

Expressiveness of Deep Learning

Alexander Bentkamp

May 14, 2024

Abstract

Deep learning has had a profound impact on computer science in recent years, with applications to search engines, image recognition and language processing, bioinformatics, and more. Recently, Cohen et al. [2] provided theoretical evidence for the superiority of deep learning over shallow learning. For my master's thesis [1], I formalized their mathematical proof using Isabelle/HOL. This formalization simplifies and generalizes the original proof, while working around the limitations of the Isabelle type system. To support the formalization, I developed reusable libraries of formalized mathematics, including results about the matrix rank, the Lebesgue measure, and multivariate polynomials, as well as a library for tensor analysis.

Contents

1	Tensor	2
2	Subtensors	8
3	Tensor Addition	11
4	Tensor Scalar Multiplication	17
5	Tensor Product	20
6	Unit Vectors as Tensors	27
7	Tensor CP-Rank	29
8	Tensor Matricization	33
9	CP-Rank and Matrix Rank	39
10	Matrix to Vector Conversion	42
11	Deep Learning Networks	43

12 Concrete Matrices	60
13 Missing Lemmas of Finite_Set	62
14 Deep Network Model	63
15 Polynomials representing the Deep Network Model	87
16 Alternative Lebesgue Measure Definition	94
17 Lebesgue Measure of Polynomial Zero Sets	96
18 Shallow Network Model	102
19 Fundamental Theorem of Network Capacity	105

1 Tensor

```
theory Tensor
imports Main
begin
```

```
typedef 'a tensor = {t::nat list × 'a list. length (snd t) = prod-list (fst t)}
by (simp add: Ex-list-of-length)
```

```
definition dims::'a tensor ⇒ nat list where
  dims A = fst (Rep-tensor A)
```

```
definition vec::'a tensor ⇒ 'a list where
  vec A = snd (Rep-tensor A)
```

```
definition tensor-from-vec::nat list ⇒ 'a list ⇒ 'a tensor where
  tensor-from-vec d v = Abs-tensor (d,v)
```

```
lemma
  assumes length v = prod-list d
  shows dims-tensor[simp]: dims (tensor-from-vec d v) = d
  and vec-tensor[simp]: vec (tensor-from-vec d v) = v
  by (simp add: Abs-tensor-inverse assms dims-def tensor-from-vec-def vec-def)+
```

```
lemma tensor-from-vec-simp[simp]: tensor-from-vec (dims A) (vec A) = A
  by (simp add: Rep-tensor-inverse Tensor.vec-def dims-def tensor-from-vec-def)
```

```
lemma length-vec: length (vec A) = prod-list (dims A)
  by (metis (mono-tags, lifting) Rep-tensor Tensor.vec-def dims-def mem-Collect-eq)
```

```
lemma tensor-eqI[intro]:
```

assumes $\text{dims } A = \text{dims } B$ **and** $\text{vec } A = \text{vec } B$
shows $A=B$
by (*metis assms tensor-from-vec-simp*)

abbreviation $\text{order}::'a \text{ tensor} \Rightarrow \text{nat}$ **where**
 $\text{order } t == \text{length } (\text{dims } t)$

inductive $\text{valid-index}::\text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{bool}$ (**infix** $\triangleleft 50$) **where**
 $\text{Nil}: [] \triangleleft [] \mid$
 $\text{Cons}: is \triangleleft ds \Longrightarrow i < d \Longrightarrow i \# is \triangleleft d \# ds$

inductive-cases $\text{valid-indexE}[\text{elim}]: is \triangleleft ds$
inductive-cases $\text{valid-index-dimsE}[\text{elim}]: is \triangleleft \text{dims } A$

lemma $\text{valid-index-length}: is \triangleleft ds \Longrightarrow \text{length } is = \text{length } ds$
by (*induction rule:valid-index.induct; auto*)

lemma $\text{valid-index-lt}: is \triangleleft ds \Longrightarrow m < \text{length } ds \Longrightarrow is!m < ds!m$
proof (*induction arbitrary:m rule:valid-index.induct*)
case *Nil*
then show ?*case* **by** *auto*
next
case *Cons*
then show ?*case* **by** (*metis gr0-conv-Suc length-Cons linorder-neqE-nat not-less-eq nth-Cons' nth-Cons-Suc*)
qed

lemma valid-indexI :
assumes $\text{length } is = \text{length } ds$ **and** $\bigwedge m. m < \text{length } ds \Longrightarrow is!m < ds!m$
shows $is \triangleleft ds$
using *assms* **proof** (*induction is arbitrary:ds*)
case *Nil*
then show ?*case* **by** (*metis length-0-conv valid-index.simps*)
next
case (*Cons a is ds*)
then obtain $d \ ds'$ **where** $ds = d \# ds'$ **by** (*metis length-Suc-conv*)
then have $is \triangleleft ds'$ **using** *Cons* **by** (*metis length-Cons less-irrefl linorder-neqE-nat not-less-eq nth-Cons-Suc*)
then show ?*case* **using** *Cons.prem2* $\langle ds = d \# ds' \rangle$ *valid-index.Cons* **by**
fastforce
qed

lemma $\text{valid-index-append}$:
assumes $is1\text{-valid}:is1 \triangleleft ds1$ **and** $is2\text{-valid}:is2 \triangleleft ds2$
shows $is1 @ is2 \triangleleft ds1 @ ds2$
apply (*rule valid-indexI[of is1 @ is2 ds1 @ ds2]*)
unfolding *nth-append*
using *valid-index-lt[OF is2-valid] valid-index-lt[OF is1-valid] valid-index-length[OF*

is1-valid] *valid-index-length*[*OF is2-valid*] *length-append*
by (*auto simp add: <length is1 = length ds1 >*)

lemma *valid-index-list-all2-iff*: *is < ds <math>\iff list-all2 (<) is ds*
by (*metis list-all2-conv-all-nth list-all2-nthD valid-indexI valid-index-length valid-index-lt*)

definition *fixed-length-sublist*::*'a list* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *'a list* **where**
*fixed-length-sublist xs l i = (take l (drop (l*i) xs))*

fun *lookup-base*::*nat list* \Rightarrow *'a list* \Rightarrow *nat list* \Rightarrow *'a* **where**
lookup-base-Nil: *lookup-base* [] *v* [] = *hd v* |
lookup-base-Cons: *lookup-base* (*d # ds*) *v* (*i # is*) =
lookup-base ds (fixed-length-sublist v (prod-list ds) i) is

definition *lookup*::*'a tensor* \Rightarrow *nat list* \Rightarrow *'a* **where**
lookup A = lookup-base (dims A) (vec A)

fun *tensor-vec-from-lookup*::*nat list* \Rightarrow (*nat list* \Rightarrow *'a*) \Rightarrow *'a list* **where**
tensor-vec-from-lookup-Nil: *tensor-vec-from-lookup* [] *e* = [*e*] |
tensor-vec-from-lookup-Cons: *tensor-vec-from-lookup* (*d # ds*) *e* = *concat (map*
($\lambda i.$ *tensor-vec-from-lookup ds (λis. e (i # is))*) [0..*d*])

definition *tensor-from-lookup*::*nat list* \Rightarrow (*nat list* \Rightarrow *'a*) \Rightarrow *'a tensor* **where**
tensor-from-lookup ds e = tensor-from-vec ds (tensor-vec-from-lookup ds e)

lemma *concat-parts-leq*:
assumes *a * d* \leq *length v*
shows *concat (map (fixed-length-sublist v d) [0..*a*]) = take (a*d) v*
using *assms* **proof** (*induction a*)
 case 0
 then show ?*case* **by** *simp*
next
 case (*Suc a*)
 then have *concat (map (fixed-length-sublist v d) [0..*a*]) = take (a * d) v* **by**
auto
 then have *concat (map (fixed-length-sublist v d) [0..*Suc a*]) =*
*take (a * d) v @ fixed-length-sublist v d a* **using** *fixed-length-sublist-def* **by**
auto
 then show ?*case* **using** *Suc* **by** (*metis add.commute mult.commute mult-Suc*
take-add fixed-length-sublist-def)
qed

lemma *concat-parts-eq*:
assumes *a * d = length v*
shows *concat (map (fixed-length-sublist v d) [0..*a*]) = v*
by (*simp add: concat-parts-leq assms*)

lemma *tensor-lookup-base*:
assumes *length v = prod-list ds*

and $\bigwedge is. is \triangleleft ds \implies \text{lookup-base } ds \ v \ is = e \ is$
shows $\text{tensor-vec-from-lookup } ds \ e = v$
using $assms$ **proof** ($\text{induction } ds \ \text{arbitrary}; v \ e$)
case Nil
then show $?case \ \text{unfolding } \text{tensor-vec-from-lookup.simps}$
by ($\text{metis } One\text{-nat-def } Tensor.\text{lookup-base-}Nil \ \text{length-0-conv } \text{length-Suc-conv}$
 $\text{list.sel}(1) \ \text{prod-list.Nil } \text{valid-index.Nil}$)
next
case ($Cons \ a \ ds$)
then have $\langle \text{length } v = \text{prod-list } ds * a \rangle$ **by** $auto$
{
fix i **assume** $\langle i < a \rangle$
then have $\langle Suc \ i \leq a \rangle$
by $simp$
then have $\text{prod-list } ds * Suc \ i \leq \text{length } v$
using $\langle \text{length } v = \text{prod-list } ds * a \rangle$
by ($\text{simp only: mult-le-mono2}$)
have $\bigwedge is'. is' \triangleleft ds \implies e \ (i \# is') = \text{lookup-base } ds \ (\text{fixed-length-sublist } v$
 $(\text{prod-list } ds) \ i) \ is'$
using $\langle i < a \rangle$ **by** ($\text{metis } Cons.\text{prems}(2) \ Tensor.\text{lookup-base-}Cons \ \text{valid-index.simps}$)
then have $\text{tensor-vec-from-lookup } ds \ (\lambda is'. e \ (i \# is')) = \text{fixed-length-sublist } v$
 $(\text{prod-list } ds) \ i$
using $Cons$ **using** $\langle \text{prod-list } ds * Suc \ i \leq \text{length } v \rangle$ **by** ($\text{simp add: } Cons.IH$
 $\text{fixed-length-sublist-def}$)
}
then show $?case \ \text{unfolding } \text{tensor-vec-from-lookup-}Cons \ \text{lookup-base-}Cons$
using $\text{atLeastLessThan-iff } \text{map-eq-conv } \text{set-upt } Cons \ \text{concat-parts-eq } \text{prod-list.Cons}$
by ($\text{metis } (no\text{-types, lifting})$)
qed

lemma tensor-lookup :

assumes $\bigwedge is. is \triangleleft \text{dims } A \implies \text{lookup } A \ is = e \ is$
shows $\text{tensor-from-lookup } (\text{dims } A) \ e = A$
using $\text{tensor-lookup-base } \text{lookup-def } \text{length-vec } \text{tensor-from-lookup-def}$ **by** (metis
 $assms \ \text{tensor-from-vec-simp}$)

lemma $\text{concat-equal-length}$:

assumes $\bigwedge xs. xs \in \text{set } xss \implies \text{length } xs = l$
shows $\text{length } (\text{concat } xss) = \text{length } xss * l$
using $assms$ **by** ($\text{induction } xss; auto$)

lemma $\text{concat-equal-length-map}$:

assumes $\bigwedge i. i < a \implies \text{length } (f \ i) = d$
shows $\text{length } (\text{concat } (\text{map } (\lambda i. f \ i) \ [0..<a])) = a * d$
using $assms$ **by** ($\text{induction } a; auto$)

lemma concat-parts :

assumes $\bigwedge xs. xs \in \text{set } xss \implies \text{length } xs = d$ **and** $i < \text{length } xss$
shows $\text{fixed-length-sublist } (\text{concat } xss) \ d \ i = xss \ ! \ i$

```

using assms proof (induction xss arbitrary:i)
  case Nil
  then show ?case by simp
next
  case (Cons xs xss)
  then have length (concat xss) = length xss * d by (simp add: Cons.prem1)
concat-equal-length)
  show ?case
  proof (cases i)
    case 0
    then have fixed-length-sublist (concat (xs # xss)) d i = xs
      unfolding fixed-length-sublist-def by (simp add: Cons.prem1)
    then show ?thesis using 0 by auto
  next
  case (Suc i')
  then have fixed-length-sublist (concat xss) d i' = xss ! i' using Cons by auto
  then show ?thesis unfolding fixed-length-sublist-def using Suc Cons.prem1)
by auto
  qed
qed

```

lemma *concat-parts'*:

assumes $\bigwedge i. i < a \implies \text{length } (f i) = d$

and $i < a$

shows *fixed-length-sublist (concat (map ($\lambda i. f i$) [0..*a*])) d i = f i*

using *assms* **proof** (*induction a*)

case *0*

then show *?case* **by** *simp*

next

case (*Suc a*)

then have $(\bigwedge i. i < a \implies \text{length } (f i) = d)$ **by** *auto*

then have *length (concat (map f [0..*a*])) = a * d* **using** *concat-equal-length-map*

by *auto*

show *?case*

proof (*cases i=a*)

assume $i=a$

then have *fixed-length-sublist (concat (map f [0..*Suc a*])) d i = f a*

by (*simp add: Suc.prem1*) $\langle \text{length } (\text{concat } (\text{map } f [0..*a*])) = a * d \rangle$

fixed-length-sublist-def)

then show *?case* **using** $\langle i=a \rangle$ **by** *auto*

next

assume $i \neq a$

then have *fixed-length-sublist (concat (map f [0..*a*])) d i = f i*

*concat (map f [0..*Suc a*]) = concat (map f [0..*a*) @ f a* **using** *Suc* **by**

auto

show *?case* **unfolding** $\langle \text{concat } (\text{map } f [0..*Suc a*]) = \text{concat } (\text{map } f [0..*a*) @ f a \rangle$

@ f a

unfolding *fixed-length-sublist-def drop-append*

using $\langle \text{length } (\text{concat } (\text{map } f [0..*a*])) = a * d \rangle$ $\langle \text{fixed-length-sublist } (\text{concat}$

$(\text{map } f [0..<a]) d \ i = f \ i$
using *append-assoc append-eq-conv-conj append-take-drop-id assms(1) assms(2)*
fixed-length-sublist-def
by *metis*
qed
qed

lemma *length-tensor-vec-from-lookup*:
 $\text{length } (\text{tensor-vec-from-lookup } ds \ e) = \text{prod-list } ds$
by (*induction ds arbitrary:e; auto simp add: concat-equal-length-map*)

lemma *lookup-tensor-vec*:
assumes $is \triangleleft ds$
shows $\text{lookup-base } ds \ (\text{tensor-vec-from-lookup } ds \ e) \ is = e \ is$
using *assms proof (induction arbitrary:e rule:valid-index.induct)*
case *Nil*
then show *?case by simp*
next
case (*Cons is ds i d e*)
then show *?case unfolding tensor-vec-from-lookup-Cons lookup-base-Cons*
by (*simp add: length-tensor-vec-from-lookup concat-parts'[of d $\lambda i. \text{tensor-vec-from-lookup } ds \ (\lambda is. e \ (i \ \# \ is)) \ \text{prod-list } ds \ i] \ (i < d)$*)
qed

lemma *lookup-tensor-from-lookup*:
assumes $is \triangleleft ds$
shows $\text{lookup } (\text{tensor-from-lookup } ds \ e) \ is = e \ is$
unfolding *lookup-def tensor-from-lookup-def*
by (*simp add: lookup-tensor-vec assms length-tensor-vec-from-lookup*)

lemma *dims-tensor-from-lookup*: $\text{dims } (\text{tensor-from-lookup } ds \ e) = ds$
unfolding *tensor-from-lookup-def*
by (*simp add: length-tensor-vec-from-lookup*)

lemma *tensor-lookup-cong*:
assumes $\text{tensor-from-lookup } ds \ e_1 = \text{tensor-from-lookup } ds \ e_2$
and $is \triangleleft ds$
shows $e_1 \ is = e_2 \ is$ **using** *assms lookup-tensor-from-lookup by metis*

lemma *tensor-from-lookup-eqI*:
assumes $\bigwedge is. is \triangleleft ds \implies e_1 \ is = e_2 \ is$
shows $\text{tensor-from-lookup } ds \ e_1 = \text{tensor-from-lookup } ds \ e_2$
by (*metis assms lookup-tensor-vec length-tensor-vec-from-lookup tensor-lookup-base tensor-from-lookup-def*)

lemma *tensor-lookup-eqI*:
assumes $\text{dims } A = \text{dims } B$ **and** $\bigwedge is. is \triangleleft (\text{dims } A) \implies \text{lookup } A \ is = \text{lookup } B \ is$
shows $A = B$ **by** (*metis assms(1) assms(2) tensor-lookup*)

end

2 Subtensors

theory *Tensor-Subtensor*
imports *Tensor*
begin

definition *subtensor*::'a tensor \Rightarrow nat \Rightarrow 'a tensor **where**
 subtensor A i = *tensor-from-vec* (tl (dims A)) (*fixed-length-sublist* (vec A) (*prod-list*
 (tl (dims A)))) i

definition *subtensor-combine*::nat list \Rightarrow 'a tensor list \Rightarrow 'a tensor **where**
 subtensor-combine ds As = *tensor-from-vec* (length As # ds) (*concat* (*map* vec
 As))

lemma *length-fixed-length-sublist*[simp]:
assumes (Suc i)*l \leq length xs
shows length (*fixed-length-sublist* xs l i) = l
 unfolding *fixed-length-sublist-def*
 by (*metis* *assms* *diff-add-inverse2* *length-drop* *length-take* *min.absorb2* *mult.commute*
 mult-Suc *take-drop*)

lemma *vec-subtensor*[simp]:
assumes dims A \neq [] **and** i < hd (dims A)
shows vec (*subtensor* A i) = *fixed-length-sublist* (vec A) (*prod-list* (tl (dims A))) i
 by (*metis* (*no-types*, *lifting*) *Suc-leI* *assms(1)* *assms(2)* *hd-Cons-tl* *length-fixed-length-sublist*
 length-vec *prod-list.Cons* *mult-le-mono1* *subtensor-def* *vec-tensor*)

lemma *dims-subtensor*[simp]:
assumes dims A \neq [] **and** i < hd (dims A)
shows dims (*subtensor* A i) = tl (dims A)
 using *Suc-leI* *assms(1)* *assms(2)* *dims-tensor* *length-fixed-length-sublist* *length-vec*
 list.collapse *prod-list.Cons* *mult-le-mono1* *subtensor-def*
 by *metis*

lemma *subtensor-combine-subtensor*[simp]:
assumes dims A \neq []
shows *subtensor-combine* (tl (dims A)) (*map* (*subtensor* A) [0..*hd* (dims A)]) =
 A
proof –
 have *length-vec-A*: *hd* (dims A) * *prod-list* (tl (dims A)) = length (*Tensor.vec* A)
 by (*metis* *assms* *length-vec* *list.collapse* *prod-list.Cons*)
 let ?*subtensor-vec* = *fixed-length-sublist* (vec A) (*prod-list* (tl (dims A)))
 {
 fix i **assume** i < *hd* (dims A)
 then have (Suc i)*(*prod-list* (tl (dims A))) \leq length (vec A)
 by (*metis* *Suc-leI* *length-vec-A* *mult-le-mono1*)
 then have (vec \circ (λ i. *tensor-from-vec* (tl (dims A)) (?*subtensor-vec* i))) i =


```

?subtensor-vec i
  by simp
}
then have 1:map (Tensor.vec ◦ (λi. tensor-from-vec (tl (dims A)) (?subtensor-vec
i))) [0..<hd (dims A)]
  = map ?subtensor-vec [0..<hd (dims A)] by auto
then have subtensor-combine (tl (dims A)) (map (λi. subtensor A i) [0..<hd
(dims A)]) = A
  unfolding subtensor-combine-def subtensor-def using concat-parts-eq[OF length-vec-A]
  by (auto simp add: 1 assms)
then show ?thesis by auto
qed

```

lemma

```

assumes ∧A. A∈set As ⇒ dims A = ds
shows subtensor-combine-dims[simp]: dims (subtensor-combine ds As) = length As
# ds (is ?D)
and subtensor-combine-vec[simp]: vec (subtensor-combine ds As) = concat (map
vec As) (is ?V)
proof –
  have ∧v. v∈set (map Tensor.vec As) ⇒ length v = prod-list ds using assms
length-vec by fastforce
  then have length As * prod-list ds = length (concat (map Tensor.vec As)) using
concat-equal-length
  by (metis length-map)
  then show ?D ?V unfolding subtensor-combine-def by simp+
qed

```

lemma subtensor-subtensor-combine:

```

assumes ∧A. A∈set As ⇒ dims A = ds and i < length As
shows subtensor (subtensor-combine ds As) i = As ! i
proof –
  have fixed-length-sublist (concat (map vec As)) (prod-list ds) i = vec (As ! i)
  using concat-parts[of map vec As prod-list ds i] assms imageE length-map
length-vec
  nth-map set-map in-set-conv-nth by fastforce
  then show ?thesis
  unfolding subtensor-def using subtensor-combine-dims subtensor-combine-vec
  by (metis assms list.sel(3) nth-mem tensor-from-vec-simp)
qed

```

lemma subtensor-induct[case-names order-0 order-step]:

```

assumes order-0: ∧A. dims A = [] ⇒ P A
and order-step: ∧A. dims A ≠ [] ⇒ (∧i. i < hd (dims A) ⇒ P (subtensor A
i)) ⇒ P A
shows P B
using assms proof (induction dims B arbitrary:B)
  case Nil
  then show ?case by auto

```

```

next
  case Cons
  then show ?case by (metis dims-subtensor list.sel(3))
qed

lemma subtensor-combine-induct[case-names order-0 order-step]:
assumes order-0:  $\bigwedge A. \text{dims } A = [] \implies P A$ 
and order-step:  $\bigwedge As ds. (\bigwedge A. A \in \text{set } As \implies P A) \implies (\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds) \implies P (\text{subtensor-combine } ds As)$ 
shows  $P A$ 
proof (induction A rule: subtensor-induct)
  case (order-0 A)
  then show ?case by (simp add: assms(1))
next
  case (order-step A)
  have  $P (\text{subtensor-combine } (\text{tl } (\text{dims } A)) (\text{map } (\text{subtensor } A) [0..<\text{hd } (\text{dims } A)]))$ 
  apply (rule assms(2))
  using atLeastLessThan-iff dims-subtensor imageE set-map set-upt order-step
  by auto
  then show ?case using subtensor-combine-subtensor[OF order-step.hyps] by
  metis
qed

lemma lookup-subtensor1[simp]:
assumes  $i \# is \triangleleft \text{dims } A$ 
shows  $\text{lookup } (\text{subtensor } A i) is = \text{lookup } A (i \# is)$ 
using assms
proof (induction A rule: subtensor-combine-induct)
  case order-0
  then show ?case by auto
next
  case (order-step As ds)
  have  $0:\text{subtensor } (\text{subtensor-combine } ds As) i = As ! i$ 
  by (metis list.discI list.sel(1) order-step.hyps order-step.prem1 subtensor-combine-dims
  subtensor-subtensor-combine valid-index-dimsE)
  have  $1:\text{dims } (\text{subtensor-combine } ds As) = \text{length } As \# ds$ 
  using order-step subtensor-combine-def subtensor-combine-dims by force
  show ?case unfolding 0 lookup-def 1 unfolding lookup-base-Cons using
  order-step.prem1
  using Tensor.lookup-base-Cons dims-subtensor lookup-def list.discI list.sel(1)
  list.sel(3) valid-index-dimsE vec-subtensor by (metis 0 1)
qed

lemma lookup-subtensor:
assumes  $is \triangleleft \text{dims } A$ 
shows  $\text{lookup } A is = \text{hd } (\text{vec } (\text{fold } (\lambda i A. \text{subtensor } A i) is A))$ 
using assms proof (induction is arbitrary: A)
  case Nil
  then show ?case by (metis Tensor.lookup-base-Nil lookup-def fold-simps(1))

```

```

length-0-conv valid-index-length)
next
  case (Cons a is A)
  then show ?case
  using dims-subtensor lookup-subtensor1 fold-simps(2) list.discI list.sel(1) list.sel(3)
  valid-indexE by (metis (no-types, lifting))
qed

lemma subtensor-eqI:
assumes dims A ≠ []
and dims-eq: dims A = dims B
and  $\bigwedge i. i < \text{hd} (\text{dims } A) \implies \text{subtensor } A \ i = \text{subtensor } B \ i$ 
shows A=B
proof -
  {
    fix is assume is < dims A
    then obtain i is' where is-Cons: is = i # is' using assms(1) by blast
    then have lookup A is = lookup B is
      using lookup-subtensor1 assms by (metis <is < dims A> is-Cons list.sel(1)
valid-index-dimsE)
  }
  then show ?thesis using tensor-lookup-eqI[OF dims-eq] by auto
qed

end

```

3 Tensor Addition

```

theory Tensor-Plus
imports Tensor-Subtensor
begin

```

```

definition vec-plus a b = map ( $\lambda(x,y). \text{plus } x \ y$ ) (zip a b)

```

```

definition plus-base:: 'a::semigroup-add tensor  $\Rightarrow$  'a tensor  $\Rightarrow$  'a tensor
where plus-base A B = (tensor-from-vec (dims A) (vec-plus (vec A) (vec B)))

```

```

instantiation tensor:: (semigroup-add) plus
begin

```

```

  definition plus-def: A + B = (if (dims A = dims B)
    then plus-base A B
    else undefined)

```

```

  instance ..
end

```

```

lemma plus-dim1[simp]: dims A = dims B  $\implies$  dims (A + B) = dims A unfolding
plus-def plus-base-def

```

using *dims-tensor length-vec length-map map-fst-zip vec-plus-def* **by** (*metis (full-types)*)
lemma *plus-dim2[simp]*: $\text{dims } A = \text{dims } B \implies \text{dims } (A + B) = \text{dims } B$ **using**
plus-dim1 **by** *metis*
lemma *plus-base*: $\text{dims } A = \text{dims } B \implies A + B = \text{plus-base } A B$ **unfolding**
plus-def **by** *metis*

lemma *fixed-length-sublist-plus*:
assumes $\text{length } xs1 = c * l$ $\text{length } xs2 = c * l$ $i < c$
shows *fixed-length-sublist* (*vec-plus* *xs1* *xs2*) *l* *i*
 $= \text{vec-plus } (\text{fixed-length-sublist } xs1 \ l \ i) \ (\text{fixed-length-sublist } xs2 \ l \ i)$
unfolding *vec-plus-def fixed-length-sublist-def* **using** *drop-map drop-zip take-map*
take-zip **by** *metis*

lemma *vec-plus[simp]*:
assumes $\text{dims } A = \text{dims } B$
shows $\text{vec } (A+B) = \text{vec-plus } (\text{vec } A) (\text{vec } B)$
unfolding *plus-def plus-base-def vec-plus-def* **using** *assms*
by (*auto*; *metis (no-types, lifting) length-map length-tensor-vec-from-lookup map-fst-zip*
tensor-lookup tensor-from-lookup-def vec-tensor)

lemma *subtensor-plus*:
fixes *A::'a::semigroup-add tensor* **and** *B::'a::semigroup-add tensor*
assumes $i < \text{hd } (\text{dims } A)$
and $\text{dims } A = \text{dims } B$
and $\text{dims } A \neq []$
shows $\text{subtensor } (A + B) \ i = \text{subtensor } A \ i + \text{subtensor } B \ i$
proof –
have $\text{length } (\text{vec } A) = \text{hd } (\text{dims } A) * \text{prod-list } (\text{tl } (\text{dims } A))$
 $\text{length } (\text{Tensor.vec } B) = \text{hd } (\text{dims } A) * \text{prod-list } (\text{tl } (\text{dims } A))$
using *length-vec prod-list.Cons assms* **by** (*metis (no-types) list.exhaust-sel*) +
then show *?thesis*
using *Tensor-Plus.vec-plus assms fixed-length-sublist-plus vec-subtensor ten-*
tensor-eqI
dims-subtensor plus-dim1 **by** *fastforce*
qed

lemma *lookup-plus[simp]*:
assumes $\text{dims } A = \text{dims } B$
and $is \triangleleft \text{dims } A$
shows $\text{lookup } (A + B) \ is = \text{lookup } A \ is + \text{lookup } B \ is$
using *assms* **proof** (*induction A+B arbitrary:A B is rule: subtensor-induct*)
case (*order-0 A B is*)
then have $is = []$ **by** *auto*
have $1: [] \triangleleft \text{dims } A$ **using** *order-0 <is = []>* **by** *auto*
have $2: [] \triangleleft \text{dims } B$ **using** *order-0 <is = []>* **by** *auto*
have $3: [] \triangleleft \text{dims } (A + B)$ **using** *order-0 <is = []>* **by** *auto*
have $\text{length } (\text{vec } A) = 1$ $\text{length } (\text{vec } B) = 1$
by (*metis length-vec prod-list.Nil order-0.hyps order-0.prem1 plus-dim1*) +
then show *?case* **unfolding** *lookup-subtensor[OF 1] lookup-subtensor[OF 2]*

```

lookup-subtensor[OF 3] ‹is = []›
  fold-simps(1) vec-plus[OF order-0.premis(1)] unfolding vec-plus-def using or-
der-0.premis length-map
  list.map-sel(1) list.size(3) map-fst-zip map-snd-zip order-0.hyps
  zero-neq-one case-prod-unfold length-vec by metis
next
case (order-step A B is)
then obtain i is' where is = i # is' by auto
have 1:is < dims A using order-step by auto
have 2:is < dims B using order-step by auto
have 3:is < dims (A + B) using order-step by auto
have lookup (subtensor A i + subtensor B i) is' = lookup (subtensor A i) is' +
lookup (subtensor B i) is'
  apply (rule order-step.hyps(2)[of i])
  using ‹is = i # is'› 3 hd-conv-nth length-greater-0-conv nth-Cons-0 or-
der-step.hyps(1) valid-index-lt
  apply auto[1]
  apply (metis 2 ‹is = i # is'› list.inject list.sel(1) list.simps(3) order-step.premis(1)
subtensor-plus valid-index.cases)
  using 1 ‹is = i # is'› order-step.premis(1) plus-dim1 apply auto[1]
  using 1 ‹is = i # is'› plus-dim1 by auto
then show ?case using lookup-subtensor[OF 1] lookup-subtensor[OF 2] lookup-subtensor[OF
3]
  using order-step ‹is = i # is'› plus-dim1 lookup-subtensor1 list.sel(1) subten-
sor-plus valid-index-dimsE by metis
qed

```

lemma plus-assoc:

```

assumes dimsA:dims A = ds and dimsB:dims B = ds and dimsC:dims C = ds
shows (A + B) + C = A + (B + C)
by (rule tensor-lookup-eqI; simp add: dimsA dimsB dimsC add.assoc)+

```

lemma tensor-comm[simp]:

fixes A::'a::ab-semigroup-add tensor

shows A + B = B + A

proof (cases dims A = dims B)

case True

then show ?thesis unfolding plus-def plus-base-def

```

  using add commute lookup-plus[OF True] plus-dim1[OF True] tensor-lookup-eqI[OF
True] vec-plus[OF True]

```

by (metis lookup-plus plus-dim1 tensor-lookup-eqI vec-plus)

next

case False

then show ?thesis unfolding plus-def plus-base-def by simp

qed

definition vec0 n = replicate n 0

definition tensor0::nat list \Rightarrow 'a::zero tensor **where**

$tensor0\ d = tensor\ from\ vec\ d\ (vec0\ (prod\ list\ d))$

lemma $dims\ tensor0[simp]$: $dims\ (tensor0\ d) = d$
and $vec\ tensor0[simp]$: $vec\ (tensor0\ d) = vec0\ (prod\ list\ d)$
unfolding $tensor0\ def\ vec0\ def$ **by** $simp\ all$

lemma $lookup\ is\ in\ vec$: $is\ \triangleleft\ (dims\ A) \implies lookup\ A\ is \in set\ (vec\ A)$

proof ($induction\ arbitrary:is\ rule:subtensor\ induct$)

case $order\ 0$

then show $?case$ **unfolding** $lookup\ def$ **using** $lookup\ base\ Nil$

by ($metis\ length\ 0\ conv\ length\ vec\ list.set\ sel(1)\ prod\ list.Nil\ valid\ index\ length\ zero\ neq\ one$)

next

case ($order\ step\ A\ is$)

then obtain $i\ is'$ **where** $is = i \# is'$ **using** $valid\ index\ dimsE$ **by** $blast$

then have $1:i < hd\ (dims\ A)$ **using** $dims\ def\ order\ step.prem$ **by** $auto$

have $2:is' \triangleleft dims\ (subtensor\ A\ i)$ **using** $\langle is = i \# is' \rangle\ dims\ subtensor\ order\ step.prem$ **by** $auto$

have $lookup\ A\ is \in set\ (Tensor.vec\ (subtensor\ A\ i))$

using $order\ step.IH\ [OF\ 1\ 2]\ lookup\ subtensor1\ \langle is = i \# is' \rangle\ order\ step.prem$

by $auto$

then show $?case$ **using** $vec\ subtensor\ fixed\ length\ sublist\ def$ **by** ($metis\ 1\ in\ set\ dropD\ in\ set\ takeD\ order\ step.hyps$)

qed

lemma $lookup\ tensor0$:

assumes $is \triangleleft ds$

shows $lookup\ (tensor0\ ds)\ is = 0$

proof $-$

have $lookup\ (tensor0\ ds)\ is \in set\ (vec\ (tensor0\ ds))$ **using** $lookup\ is\ in\ vec\ assms$
by ($metis\ dims\ tensor0$)

moreover have $set\ (vec\ (tensor0\ ds)) \subseteq \{0\}$ **unfolding** $vec\ tensor0\ vec0\ def$
by ($metis\ in\ set\ replicate\ singleton\ iff\ subsetI$)

ultimately show $?thesis$ **by** $auto$

qed

lemma

fixes $A::'a::monoid\ add\ tensor$

shows $tensor\ add\ 0\ right[simp]$: $A + tensor0\ (dims\ A) = A$

unfolding $plus\ def\ plus\ base\ def\ dims\ tensor0$

apply ($simp\ all$)

apply ($rule\ tensor\ lookup\ eqI$)

apply ($metis\ (no\ types,\ lifting)\ dims\ tensor\ dims\ tensor0\ length\ vec\ plus\ dim2\ vec\ plus\ vec\ tensor0$)

by ($metis\ add.\ right\ neutral\ dims\ tensor0\ lookup\ plus\ lookup\ tensor0\ plus\ dim2\ tensor\ from\ vec\ simp\ vec\ plus\ vec\ tensor0$)

lemma

fixes $A::'a::monoid\ add\ tensor$

shows *tensor-add-0-left*[*simp*]: $\text{tensor0 } (\text{dims } A) + A = A$
unfolding *plus-def plus-base-def dims-tensor0*
apply (*simp-all*)
apply (*rule tensor-lookup-eqI*)
apply (*metis (no-types, lifting) dims-tensor dims-tensor0 length-vec plus-dim2 vec-plus vec-tensor0*)
by (*metis add.left-neutral dims-tensor0 lookup-plus lookup-tensor0 plus-dim2 tensor-from-vec-simp vec-plus vec-tensor0*)

definition *listsum*:: $\text{nat list} \Rightarrow 'a::\text{monoid-add tensor list} \Rightarrow 'a \text{ tensor}$ **where**
listsum ds As = foldr (+) As (tensor0 ds)

definition *listsum'*:: $'a::\text{monoid-add tensor list} \Rightarrow 'a \text{ tensor}$ **where**
listsum' As = listsum (dims (hd As)) As

lemma *listsum-Nil*: $\text{listsum } ds [] = \text{tensor0 } ds$ **by** (*simp add: Tensor-Plus.listsum-def*)

lemma *listsum-one*: $\text{listsum } (\text{dims } A) [A] = A$ **unfolding** *listsum-def* **by** *simp*

lemma *listsum-Cons*: $\text{listsum } ds (A \# As) = A + \text{listsum } ds As$
unfolding *listsum-def* **by** *auto*

lemma *listsum-dims*:
assumes $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$
shows $\text{dims } (\text{listsum } ds As) = ds$
using *assms* **proof** (*induction As*)
case *Nil*
then show *?case* **by** (*metis dims-tensor0 listsum-Nil*)
next
case (*Cons A As*)
then show *?case* **using** *listsum-Cons*
by (*metis list.set-intros(1) list.set-intros(2) plus-dim2*)
qed

lemma *subtensor0*:
assumes $ds \neq []$ **and** $i < \text{hd } ds$
shows $\text{subtensor } (\text{tensor0 } ds) i = \text{tensor0 } (\text{tl } ds)$
proof (*rule tensor-lookup-eqI*)
show $1:\text{dims } (\text{subtensor } (\text{tensor0 } ds) i) = \text{dims } (\text{tensor0 } (\text{tl } ds))$ **by** (*simp add: assms(1) assms(2)*)
fix *is* **assume** $is \triangleleft \text{dims } (\text{subtensor } (\text{tensor0 } ds) i)$
then have $i \# is \triangleleft \text{dims } (\text{tensor0 } ds)$ **using** *assms(1) assms(2) valid-index.Cons*
by *fastforce*
then show $\text{lookup } (\text{subtensor } (\text{tensor0 } ds) i) is = \text{lookup } (\text{tensor0 } (\text{tl } ds)) is$
using *lookup-subtensor1 1 < is < dims (subtensor (tensor0 ds) i) > dims-tensor0 lookup-tensor0*
by *metis*
qed

lemma *subtensor-listsum*:
assumes $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$
and $ds \neq []$ **and** $i < \text{hd } ds$
shows $\text{subtensor } (\text{listsum } ds \text{ } As) \ i = \text{listsum } (\text{tl } ds) \ (\text{map } (\lambda A. \text{subtensor } A \ i) \ As)$
using *assms* **proof** (*induction* *As*)
 case *Nil*
 then show *?case* **using** *lookup-tensor0* *assms(2)* *assms(3)* *subtensor0* **by** (*auto*
simp add: listsum-Nil)
 next
 case (*Cons* *A* *As*)
 then show *?case* **by** (*simp add: listsum-Cons; metis subtensor-plus listsum-dims*)
qed

lemma *listsum0*:
assumes $\bigwedge A. A \in \text{set } As \implies A = \text{tensor0 } ds$
shows $\text{listsum } ds \text{ } As = \text{tensor0 } ds$
using *assms* **proof** (*induction* *As*)
 case *Nil*
 show *?case* **by** (*simp add: listsum-Nil*)
 next
 case *Cons*
 then show *?case* **using** *listsum-Cons*
 by (*metis dims-tensor0 list.set-intros(1) set-subset-Cons subsetCE tensor-add-0-right*)
qed

lemma *listsum-all-0-but-one*:
assumes $\bigwedge i. i \neq j \implies i < \text{length } As \implies As!i = \text{tensor0 } ds$
and $\text{dims } (As!j) = ds$
and $j < \text{length } As$
shows $\text{listsum } ds \text{ } As = As!j$
using *assms* **proof** (*induction* *As* *arbitrary:j*)
 case *Nil*
 then show *?case* **by** *auto*
 next
 case (*Cons* *A* *As* *j*)
 then show *?case*
 proof (*cases* *j*)
 case *0*
 then have $\bigwedge i. i < \text{length } As \implies As!i = \text{tensor0 } ds$ **using** *Cons* **using**
Suc-less-eq length-Cons list.sel(3) nat.simps(3) nth-tl **by** *fastforce*
 then have $\text{listsum } ds \text{ } As = \text{tensor0 } ds$ **using** *listsum0* **by** (*metis in-set-conv-nth*)
 then show *?thesis* **by** (*metis 0 Cons.prem(2) listsum-Cons nth-Cons-0 tensor-add-0-right*)
 next
 case (*Suc* *j'*)
 then have $\text{listsum } ds \text{ } As = As!j'$ **by** (*metis (no-types, lifting) Cons.IH Cons.prem(1) Cons.prem(2) Cons.prem(3) Suc-less-eq length-Cons less-Suc-eq list.sel(3) not-less-eq*)

nth-tl)
then show *?thesis* **by** (*metis Cons.premis(1) Cons.premis(2) Suc length-greater-0-conv list.simps(3) listsum-Cons nat.simps(3) nth-Cons-0 nth-Cons-Suc tensor-add-0-left*)
qed
qed

lemma *lookup-listsum*:
assumes *is* \triangleleft *ds*
and $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$
shows *lookup (listsum ds As) is* = $(\sum A \leftarrow As. \text{lookup } A \text{ is})$
using *assms* **proof** (*induction As*)
 case *Nil*
 then show *?case* **by** (*simp add: assms(1) listsum-Nil lookup-tensor0*)
next
 case (*Cons A As*)
 then show *?case* **by** (*simp add: listsum-Cons list.set-intros listsum-dims*)
qed

end

4 Tensor Scalar Multiplication

theory *Tensor-Scalar-Mult*
imports *Tensor-Plus Tensor-Subtensor*
begin

definition *vec-smult::'a::ring \Rightarrow 'a list \Rightarrow 'a list* **where**
vec-smult α β = map (() α) β*

lemma *vec-smult0: vec-smult 0 as = vec0 (length as)*
by (*induction as; auto simp add:vec0-def vec-smult-def*)

lemma *vec-smult-distr-right*:
shows *vec-smult ($\alpha + \beta$) as = vec-plus (vec-smult α as) (vec-smult β as)*
 unfolding *vec-smult-def vec-plus-def*
 by (*induction as; simp add: distrib-right*)

lemma *vec-smult-Cons*:
shows *vec-smult α (a # as) = ($\alpha * a$) # vec-smult α as* **by** (*simp add: vec-smult-def*)

lemma *vec-plus-Cons*:
shows *vec-plus (a # as) (b # bs) = (a+b) # vec-plus as bs* **by** (*simp add: vec-plus-def*)

lemma *vec-smult-distr-left*:
assumes *length as = length bs*
shows *vec-smult α (vec-plus as bs) = vec-plus (vec-smult α as) (vec-smult α bs)*
using *assms* **proof** (*induction as arbitrary:bs*)

```

case Nil
then show ?case unfolding vec-smult-def vec-plus-def by simp
next
  case (Cons a as')
  then obtain b bs' where  $bs = b \# bs'$  by (metis Suc-length-conv)
  then have  $0 : \text{vec-smult } \alpha \ (\text{vec-plus } (a \# as') \ bs) = (\alpha * (a+b)) \# \text{vec-smult } \alpha \ (\text{vec-plus } as' \ bs')$ 
  unfolding vec-smult-def vec-plus-def using Cons.IH[of bs'] by simp
  have  $\text{length } bs' = \text{length } as'$  using Cons.prems  $\langle bs = b \# bs' \rangle$  by auto
  then show ?case unfolding 0 unfolding  $\langle bs = b \# bs' \rangle$  vec-smult-Cons
vec-plus-Cons
  by (simp add: Cons.IH distrib-left)
qed

```

lemma *length-vec-smult*: $\text{length } (\text{vec-smult } \alpha \ v) = \text{length } v$ **unfolding** *vec-smult-def* **by** *simp*

definition *smult*:: $'a::\text{ring} \Rightarrow 'a \ \text{tensor} \Rightarrow 'a \ \text{tensor}$ (**infixl** \cdot 70) **where**
smult $\alpha \ A = (\text{tensor-from-vec } (\text{dims } A) \ (\text{vec-smult } \alpha \ (\text{vec } A)))$

lemma *tensor-smult0*: **fixes** $A::'a::\text{ring} \ \text{tensor}$
shows $0 \cdot A = \text{tensor0 } (\text{dims } A)$
unfolding *smult-def* *tensor0-def* *vec-smult-def* **using** *vec-smult0* *length-vec*
by (*metis (no-types) vec-smult-def*)

lemma *dims-smult*[*simp*]: $\text{dims } (\alpha \cdot A) = \text{dims } A$
and *vec-smult*[*simp*]: $\text{vec } (\alpha \cdot A) = \text{map } ((*) \ \alpha) \ (\text{vec } A)$
unfolding *smult-def* *vec-smult-def* **by** (*simp add: length-vec*) $+$

lemma *tensor-smult-distr-right*: $(\alpha + \beta) \cdot A = \alpha \cdot A + \beta \cdot A$
unfolding *plus-def* *plus-base-def*
by (*auto*; *metis smult-def vec-smult-def vec-smult-distr-right*)

lemma *tensor-smult-distr-left*: $\text{dims } A = \text{dims } B \Longrightarrow \alpha \cdot (A + B) = \alpha \cdot A + \alpha \cdot B$

proof –

```

assume a1:  $\text{dims } A = \text{dims } B$ 
then have f2:  $\text{length } (\text{vec-plus } (\text{vec } A) \ (\text{vec } B)) = \text{length } (\text{vec } A)$ 
  by (simp add: length-vec vec-plus-def)
have f3:  $\text{dims } (\text{tensor-from-vec } (\text{dims } B) \ (\text{vec-smult } \alpha \ (\text{vec } A))) = \text{dims } B$ 
  using a1 by (simp add: length-vec vec-smult-def)
have f4:  $\text{vec } (\alpha \cdot A) = \text{vec-smult } \alpha \ (\text{vec } A)$ 
  by (simp add: vec-smult-def)
have  $\text{length } (\text{vec-smult } \alpha \ (\text{vec } B)) = \text{length } (\text{vec } B)$ 
  by (simp add: vec-smult-def)
then show ?thesis
  unfolding plus-def plus-base-def using f4 f3 f2 a1
  by (simp add: length-vec smult-def vec-smult-distr-left)

```

qed

lemma *smult-fixed-length-sublist*:

assumes $\text{length } xs = l * c \ i < c$

shows $\text{fixed-length-sublist } (\text{vec-smult } \alpha \ xs) \ l \ i = \text{vec-smult } \alpha \ (\text{fixed-length-sublist } xs \ l \ i)$

unfolding *fixed-length-sublist-def vec-smult-def* **by** (*simp add: drop-map take-map*)

lemma *smult-subtensor*:

assumes $\text{dims } A \neq [] \ i < \text{hd } (\text{dims } A)$

shows $\alpha \cdot \text{subtensor } A \ i = \text{subtensor } (\alpha \cdot A) \ i$

proof (*rule tensor-eqI*)

show $\text{dims } (\alpha \cdot \text{subtensor } A \ i) = \text{dims } (\text{subtensor } (\alpha \cdot A) \ i)$

using *dims-smult dims-subtensor assms(1) assms(2)* **by** *simp*

show $\text{vec } (\alpha \cdot \text{subtensor } A \ i) = \text{vec } (\text{subtensor } (\alpha \cdot A) \ i)$

unfolding *vec-smult*

unfolding *vec-subtensor[OF <dims A ≠ []> <i < hd (dims A)>]*

using *vec-subtensor[of α · A i]*

by (*simp add: assms(1) assms(2) drop-map fixed-length-sublist-def take-map*)

qed

lemma *lookup-smult*:

assumes $is \triangleleft \text{dims } A$

shows $\text{lookup } (\alpha \cdot A) \ is = \alpha * \text{lookup } A \ is$

using *assms* **proof** (*induction A arbitrary:is rule:subtensor-induct*)

case (*order-0 A is*)

then have $\text{length } (\text{vec } A) = 1$ **by** (*simp add: length-vec*)

then have $\text{hd } (\text{vec-smult } \alpha \ (\text{vec } A)) = \alpha * \text{hd } (\text{vec } A)$ **unfolding** *vec-smult-def*

by (*metis list.map-sel(1) list.size(3) zero-neq-one*)

moreover have $is = []$ **using** *order-0* **by** *auto*

ultimately show *?case* **unfolding** *smult-def* **by** (*auto simp add: <length (Tensor.vec A) = 1> lookup-def length-vec-smult order-0.hyps*)

next

case (*order-step A is*)

then obtain $i \ is'$ **where** $is = i \# \ is'$ **by** *blast*

then have $\text{lookup } (\alpha \cdot \text{subtensor } A \ i) \ is' = \alpha * \text{lookup } (\text{subtensor } A \ i) \ is'$

by (*metis (no-types, lifting) dims-subtensor list.sel(1) list.sel(3) order-step.IH order-step.hyps order-step.premis valid-index-dimsE*)

then show *?case* **using** *smult-subtensor <is = i # is'> dims-smult lookup-subtensor1*

list.sel(1) order-step.hyps order-step.premis valid-index-dimsE

by *metis*

qed

lemma *tensor-smult-assoc*:

fixes $A::'a::\text{ring tensor}$

shows $\alpha \cdot (\beta \cdot A) = (\alpha * \beta) \cdot A$

by (*rule tensor-lookup-eqI, simp, metis lookup-smult dims-smult mult.assoc*)

end

5 Tensor Product

theory *Tensor-Product*

imports *Tensor-Scalar-Mult Tensor-Subtensor*

begin

instantiation *tensor:: (ring) semigroup-mult*

begin

definition *tensor-prod-def*: $A * B = \text{tensor-from-vec } (\text{dims } A @ \text{dims } B) (\text{concat } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A)))$

abbreviation *tensor-prod-otimes* :: $'a \text{ tensor} \Rightarrow 'a \text{ tensor} \Rightarrow 'a \text{ tensor}$ (**infixl** \otimes 70)

where $A \otimes B \equiv A * B$

lemma *vec-tensor-prod[simp]*: $\text{vec } (A \otimes B) = \text{concat } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A))$ (**is** $?V$)

and *dims-tensor-prod[simp]*: $\text{dims } (A \otimes B) = \text{dims } A @ \text{dims } B$ (**is** $?D$)

proof –

have $\text{length } (\text{concat } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A))) = \text{prod-list } (\text{dims } A @ \text{dims } B)$

proof –

have $\bigwedge xs. xs \in \text{set } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A)) \implies \text{length } xs = \text{length } (\text{vec } B)$

using *length-vec-smult by force*

then show *?thesis using concat-equal-length by (metis length-map length-vec prod-list.append)*

qed

then show $?V ?D$ **by** (*simp add: tensor-prod-def*)+

qed

lemma *tensorprod-subtensor-base*:

shows $\text{concat } (\text{map } f (\text{concat } xss)) = \text{concat } (\text{map } (\lambda xs. \text{concat } (\text{map } f xs)) xss)$

by (*induction xss; auto*)

lemma *subtensor-combine-tensor-prod*:

assumes $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$

shows $\text{subtensor-combine } ds As \otimes B = \text{subtensor-combine } (ds @ \text{dims } B) (\text{map } (\lambda A. A \otimes B) As)$

proof –

let $?f = \lambda a. \text{vec-smult } a (\text{Tensor.vec } B)$

let $?xss = \text{map } \text{Tensor.vec } As$

have 1: $\text{prod-list } (\text{length } As \# ds) = \text{length } (\text{concat } ?xss)$ **by** (*metis assms length-vec subtensor-combine-dims subtensor-combine-vec*)

have 2: $\bigwedge A. A \in \text{set } As \implies \text{prod-list } (\text{dims } A @ \text{dims } B) = \text{length } (\text{concat } (\text{map } ?f (\text{Tensor.vec } A)))$

by (*metis dims-tensor-prod length-vec vec-tensor-prod*)

have 3: $\text{length } As \# ds @ \text{dims } B = (\text{length } (\text{map } (\lambda A. \text{tensor-from-vec } (\text{dims } A @ \text{dims } B) (\text{concat } (\text{map } ?f (\text{Tensor.vec } A))))))$

$A @ \text{dims } B$
 $(\text{concat } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A)))) \text{ As} \# \text{ ds } @ \text{ dims } B$ **by**
simp
have $4:(\text{concat } (\text{map } (\lambda xs. \text{concat } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) xs)) (\text{map } \text{vec } \text{As}))))$
 $= (\text{concat } (\text{map } \text{vec } (\text{map } (\lambda A. \text{tensor-from-vec } (\text{dims } A @ \text{dims } B) (\text{concat } (\text{map } (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A)))) \text{ As})))$
unfolding *map-map[unfolded comp-def]* **using** *vec-tensor* **by** (*metis* (*no-types*,
lifting) 2 *map-eq-conv*)
have *subtensor-combine* $\text{ds } \text{As} \otimes B = \text{tensor-from-vec } (\text{length } \text{As} \# \text{ ds } @ \text{ dims } B) (\text{concat } (\text{map } ?f (\text{concat } (?xss))))$
unfolding *subtensor-combine-def* *tensor-prod-def* **using** 1 **by** *auto*
also have $\dots = \text{tensor-from-vec } (\text{length } \text{As} \# \text{ ds } @ \text{ dims } B) (\text{concat } (\text{map } (\lambda xs. \text{concat } (\text{map } ?f xs)) ?xss))$
using *tensorprod-subtensor-base*[*of ?f ?xss*] **by** *auto*
also have $\dots = \text{subtensor-combine } (\text{ds } @ \text{ dims } B) (\text{map } (\lambda A. A \otimes B) \text{As})$
unfolding *subtensor-combine-def* *tensor-prod-def* **using** 3 4 **by** *metis*
finally show *?thesis* **by** *metis*
qed

lemma *subtensor-tensor-prod*:
assumes $\text{dims } A \neq []$ **and** $i < \text{hd } (\text{dims } A)$
shows *subtensor* $(A \otimes B) i = \text{subtensor } A i \otimes B$
using *assms* **proof** (*induction* A *rule:subtensor-combine-induct*)
case *order-0*
then show *?case* **by** *auto*
next
case (*order-step* $\text{As } \text{ds}$)
have $1:i < \text{length } (\text{map } (\lambda A. A \otimes B) \text{As})$ **using** *order-step* **by** (*simp* *add:order-step.hyps* *order-step.premis*(1))
have $2:(\bigwedge A. A \in \text{set } (\text{map } (\lambda A. A \otimes B) \text{As}) \implies \text{dims } A = \text{ds } @ \text{ dims } B)$
using *order-step* **by** *auto*
have *subtensor* $(\text{subtensor-combine } \text{ds } \text{As} \otimes B) i = \text{subtensor } (\text{subtensor-combine } (\text{ds } @ \text{ dims } B) (\text{map } (\lambda A. A \otimes B) \text{As})) i$
using *subtensor-combine-tensor-prod* *order-step* **by** *metis*
also have $\dots = \text{As } ! i \otimes B$
using *order-step* *subtensor-subtensor-combine*[*of* $(\text{map } (\lambda A. A \otimes B) \text{As}) \text{ ds } @ \text{ dims } B i]$ 1 2 **by** *auto*
also have $\dots = \text{subtensor } (\text{subtensor-combine } \text{ds } \text{As}) i \otimes B$
by (*metis* 1 *length-map* *order-step.hyps* *subtensor-subtensor-combine*)
finally show *?case* **by** *auto*
qed

lemma *lookup-tensor-prod*[*simp*]:
assumes *is1-valid*: $is1 \triangleleft \text{dims } A$ **and** *is2-valid*: $is2 \triangleleft \text{dims } B$
shows *lookup* $(A \otimes B) (is1 @ is2) = \text{lookup } A is1 * \text{lookup } B is2$
using *assms* **proof** (*induction* A *arbitrary:is1* *rule:subtensor-induct*)
case (*order-0* A *is1*)

```

then obtain  $a$  where  $\text{vec } A = [a]$ 
using Suc-length-conv Tensor.tensor-vec-from-lookup-Nil length-0-conv length-tensor-vec-from-lookup
length-vec by metis
then have  $A \otimes B = a \cdot B$  unfolding tensor-prod-def smult-def using order-0
by simp
moreover have  $\text{lookup } A [] = a$  by (simp add: ⟨Tensor.vec A = [a]⟩ lookup-def
order-0.hyps)
ultimately have  $\text{lookup } (A \otimes B) (is2) = a * \text{lookup } B is2$  by (simp add:
lookup-smult is2-valid)
then show  $?case$  using  $\langle \text{lookup } A [] = a \rangle$  null-rec(1) order-0.hyps order-0.prem(1)
by auto
next
case (order-step A is1)
then obtain  $i is1'$  where  $i \# is1' = is1$  by blast
have  $\text{lookup } (\text{subtensor } A i \otimes B) (is1' @ is2) = \text{lookup } (\text{subtensor } A i) is1' * \text{lookup } B is2$ 
using order-step
by (metis ⟨i # is1' = is1⟩ dims-subtensor list.sel(1) list.sel(3) valid-index-dimsE)
then show  $\text{lookup } (A \otimes B) (is1 @ is2) = \text{lookup } A is1 * \text{lookup } B is2$ 
using lookup-subtensor1[of i is1' A] lookup-subtensor1[of i is1' @ is2 A ⊗ B]
subtensor-tensor-prod[of A i B]
Cons-eq-appendI ⟨i # is1' = is1⟩ dims-tensor-prod is2-valid list.sel(1) order-step.hyps order-step.prem(1) valid-index-append valid-index-dimsE
by metis
qed

lemma valid-index-split:
assumes  $is \triangleleft ds1 @ ds2$ 
obtains  $is1 is2$  where  $is1 @ is2 = is$   $is1 \triangleleft ds1$   $is2 \triangleleft ds2$ 
proof
assume  $a: \bigwedge is1 is2. is1 @ is2 = is \implies is1 \triangleleft ds1 \implies is2 \triangleleft ds2 \implies thesis$ 
have  $\text{length-is:length } is = \text{length } ds1 + \text{length } ds2$  using valid-index-length
using assms by auto
show  $\text{take } (\text{length } ds1) is \triangleleft ds1$ 
apply (rule valid-indexI)
using valid-index-length using assms apply auto[1]
by (metis add-leD1 assms length-append not-less nth-append nth-take valid-index-lt)
show  $\text{drop } (\text{length } ds1) is \triangleleft ds2$ 
apply (rule valid-indexI)
using valid-index-length using assms apply auto[1]
using nth-drop[of length ds1 is] valid-index-lt[OF assms(1)] nth-append[of ds1 ds2] length-is
by (metis length-append nat-add-left-cancel-less nat-le-iff-add nth-append-length-plus)
show  $\text{take } (\text{length } ds1) is @ \text{drop } (\text{length } ds1) is = is$  using length-is by auto
qed

instance proof
fix  $A B C :: 'a :: \text{ring}$  tensor
show  $(A \otimes B) \otimes C = A \otimes (B \otimes C)$ 
proof (rule tensor-lookup-eqI, simp)

```

```

fix is assume  $is \triangleleft \text{dims } ((A \otimes B) \otimes C)$ 
obtain is1 is23 where  $is1 \triangleleft \text{dims } A$   $is23 \triangleleft \text{dims } (B \otimes C)$   $is1 @ is23 = is$ 
by (metis (mono-tags, lifting)  $\langle is \triangleleft \text{dims } ((A \otimes B) \otimes C) \rangle$  Tensor-Product.dims-tensor-prod
append-assoc valid-index-split)
obtain is2 is3 where  $is2 \triangleleft \text{dims } B$   $is3 \triangleleft \text{dims } C$   $is2 @ is3 = is23$ 
by (metis  $\langle is23 \triangleleft \text{dims } (\text{local.tensor-prod-otimes } B \ C) \rangle$  dims-tensor-prod
valid-index-split)
define is12 where  $is12 = is1 @ is2$ 
have  $is12 \triangleleft \text{dims } (A \otimes B)$  by (simp add:  $\langle is1 \triangleleft \text{dims } A \rangle \langle is2 \triangleleft \text{dims } B \rangle$ 
is12-def valid-index-append)
have  $is12 @ is3 = is$  by (simp add:  $\langle is1 @ is23 = is \rangle \langle is2 @ is3 = is23 \rangle$ 
is12-def)
show  $\text{lookup } ((A \otimes B) \otimes C) \ is = \text{lookup } (A \otimes (B \otimes C)) \ is$ 
unfolding lookup-tensor-prod[OF  $\langle is1 \triangleleft \text{dims } A \rangle \langle is23 \triangleleft \text{dims } (B \otimes C) \rangle$ ,
unfolded  $\langle is1 @ is23 = is \rangle$ ]
 $\text{lookup-tensor-prod}$ [OF  $\langle is12 \triangleleft \text{dims } (A \otimes B) \rangle \langle is3 \triangleleft \text{dims } C \rangle$ , unfolded
 $\langle is12 @ is3 = is \rangle$ ]
using  $\langle is1 \triangleleft \text{dims } A \rangle \langle is2 @ is3 = is23 \rangle \langle is2 \triangleleft \text{dims } B \rangle \langle is3 \triangleleft \text{dims } C \rangle$ 
is12-def mult.assoc by fastforce
qed
qed

```

end

lemma *tensor-prod-distr-left:*

assumes $\text{dims } A = \text{dims } B$

shows $(A + B) \otimes C = (A \otimes C) + (B \otimes C)$

proof –

have $\bigwedge is. is \triangleleft \text{dims } A @ \text{dims } C \implies \text{lookup } ((A + B) \otimes C) \ is = \text{lookup } (A \otimes C + B \otimes C) \ is$

proof –

fix *is* **assume** $is \triangleleft \text{dims } A @ \text{dims } C$

obtain *is1 is2* **where** $is = is1 @ is2$ $is1 \triangleleft \text{dims } A$ $is2 \triangleleft \text{dims } C$ **using**
valid-index-split **using** $\langle is \triangleleft \text{dims } A @ \text{dims } C \rangle$ **by** *blast*

then show $\text{lookup } ((A + B) \otimes C) \ is = \text{lookup } ((A \otimes C) + (B \otimes C)) \ is$

using *lookup-plus*

$\langle is1 \triangleleft \text{dims } A \rangle \langle is2 \triangleleft \text{dims } C \rangle$ *assms plus-dim1 dims-tensor-prod lookup-tensor-prod*
ring-class.ring-distrib(2) *valid-index-append*

by *fastforce*

qed

moreover have *tensor-from-lookup* ($\text{dims } A @ \text{dims } C$) ($\text{lookup } ((A + B) \otimes C)$)
 $= (A + B) \otimes C$

tensor-from-lookup ($\text{dims } A @ \text{dims } C$) ($\text{lookup } ((A \otimes C) + (B \otimes C))$) $= (A \otimes C) + (B \otimes C)$

by (*metis* (*no-types*, *lifting*) *assms plus-dim1 dims-tensor-prod tensor-lookup*)**+**

ultimately show *?thesis* **using** *tensor-from-lookup-eqI*

by (*metis* $\langle \bigwedge is. is \triangleleft \text{dims } A @ \text{dims } C \implies \text{lookup } ((A + B) \otimes C) \ is = \text{lookup } (A \otimes C + B \otimes C) \ is \rangle$)

qed

lemma *tensor-prod-distr-right*:
assumes $\text{dims } A = \text{dims } B$
shows $C \otimes (A + B) = (C \otimes A) + (C \otimes B)$
proof –
have $\bigwedge is. is \triangleleft \text{dims } C @ \text{dims } A \implies \text{lookup } (C \otimes (A + B)) \text{ is} = \text{lookup } (C \otimes A + C \otimes B) \text{ is}$
proof –
fix is **assume** $is \triangleleft \text{dims } C @ \text{dims } A$
obtain $is1 \ is2$ **where** $is = is1 @ is2$ $is1 \triangleleft \text{dims } C$ $is2 \triangleleft \text{dims } A$ **using** *valid-index-split* **using** $\langle is \triangleleft \text{dims } C @ \text{dims } A \rangle$ **by** *blast*
then show $\text{lookup } (C \otimes (A + B)) \text{ is} = \text{lookup } ((C \otimes A) + (C \otimes B)) \text{ is}$
using *lookup-plus*
using $\langle is2 \triangleleft \text{dims } A \rangle \langle is1 \triangleleft \text{dims } C \rangle$ *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distrib(1) valid-index-append*
by *fastforce*
qed
moreover have $\text{tensor-from-lookup } (\text{dims } C @ \text{dims } A) (\text{lookup } (C \otimes (A + B))) = C \otimes (A + B)$
 $\text{tensor-from-lookup } (\text{dims } C @ \text{dims } A) (\text{lookup } ((C \otimes A) + (C \otimes B))) = (C \otimes A) + (C \otimes B)$
by (*metis (no-types, lifting) assms plus-dim1 dims-tensor-prod tensor-lookup*)
ultimately show *?thesis* **using** *tensor-from-lookup-eqI*
by (*metis* $\langle \bigwedge is. is \triangleleft \text{dims } C @ \text{dims } A \implies \text{lookup } (C \otimes (A + B)) \text{ is} = \text{lookup } (C \otimes A + C \otimes B) \text{ is} \rangle$)
qed

instantiation *tensor* :: (*ring-1*) *monoid-mult*

begin

definition *tensor-one-def:1* = *tensor-from-vec* [] [1]

lemma *tensor-one-from-lookup: 1* = *tensor-from-lookup* [] ($\lambda-. 1$)

unfolding *tensor-one-def* **by** (*rule tensor-eqI; simp-all add: tensor-from-lookup-def*)
)

instance proof

fix $A::'a::\text{ring-1}$ *tensor*

show $A * 1 = A$ **unfolding** *tensor-one-from-lookup*

by (*rule tensor-lookup-eqI; metis lookup-tensor-prod[of - A [] tensor-from-lookup*
[] ($\lambda-. 1$))

lookup-tensor-from-lookup valid-index.Nil append-Nil2 dims-tensor dims-tensor-prod length-tensor-vec-from-lookup mult.right-neutral tensor-from-lookup-def)

next

fix $A::'a::\text{ring-1}$ *tensor*

show $1 * A = A$ **unfolding** *tensor-one-from-lookup*

by (*rule tensor-lookup-eqI; metis lookup-tensor-prod[of [] tensor-from-lookup*
[] ($\lambda-. 1$) - A)

lookup-tensor-from-lookup valid-index.Nil List.append.append-Nil dims-tensor dims-tensor-prod


```

length-tensor-vec-from-lookup mult.left-neutral tensor-from-lookup-def)
qed
end

lemma order-tensor-one: order 1 = 0 unfolding tensor-one-def by simp

lemma smult-prod-extract1:
fixes a::'a::comm-ring-1
shows a · (A ⊗ B) = (a · A) ⊗ B
proof (rule tensor-lookup-eqI)
  show dims (a · (A ⊗ B)) = dims ((a · A) ⊗ B) by simp
  fix is assume is < dims (a · (A ⊗ B))
  then have is < dims (A ⊗ B) by auto
  then obtain is1 is2 where is1 < dims A is2 < dims B is = is1 @ is2 by
(metis dims-tensor-prod valid-index-split)
  then have is1 < dims (a · A) by auto
  show lookup (a · (A ⊗ B)) is = lookup (a · A ⊗ B) is
  using lookup-tensor-prod[OF <is1 < dims A> <is2 < dims B>] lookup-tensor-prod[OF
<is1 < dims (a · A)> <is2 < dims B>]
  lookup-smult[OF <is < dims (A ⊗ B)>] lookup-smult[OF <is1 < dims A>] <is
= is1 @ is2> by simp
qed

lemma smult-prod-extract2:
fixes a::'a::comm-ring-1
shows a · (A ⊗ B) = A ⊗ (a · B)
proof (rule tensor-lookup-eqI)
  show dims (a · (A ⊗ B)) = dims (A ⊗ (a · B)) by simp
  fix is assume is < dims (a · (A ⊗ B))
  then have is < dims (A ⊗ B) by auto
  then obtain is1 is2 where is1 < dims A is2 < dims B is = is1 @ is2 by
(metis dims-tensor-prod valid-index-split)
  then have is2 < dims (a · B) by auto
  show lookup (a · (A ⊗ B)) is = lookup (A ⊗ (a · B)) is
  using lookup-tensor-prod[OF <is1 < dims A> <is2 < dims B>] lookup-tensor-prod[OF
<is1 < dims A> <is2 < dims (a · B)>]
  lookup-smult[OF <is < dims (A ⊗ B)>] lookup-smult[OF <is2 < dims B>]
<is = is1 @ is2> by simp
qed

lemma order-0-multiple-of-one:
assumes order A = 0
obtains a where A = a · 1
proof
  assume (∧ a. A = a · 1 ⇒ thesis)
  have length (vec A) = 1 using assms by (simp add:length-vec)
  then obtain a where vec A = [a] by (metis One-nat-def Suc-length-conv
length-0-conv)

```

moreover have $\text{vec } (a \cdot 1) = [a]$ **unfolding** *smult-def tensor-one-def* **by** (*simp add: vec-smult-def*)
ultimately have $A = a \cdot 1$ **using** *tensor-eqI* **by** (*metis assms dims-smult length-0-conv order-tensor-one*)
then show $A = \text{hd } (\text{vec } A) \cdot 1$ **using** $\langle \text{vec } A = [a] \rangle$ **by** *auto*
qed

lemma *smult-1*:
fixes $A::'a::\text{ring-1 tensor}$
shows $A = 1 \cdot A$ **unfolding** *smult-def tensor-one-def*
apply (*rule tensor-eqI*)
apply (*simp add: length-vec length-vec-smult*)
by (*metis dims-tensor length-vec length-vec-smult lookup-smult mult.left-neutral smult-def tensor-lookup-eqI*)

lemma *tensor0-prod-right[simp]*: $A \otimes \text{tensor0 } ds = \text{tensor0 } (\text{dims } A @ ds)$
proof (*rule tensor-lookup-eqI, simp*)
fix is **assume** $is \triangleleft \text{dims } (A \otimes \text{tensor0 } ds)$
then obtain $is1 is2$ **where** $is1 \triangleleft \text{dims } A$ $is2 \triangleleft \text{dims } (\text{tensor0 } ds)$ $is = is1 @ is2$
by (*metis dims-tensor0 dims-tensor-prod valid-index-split*)
then show $\text{lookup } (A \otimes \text{tensor0 } ds) is = \text{lookup } (\text{tensor0 } (\text{dims } A @ ds)) is$
by (*metis (no-types, lifting) $\langle is \triangleleft \text{dims } (A \otimes \text{tensor0 } ds) \rangle$ dims-tensor0 dims-tensor-prod lookup-tensor0 lookup-tensor-prod mult-zero-right*)
qed

lemma *tensor0-prod-left[simp]*: $\text{tensor0 } ds \otimes A = \text{tensor0 } (ds @ \text{dims } A)$
proof (*rule tensor-lookup-eqI, simp*)
fix is **assume** $is \triangleleft \text{dims } (\text{tensor0 } ds \otimes A)$
then obtain $is1 is2$ **where** $is1 \triangleleft \text{dims } (\text{tensor0 } ds)$ $is2 \triangleleft \text{dims } A$ $is = is1 @ is2$
by (*metis dims-tensor0 dims-tensor-prod valid-index-split*)
then show $\text{lookup } (\text{tensor0 } ds \otimes A) is = \text{lookup } (\text{tensor0 } (ds @ \text{dims } A)) is$
by (*metis (no-types, lifting) $\langle is \triangleleft \text{dims } (\text{tensor0 } ds \otimes A) \rangle$ dims-tensor0 dims-tensor-prod lookup-tensor0 lookup-tensor-prod mult-zero-left*)
qed

lemma *subtensor-prod-with-vec*:
assumes $\text{order } A = 1$ $i < \text{hd } (\text{dims } A)$
shows $\text{subtensor } (A \otimes B) i = \text{lookup } A [i] \cdot B$
proof (*rule tensor-lookup-eqI*)
have $\text{dims } (A \otimes B) \neq []$ **using** *assms(1)* **by** *auto*
have $\text{hd } (\text{dims } A) = \text{hd } (\text{dims } (A \otimes B))$
by (*metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod list.sel(1)*)
show $\text{dims } (\text{subtensor } (A \otimes B) i) = \text{dims } (\text{lookup } A [i] \cdot B)$
unfolding *dims-smult dims-subtensor[OF $\langle \text{dims } (A \otimes B) \neq [] \rangle$, $\langle i < \text{hd } (\text{dims } A) \rangle$]* $[\text{unfolded } \langle \text{hd } (\text{dims } A) = \text{hd } (\text{dims } (A \otimes B)) \rangle]$

```

  by (metis One-nat-def Suc-length-conv append.simps(2) append-self-conv2 assms(1)
      dims-tensor-prod length-0-conv list.sel(3))
next
  fix is assume is < dims (subtensor (A ⊗ B) i)
  have dims (A ⊗ B) ≠ [] using assms(1) by auto
  have hd (dims A) = hd (dims (A ⊗ B))
  by (metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod
      list.sel(1))
  then have is < dims B
  using < is < dims (subtensor (A ⊗ B) i) [unfolded dims-subtensor[OF <dims
      (A ⊗ B) ≠ []> <i < hd (dims A)> [unfolded <hd (dims A) = hd (dims (A ⊗ B))>]] ]
  by (metis One-nat-def Suc-length-conv append-self-conv2 assms(1) dims-tensor-prod
      length-0-conv list.sel(3) list.simps(3) tl-append2)
  have [i] < dims A using assms by (metis One-nat-def Suc-length-conv length-0-conv
      list.sel(1) valid-index.Nil valid-index.simps)
  then have i # is < dims (A ⊗ B) using < is < dims (subtensor (A ⊗ B) i)
      dims-subtensor valid-index.Cons by auto
  then show lookup (subtensor (A ⊗ B) i) is = lookup (lookup A [i] · B) is
  unfolding lookup-subtensor1[OF <i # is < dims (A ⊗ B)>]
  using lookup-tensor-prod[OF <[i] < dims A> <is < dims B>] lookup-smult
      <is < dims B> using append-Cons by fastforce
qed

end

```

6 Unit Vectors as Tensors

```

theory Tensor-Unit-Vec
imports Tensor-Product
begin

```

```

definition unit-vec::nat ⇒ nat ⇒ 'a::ring-1 tensor
where unit-vec n i = tensor-from-lookup [n] (λx. if x=[i] then 1 else 0)

```

```

lemma dims-unit-vec: dims (unit-vec n i) = [n] unfolding unit-vec-def by (simp
add: tensor-from-lookup-def)

```

```

lemma lookup-unit-vec:
assumes j < n
shows lookup (unit-vec n i) [j] = (if i=j then 1 else 0)
proof -
  have [j] < [n] by (simp add: assms valid-index.Cons valid-index.Nil)
  then have lookup (unit-vec n i) [j] = (λx. if x=[i] then 1 else 0) [j]
  by (simp add: lookup-tensor-from-lookup unit-vec-def)
  then show ?thesis by auto
qed

```

```

lemma subtensor-prod-with-unit-vec:
fixes A::'a::ring-1 tensor

```

assumes $j < n$
shows $\text{subtensor } (\text{unit-vec } n \ i \otimes A) \ j = (\text{if } i=j \ \text{then } A \ \text{else } (\text{tensor0 } (\text{dims } A)))$
proof –
have $0:\text{lookup } (\text{unit-vec } n \ i) \ [j] = (\text{if } i=j \ \text{then } 1 \ \text{else } 0)$ **unfolding** unit-vec-def
by ($\text{simp add: assms lookup-tensor-from-lookup valid-index.Cons valid-index.Nil}$)
have $1:\text{order } (\text{unit-vec } n \ i) = 1$ **unfolding** unit-vec-def **by** ($\text{simp add: tensor-from-lookup-def}$)
from assms **have** $2:j < \text{hd } (\text{dims } (\text{tensor-from-lookup } [n] \ (\lambda x. \ \text{if } x = [i] \ \text{then } 1 \ \text{else } 0)))$
by ($\text{simp add: dims-tensor-from-lookup}$)
show $?thesis$ **using** $\text{unit-vec-def subtensor-prod-with-vec 1 2 0 smult-1 tensor-smult0}$
by ($\text{metis (no-types, lifting) tensor-from-lookup-eqI}$)
qed

lemma *subtensor-decomposition*:

assumes $\text{dims } A \neq []$
shows $\text{listsum } (\text{dims } A) \ (\text{map } (\lambda i. \ \text{unit-vec } (\text{hd } (\text{dims } A)) \ i \otimes \text{subtensor } A \ i) \ [0..\text{hd } (\text{dims } A)]) = A$ (**is** $?LS = A$)
proof –
let $?f = \lambda i. \ \text{unit-vec } (\text{hd } (\text{dims } A)) \ i \otimes \text{subtensor } A \ i$
have $\text{correct-dims}:\bigwedge B. B \in \text{set } (\text{map } ?f \ [0..\text{hd } (\text{dims } A)]) \implies \text{dims } B = \text{dims } A$
proof –
fix B
assume $B \in \text{set } (\text{map } ?f \ [0..\text{hd } (\text{dims } A)])$
then obtain i **where** $B:B = ?f \ i$ **and** $i < \text{hd } (\text{dims } A)$ **by** *auto*
then have $\text{dims } (\text{subtensor } A \ i) = \text{tl } (\text{dims } A)$ **using** dims-subtensor **using** assms **by** *blast*
then show $\text{dims } B = \text{dims } A$ **unfolding** B
by ($\text{metis append-Cons assms dims-tensor-prod dims-unit-vec list.exhaust-sel self-append-conv2}$)
qed
have $\bigwedge j. j < \text{hd } (\text{dims } A) \implies \text{subtensor } ?LS \ j = \text{subtensor } A \ j$
proof –
fix j
assume $j < \text{hd } (\text{dims } A)$
have $1:\text{subtensor } ?LS \ j = \text{listsum } (\text{tl } (\text{dims } A)) \ (\text{map } (\lambda A. \ \text{subtensor } A \ j) \ (\text{map } ?f \ [0..\text{hd } (\text{dims } A)]))$
using $\text{subtensor-listsum[of } (\text{map } (\lambda i. \ ?f \ i) \ [0..\text{hd } (\text{dims } A)]) \ \text{dims } A \ j, \text{OF correct-dims assms } \langle j < \text{hd } (\text{dims } A) \rangle]$
by *linarith*
also have $\dots = \text{listsum } (\text{tl } (\text{dims } A)) \ (\text{map } (\lambda i. \ \text{subtensor } (?f \ i) \ j) \ [0..\text{hd } (\text{dims } A)])$
proof –
have $\text{map } (\lambda A. \ \text{subtensor } A \ j) \ (\text{map } ?f \ [0..\text{hd } (\text{dims } A)]) = \text{map } (\lambda i. \ \text{subtensor } (?f \ i) \ j) \ [0..\text{hd } (\text{dims } A)]$
unfolding $\text{map-map[of } (\lambda A. \ \text{subtensor } A \ j) \ ?f \ [0..\text{hd } (\text{dims } A)]]$ **by** *simp*
with 1 show $?thesis$ **by** *metis*
qed

```

    also have ... = map (λi. if i = j then subtensor A i else tensor0 (dims
(subtensor A i))) [0..<hd (dims A)] ! j
    unfolding subtensor-prod-with-unit-vec[OF ⟨j < hd (dims A)⟩]
    using listsum-all-0-but-one[of j (map (λi. if i = j then subtensor A i else
tensor0 (dims (subtensor A i))) [0..<hd (dims A)])] tl (dims A)
    by (simp add: ⟨j < hd (dims A)⟩ assms)
    also have ... = subtensor A j by (simp add: ⟨j < hd (dims A)⟩)
    finally show subtensor ?LS j = subtensor A j by auto
qed
moreover have dims ?LS = dims A using correct-dims listsum-dims by blast
ultimately show ?thesis using subtensor-eqI by (metis (no-types, lifting) assms)
qed

end

```

7 Tensor CP-Rank

```

theory Tensor-Rank
imports Tensor-Unit-Vec
begin

```

```

inductive cprank-max1::'a::ring-1 tensor ⇒ bool where
order1: order A ≤ 1 ⇒ cprank-max1 A |
higher-order: order A = 1 ⇒ cprank-max1 B ⇒ cprank-max1 (A ⊗ B)

```

```

lemma cprank-max1-order0: cprank-max1 B ⇒ order A = 0 ⇒ cprank-max1
(A ⊗ B)

```

```

proof (induction B rule:cprank-max1.induct)
  case order1
  then show ?case by (simp add: cprank-max1.order1)
next
  case (higher-order A' B)
  then have order (A ⊗ A') = 1 by simp
  then show ?case using higher-order cprank-max1.higher-order by (metis mult.assoc)
qed

```

```

lemma cprank-max1-order-le1: order A ≤ 0 ⇒ cprank-max1 B ⇒ cprank-max1
(A ⊗ B)
by (simp add: cprank-max1-order0)

```

```

lemma cprank-max1-prod: cprank-max1 A ⇒ cprank-max1 B ⇒ cprank-max1
(A ⊗ B)
  apply (induction A rule: cprank-max1.induct)
  apply (meson higher-order le-neq-trans less-one cprank-max1-order0)
  by (simp add: higher-order mult.assoc)

```

```

lemma cprank-max1-prod-list:
assumes ⋀B. B ∈ set Bs ⇒ cprank-max1 B
shows cprank-max1 (prod-list Bs)

```

using *assms* **by** (*induction* *Bs*, *metis* *dims-smult* *dims-tensor0* *list.size(3)* *prod-list.Nil* *order1* *order-0-multiple-of-one* *zero-le-one*, *simp* *add: cprank-max1-prod*)

lemma *cprank-max1-prod-listE*:

fixes *A::'a::comm-ring-1* *tensor*

assumes *cprank-max1 A*

obtains *Bs a* **where** $\bigwedge B. B \in \text{set } Bs \implies \text{order } B = 1 \ a \cdot \text{prod-list } Bs = A$

using *assms* **proof** (*induction* *A* *arbitrary:thesis* *rule:cprank-max1.induct*)

case (*order1 A*)

then show *?case*

proof (*cases* *order A = 0*)

case *True*

then obtain *a* **where** $A = a \cdot \text{prod-list } []$ **using** *order-0-multiple-of-one* **using** *prod-list.Nil* **by** *auto*

then show *?thesis* **using** *length-pos-if-in-set* *order1.premis* **by** *fastforce*

next

case *False*

then have $\text{order } A = 1$ **using** *order1* **by** *linarith*

then have $A = 1 \cdot \text{prod-list } [A]$ **by** (*simp* *add: smult-1*)

then show *?thesis* **by** (*metis* $\langle \text{order } A = 1 \rangle$ *length-greater-0-conv* *length-pos-if-in-set* *order1.premis* *set-ConsD*)

qed

next

case (*higher-order A B*)

then obtain *Bs b* **where** $\bigwedge B'. B' \in \text{set } Bs \implies \text{order } B' = 1$ $b \cdot \text{prod-list } Bs = B$ **by** *metis*

then have $\bigwedge B. B \in \text{set } (A \# Bs) \implies \text{order } B = 1$ **using** *higher-order* **by** *auto*

have $A \otimes B = b \cdot (A \otimes \text{prod-list } Bs)$ **using** *smult-prod-extract2* $\langle b \cdot \text{prod-list } Bs = B \rangle$ **by** *metis*

then show *?case* **by** (*metis* $\langle \bigwedge Ba. Ba \in \text{set } (A \# Bs) \implies \text{order } Ba = 1 \rangle$ *higher-order.premis* *prod-list.Cons*)

qed

inductive *cprank-max* **::** *nat* \Rightarrow *'a::ring-1* *tensor* \Rightarrow *bool* **where**

cprank-max0: *cprank-max* *0* (*tensor0* *ds*) |

cprank-max-Suc: $\text{dims } A = \text{dims } B \implies \text{cprank-max1 } A \implies \text{cprank-max } j \ B \implies$

cprank-max (*Suc* *j*) (*A+B*)

lemma *cprank-max1*: *cprank-max1 A* \implies *cprank-max 1 A*

by (*metis* *One-nat-def* *dims-tensor0* *cprank-max.simps* *cprank-max0* *tensor-add-0-right*)

lemma *cprank-max-plus*: *cprank-max i A* \implies *cprank-max j B* \implies $\text{dims } A = \text{dims } B \implies \text{cprank-max } (i+j) \ (A+B)$

apply (*induction* *A* *rule:cprank-max.induct*)

apply *auto[1]*

by (*metis* *add-Suc* *plus-assoc* *plus-dim1* *cprank-max.intros(2)*)

lemma *cprank-max-listsum*:

assumes $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$
and $\bigwedge A. A \in \text{set } As \implies \text{cprank-max } n \ A$
shows $\text{cprank-max } (n * \text{length } As) (\text{listsum } ds \ As)$
using *assms* **proof** (*induction* *As*)
 case *Nil*
 then show *?case* **using** *listsum-Nil* *cprank-max.simps* **by** *fastforce*
next
 case (*Cons* *A* *As*)
 then show *?case* **using** *cprank-max-plus*[*of* *n* *A* *n * length* *As* *listsum* *ds* *As*]
 by (*simp* *add: length-Cons* *list.set-intros(1)* *listsum-Cons* *listsum-dims* *set-subset-Cons* *subsetCE*)
qed

lemma *cprank-maxE*:
assumes $\text{cprank-max } n \ A$
obtains *BS* **where** $(\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B)$ **and** $(\bigwedge B. B \in \text{set } BS \implies \text{dims } A = \text{dims } B)$ **and** $\text{listsum } (\text{dims } A) \ BS = A$ **and** $\text{length } BS = n$
using *assms* **proof** (*induction* *arbitrary:thesis* *rule:cprank-max.induct*)
 case (*cprank-max0* *ds*)
 have *Tensor-Plus.listsum* (*dims* (*tensor0* *ds*)) [] = *tensor0* *ds* **by** (*simp* *add: listsum-Nil*)
 then show *?case* **using** *cprank-max0.prem*s **by** *fastforce*
next
 case (*cprank-max-Suc* *A* *B* *j*)
 then obtain *BS* **where** *BS-def*: $(\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B)$ $(\bigwedge B'. B' \in \text{set } BS \implies \text{dims } B' = \text{dims } B)$
 $\text{listsum } (\text{dims } B) \ BS = B$ $\text{length } BS = j$ **by** *metis*
 then have $\text{listsum } (\text{dims } (A + B)) (A \# BS) = A + B$
 by (*simp* *add: listsum-Cons* *cprank-max-Suc.hyps(1)*)
 then show *?case* **using** *BS-def* *length-Cons* *cprank-max-Suc.hyps(2)* *cprank-max-Suc.prem*s *set-ConsD*
 by (*metis* *plus-dim1* *cprank-max-Suc.hyps(1)*)
qed

lemma *cprank-maxI*:
assumes $\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B$
and $\bigwedge B. B \in \text{set } BS \implies \text{dims } B = ds$
shows $\text{cprank-max } (\text{length } BS) (\text{listsum } ds \ BS)$
using *assms* **proof** (*induction* *BS*)
 case *Nil*
 then show *?case* **by** (*simp* *add: listsum-Nil* *cprank-max0*)
next
 case (*Cons* *B* *BS*)
 then show *?case*
 by (*simp* *add: length-Cons* *list.set-intros(1)* *list.set-intros(2)* *listsum-Cons* *listsum-dims* *cprank-max-Suc*)
qed

lemma *cprank-max-0E*: $\text{cprank-max } 0 \ A \implies A = \text{tensor0 } (\text{dims } A)$ **by** (*metis*

dims-tensor0 length-0-conv cprank-max0 cprank-maxE)

lemma *listsum-prod-distr-right*:

assumes $(\bigwedge C. C \in \text{set } CS \implies \text{dims } C = ds)$

shows $A \otimes \text{listsum } ds \text{ } CS = \text{listsum } (\text{dims } A @ ds) (\text{map } (\lambda C. A \otimes C) \text{ } CS)$

using *assms* **proof** (*induction* *CS*)

case *Nil*

then show *?case* **by** (*simp add:listsum-Nil*)

next

case (*Cons* *C* *CS*)

then have $\text{dims } C = \text{dims } (\text{listsum } ds \text{ } CS)$ **by** (*simp add: list.set-intros(1) list.set-intros(2) listsum-dims*)

then show *?case* **unfolding** *listsum-Cons list.map(2)*

using *tensor-prod-distr-right Cons.IH Cons.premis list.set-intros(2)* **by** *fastforce*
qed

lemma *cprank-max-prod-order1*:

assumes $\text{order } A = 1$

and *cprank-max* *n* *B*

shows *cprank-max* *n* ($A \otimes B$)

proof –

obtain *CS* **where** $(\bigwedge C. C \in \text{set } CS \implies \text{cprank-max1 } C)$

and $(\bigwedge C. C \in \text{set } CS \implies \text{dims } C = \text{dims } B)$

and $\text{listsum } (\text{dims } B) \text{ } CS = B$

and $\text{length } CS = n$

using *assms(2) cprank-maxE* **by** *metis*

define *CS'* **where** $CS' = \text{map } (\lambda C. A \otimes C) \text{ } CS$

then have $\bigwedge C'. C' \in \text{set } CS' \implies \text{cprank-max1 } C'$

using *assms(1) higher-order* $\langle \bigwedge C. C \in \text{set } CS \implies \text{cprank-max1 } C \rangle$ *imageE set-map* **by** *auto*

have $\text{listsum } (\text{dims } A @ \text{dims } B) \text{ } CS' = A \otimes B$ **using** *CS'-def* $\langle \text{Tensor-Plus.listsum } (\text{dims } B) \text{ } CS = B \rangle$

using $\langle \bigwedge Ca. Ca \in \text{set } CS \implies \text{dims } Ca = \text{dims } B \rangle$ *listsum-prod-distr-right* **by** *fastforce*

then show *?thesis* **by** (*metis (mono-tags, lifting) CS'-def* $\langle \bigwedge C'. C' \in \text{set } CS' \implies \text{cprank-max1 } C' \rangle$ $\langle \bigwedge Ca. Ca \in \text{set } CS \implies \text{dims } Ca = \text{dims } B \rangle$ $\langle \text{length } CS = n \rangle$ *dims-tensor-prod imageE length-map cprank-maxI set-map*)

qed

lemma *cprank-max-upper-bound*:

shows *cprank-max* (*prod-list* (*dims* *A*)) *A*

proof (*induction* *A* *rule:subtensor-induct*)

case (*order-0* *A*)

then have *cprank-max* *1* *A* **using** *order1 cprank-max1* **by** *force*

then show *?case* **using** *order-0* **by** *auto*

next

case (*order-step* *A*)

define *Bs* **where** $Bs = \text{map } (\lambda i. \text{unit-vec } (\text{hd } (\text{dims } A)) \ i \otimes \text{subtensor } A \ i)$
 $[0..<\text{hd } (\text{dims } A)]$


```

have  $\bigwedge B. B \in \text{set } Bs \implies \text{dims } A = \text{dims } B$ 
proof -
  fix B assume B  $\in$  set Bs
  obtain i where  $i < \text{hd } (\text{dims } A)$   $Bs ! i = B$  using Bs-def  $\langle B \in \text{set } Bs \rangle$  by auto
  then have  $\text{dims } (\text{unit-vec } (\text{hd } (\text{dims } A)) i \otimes \text{subtensor } A i) = \text{dims } A$ 
    using dims-unit-vec order-step.hyps
  by (metis append-Cons dims-subtensor dims-tensor-prod list.exhaust-sel self-append-conv2)
  then show  $\text{dims } A = \text{dims } B$  using Bs-def  $\langle Bs ! i = B \rangle$   $\langle i < \text{hd } (\text{dims } A) \rangle$ 
by auto
qed
have  $\bigwedge B. B \in \text{set } Bs \implies \text{cprank-max } (\text{prod-list } (\text{tl } (\text{dims } A))) B$ 
proof -
  fix B assume B  $\in$  set Bs
  obtain i where  $i < \text{hd } (\text{dims } A)$   $Bs ! i = B$  using Bs-def  $\langle B \in \text{set } Bs \rangle$  by auto
  then have  $\text{cprank-max } (\text{prod-list } (\text{tl } (\text{dims } A))) (\text{unit-vec } (\text{hd } (\text{dims } A)) i \otimes$ 
subtensor A i)
    by (metis One-nat-def dims-subtensor dims-unit-vec length-Cons list.size(3)
order-step.IH order-step.hyps cprank-max-prod-order1)
  then show  $\text{cprank-max } (\text{prod-list } (\text{tl } (\text{dims } A))) B$  using Bs-def  $\langle Bs ! i = B \rangle$ 
 $\langle i < \text{hd } (\text{dims } A) \rangle$  by auto
qed
then show ?case using subtensor-decomposition[OF order-step.hyps] cprank-max-listsum
  by (metis (no-types, lifting) Bs-def  $\langle \bigwedge Ba. Ba \in \text{set } Bs \implies \text{dims } A = \text{dims } Ba \rangle$ 
diff-zero length-map length-upt list.exhaust-sel prod-list.Cons mult.commute or-
der-step.hyps)
qed

definition cprank :: 'a::ring-1 tensor  $\Rightarrow$  nat where
cprank A = (LEAST n. cprank-max n A)

lemma cprank-upper-bound: cprank A  $\leq$  prod-list (dims A)
unfolding cprank-def using cprank-max-upper-bound Least-le by fastforce

lemma cprank-max-cprank: cprank-max (cprank A) A
unfolding cprank-def using cprank-max-upper-bound by (metis LeastI)

end

```

8 Tensor Matricization

```

theory Tensor-Matricization
imports Tensor-Plus
Jordan-Normal-Form.Matrix Jordan-Normal-Form.DL-Missing-Sublist
begin

fun digit-decode :: nat list  $\Rightarrow$  nat list  $\Rightarrow$  nat where
digit-decode [] [] = 0 |
digit-decode (d # ds) (i # is) = i + d * digit-decode ds is

```

```

fun digit-encode :: nat list  $\Rightarrow$  nat  $\Rightarrow$  nat list where
  digit-encode [] a = [] |
  digit-encode (d # ds) a = a mod d # digit-encode ds (a div d)

lemma digit-encode-decode[simp]:
assumes is  $\triangleleft$  ds
shows digit-encode ds (digit-decode ds is) = is
  using assms apply (induction rule:valid-index.induct)
  unfolding digit-decode.simps digit-encode.simps
  by simp-all

lemma digit-decode-encode[simp]:
shows digit-decode ds (digit-encode ds a) = a mod (prod-list ds)
by (induction ds arbitrary:a; simp add: Divides.mod-mult2-eq add.commute)

lemma digit-decode-encode-lt[simp]:
assumes a < prod-list ds
shows digit-decode ds (digit-encode ds a) = a
by (simp add: assms)

lemma digit-decode-lt:
assumes is  $\triangleleft$  ds
shows digit-decode ds is < prod-list ds
using assms proof (induction rule:valid-index.induct)
  case Nil
  then show ?case by simp
next
  case (Cons is ds i d)
  have (i + d * digit-decode ds is) div (d * prod-list ds) = 0
    using Cons.IH Cons.hyps(2) div-mult2-eq by force
  then show ?case unfolding digit-decode.simps prod-list.Cons
    by (metis (no-types) Cons.IH Cons.hyps(2) div-eq-0-iff mult-eq-0-iff not-less0)
qed

lemma digit-encode-valid-index:
assumes a < prod-list ds
shows digit-encode ds a  $\triangleleft$  ds
using assms proof (induction ds arbitrary:a)
  case Nil
  show ?case by (simp add: valid-index.Nil)
next
  case (Cons d ds a)
  then have a < d * prod-list ds
    by simp
  then have a div d < prod-list ds
    by (metis div-eq-0-iff div-mult2-eq mult-0-right not-less0)
  then have digit-encode ds (a div d)  $\triangleleft$  ds
    by (rule Cons)
  moreover have d > 0

```

using $\langle a < d * \text{prod-list } ds \rangle$ **by** (cases $d = 0$) *simp-all*
then have $a \bmod d < d$
by *simp*
ultimately show *?case*
by (*simp add: valid-index.Cons*)
qed

lemma *length-digit-encode*:
shows $\text{length } (\text{digit-encode } ds \ a) = \text{length } ds$
by (*induction ds arbitrary:a; simp-all*)

lemma *digit-encode-0*:
 $\text{prod-list } ds \ dvd \ a \implies \text{digit-encode } ds \ a = \text{replicate } (\text{length } ds) \ 0$
proof (*induction ds arbitrary:a*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons d ds a*)
then have $\text{prod-list } ds \ dvd \ (a \ \text{div } d)$ **unfolding** *prod-list.Cons*
by (*metis dvd-0-right dvd-div-iff-mult dvd-mult-left mult.commute split-div*)
then show *?case* **unfolding** *digit-encode.simps length-Cons replicate-Suc prod-list.Cons*
using *Cons*
using *dvd-imp-mod-0 dvd-mult-left prod-list.Cons* **by** *force*
qed

lemma *valid-index-weave*:
assumes $is1 \triangleleft (\text{nths } ds \ A)$
and $is2 \triangleleft (\text{nths } ds \ (-A))$
shows $\text{weave } A \ is1 \ is2 \triangleleft ds$
and $\text{nths } (\text{weave } A \ is1 \ is2) \ A = is1$
and $\text{nths } (\text{weave } A \ is1 \ is2) \ (-A) = is2$
proof –
have $\text{length-}ds: \text{length } is1 + \text{length } is2 = \text{length } ds$
using *valid-index-length[OF assms(1)] valid-index-length[OF assms(2)]*
length-weave weave-complementary-nthss **by** *metis*
have $1:\text{length } is1 = \text{card } \{i \in A. i < \text{length } is1 + \text{length } is2\}$ **unfolding** *length-ds*
using *length-nths' assms(1) valid-index-length* **by** *auto*
have $2:\text{length } is2 = \text{card } \{i \in -A. i < \text{length } is1 + \text{length } is2\}$ **unfolding**
length-ds
using *length-nths'[of ds -A] assms(2) valid-index-length* **by** *auto*
show $\text{nths } (\text{weave } A \ is1 \ is2) \ A = is1$ $\text{nths } (\text{weave } A \ is1 \ is2) \ (-A) = is2$ **using**
nths-weave[OF 1 2] **by** *blast+*
then have $\text{nths } (\text{weave } A \ is1 \ is2) \ A \triangleleft (\text{nths } ds \ A)$
 $\text{nths } (\text{weave } A \ is1 \ is2) \ (-A) \triangleleft (\text{nths } ds \ (-A))$ **using** *assms* **by** *auto*
then show $\text{weave } A \ is1 \ is2 \triangleleft ds$ **using** *list-all2-nths valid-index-list-all2-iff* **by**
blast
qed

definition *matricize* $:: \text{nat set} \Rightarrow 'a \ \text{tensor} \Rightarrow 'a \ \text{mat}$ **where**

```

matricize rmodes T = mat
  (prod-list (nth (Tensor.dims T) rmodes))
  (prod-list (nth (Tensor.dims T) (-rmodes)))
  (λ(r, c). Tensor.lookup T (weave rmodes
    (digit-encode (nth (Tensor.dims T) rmodes) r)
    (digit-encode (nth (Tensor.dims T) (-rmodes)) c)
  ))

```

definition *dematricize::nat set ⇒ 'a mat ⇒ nat list ⇒ 'a tensor where*
dematricize rmodes A ds = tensor-from-lookup ds
(λis. A \$\$ (digit-decode (nth ds rmodes) (nth is rmodes),
digit-decode (nth ds (-rmodes)) (nth is (-rmodes))))
)

lemma *dims-matricize:*
dim-row (matricize rmodes T) = prod-list (nth (Tensor.dims T) rmodes)
dim-col (matricize rmodes T) = prod-list (nth (Tensor.dims T) (-rmodes))
unfolding *matricize-def using dim-row-mat by simp-all*

lemma *dims-dematricize: Tensor.dims (dematricize rmodes A ds) = ds*
by *(simp add: dematricize-def dims-tensor-from-lookup)*

lemma *valid-index-nths:*
assumes *is < ds*
shows *nths is A < nths ds A*
using *assms proof (induction arbitrary:A rule:valid-index.induct)*
case *Nil*
then show *?case using nths-nil valid-index.simps by blast*
next
case *(Cons is ds i d)*
then have *nths is {j. Suc j ∈ A} < nths ds {j. Suc j ∈ A}*
by *simp*
then show *?case unfolding nths-Cons*
by *(cases 0∈A; simp-all add: Cons.hyps(2) valid-index.Cons)*
qed

lemma *dematricize-matricize:*
shows *dematricize rmodes (matricize rmodes T) (Tensor.dims T) = T*
proof *(rule tensor-lookup-eqI)*
show *1:Tensor.dims (dematricize rmodes (matricize rmodes T) (Tensor.dims T)) = Tensor.dims T*
by *(simp add: dematricize-def dims-tensor-from-lookup)*
fix *is assume is < Tensor.dims (dematricize rmodes (matricize rmodes T) (Tensor.dims T))*
then have *is < Tensor.dims T using 1 by auto*
let *?rds = (nth (Tensor.dims T) rmodes)*
let *?c ds = (nth (Tensor.dims T) (-rmodes))*

```

have decode-r: digit-decode ?rds (nth $s$  is rmodes) < prod-list ?rds
  by (simp add: <is < Tensor.dims T> valid-index-nths digit-decode-lt)
have decode-c: digit-decode ?c $s$  (nth $s$  is (-rmodes)) < prod-list ?c $s$ 
  by (simp add: <is < Tensor.dims T> valid-index-nths digit-decode-lt)
have (matricize rmodes T) $$
  (digit-decode ?rds (nth $s$  is rmodes),
   digit-decode ?c $s$  (nth $s$  is (- rmodes))) =
  Tensor.lookup T is
unfolding matricize-def
  by (simp add: decode-r decode-c <is < Tensor.dims T> valid-index-nths)
then show Tensor.lookup (dematricize rmodes (matricize rmodes T)) (Tensor.dims
T) is = Tensor.lookup T is
  by (simp add: dematricize-def dims-tensor-from-lookup lookup-tensor-from-lookup[OF
<is < Tensor.dims T>])
qed

```

lemma matricize-dematricize:

```

assumes dim-row A = prod-list (nth $s$  ds rmodes)
and dim-col A = prod-list (nth $s$  ds (-rmodes))
shows matricize rmodes (dematricize rmodes A ds) = A
proof (rule eq-matI)
  show dim-row (matricize rmodes (dematricize rmodes A ds)) = dim-row A
    unfolding assms(1) dematricize-def dims-tensor-from-lookup matricize-def dim-row-mat
by metis
  show dim-col (matricize rmodes (dematricize rmodes A ds)) = dim-col A
    unfolding assms(2) dematricize-def dims-tensor-from-lookup matricize-def dim-col-mat
by metis
  fix r c assume r < dim-row A c < dim-col A
  have valid1:digit-encode (nth $s$  ds rmodes) r < nth $s$  ds rmodes and
    valid2:digit-encode (nth $s$  ds (- rmodes)) c < nth $s$  ds (- rmodes)
  using <r < dim-row A> assms(1) <c < dim-col A> assms(2) digit-encode-valid-index
by auto
  have 0:Tensor.lookup (dematricize rmodes A ds)
    (weave rmodes
     (digit-encode (nth $s$  (Tensor.dims (dematricize rmodes A ds)) rmodes) r)
     (digit-encode (nth $s$  (Tensor.dims (dematricize rmodes A ds)) (- rmodes)) c)
    ) = A $$ (r, c)
  unfolding dematricize-def unfolding dims-tensor-from-lookup
unfolding lookup-tensor-from-lookup[OF valid-index-weave(1)[OF valid1 valid2]]
  using digit-decode-encode-lt[OF <c < dim-col A>[unfolded assms(2)]]
    digit-decode-encode-lt[OF <r < dim-row A>[unfolded assms(1)]]
    valid-index-weave(2)[OF valid1 valid2] valid-index-weave(3)[OF valid1 valid2]
  by presburger
  from <r < dim-row A> have r-le: r < prod-list (nth $s$  (Tensor.dims (dematricize
rmodes A ds)) rmodes)
  by (metis <dim-row (matricize rmodes (dematricize rmodes A ds)) = dim-row
A> matricize-def dim-row-mat(1))
  from <c < dim-col A> have c-le: c < prod-list (nth $s$  (Tensor.dims (dematricize
rmodes A ds)) (- rmodes))

```

by (*metis* $\langle \text{dim-col (matricize rmodes (dematricize rmodes A ds))} = \text{dim-col A} \rangle$
matricize-def dim-col-mat(1))
then show (*matricize rmodes (dematricize rmodes A ds)*) $\$\$ (r, c) = A \$\$ (r,$
 $c)$
unfolding *matricize-def using r-le c-le 0 by simp*
qed

lemma *matricize-add:*

assumes *dims A = dims B*

shows *matricize I A + matricize I B = matricize I (A+B)*

proof (*rule eq-matI*)

show *dim-row (matricize I A + matricize I B) = dim-row (matricize I (A + B))* **by** (*simp add: assms dims-matricize(1)*)

show *dim-col (matricize I A + matricize I B) = dim-col (matricize I (A + B))*

by (*simp add: assms dims-matricize(2)*)

fix *i j* **assume** *ij-le1:i < dim-row (matricize I (A + B)) j < dim-col (matricize I (A + B))*

then have

ij-le2:i < prod-list (nth (Tensor.dims A) I) j < prod-list (nth (Tensor.dims A) (-I)) **and**

ij-le3:i < prod-list (nth (Tensor.dims B) I) j < prod-list (nth (Tensor.dims B) (-I)) **and**

ij-le4:i < prod-list (nth (Tensor.dims (A + B)) I) j < prod-list (nth (Tensor.dims (A + B)) (-I))

by (*simp-all add: assms dims-matricize*)

then have *ij-le5:i < dim-row (matricize I B) j < dim-col (matricize I B)*

by (*simp-all add: assms dims-matricize*)

show (*matricize I A + matricize I B*) $\$\$ (i, j) = \text{matricize I (A + B)} \mathbb{\$} (i, j)$

unfolding *index-add-mat(1)[OF ij-le5]* **unfolding** *matricize-def* **unfolding** *index-mat[OF ij-le2]* *index-mat[OF ij-le3]* *index-mat[OF ij-le4]*

using *assms digit-encode-valid-index ij-le2(1) ij-le2(2) valid-index-weave(1)*

by *auto*

qed

lemma *matricize-0:*

shows *matricize I (tensor0 ds) = 0_m (dim-row (matricize I (tensor0 ds))) (dim-col (matricize I (tensor0 ds)))*

proof (*rule eq-matI*)

show *dim-row (matricize I (tensor0 ds)) = dim-row (0_m (dim-row (matricize I (tensor0 ds))) (dim-col (matricize I (tensor0 ds))))*

unfolding *zero-mat-def dim-row-mat* **by** (*simp add: dims-matricize(1)*)

show *dim-col (matricize I (tensor0 ds)) = dim-col (0_m (dim-row (matricize I (tensor0 ds))) (dim-col (matricize I (tensor0 ds))))*

unfolding *zero-mat-def dim-row-mat* **by** (*simp add: dims-matricize(2)*)

fix *i j* **assume** *ij-le1: i < dim-row (0_m (dim-row (matricize I (tensor0 ds))) (dim-col (matricize I (tensor0 ds))))*

j < dim-col (0_m (dim-row (matricize I (tensor0 ds))) (dim-col (matricize I (tensor0 ds))))

then have *ij-le2:i < dim-row (matricize I (tensor0 ds)) j < dim-col (matricize I (tensor0 ds))*

```

I (tensor0 ds)
  unfolding zero-mat-def dim-row-mat by (simp-all add: dims-matricize)
  show matricize I (tensor0 ds) $$ (i, j) = 0_m (dim-row (matricize I (tensor0
ds))) (dim-col (matricize I (tensor0 ds))) $$ (i, j)
  unfolding zero-mat-def index-mat[OF ij-le2] unfolding matricize-def in-
dex-mat[OF ij-le2[unfolded dims-matricize]]
  by (simp, metis lookup-tensor0 digit-encode-valid-index dims-matricize(1) dims-matricize(2)
dims-tensor0
  ij-le2(1) ij-le2(2) valid-index-weave(1))
qed

end

```

9 CP-Rank and Matrix Rank

```

theory DL-Rank-CP-Rank
imports Tensor-Rank Jordan-Normal-Form.DL-Rank Tensor-Matricization
  Jordan-Normal-Form.DL-Submatrix Jordan-Normal-Form.Missing-VectorSpace
begin

abbreviation mrank A == vec-space.rank (dim-row A) A

no-notation normal-rel (infixl < 60)

lemma lookup-order1-prod:
assumes  $\bigwedge B. B \in \text{set } Bs \implies \text{Tensor.order } B = 1$ 
assumes  $is \triangleleft \text{dims (prod-list } Bs)$ 
shows lookup (prod-list Bs) is = prod-list (map ( $\lambda(i,B). \text{lookup } B [i]$ ) (zip is Bs))
using assms proof (induction Bs arbitrary:is)
  case Nil
  then show ?case unfolding prod-list.Nil unfolding zip.simps tensor-one-def
  by (metis (no-types, lifting) dims-tensor-from-lookup length-greater-0-conv length-map
prod-list.Nil
  lookup-tensor-from-lookup tensor-one-def tensor-one-from-lookup)
next
  case (Cons B Bs is')
  then obtain i is where  $is' = i \# is$ 
  by (metis append-is-Nil-conv dims-tensor-prod length-0-conv list.set-intros(1)
prod-list.Cons valid-index.simps zero-neq-one)
  have Tensor.order B = 1 using Cons by auto
  then have  $\text{valid1}[i] \triangleleft \text{dims } B$ 
  using  $\langle is' \triangleleft \text{dims (prod-list (B \# Bs))} \rangle$ [unfolded prod-list.Cons dims-tensor-prod
 $\langle is' = i \# is \rangle$ ]
  by (metis One-nat-def Suc-length-conv hd-append2 length-0-conv list.sel(1)
list.simps(3) valid-index.Nil valid-index.simps)
  have  $\text{valid2}: is \triangleleft \text{dims (prod-list } Bs)$ 
  using  $\langle is' \triangleleft \text{dims (prod-list (B \# Bs))} \rangle$ [unfolded prod-list.Cons dims-tensor-prod
 $\langle is' = i \# is \rangle$ ]  $\langle \text{Tensor.order } B = 1 \rangle$ 
  by (metis One-nat-def Suc-length-conv append-eq-Cons-conv length-0-conv list.sel(3))

```

list.simps(3) self-append-conv2 valid-indexE
show *?case unfolding* $\langle is' = i \# is \rangle$ *List.zip-Cons-Cons List.list.map(2) prod-list.Cons*
lookup-tensor-prod[OF valid1 valid2, simplified] **by** *(simp add: Cons.IH Cons.prem1)*
valid2)
qed

lemma *matricize-cprank-max1:*

fixes *A::'a::field tensor*

assumes *cprank-max1 A*

shows *mrank (matricize I A) ≤ 1*

proof –

obtain *Bs a* **where** $\bigwedge B. B \in \text{set } Bs \implies \text{Tensor.order } B = 1$ *a · prod-list Bs =*
A

using *cprank-max1-prod-listE assms* **by** *metis*

define *row-factor*

where *row-factor ris = a * prod-list (map ($\lambda(i,B).$ lookup B [i]) (zip ris (nth*
Bs I)))

for *ris*

define *col-factor*

where *col-factor cis = prod-list (map ($\lambda(i,B).$ lookup B [i]) (zip cis (nth*
Bs (-I))))

for *cis*

have $\bigwedge is. is \triangleleft \text{dims } A \implies \text{lookup } A \text{ is} = \text{row-factor (nth is I)} * \text{col-factor}$
(nth is (-I))

proof –

fix *is* **assume** *is \triangleleft dims A*

then have *lookup A is = a * (prod-list (map ($\lambda(i,B).$ lookup B [i]) (zip is Bs)))*

using *lookup-order1-prod[OF $\langle \bigwedge B. B \in \text{set } Bs \implies \text{Tensor.order } B = 1 \rangle$*
lookup-smult

using $\langle a \cdot \text{prod-list } Bs = A \rangle$ *dims-smult* **by** *fastforce*

also have $\dots = a * (\text{prod-list (map ($\lambda(i,B).$ lookup B [i]) (nth (zip is Bs) I)))$

*

$(\text{prod-list (map ($\lambda(i,B).$ lookup B [i]) (nth (zip is Bs) (-I))))$

using *prod-list-complementary-nthss* **by** *auto*

also have $\dots = \text{row-factor (nth is I)} * \text{col-factor (nth is (-I))}$

using *nths-zip row-factor-def col-factor-def* **by** *metis*

finally show *lookup A is = row-factor (nth is I) * col-factor (nth is (-I)) .*

qed

define *row-factor'*

where *row-factor' r = row-factor (digit-encode (nth (Tensor.dims A) I) r)*

for *r*

define *col-factor'*

where *col-factor' c = col-factor (digit-encode (nth (Tensor.dims A) (-I)) c)*

for *c*

have $\bigwedge r c. r < \text{dim-row (matricize I A)} \implies c < \text{dim-col (matricize I A)} \implies \text{ma-}$
*tricize I A \$\$\$ (r,c) = row-factor' r * col-factor' c*

proof –

fix *r c* **assume** *r < dim-row (matricize I A) c < dim-col (matricize I A)*

then have *matricize I A \$\$\$ (r,c) = Tensor.lookup A (weave I*


```

    (digit-encode (nth (Tensor.dims A) I) r)
    (digit-encode (nth (Tensor.dims A) (-I)) c)
  ) unfolding dims-matricize unfolding matricize-def by simp
  also have ... = row-factor' r * col-factor' c
  using ⟨ $\wedge is. is \triangleleft \text{dims } A \implies \text{lookup } A \text{ is} = \text{row-factor } (nth \text{ is } I) * \text{col-factor}$ 
  (nth is (- I))⟩
    valid-index-weave[OF
    digit-encode-valid-index[OF ⟨ $r < \text{dim-row } (\text{matricize } I A)$ ⟩[unfolding dims-matricize]]
    digit-encode-valid-index[OF ⟨ $c < \text{dim-col } (\text{matricize } I A)$ ⟩[unfolding dims-matricize]]]
    valid-index-weave(2) valid-index-weave(3) row-factor'-def col-factor'-def by
metis
  finally show matricize I A $$$ (r,c) = row-factor' r * col-factor' c .
  qed
  then show ?thesis using vec-space.rank-le-1-product-entries[of matricize I A] by
blast
qed

```

lemma *matrix-rank-le-cprank-max*:

fixes $A :: ('a::field) \text{ tensor}$

assumes *cprank-max* r A

shows $\text{mrnk } (\text{matricize } I A) \leq r$

using *assms*

proof (*induction rule:cprank-max.induct*)

fix $ds :: \text{nat list}$

have $\text{matricize } I (\text{tensor0 } ds) = 0_m (\text{dim-row } (\text{matricize } I (\text{tensor0 } ds))) (\text{dim-col } (\text{matricize } I (\text{tensor0 } ds)))$

using *matricize-0* **by** *auto*

then show $\text{mrnk } (\text{matricize } I (\text{tensor0 } ds)) \leq 0$

using *eq-imp-le vec-space.rank-0I* **by** *metis*

next

fix $A B :: 'a \text{ tensor}$ **and** $j :: \text{nat}$

assume $\text{dims } A = \text{dims } B$

assume *cprank-max1* A

assume $\text{mrnk } (\text{matricize } I B) \leq j$

have $\text{mrnk } (\text{matricize } I A) \leq 1$ **using** ⟨*cprank-max1* A⟩ *matricize-cprank-max1* **by** *auto*

have $\text{mrnk } (\text{matricize } I (A + B)) \leq \text{mrnk } (\text{matricize } I A) + \text{mrnk } (\text{matricize } I B)$

using *matricize-add vec-space.rank-subadditive dims-matricize*

carrier-matI index-add-mat(2) ⟨ $\text{dims } A = \text{dims } B$ ⟩ **by** *metis*

then show $\text{mrnk } (\text{matricize } I (A + B)) \leq \text{Suc } j$

using ⟨ $\text{mrnk } (\text{matricize } I A) \leq 1$ ⟩ ⟨ $\text{mrnk } (\text{matricize } I B) \leq j$ ⟩ **by** *linarith*

qed

lemma *matrix-rank-le-cp-rank*:

fixes $A :: ('a::field) \text{ tensor}$

shows $\text{mrnk } (\text{matricize } I A) \leq \text{cprank } A$

using *matrix-rank-le-cprank-max* **using** *cprank-max-cprank* **by** *auto*

end

10 Matrix to Vector Conversion

theory *DL-Flatten-Matrix*
imports *Jordan-Normal-Form.Matrix*
begin

definition *extract-matrix* :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow nat \Rightarrow 'a mat **where**
extract-matrix a m n = mat m n ($\lambda(i,j). a (i*n + j)$)

definition *flatten-matrix* :: 'a mat \Rightarrow (nat \Rightarrow 'a) **where**
flatten-matrix A k = A \$\$ (k div dim-col A, k mod dim-col A)

lemma *two-digit-le*:

*i * n + j < m * n* **if** *i < m* *j < n* **for** *i j :: nat*
using *that* **by** (auto dest!: less-imp-Suc-add simp add: algebra-simps)

lemma *extract-matrix-cong*:

assumes $\bigwedge i. i < m * n \implies a i = b i$
shows *extract-matrix* a m n = *extract-matrix* b m n

proof –

have $\bigwedge i j. i < m \implies j < n \implies a (i*n + j) = b (i*n + j)$ **using** *two-digit-le*
assms **by** blast

then show *?thesis* **unfolding** *extract-matrix-def* **by** auto
qed

lemma *extract-matrix-flatten-matrix*:

extract-matrix (*flatten-matrix* A) (dim-row A) (dim-col A) = A
unfolding *extract-matrix-def* *flatten-matrix-def* **by** auto

lemma *extract-matrix-flatten-matrix-cong*:

assumes $\bigwedge x. x < \text{dim-row } A * \text{dim-col } A \implies f x = \text{flatten-matrix } A x$
shows *extract-matrix* f (dim-row A) (dim-col A) = A

unfolding *extract-matrix-def*

by (metis *assms* *extract-matrix-cong* *extract-matrix-def* *extract-matrix-flatten-matrix*)

lemma *flatten-matrix-extract-matrix*:

flatten-matrix (*extract-matrix* a m n) k = a k **if** *k < m * n*

proof –

from *that* **have** *m * n > 0*

by (cases *m * n = 0*) *simp-all*

then have *m > 0* **and** *n > 0*

by *simp-all*

with *that* **have** *k div n < m*

by (metis *div-eq-0-iff* *div-mult2-eq* *mult.commute* *neg0-conv*)

moreover have *k mod n < n*

using $\langle n > 0 \rangle$ **by** *simp*

ultimately show *?thesis*

by (auto simp add: extract-matrix-def flatten-matrix-def)
qed

lemma *index-extract-matrix*:
assumes $i < m$ $j < n$
shows *extract-matrix* a m n $\$ \$ (i, j) = a (i * n + j)$
unfolding *extract-matrix-def* **using** *assms* **by** *simp*

lemma *dim-extract-matrix*:
shows *dim-row* (*extract-matrix* a m n) = m
and *dim-col* (*extract-matrix* a m n) = n
unfolding *extract-matrix-def* **by** *simp-all*

end

11 Deep Learning Networks

theory *DL-Network*
imports *Tensor-Product*
Jordan-Normal-Form.Matrix Tensor-Unit-Vec DL-Flatten-Matrix
Jordan-Normal-Form.DL-Missing-List
begin

This symbol is used for the Tensor product:

no-notation *Group.monoid.mult* (**infixl** \otimes_1 70)

notation *Matrix.unit-vec* ($unit_v$)
hide-const (**open**) *Matrix.unit-vec*

datatype $'a$ *convnet* = *Input* nat | *Conv* $'a$ $'a$ *convnet* | *Pool* $'a$ *convnet* $'a$ *convnet*

fun *input-sizes* :: $'a$ *convnet* \Rightarrow *nat list* **where**
input-sizes (*Input* M) = $[M]$ |
input-sizes (*Conv* A m) = *input-sizes* m |
input-sizes (*Pool* $m1$ $m2$) = *input-sizes* $m1$ @ *input-sizes* $m2$

fun *count-weights* :: $bool \Rightarrow (nat \times nat)$ *convnet* \Rightarrow *nat* **where**
count-weights shared (*Input* M) = 0 |
count-weights shared (*Conv* ($r0, r1$) m) = $r0 * r1 +$ *count-weights shared* m |
count-weights shared (*Pool* $m1$ $m2$) =
(if *shared*
then max (*count-weights shared* $m1$) (*count-weights shared* $m2$)
else *count-weights shared* $m1 +$ *count-weights shared* $m2$)

fun *output-size* :: $(nat \times nat)$ *convnet* \Rightarrow *nat* **where**
output-size (*Input* M) = M |
output-size (*Conv* ($r0, r1$) m) = $r0$ |
output-size (*Pool* $m1$ $m2$) = *output-size* $m1$

inductive *valid-net* :: (nat × nat) convnet ⇒ bool **where**
valid-net (Input M) |
output-size m = r1 ⇒ *valid-net* m ⇒ *valid-net* (Conv (r0,r1) m) |
output-size m1 = *output-size* m2 ⇒ *valid-net* m1 ⇒ *valid-net* m2 ⇒ *valid-net*
(Pool m1 m2)

fun *insert-weights* :: bool ⇒ (nat × nat) convnet ⇒ (nat ⇒ real) ⇒ real mat
convnet **where**
insert-weights shared (Input M) w = Input M |
insert-weights shared (Conv (r0,r1) m) w = Conv
(extract-matrix w r0 r1)
(insert-weights shared m (λi. w (i+r0*r1))) |
insert-weights shared (Pool m1 m2) w = Pool
(insert-weights shared m1 w)
(insert-weights shared m2 (if shared then w else (λi. w (i+(count-weights shared
m1))))))

fun *remove-weights* :: real mat convnet ⇒ (nat × nat) convnet **where**
remove-weights (Input M) = Input M |
remove-weights (Conv A m) = Conv (dim-row A, dim-col A) (remove-weights m) |
remove-weights (Pool m1 m2) = Pool (remove-weights m1) (remove-weights m2)

abbreviation *output-size'* == (λm. *output-size* (remove-weights m))
abbreviation *valid-net'* == (λm. *valid-net* (remove-weights m))

fun *evaluate-net* :: real mat convnet ⇒ real vec list ⇒ real vec **where**
evaluate-net (Input M) inputs = hd inputs |
evaluate-net (Conv A m) inputs = A *_v evaluate-net m inputs |
evaluate-net (Pool m1 m2) inputs = component-mult
(evaluate-net m1 (take (length (input-sizes m1)) inputs))
(evaluate-net m2 (drop (length (input-sizes m1)) inputs))

definition *mat-tensorlist-mult* :: real mat ⇒ real tensor vec ⇒ nat list ⇒ real
tensor vec

where *mat-tensorlist-mult* A Ts ds
= Matrix.vec (dim-row A) (λj. tensor-from-lookup ds (λis. (A *_v (map-vec (λT.
Tensor.lookup T is) Ts)) \$j))

lemma *insert-weights-cong*:

assumes (∧i. i < count-weights s m ⇒ w1 i = w2 i)

shows *insert-weights* s m w1 = *insert-weights* s m w2

using *assms* **proof** (induction m arbitrary: w1 w2)

case Input

then show ?case by simp

next

case (Conv r01 m)

then obtain r0 r1 **where** r01 = (r0,r1) **by** (meson surj-pair)

```

have 2:insert-weights s m (λi. w1 (i + r0 * r1)) = insert-weights s m (λi. w2
(i + r0 * r1)) using Conv
using ⟨r01 = (r0, r1)⟩ add.commute add-less-cancel-right count-weights.simps(2)
by fastforce
then show ?case unfolding ⟨r01 = (r0, r1)⟩ insert-weights.simps
by (metis Conv.premis ⟨r01 = (r0, r1)⟩ count-weights.simps(2) extract-matrix-cong
trans-less-add1)
next
case (Pool m1 m2)
have 1:insert-weights s m1 w1 = insert-weights s m1 w2
using Pool(1)[of w1 w2] Pool(3)[unfolded count-weights.simps]
by (cases s; auto)
have shared:s=True ⇒ insert-weights s m2 w1 = insert-weights s m2 w2
using Pool(2)[of w1 w2] Pool(3)[unfolded count-weights.simps] by auto
have unshared:s=False ⇒ insert-weights s m2 (λi. w1 (i + count-weights s
m1)) = insert-weights s m2 (λi. w2 (i + count-weights s m1))
using Pool(2) Pool(3) count-weights.simps by fastforce
show ?case unfolding insert-weights.simps 1 using unshared shared by simp
qed

```

lemma *dims-mat-tensorlist-mult*:

assumes $T \in \text{set}_v (\text{mat-tensorlist-mult } A \text{ } Ts \text{ } ds)$

shows $\text{Tensor.dims } T = ds$

proof –

obtain j **where** $T = \text{tensor-from-lookup } ds (\lambda is. (A *_v (\text{map-vec } (\lambda T. \text{Tensor.lookup } T \text{ is}) \text{ } Ts))) \j

using $\text{vec-setE}[OF \text{ } \text{assms}, \text{ } \text{unfolded } \text{mat-tensorlist-mult-def}]$ **by** (metis *dim-vec index-vec*)

then show ?thesis **by** (simp add: *length-tensor-vec-from-lookup tensor-from-lookup-def*)

qed

fun *tensors-from-net* :: *real mat convnet* ⇒ *real tensor vec* **where**

tensors-from-net (Input M) = *Matrix.vec* M (λi. *unit-vec* M i) |

tensors-from-net (Conv A m) = *mat-tensorlist-mult* A (*tensors-from-net* m) (*input-sizes* m) |

tensors-from-net (Pool $m1$ $m2$) = *component-mult* (*tensors-from-net* $m1$) (*tensors-from-net* $m2$)

lemma *output-size-correct-tensors*:

assumes *valid-net'* m

shows *output-size'* m = *dim-vec* (*tensors-from-net* m)

using *assms* **proof** (*induction* m)

case *Input*

then show ?case **by** simp

next

case (Conv A m)

then show ?case

unfolding *remove-weights.simps output-size.simps tensors-from-net.simps*

using *mat-tensorlist-mult-def* **by** auto

```

next
  case (Pool m1 m2)
  then show ?case by (metis convnet.distinct(3) convnet.distinct(5) convnet.inject(3)
dim-component-mult
  min.idem output-size.simps(3) remove-weights.simps(3) tensors-from-net.simps(3)
valid-net.simps)
qed

lemma output-size-correct:
assumes valid-net' m
and map dim-vec inputs = input-sizes m
shows output-size' m = dim-vec (evaluate-net m inputs)
using assms proof (induction m arbitrary:inputs)
  case Input
    then show ?case using length-Cons list.map-sel(1) list.sel(1) list.simps(8)
list.size(3) nat.simps(3) by auto
  next
    case (Conv A m)
    then show ?case unfolding evaluate-net.simps remove-weights.simps output-size.simps
dim-mult-mat-vec
      by auto
  next
    case (Pool m1 m2)
    then have valid-net' m1 valid-net' m2
      using convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-weights.simps(3)
valid-net.cases by fastforce+
      moreover have map dim-vec (take (length (input-sizes m1)) inputs) = in-
put-sizes m1
        map dim-vec (drop (length (input-sizes m1)) inputs) = input-sizes m2
      using Pool.prems(2) by (metis append-eq-conv-conj drop-map input-sizes.simps(3)
take-map)+
      ultimately have
        output-size' m1 = dim-vec (evaluate-net m1 (take (length (input-sizes m1))
inputs))
        output-size' m2 = dim-vec (evaluate-net m2 (drop (length (input-sizes m1))
inputs))
      using Pool.IH by blast+
    then show ?case unfolding evaluate-net.simps remove-weights.simps output-size.simps
      by (metis Pool.prems(1) valid-net' m1 valid-net' m2) dim-component-mult
output-size.simps(3) output-size-correct-tensors remove-weights.simps(3) ten-
sors-from-net.simps(3))
qed

```

```

lemma input-sizes-remove-weights: input-sizes m = input-sizes (remove-weights
m)
  by (induction m; simp)

```

```

lemma dims-tensors-from-net:

```

```

assumes  $T \in \text{set}_v$  (tensors-from-net  $m$ )
shows  $\text{Tensor.dims } T = \text{input-sizes } m$ 
using assms proof (induction  $m$  arbitrary: $T$ )
  case (Input  $M$ )
    then obtain  $j$  where  $T = \text{unit-vec } M j$ 
      using vec-setE tensors-from-net.simps(1) by (metis dim-vec index-vec)
      then show ?case by (simp add: dims-unit-vec)
  next
    case (Conv  $A$   $m$ )
      then show ?case unfolding remove-weights.simps input-sizes.simps
        using dims-mat-tensorlist-mult by (simp add: input-sizes-remove-weights)
  next
    case (Pool  $m1$   $m2$   $T$ )
      then obtain  $i$  where
        component-mult (tensors-from-net  $m1$ ) (tensors-from-net  $m2$ )  $\$ i = T$ 
         $i < \text{dim-vec}$  (tensors-from-net  $m1$ )  $i < \text{dim-vec}$  (tensors-from-net  $m2$ )
        using tensors-from-net.simps vec-setE dim-component-mult by (metis min.strict-boundedE)
      then obtain  $T1$   $T2$  where  $T = T1 \otimes T2$   $T1 \in \text{set}_v$  (tensors-from-net  $m1$ )  $T2 \in \text{set}_v$  (tensors-from-net  $m2$ )
        using vec-setI by (metis index-component-mult)
      then show ?case unfolding remove-weights.simps input-sizes.simps by (simp add: Pool.IH(1) Pool.IH(2))
qed

```

definition *base-input* :: *real mat convnet* \Rightarrow *nat list* \Rightarrow *real vec list* **where**
base-input m *is* = (*map* ($\lambda(n, i). \text{unit}_v$ n i) (*zip* (*input-sizes* m) *is*))

lemma *base-input-length*:

assumes $is \triangleleft \text{input-sizes } m$

shows $\text{input-sizes } m = \text{map } \text{dim-vec} (\text{base-input } m \text{ is})$

proof (*rule* *nth-equalityI*)

have $\text{length} (\text{input-sizes } m) = \text{length } is$ **using** *assms* *valid-index-length* **by** *auto*

then show $\text{length} (\text{input-sizes } m) = \text{length} (\text{map } \text{dim-vec} (\text{base-input } m \text{ is}))$

unfolding *base-input-def* **by** *auto*

{

fix i

assume $i < \text{length} (\text{input-sizes } m)$

then have $\text{map} (\lambda(n, i). \text{unit}_v$ n i) (*zip* (*input-sizes* m) *is*) $! i = \text{unit}_v$ (*input-sizes* m $! i$) (*is* $! i$)

using $\langle \text{length} (\text{input-sizes } m) = \text{length } is \rangle$ **by** *auto*

then have $\text{input-sizes } m ! i = \text{map } \text{dim-vec} (\text{base-input } m \text{ is}) ! i$ **unfolding** *base-input-def* **using** *index-unit-vec*(3)

using $\langle i < \text{length} (\text{input-sizes } m) \rangle$ $\langle \text{length} (\text{input-sizes } m) = \text{length} (\text{map } \text{dim-vec} (\text{base-input } m \text{ is})) \rangle$

base-input-def *assms* *length-map* *nth-map* *valid-index-lt* **by** (*simp* *add*: *input-sizes-remove-weights*)

 }

then show $\bigwedge i. i < \text{length} (\text{input-sizes } m) \Longrightarrow \text{input-sizes } m ! i = \text{map } \text{dim-vec} (\text{base-input } m \text{ is}) ! i$ **by** *auto*

qed

lemma *nth-mat-tensorlist-mult*:

assumes $\bigwedge A. A \in \text{set}_v \text{ Ts} \implies \text{dims } A = ds$

assumes $i < \text{dim-row } A$

assumes $\text{dim-vec } \text{Ts} = \text{dim-col } A$

shows $\text{mat-tensorlist-mult } A \text{ Ts } ds \ \$ i = \text{listsum } ds \ (\text{map } (\lambda j. (A \ \$\$ (i,j)) \cdot \text{Ts } \$ j) \ [0..<\text{dim-vec } \text{Ts}])$

(**is** $= \text{listsum } ds \ ?\text{Ts}'$)

proof (*rule tensor-lookup-eqI*)

have $\text{dims-Ts}' : \bigwedge T. T \in \text{set } ?\text{Ts}' \implies \text{dims } T = ds$

proof –

fix T **assume** $T \in \text{set } ?\text{Ts}'$

then obtain k **where** $T = ?\text{Ts}' ! k$ **and** $k < \text{length } ?\text{Ts}' \ k < \text{dim-vec } \text{Ts}$ **using** *in-set-conv-nth* **by force**

show $\text{dims } T = ds$ **unfolding** $\langle T = ?\text{Ts}' ! k \rangle$ *nth-map[OF $\langle k < \text{length } ?\text{Ts}' \rangle$]* *[unfolded length-map]*

using *assms(1)* $\langle k < \text{dim-vec } \text{Ts} \rangle$

by (*simp add: $\langle k < \text{length } (\text{map } (\lambda j. A \ \$\$ (i, j)) \cdot \text{Ts } \$ j) \ [0..<\text{dim-vec } \text{Ts}]) \rangle$* *vec-setI*)

qed

then show $\text{dims-eq} : \text{dims } (\text{mat-tensorlist-mult } A \ \text{Ts } ds \ \$ i) = \text{dims } (\text{Tensor-Plus.listsum } ds \ (\text{map } (\lambda j. A \ \$\$ (i, j)) \cdot \text{Ts } \$ j) \ [0..<\text{dim-vec } \text{Ts}])$

using *dims-mat-tensorlist-mult assms mat-tensorlist-mult-def listsum-dims*

by (*metis (no-types, lifting) dim-vec vec-setI*)

fix is **assume** $is\text{-valid} : is \triangleleft \text{dims } (\text{mat-tensorlist-mult } A \ \text{Ts } ds \ \$ i)$

then have $is \triangleleft ds$ **using** *dims-eq dims-Ts' listsum-dims* **by** (*metis (no-types, lifting)*)

have *summand-eq*: $\bigwedge j. j \in \{0 ..<\text{dim-vec } \text{Ts}\} \implies \text{row } A \ i \ \$ j * (\text{map-vec } (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts}) \ \$ j = \text{lookup } (A \ \$\$ (i, j)) \cdot \text{Ts } \$ j$ is

using *index-vec $\langle i < \text{dim-row } A \rangle$ row-def $\langle \text{dim-vec } \text{Ts} = \text{dim-col } A \rangle$*

$\langle is \triangleleft ds \rangle$ *assms(1) lookup-smult atLeastLessThan-iff index-map-vec(1) vec-setI*

by *metis*

have $\text{lookup } (\text{mat-tensorlist-mult } A \ \text{Ts } ds \ \$ i) \ is = (A *_{\text{v}} (\text{map-vec } (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts})) \ \$ i$

unfolding *mat-tensorlist-mult-def* **using** *lookup-tensor-from-lookup[OF $\langle is \triangleleft ds \rangle$]* **using** $\langle i < \text{dim-row } A \rangle$ **by** *auto*

also have $\dots = \text{row } A \ i \cdot \text{map-vec } (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts}$

using $\langle i < \text{dim-row } A \rangle$ **by** *simp*

also have $\dots = (\sum j \in \{0 ..<\text{dim-vec } \text{Ts}\}. \text{row } A \ i \ \$ j * (\text{map-vec } (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts}) \ \$ j)$

unfolding *scalar-prod-def nth-rows[OF $\langle i < \text{dim-row } A \rangle$]* **by** *simp*

also have $\dots = (\sum j \in \{0 ..<\text{dim-vec } \text{Ts}\}. \text{lookup } (A \ \$\$ (i, j)) \cdot \text{Ts } \$ j) \ is$ **using** *summand-eq* **by force**

also have $\dots = (\sum A \leftarrow ?\text{Ts}'. \text{lookup } A \ is)$ **unfolding** *map-map*

Groups-List.sum-set-upt-conv-sum-list-nat[symmetric] atLeastLessThan-upt[symmetric]

by *auto*
also have ... = *lookup* (*listsum ds ?Ts'*) *is* **using** *lookup-listsum[OF ‹is ‹ ds›]*
dims-Ts' **by** *fastforce*
finally show *lookup* (*mat-tensorlist-mult A Ts ds \$ i*) *is* = *lookup* (*listsum ds*
?Ts') *is* **by** *metis*
qed

lemma *lookup-tensors-from-net*:
assumes *valid-net' m*
and *is ‹ input-sizes m*
and *j < output-size' m*
shows *Tensor.lookup* (*tensors-from-net m \$ j*) *is* = *evaluate-net m* (*base-input m*
is) *\$ j*
using *assms proof* (*induction m arbitrary:j is*)
case (*Input M*)
then have *j < M* **using** *output-size.simps(1)* **using** *Input* **by** *auto*
then have *1:tensors-from-net* (*Input M*) *\$ j* = *unit-vec M j* **by** *simp*
obtain *i* **where** *is* = [*i*] *i < M* **using** *Input Suc-length-conv input-sizes.simps(1)*
length-0-conv list.size(3) valid-index-length **by** *auto*
then have *2:Tensor.lookup* (*tensors-from-net* (*Input M*) *\$ j*) *is* = (*if i=j then 1*
else 0) **using** *lookup-unit-vec 1* **by** *metis*
have *evaluate-net* (*Input M*) (*map* ($\lambda(n, i). \text{unit}_v n i$) (*zip* (*input-sizes* (*Input*
M)) *is*)) = *unit_v M i* **using** $\langle is = [i] \rangle$ **by** *auto*
then show *?case* **using** $\langle j < M \rangle$ *base-input-def* **by** (*simp add: ‹ i < M ‹*)
next
case (*Conv A m j is*)
have *is-valid:is ‹ input-sizes m* **using** *Conv.premis* **by** *simp*
have *valid-net:valid-net' m* **using** *Conv.premis(1)* **unfolding** *remove-weights.simps*
using *valid-net.simps convnet.distinct(1) convnet.distinct(5) convnet.inject(2)*
by *blast*
then have *length-em: dim-vec* (*evaluate-net m* (*base-input m is*)) = *output-size'*
m
using *output-size-correct base-input-length is-valid* **by** *metis*

have *IH':map-vec* ($\lambda T. \text{Tensor.lookup } T \text{ is}$) (*tensors-from-net m*) =
evaluate-net m (*base-input m is*)
proof (*rule eq-vecI*)
show *equal-lengths: dim-vec* (*map-vec* ($\lambda T. \text{lookup } T \text{ is}$) (*tensors-from-net m*))
= *dim-vec* (*evaluate-net m* (*base-input m is*)) **using** *length-em*
by (*simp add: output-size-correct-tensors valid-net*)
show $\bigwedge i. i < \text{dim-vec} (\text{evaluate-net } m (\text{base-input } m \text{ is})) \implies$
map-vec ($\lambda T. \text{lookup } T \text{ is}$) (*tensors-from-net m*) *\$ i* = *evaluate-net m*
(*base-input m is*) *\$ i*
proof –
fix *i*
assume *i < dim-vec* (*evaluate-net m* (*base-input m is*))
then have *i < output-size' m* **using** *equal-lengths length-em* **by** *auto*
then show *map-vec* ($\lambda T. \text{lookup } T \text{ is}$) (*tensors-from-net m*) *\$ i*
= *evaluate-net m* (*base-input m is*) *\$ i*

```

      using Conv.IH is-valid equal-lengths valid-net base-input-def length-em
nth-map-upt
      length-map nth-map by auto
    qed
  qed

  have Tensor.lookup ((tensors-from-net (Conv A m)) $ j) is =
    (A *_v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))) $ j
  proof -
    have dim-vec (tensors-from-net (Conv A m)) = output-size' (Conv A m)
      using Conv by (simp add: mat-tensorlist-mult-def)
    then have j < dim-vec (tensors-from-net (Conv A m)) using Conv.prem by
auto
    then have (tensors-from-net (Conv A m)) $ j = tensor-from-lookup (input-sizes
m)
      (λis. (A *_v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))))
    $ j
    unfolding tensors-from-net.simps mat-tensorlist-mult-def by fastforce
    then show ?thesis
      using lookup-tensor-from-lookup[OF is-valid] by auto
  qed
  also have (A *_v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))) $ j
    = (A *_v (evaluate-net m (base-input m is))) $ j using IH' by auto
  also have ... = evaluate-net (Conv A m) (base-input (Conv A m) is) $ j
    unfolding base-input-def using evaluate-net.simps by auto
  finally show ?case by auto
next
  case (Pool m1 m2 j is)

```

We split "is" into two parts for each subnet:

```

  obtain is1 is2 where is12-def:is = is1 @ is2 is1 < input-sizes m1 is2 < in-
put-sizes m2
  by (metis Pool.prem(2) input-sizes.simps(3) valid-index-split)

```

Apply the induction hypothesis to the subnets:

```

  have IH:Tensor.lookup (tensors-from-net m1 $ j) is1
    = evaluate-net m1 (map (λ(x, y). unit_v x y) (zip (input-sizes m1) is1)) $ j
      Tensor.lookup (tensors-from-net m2 $ j) is2
    = evaluate-net m2 (map (λ(x, y). unit_v x y) (zip (input-sizes m2) is2)) $ j
  using Pool convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-weights.simps(3)
  valid-net.simps ⟨is1 < input-sizes m1⟩ ⟨is2 < input-sizes m2⟩ output-size.simps(3)
  by (metis base-input-def)+

```

In the Pool layer tensor entries get multiplied:

```

  have lookup-prod: Tensor.lookup (tensors-from-net (Pool m1 m2) $ j) is
    = Tensor.lookup (tensors-from-net m1 $ j) is1 * Tensor.lookup (tensors-from-net
m2 $ j) is2
  proof -
    have j-small: j < dim-vec (tensors-from-net m1) j < dim-vec (tensors-from-net
m2)

```

```

    by (metis Pool.premis(1) Pool.premis(3) convnet.distinct(3) convnet.inject(3)
convnet.simps(9)
    output-size.simps(3) output-size-correct-tensors remove-weights.simps(3) valid-net.cases)+
    then have 0:tensors-from-net (Pool m1 m2) $ j = tensors-from-net m1 $ j  $\otimes$ 
tensors-from-net m2 $ j
    unfolding tensors-from-net.simps using j-small index-component-mult by
blast
    have Tensor.dims (tensors-from-net m1 $ j) = input-sizes m1
    Tensor.dims (tensors-from-net m2 $ j) = input-sizes m2
    using dims-tensors-from-net j-small nth-mem by (simp-all add: vec-setI)
    then have is12-valid:
        is1  $\triangleleft$  Tensor.dims (tensors-from-net m1 $ j)
        is2  $\triangleleft$  Tensor.dims (tensors-from-net m2 $ j)
    using is12-def by presburger+
    then show ?thesis
    unfolding 0 using lookup-tensor-prod[OF is12-valid] is12-def by auto
qed

```

Output values get multiplied in the Pool layer as well:

```

    have evaluate-net (Pool m1 m2) (base-input (Pool m1 m2) is) $ j
    = evaluate-net m1 (base-input m1 is1) $ j * evaluate-net m2 (base-input m2
is2) $ j
    proof -
    have valid-net' m1 valid-net' m2
    using remove-weights.simps valid-net.simps Pool.premis
    by (metis convnet.distinct(3) convnet.distinct(5) convnet.inject(3))+
    have input-sizes m1 = map dim-vec (base-input m1 is1)
    input-sizes m2 = map dim-vec (base-input m2 is2)
    using base-input-def base-input-length base-input-def is12-def by auto
    have j < dim-vec (evaluate-net m1 (base-input m1 is1)) j < dim-vec (evaluate-net
m2 (base-input m2 is2))
    using Pool.premis <input-sizes m1 = map dim-vec (base-input m1 is1)>
<valid-net' m1>
    output-size-correct by (auto,metis Pool.premis(1) Pool.premis(3) <input-sizes
m2 = map dim-vec (base-input m2 is2)>
convnet.distinct(3) convnet.distinct(5) convnet.inject(3) output-size.simps(3)
output-size-correct
remove-weights.simps(3) valid-net.cases)
    then show ?thesis unfolding evaluate-net.simps unfolding base-input-def
    using is12-def(1) is12-def(2) valid-index-length by (simp add: append-eq-conv-conj
drop-map
drop-zip index-component-mult input-sizes-remove-weights take-map take-zip)
    qed

```

```

    then show ?case using lookup-prod IH base-input-def by auto
qed

```

```

primrec extract-weights::bool  $\Rightarrow$  real mat convnet  $\Rightarrow$  nat  $\Rightarrow$  real where
    extract-weights-Input: extract-weights shared (Input M) = ( $\lambda x.$  0)

```

```

| extract-weights-Conv: extract-weights shared (Conv A m) =
  (λx. if x < dim-row A * dim-col A then flatten-matrix A x
    else extract-weights shared m (x - dim-row A * dim-col A))
| extract-weights-Pool: extract-weights shared (Pool m1 m2) =
  (λx. if x < count-weights shared (remove-weights m1)
    then extract-weights shared m1 x
    else extract-weights shared m2 (x - count-weights shared (remove-weights
m1)))

```

```

inductive balanced-net::(nat × nat) convnet ⇒ bool where
  balanced-net-Input: balanced-net (Input M)
| balanced-net-Conv: balanced-net m ⇒ balanced-net (Conv A m)
| balanced-net-Pool: balanced-net m1 ⇒ balanced-net m2 ⇒
  count-weights True m1 = count-weights True m2 ⇒ balanced-net (Pool m1
m2)

```

```

inductive shared-weight-net::real mat convnet ⇒ bool where
  shared-weight-net-Input: shared-weight-net (Input M)
| shared-weight-net-Conv: shared-weight-net m ⇒ shared-weight-net (Conv A m)
| shared-weight-net-Pool: shared-weight-net m1 ⇒ shared-weight-net m2 ⇒
  count-weights True (remove-weights m1) = count-weights True (remove-weights
m2) ⇒
  (∧x. x < count-weights True (remove-weights m1) ⇒ extract-weights True m1
x = extract-weights True m2 x)
  ⇒ shared-weight-net (Pool m1 m2)

```

lemma insert-extract-weights-cong-shared:

assumes shared-weight-net m

assumes $\bigwedge x. x < \text{count-weights True (remove-weights m)} \implies f x = \text{extract-weights True m } x$

shows $m = \text{insert-weights True (remove-weights m)} f$

using *assms* **proof** (induction m arbitrary:f)

case (shared-weight-net-Input M)

then show ?case

by *simp*

next

case (shared-weight-net-Conv m A)

have $\text{extract-matrix } f \text{ (dim-row A) (dim-col A)} = A$

by (*simp add: extract-matrix-cong extract-matrix-flatten-matrix shared-weight-net-Conv.prem*s)

then show ?case

using *shared-weight-net-Conv.IH*[of (λi. f (i + dim-row A * dim-col A))]

using *shared-weight-net-Conv.prem*s **by** *auto*

next

case (shared-weight-net-Pool m1 m2)

have $m1 = \text{insert-weights True (remove-weights m1)} f$

using *shared-weight-net-Pool.IH*(1) *shared-weight-net-Pool.prem*s **by** *auto*

have $m2 = \text{insert-weights True (remove-weights m2)} f$

using *local.shared-weight-net-Pool*(3) *shared-weight-net-Pool.IH*(2)

shared-weight-net-Pool.hyps(4) *shared-weight-net-Pool.prem*s **by** *fastforce*

```

then show ?case
  using ⟨m1 = insert-weights True (remove-weights m1) f⟩ by auto
qed

```

```

lemma insert-extract-weights-cong-unshared:
assumes  $\bigwedge x. x < \text{count-weights False (remove-weights m)} \implies f x = \text{extract-weights False m } x$ 
shows  $m = \text{insert-weights False (remove-weights m) } f$ 
using assms proof (induction m arbitrary:f)
case (Input M)
  then show ?case
    by simp
next
  case (Conv A m)
    then have  $\text{extract-matrix } f \text{ (dim-row } A) \text{ (dim-col } A) = A$ 
      by (metis count-weights.simps(2) extract-matrix-flatten-matrix-cong extract-weights-Conv
remove-weights.simps(2) trans-less-add1)
    then show ?case
      using Conv.IH Conv.premis by auto
next
  case (Pool m1 m2)
    then show ?case
      using Pool.IH(1) Pool.IH(2) Pool.premis by auto
qed

```

```

lemma remove-insert-weights:
shows  $\text{remove-weights (insert-weights } s \text{ m } w) = m$ 
proof (induction m arbitrary:w)
  case Input
    then show ?case by simp
next
  case (Conv r12 m)
    then obtain r1 r2 where  $r12 = (r1, r2)$  by fastforce
    then have  $\text{remove-weights (insert-weights } s \text{ m } w) = m$  using Conv.IH by blast
    then have  $\text{remove-weights (insert-weights } s \text{ (Conv (r1,r2) m) } w) = \text{Conv (r1,r2) } m$ 
      unfolding insert-weights.simps remove-weights.simps
      using extract-matrix-def Conv.IH dim-extract-matrix(1) by (metis dim-col-mat(1)
)
    then show ?case using ⟨r12 = (r1, r2)⟩ by blast
next
  case (Pool m1 m2 w)
    then show ?case unfolding insert-weights.simps remove-weights.simps using
Pool.IH by blast
qed

```

```

lemma extract-insert-weights-shared:
assumes  $x < \text{count-weights True } m$ 
and balanced-net m

```

```

shows extract-weights True (insert-weights True m w) x = w x
using assms
proof (induction m arbitrary:w x)
  case (Input x)
  then show ?case
    by simp
next
  case (Conv r01 m)
  obtain r0 r1 where r01 = (r0,r1) by force
  then show ?case unfolding ⟨r01 = (r0,r1)⟩ insert-weights.simps extract-weights.simps

    apply (cases x < dim-row (extract-matrix w r0 r1) * dim-col (extract-matrix
w r0 r1))
    apply (auto simp add: dim-extract-matrix(1) dim-extract-matrix(2) flat-
ten-matrix-extract-matrix)
    using Conv.IH[of - λi. w (i + r0 * r1)] Conv.prem(1) Conv.prem(2) ⟨r01
= (r0, r1)⟩ balanced-net.cases by force
next
  case (Pool m1 m2)
  then show ?case unfolding insert-weights.simps extract-weights.simps remove-insert-weights
    apply (cases x < count-weights True m1)
    apply (metis balanced-net.simps convnet.distinct(5) convnet.inject(3) count-weights.simps(1)
not-less-zero)
    by (metis (no-types, lifting) balanced-net.simps convnet.distinct(5) convnet.inject(3)
count-weights.simps(1) count-weights.simps(3) less-max-iff-disj not-less-zero)
qed

lemma shared-weight-net-insert-weights: balanced-net m  $\implies$  shared-weight-net (insert-weights
True m w)
proof (induction m arbitrary:w)
  case (Input x)
  then show ?case using insert-weights.simps balanced-net.simps shared-weight-net.simps
by metis
next
  case (Conv r01 m)
  then obtain r0 r1 where r01 = (r0,r1) by force
  then show ?case unfolding ⟨r01 = (r0,r1)⟩ insert-weights.simps
    by (metis Conv.IH Conv.prem balanced-net.simps convnet.distinct(1) conv-
net.distinct(5) convnet.inject(2) shared-weight-net-Conv)
next
  case (Pool m1 m2)
  have balanced-net m1 balanced-net m2
    using Pool.prem balanced-net.simps by blast+
  have  $\bigwedge x. x < \text{count-weights True } m1 \implies$ 
    extract-weights True (insert-weights True m1 w) x = extract-weights True
(insert-weights True m2 w) x
    using extract-insert-weights-shared
    by (metis Pool.prem balanced-net.simps convnet.distinct(3) convnet.distinct(5)
convnet.inject(3))

```

then show *?case unfolding insert-weights.simps using Pool(1)[of w] Pool(2)[of w]*
by (*metis Pool.premis balanced-net.simps convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-insert-weights shared-weight-net-Pool*)
qed

lemma *finite-valid-index: finite {is. is < d}*
proof (*induction ds*)
case *Nil*
then show *?case by (metis List.finite-set finite-subset length-0-conv list.set-intros(1) mem-Collect-eq subsetI valid-index-length)*
next
case (*Cons d ds*)
have $\{is. is < d \# ds\} \subseteq (\bigcup i < d. \{i \# is \mid is. is < ds\})$
proof (*rule subsetI*)
fix *is* **assume** $is \in \{is. is < d \# ds\}$
then have $is < d \# ds$ **by** *auto*
then obtain *i is'* **where** $is = i \# is'$ **by** *blast*
then have $i < d$ **using** $\langle is < d \# ds \rangle$ **by** *blast*
have $is' < ds$ **using** $\langle is = i \# is' \rangle \langle is < d \# ds \rangle$ **by** *blast*
have $is \in \{i \# is \mid is. is < ds\}$ **by** (*simp add: \langle is = i \# is' \rangle \langle is' < ds \rangle*)
then show $is \in (\bigcup i < d. \{i \# is \mid is. is < ds\})$ **using** $\langle i < d \rangle$ **by** *blast*
qed
moreover have $\bigwedge i. finite \{i \# is \mid is. is < ds\}$ **by** (*simp add: Cons.IH*)
ultimately show $finite \{is. is < d \# ds\}$ **by** (*simp add: finite-subset*)
qed

lemma *setsum-valid-index-split:*
 $(\sum is \mid is < ds1 \ @ \ ds2. f \ is) = (\sum is1 \mid is1 < ds1. (\sum is2 \mid is2 < ds2. f \ (is1 \ @ \ is2)))$
proof –
have $1: ((\lambda(is1, is2). is1 \ @ \ is2) \ ‘ (\{is1. is1 < ds1\} \times \{is2. is2 < ds2\})) = \{is. is < ds1 \ @ \ ds2\}$ (**is** *?A = ?B*)
proof (*rule subset-antisym; rule subsetI*)
fix *x* **assume** $x \in ?A$
then show $x \in ?B$ **using** *valid-index-append by auto*
next
fix *x* **assume** $x \in ?B$
then have $x < ds1 \ @ \ ds2$ **by** *auto*
then obtain *x1 x2* **where** $x = x1 \ @ \ x2$ $x1 < ds1$ $x2 < ds2$ **by** (*metis valid-index-split*)
then have $(x1, x2) \in (\{is1. is1 < ds1\} \times \{is2. is2 < ds2\})$ **by** *auto*
then show $x \in ?A$ **using** *imageI \langle x = x1 \ @ \ x2 \rangle by blast*
qed
have $2: inj-on \ (\lambda(is1, is2). is1 \ @ \ is2) \ (\{is1. is1 < ds1\} \times \{is2. is2 < ds2\})$
by (*simp add: inj-on-def valid-index-length*)
show *?thesis*
unfolding *Groups-Big.comm-monoid-add-class.sum.cartesian-product[of \lambda is1 is2. f \ (is1 \ @ \ is2)]*

using *Groups-Big.comm-monoid-add-class.sum.reindex*[*OF 2, of f*] 1
 2 *SigmaE prod.simps*(2) *sum.reindex-cong* **by** (*simp add: split-def*)
qed

lemma *prod-lessThan-split*:
fixes $g :: \text{nat} \Rightarrow \text{real}$ **shows** $\text{prod } g \{..<n+m\} = \text{prod } g \{..<n\} * \text{prod } (\lambda x. g (x+n)) \{..<m\}$
using *Groups-Big.comm-monoid-mult-class.prod.union-inter-neutral*[*of \{..<n\} \{n..<n+m\}*]
g, unfolded ivl-disj-un-one(2)[*OF le-add1*], *OF finite-lessThan finite-atLeastLessThan*]
by (*metis (no-types) add commute add.left-neutral atLeast0LessThan empty-iff ivl-disj-int-one*(2)
prod.shift-bounds-nat-ivl)

lemma *evaluate-net-from-tensors*:
assumes *valid-net' m*
and *map dim-vec inputs = input-sizes m*
and $j < \text{output-size}' m$
shows $\text{evaluate-net } m \text{ inputs } \$ j$
 $= (\sum_{is \in \{is. is \triangleleft \text{input-sizes } m\}} (\prod_{k < \text{length } inputs} \text{inputs } ! k \$ (is!k)) * \text{Tensor.lookup } (\text{tensors-from-net } m \$ j) is)$
using *assms proof (induction m arbitrary:j is inputs)*
case (*Input M*)
then have $\text{length } inputs = 1 \text{ input-sizes } (\text{Input } M) = [M]$ **by** *auto*
{
fix *is* **assume** $is \triangleleft \text{input-sizes } (\text{Input } M)$
then have $\text{length } is = 1$ **by** (*simp add: valid-index-length*)
then have $is = [\text{hd } is]$ **by** (*metis One-nat-def length-0-conv length-Suc-conv list.sel*(1))
then have $\text{Tensor.lookup } (\text{tensors-from-net } (\text{Input } M) \$ j) is = (\text{if } \text{hd } is=j \text{ then } 1 \text{ else } 0)$
by (*metis Input.prem*(3) $\langle \text{input-sizes } (\text{Input } M) = [M] \rangle \langle is \triangleleft \text{input-sizes } (\text{Input } M) \rangle \text{list.distinct}$ (1)
lookup-unit-vec nth-Cons-0 output-size.simps(1) *remove-weights.simps*(1) *tensors-from-net.simps*(1) *valid-indexE index-vec*)
then have $(\prod_{k < \text{length } inputs} \text{inputs } ! k \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } (\text{Input } M) \$ j) is =$
 $(\text{if } is=[j] \text{ then } (\prod_{k < \text{length } inputs} \text{inputs } ! k \$ (is ! k)) \text{ else } 0)$ **using**
 $\langle is = [\text{hd } is] \rangle$ **by** *auto*
}
then have $(\sum_{is \mid is \triangleleft \text{input-sizes } (\text{Input } M)} (\prod_{k < \text{length } inputs} \text{inputs } ! k \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } (\text{Input } M) \$ j) is)$
 $= (\sum_{is \mid is \triangleleft \text{input-sizes } (\text{Input } M)} (\text{if } is=[j] \text{ then } (\prod_{k < \text{length } inputs} \text{inputs } ! k \$ (is ! k)) \text{ else } 0))$ **by** *auto*
also have $(\sum_{is \mid is \triangleleft \text{input-sizes } (\text{Input } M)} (\text{if } is=[j] \text{ then } (\prod_{k < \text{length } inputs} \text{inputs } ! k \$ (is ! k)) \text{ else } 0))$
 $= (\prod_{k < \text{length } inputs} \text{inputs } ! k \$ ([j] ! k))$ **unfolding** *sum.delta*[*OF finite-valid-index*]
using *Input.prem*(3) *valid-index.Cons valid-index.Nil* **by** *auto*
also have $\dots = \text{inputs } ! 0 \$ j$ **using** $\langle \text{length } inputs = 1 \rangle$ **by** (*simp add: prod.lessThan-Suc*)
also have $\dots = \text{evaluate-net } (\text{Input } M) \text{ inputs } \$ j$ **unfolding** *evaluate-net.simps*

by (*metis* $\langle \text{length inputs} = 1 \rangle$ *hd-conv-nth list.size*(\mathcal{I}) *zero-neg-one*)
finally show *?case by auto*
next
case (*Conv A m j*)
have $j < \text{dim-row } A$ **using** *Conv.prem*s(\mathcal{I}) **by** *auto*
have $0: \bigwedge is. is \triangleleft \text{input-sizes } (Conv A m) \implies$
 $(\prod k < \text{length inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } (Conv A m)$
 $\$ j) is =$
 $(\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A j \ \$ i * ((\prod k < \text{length inputs}.$
 $\text{inputs} ! k \ \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } m \ \$ i) is))$
proof –
fix *is* **assume** $is \triangleleft \text{input-sizes } (Conv A m)$
then have $is \triangleleft \text{input-sizes } m$ **by** *simp*
have $0: \text{lookup } (\text{tensors-from-net } (Conv A m) \ \$ j) is =$
 $(\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A j \ \$ i * \text{lookup } (\text{tensors-from-net}$
 $m \ \$ i) is)$
unfolding *tensors-from-net.simps mat-tensorlist-mult-def index-vec*[*OF* $\langle j <$
 $\text{dim-row } A \rangle]$
 $\text{lookup-tensor-from-lookup}$ [*OF* $\langle is \triangleleft \text{input-sizes } m \rangle]$ *index-mult-mat-vec*[*OF* $\langle j$
 $< \text{dim-row } A \rangle]$ *scalar-prod-def*
using *index-map-vec* **by** *auto*
show $(\prod k < \text{length inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } (Conv$
 $A m) \ \$ j) is$
 $= (\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A j \ \$ i * ((\prod k < \text{length}$
 $\text{inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } m \ \$ i) is))$
unfolding 0 *sum-distrib-left* **by** (*simp add: semiring-normalization-rules*(19))
qed
have *valid-net'* m **by** (*metis* *Conv.prem*s(1) *convnet.distinct*(1) *convnet.distinct*(5)
convnet.inject(2) *remove-weights.simps*(2) *valid-net.simps*)
have $\text{map dim-vec inputs} = \text{input-sizes } m$ **by** (*simp add: Conv.prem*s(2))
have $\text{output-size}' m = \text{dim-vec } (\text{tensors-from-net } m)$ **by** (*simp add:* $\langle \text{valid-net}'$
 $m \rangle$ *output-size-correct-tensors*)
have $1: \bigwedge i. i < \text{dim-vec } (\text{tensors-from-net } m) \implies (\sum is \mid is \triangleleft \text{input-sizes } (Conv$
 $A m). ((\prod k < \text{length inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } m \ \$ i)$
 $is)) = \text{evaluate-net } m \ \text{inputs} \ \$ i$ **unfolding** *input-sizes.simps*
using *Conv.IH* $\langle \text{valid-net}' m \rangle$ $\langle \text{map dim-vec inputs} = \text{input-sizes } m \rangle$ $\langle \text{out-}$
 $\text{put-size}' m = \text{dim-vec } (\text{tensors-from-net } m) \rangle$ **by** *simp*

have $(\sum is \mid is \triangleleft \text{input-sizes } (Conv A m). (\prod k < \text{length inputs}. \text{inputs} ! k \ \$ (is !$
 $k)) * \text{lookup } (\text{tensors-from-net } (Conv A m) \ \$ j) is)$
 $= (\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). (\sum is \mid is \triangleleft \text{input-sizes } (Conv$
 $A m). \text{row } A j \ \$ i * ((\prod k < \text{length inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup}$
 $(\text{tensors-from-net } m \ \$ i) is)))$
using *Groups-Big.comm-monoid-add-class.sum.swap* 0 **by** *auto*
also have $\dots = (\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A j \ \$ i * (\sum is \mid$
 $is \triangleleft \text{input-sizes } (Conv A m). ((\prod k < \text{length inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup}$
 $(\text{tensors-from-net } m \ \$ i) is)))$
by (*simp add: sum-distrib-left*)
also have $\dots = (\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A j \ \$ i * \text{evalu-}$

```

ate-net m inputs $ i) using 1 by auto
  also have ... = row A j · evaluate-net m inputs
  by (metis (full-types) ⟨map dim-vec inputs = input-sizes m⟩ ⟨output-size' m =
dim-vec (tensors-from-net m)⟩
  ⟨valid-net' m⟩ output-size-correct scalar-prod-def)
  also have ... = (A *v evaluate-net m inputs) $ j by (simp add: ⟨j < dim-row
A⟩)
  also have ... = evaluate-net (Conv A m) inputs $ j by simp
  finally show ?case by auto
next
  case (Pool m1 m2 j)
  have valid-net' m1 valid-net' m2
  by (metis Pool.premis(1) convnet.distinct(3) convnet.inject(3) convnet.simps(9)
remove-weights.simps(3) valid-net.simps)+
  have j < output-size' m2 j < output-size' m1
  apply (metis Pool.premis(1) Pool.premis(3) convnet.distinct(3) convnet.inject(3)
convnet.simps(9)
  output-size.simps(3) remove-weights.simps(3) valid-net.simps) using Pool.premis
by auto
  then have j < dim-vec (tensors-from-net m1) j < dim-vec (tensors-from-net
m2)
  by (simp-all add: ⟨valid-net' m1⟩ ⟨valid-net' m2⟩ output-size-correct-tensors)

define inputs1 where inputs1 = take (length (input-sizes m1)) inputs
define inputs2 where inputs2 = drop (length (input-sizes m1)) inputs
have map dim-vec inputs1 = input-sizes m1 map dim-vec inputs2 = input-sizes
m2
  apply (metis Pool.premis(2) append-eq-conv-conj input-sizes.simps(3) inputs1-def
take-map)
  by (metis Pool.premis(2) append-eq-conv-conj drop-map input-sizes.simps(3)
inputs2-def)
  have inputs = inputs1 @ inputs2 by (simp add: inputs1-def inputs2-def)
  {
  fix is1 is2 assume is1 < input-sizes m1 is2 < input-sizes m2
  have length is1 = length inputs1
  using ⟨is1 < input-sizes m1⟩ ⟨map dim-vec inputs1 = input-sizes m1⟩
valid-index-length by fastforce
  have length is2 = length inputs2
  using ⟨is2 < input-sizes m2⟩ ⟨map dim-vec inputs2 = input-sizes m2⟩
valid-index-length by fastforce
  have 1:(∏ k<length inputs1. (inputs1 @ inputs2) ! k $ ((is1 @ is2) ! k)) =
(∏ k<length inputs1. inputs1 ! k $ (is1 ! k))
  using ⟨length is1 = length inputs1⟩ ⟨length is2 = length inputs2⟩
nth-append by (metis (no-types, lifting) lessThan-iff prod.cong)
  have 2:(∏ x<length inputs2. (inputs1 @ inputs2) ! (x + length inputs1) $ ((is1
@ is2) ! (x + length inputs1))) =
(∏ k<length inputs2. inputs2 ! k $ (is2 ! k))
  using ⟨length is1 = length inputs1⟩ ⟨length is2 = length inputs2⟩
  by (metis (no-types, lifting) add commute nth-append-length-plus)
}

```

```

have ( $\prod k < \text{length } \text{inputs}. \text{inputs} ! k \ \$ (is1 \ @ \ is2) ! k$ ) = ( $\prod k < \text{length } \text{inputs1}. \text{inputs1} ! k \ \$ (is1 ! k)$ ) * ( $\prod k < \text{length } \text{inputs2}. \text{inputs2} ! k \ \$ (is2 ! k)$ )
unfolding  $\langle \text{inputs} = \text{inputs1} \ @ \ \text{inputs2} \rangle$  length-append prod-lessThan-split
using 1 2 by metis
}
note 1 = this
{
fix is1 is2 assume  $is1 \triangleleft \text{input-sizes } m1$   $is2 \triangleleft \text{input-sizes } m2$ 
then have  $is1 \triangleleft \text{dims } (\text{tensors-from-net } m1 \ \$ j)$   $is2 \triangleleft \text{dims } (\text{tensors-from-net } m2 \ \$ j)$ 
using  $\langle j < \text{dim-vec } (\text{tensors-from-net } m1) \rangle$   $\langle j < \text{dim-vec } (\text{tensors-from-net } m2) \rangle$  dims-tensors-from-net vec-setI by force+
have  $\text{lookup } (\text{tensors-from-net } (\text{Pool } m1 \ m2) \ \$ j) (is1 \ @ \ is2) = \text{lookup } (\text{tensors-from-net } m1 \ \$ j) is1 * \text{lookup } (\text{tensors-from-net } m2 \ \$ j) is2$ 
unfolding tensors-from-net.simps index-component-mult[OF  $\langle j < \text{dim-vec } (\text{tensors-from-net } m1) \rangle \langle j < \text{dim-vec } (\text{tensors-from-net } m2) \rangle$ ]
 $\text{lookup-tensor-prod}[OF \langle is1 \triangleleft \text{dims } (\text{tensors-from-net } m1 \ \$ j) \rangle \langle is2 \triangleleft \text{dims } (\text{tensors-from-net } m2 \ \$ j) \rangle]$  by metis
}
note 2 = this

have j-le-eval:  $j < \text{dim-vec } (\text{evaluate-net } m1 \ (\text{take } (\text{length } (\text{input-sizes } m1)) \ \text{inputs}))$ 
 $j < \text{dim-vec } (\text{evaluate-net } m2 \ (\text{drop } (\text{length } (\text{input-sizes } m1)) \ \text{inputs}))$ 
using  $\langle j < \text{output-size}' m1 \rangle$   $\langle \text{map dim-vec } \text{inputs1} = \text{input-sizes } m1 \rangle$   $\langle \text{valid-net}' m1 \rangle$  inputs1-def output-size-correct
using  $\langle j < \text{output-size}' m2 \rangle$   $\langle \text{map dim-vec } \text{inputs2} = \text{input-sizes } m2 \rangle$   $\langle \text{valid-net}' m2 \rangle$  inputs2-def by auto
have  $(\sum is \mid is \triangleleft \text{input-sizes } (\text{Pool } m1 \ m2). (\prod k < \text{length } \text{inputs}. \text{inputs} ! k \ \$ (is ! k)) * \text{lookup } (\text{tensors-from-net } (\text{Pool } m1 \ m2) \ \$ j) is)$ 
=  $(\sum is1 \mid is1 \triangleleft \text{input-sizes } m1. \sum is2 \mid is2 \triangleleft \text{input-sizes } m2.$ 
 $(\prod k < \text{length } \text{inputs1}. \text{inputs1} ! k \ \$ (is1 ! k)) * (\prod k < \text{length } \text{inputs2}. \text{inputs2} ! k \ \$ (is2 ! k)) *$ 
 $\text{lookup } (\text{tensors-from-net } m1 \ \$ j) is1 * \text{lookup } (\text{tensors-from-net } m2 \ \$ j) is2)$ 
unfolding input-sizes.simps setsum-valid-index-split using 1 2
using mem-Collect-eq sum.cong by (simp add: mult.assoc)
also have ... =  $(\sum is1 \mid is1 \triangleleft \text{input-sizes } m1. (\prod k < \text{length } \text{inputs1}. \text{inputs1} ! k \ \$ (is1 ! k)) * \text{lookup } (\text{tensors-from-net } m1 \ \$ j) is1) *$ 
 $(\sum is2 \mid is2 \triangleleft \text{input-sizes } m2. (\prod k < \text{length } \text{inputs2}. \text{inputs2} ! k \ \$ (is2 ! k)) * \text{lookup } (\text{tensors-from-net } m2 \ \$ j) is2)$ 
unfolding sum-product by (rule sum.cong, metis, rule sum.cong, metis, simp)
also have ... =  $\text{evaluate-net } (\text{Pool } m1 \ m2) \ \text{inputs} \ \$ j$  unfolding evaluate-net.simps index-component-mult[OF j-le-eval]
using Pool.IH(1)[OF  $\langle \text{valid-net}' m1 \rangle - \langle j < \text{output-size}' m1 \rangle$ ] Pool.IH(2)[OF  $\langle \text{valid-net}' m2 \rangle - \langle j < \text{output-size}' m2 \rangle$ ]
using  $\langle \text{map dim-vec } \text{inputs1} = \text{input-sizes } m1 \rangle$   $\langle \text{map dim-vec } \text{inputs2} = \text{input-sizes } m2 \rangle$  inputs1-def inputs2-def by auto
finally show ?case by metis

```

qed

lemma *tensors-from-net-eqI*:

assumes *valid-net' m1 valid-net' m2 input-sizes m1 = input-sizes m2*

assumes $\bigwedge \text{inputs}. \text{input-sizes } m1 = \text{map dim-vec inputs} \implies \text{evaluate-net } m1 \text{ inputs} = \text{evaluate-net } m2 \text{ inputs}$

shows *tensors-from-net m1 = tensors-from-net m2*

proof –

have *map dim-vec (map 0_v (input-sizes m2)) = input-sizes m2*

map dim-vec (map 0_v (input-sizes m1)) = input-sizes m1 **by** (*auto intro:*

nth-equalityI)

then have *output-size' m1 = output-size' m2* **using**

output-size-correct[OF ⟨valid-net' m1⟩ ⟨map dim-vec (map 0_v (input-sizes m1)) = input-sizes m1⟩]

output-size-correct[OF ⟨valid-net' m2⟩ ⟨map dim-vec (map 0_v (input-sizes m2)) = input-sizes m2⟩]

assms(3) assms(4)

by (*metis (no-types)*)

have $\bigwedge is. \text{base-input } m1 \text{ is} = \text{base-input } m2 \text{ is}$

unfolding *base-input-def ⟨input-sizes m1 = input-sizes m2⟩* **by** *metis*

show *?thesis* **by** (*rule eq-vecI, rule tensor-lookup-eqI; metis*

lookup-tensors-from-net[OF ⟨valid-net' m1⟩, unfolded ⟨ $\bigwedge is. \text{base-input } m1 \text{ is} = \text{base-input } m2 \text{ is}$ ⟩ ⟨output-size' m1 = output-size' m2⟩]

lookup-tensors-from-net[OF ⟨valid-net' m2⟩] assms(3) base-input-length

assms(1) assms(2) dims-tensors-from-net output-size-correct-tensors vec-setI

⟨output-size' m1 = output-size' m2⟩ assms(4))

qed

end

12 Concrete Matrices

theory *DL-Concrete-Matrices*

imports *Jordan-Normal-Form.Matrix*

begin

The following definition allows non-square-matrices, *mat_one* (*mat_one n*) only allows square matrices.

definition *id-matrix::nat \Rightarrow nat \Rightarrow real mat*

where *id-matrix nr nc = mat nr nc ($\lambda(r, c). \text{if } r=c \text{ then } 1 \text{ else } 0$)*

lemma *id-matrix-dim: dim-row (id-matrix nr nc) = nr dim-col (id-matrix nr nc) = nc* **by** (*simp-all add: id-matrix-def*)

lemma *row-id-matrix:*

assumes *i < nr*

shows *row (id-matrix nr nc) i = unit-vec nc i*

by (*rule eq-vecI, simp add: assms id-matrix-def unit-vec-def, simp add: id-matrix-dim(2)*)

lemma *unit-eq-0*[*simp*]:
assumes $i: i \geq n$
shows $\text{unit-vec } n \ i = 0_v \ n$
by (*rule eq-vecI, insert i, auto simp: unit-vec-def*)

lemma *mult-id-matrix*:
assumes $i < nr$
shows $(\text{id-matrix } nr \ (\text{dim-vec } v) \ *_v \ v) \ \$ \ i = (\text{if } i < \text{dim-vec } v \ \text{then } v \ \$ \ i \ \text{else } 0) \ (\text{is } ?a \ \$ \ i = ?b)$
proof –
have $?a \ \$ \ i = \text{row } (\text{id-matrix } nr \ (\text{dim-vec } v)) \ i \cdot v$ **using** *index-mult-mat-vec*
assms id-matrix-dim **by** *auto*
also have $\dots = \text{unit-vec } (\text{dim-vec } v) \ i \cdot v$ **using** *row-id-matrix* *assms* **by** *auto*
also have $\dots = ?b$ **using** *scalar-prod-left-unit carrier-vecI unit-eq-0 scalar-prod-left-zero*
by *fastforce*
finally show *?thesis* **by** *auto*
qed

definition *all1-vec::nat \Rightarrow real vec*
where $\text{all1-vec } n = \text{vec } n \ (\lambda i. 1)$

definition *all1-matrix::nat \Rightarrow nat \Rightarrow real mat*
where $\text{all1-matrix } nr \ nc = \text{mat } nr \ nc \ (\lambda(r, c). 1)$

lemma *all1-matrix-dim*: $\text{dim-row } (\text{all1-matrix } nr \ nc) = nr$ $\text{dim-col } (\text{all1-matrix } nr \ nc) = nc$
by (*simp-all add: all1-matrix-def*)

lemma *row-all1-matrix*:
assumes $i < nr$
shows $\text{row } (\text{all1-matrix } nr \ nc) \ i = \text{all1-vec } nc$
apply (*rule eq-vecI*)
apply (*simp add: all1-matrix-def all1-vec-def assms*)
by (*simp add: all1-matrix-def all1-vec-def*)

lemma *all1-vec-scalar-prod*:
shows $\text{all1-vec } (\text{length } xs) \cdot (\text{vec-of-list } xs) = \text{sum-list } xs$
proof –
have $\text{all1-vec } (\text{length } xs) \cdot (\text{vec-of-list } xs) = (\sum i = 0..<\text{dim-vec } (\text{vec-of-list } xs). \text{vec-of-list } xs \ \$ \ i)$
unfolding *scalar-prod-def* **by** (*metis (no-types, lifting) all1-vec-def mult-cancel-right1 sum.ivl-cong*
vec.abs-eq dim-vec index-vec vec-of-list.abs-eq)
also have $\dots = (\sum i = 0..<\text{length } xs. xs \ ! \ i)$ **using** *vec.abs-eq dim-vec vec-of-list.abs-eq*
by (*metis sum.ivl-cong index-vec*)
also have $\dots = \text{sum-list } xs$ **by** (*simp add: sum-list-sum-nth*)
finally show *?thesis* **by** *auto*
qed

lemma *mult-all1-matrix*:
assumes $i < nr$
shows $((all1\text{-matrix } nr \ (dim\text{-vec } v)) *_{\mathbf{v}} v) \$ i = sum\text{-list } (list\text{-of-vec } v) \ (\mathbf{is } ?a \$ i = sum\text{-list } (list\text{-of-vec } v))$
proof –
 have $?a \$ i = row \ (all1\text{-matrix } nr \ (dim\text{-vec } v)) \ i \cdot v$ **using** *index-mult-mat-vec*
assms all1-matrix-dim **by** *auto*
 also have $... = sum\text{-list } (list\text{-of-vec } v)$ **unfolding** *row-all1-matrix*[*OF assms*]
using *all1-vec-scalar-prod*[*of list-of-vec v*]
 by (*metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq*)
 finally show *?thesis* **by** *auto*
qed

definition *copy-first-matrix*:: $nat \Rightarrow nat \Rightarrow real \ mat$
where *copy-first-matrix* $nr \ nc = mat \ nr \ nc \ (\lambda(r, c). \ \mathbf{if } c = 0 \ \mathbf{then } 1 \ \mathbf{else } 0)$

lemma *copy-first-matrix-dim*: $dim\text{-row } (copy\text{-first-matrix } nr \ nc) = nr \ dim\text{-col } (copy\text{-first-matrix } nr \ nc) = nc$
by (*simp-all add: copy-first-matrix-def*)

lemma *row-copy-first-matrix*:
assumes $i < nr$
shows $row \ (copy\text{-first-matrix } nr \ nc) \ i = unit\text{-vec } nc \ 0$
 apply (*rule eq-vecI*)
 apply (*auto simp add: copy-first-matrix-def assms*)[1]
 by (*simp add: copy-first-matrix-def*)

lemma *mult-copy-first-matrix*:
assumes $i < nr$ **and** $dim\text{-vec } v > 0$
shows $(copy\text{-first-matrix } nr \ (dim\text{-vec } v) *_{\mathbf{v}} v) \$ i = v \$ 0 \ (\mathbf{is } ?a \$ i = v \$ 0)$
proof –
 have $?a \$ i = row \ (copy\text{-first-matrix } nr \ (dim\text{-vec } v)) \ i \cdot v$ **using** *index-mult-mat-vec*
assms copy-first-matrix-dim **by** *auto*
 also have $... = unit\text{-vec } (dim\text{-vec } v) \ 0 \cdot v$ **using** *row-copy-first-matrix* *assms* **by** *auto*
 also have $... = v \$ 0$ **using** *assms(2)* *scalar-prod-left-unit carrier-dim-vec* **by** *blast*
 finally show *?thesis* **by** *auto*
qed

end

13 Missing Lemmas of Finite_Set

theory *DL-Missing-Finite-Set*
imports *Main*

```

begin

lemma card-even[simp]: card {a ∈ Collect even. a < 2 * n} = n
proof (induction n)
  case 0
  then show ?case by auto
next
  case (Suc n)
  have {a ∈ Collect even. a < 2 * Suc n} = insert (2*n) {a ∈ Collect even. a <
2 * n}
  using le-eq-less-or-eq less-Suc-eq-le subset-antisym by force
  show ?case
  unfolding ⟨{a ∈ Collect even. a < 2 * Suc n} = insert (2*n) {a ∈ Collect
even. a < 2 * n}⟩
  using Suc card-insert-disjoint[of {a ∈ Collect even. a < 2 * n} 2*n]
  by (simp add: finite-M-bounded-by-nat less-not-refl2)
qed

lemma card-odd[simp]: card {a ∈ Collect odd. a < 2 * n} = n
proof (induction n)
  case 0
  then show ?case by auto
next
  case (Suc n)
  have {a ∈ Collect odd. a < 2 * Suc n} = insert (2*n+1) {a ∈ Collect odd. a
< 2 * n}
  using le-eq-less-or-eq less-Suc-eq-le subset-antisym by force
  show ?case
  unfolding ⟨{a ∈ Collect odd. a < 2 * Suc n} = insert (2*n+1) {a ∈ Collect
odd. a < 2 * n}⟩
  using Suc card-insert-disjoint[of {a ∈ Collect even. a < 2 * n} 2*n]
  by (simp add: finite-M-bounded-by-nat less-not-refl2)
qed

end

```

14 Deep Network Model

```

theory DL-Deep-Model
imports DL-Network Tensor-Matricization Jordan-Normal-Form.DL-Submatrix DL-Concrete-Matrices
DL-Missing-Finite-Set Jordan-Normal-Form.DL-Missing-Sublist Jordan-Normal-Form.Determinant
begin

```

```

hide-const(open) Polynomial.order

```

```

hide-const (open) Matrix.unit-vec

```

```

fun deep-model and deep-model' where
deep-model' Y [] = Input Y |

```

$deep\text{-}model' Y (r \# rs) = Pool (deep\text{-}model Y r rs) (deep\text{-}model Y r rs) |$
 $deep\text{-}model Y r rs = Conv (Y,r) (deep\text{-}model' r rs)$

abbreviation $deep\text{-}model'\text{-}l rs == deep\text{-}model' (rs!0) (tl rs)$

abbreviation $deep\text{-}model\text{-}l rs == deep\text{-}model (rs!0) (rs!1) (tl (tl rs))$

lemma *valid-deep-model: valid-net (deep-model Y r rs)*

apply (*induction rs arbitrary: Y r*)

apply (*simp add: valid-net.intros(1) valid-net.intros(2)*)

using *valid-net.intros(2) valid-net.intros(3)* **by** *auto*

lemma *valid-deep-model': valid-net (deep-model' r rs)*

apply (*induction rs arbitrary: r*)

apply (*simp add: valid-net.intros(1)*)

by (*metis deep-model'.elims deep-model'.simps(2) deep-model.elims output-size.simps valid-net.simps*)

lemma *input-sizes-deep-model':*

assumes $length\ rs \geq 1$

shows $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = replicate (2^{length\ rs - 1}) (last\ rs)$

using *assms* **proof** (*induction butlast rs arbitrary:rs*)

case *Nil*

then have $rs = [rs!0]$

by (*metis One-nat-def diff-diff-cancel diff-zero length-0-conv length-Suc-conv length-butlast nth-Cons-0*)

then have $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = [last\ rs]$

by (*metis deep-model'.simps(1) input-sizes.simps(1) last.simps list.sel(3)*)

then show $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = replicate (2^{length\ rs - 1}) (last\ rs)$

by (*metis One-nat-def <[] = butlast rs> empty-replicate length-butlast list.size(3) power-0 replicate.simps(2)*)

next

case (*Cons r rs' rs*)

then have $IH: input\text{-}sizes (deep\text{-}model'\text{-}l (tl\ rs)) = replicate (2^{length (tl\ rs) - 1}) (last\ rs)$

by (*metis (no-types, lifting) One-nat-def butlast-tl diff-is-0-eq' last-tl length-Cons length-butlast length-tl list.sel(3) list.size(3) nat-le-linear not-one-le-zero*)

have $rs = r \# (tl\ rs)$ **by** (*metis Cons.hyps(2) Cons.prem One-nat-def append-Cons append-butlast-last-id length-greater-0-conv less-le-trans list.sel(3) zero-less-Suc*)

then have $deep\text{-}model'\text{-}l rs = Pool (deep\text{-}model\text{-}l rs) (deep\text{-}model\text{-}l rs)$

by (*metis Cons.hyps(2) One-nat-def butlast.simps(2) deep-model'.elims list.sel(3) list.simps(3) nth-Cons-0 nth-Cons-Suc*)

then have $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = input\text{-}sizes (deep\text{-}model\text{-}l rs) @ input\text{-}sizes (deep\text{-}model\text{-}l rs)$

using *input-sizes.simps(3)* **by** *metis*

also have $\dots = input\text{-}sizes (deep\text{-}model'\text{-}l (tl\ rs)) @ input\text{-}sizes (deep\text{-}model'\text{-}l (tl\ rs))$

by (*metis (no-types, lifting) Cons.hyps(2) One-nat-def deep-model.elims input-sizes.simps(2)*)

$length-Cons$ $length-butlast$ $length-greater-0-conv$ $length-tl$ $list.sel(2)$ $list.sel(3)$
 $list.size(3)$
 $nth-tl$ $one-neq-zero$
also have ... = $replicate$ ($2 \wedge (length (tl\ rs) - 1)$) ($last\ rs$) @ $replicate$ ($2 \wedge (length (tl\ rs) - 1)$) ($last\ rs$)
using IH **by** $auto$
also have ... = $replicate$ ($2 \wedge (length\ rs - 1)$) ($last\ rs$)
using $replicate-add$ [of $2 \wedge (length (tl\ rs) - 1)$ $2 \wedge (length (tl\ rs) - 1)$ $last\ rs$]
by ($metis$ $Cons.hyps(2)$ $One-nat-def$ $butlast-tl$ $length-butlast$ $list.sel(3)$ $list.size(4)$)
 $mult-2-right$
 $power-add$ $power-one-right$)
finally show $?case$ **by** $auto$
qed

lemma $input-sizes-deep-model$:
assumes $length\ rs \geq 2$
shows $input-sizes$ ($deep-model-l\ rs$) = $replicate$ ($2 \wedge (length\ rs - 2)$) ($last\ rs$)
proof –
have $input-sizes$ ($deep-model-l\ rs$) = $input-sizes$ ($deep-model'-l$ ($tl\ rs$))
by ($metis$ $One-nat-def$ $Suc-1$ $assms$ $hd-Cons-tl$ $deep-model.elims$ $input-sizes.simps(2)$)
 $length-Cons$
 $length-greater-0-conv$ $lessI$ $linorder-not-le$ $list.size(3)$ $not-numeral-le-zero$ $nth-tl$)
also have ... = $replicate$ ($2 \wedge (length\ rs - 2)$) ($last\ rs$) **using** $input-sizes-deep-model'$
by ($metis$ ($no-types$, $lifting$) $One-nat-def$ $Suc-1$ $Suc-eq-plus1$ $assms$ $diff-diff-left$ $hd-Cons-tl$ $last-tl$ $length-Cons$ $length-tl$ $linorder-not-le$ $list.size(3)$ $not-less-eq$ $not-numeral-le-zero$ $numeral-le-one-iff$ $semiring-norm(69)$)
finally show $?thesis$ **by** $auto$
qed

lemma $evaluate-net-Conv-id$:
assumes $valid-net'\ m$
and $input-sizes\ m = map\ dim-vec\ input$
and $j < nr$
shows $evaluate-net$ ($Conv$ ($id-matrix\ nr$ ($output-size'\ m$)) m) $input\ \$\ j$
= ($if\ j < output-size'\ m$ then $evaluate-net\ m\ input\ \$\ j$ else 0)
unfolding $evaluate-net.simps$ $output-size-correct$ [OF $assms(1)$ $assms(2)$][$symmetric$]
using $mult-id-matrix$ [OF $\langle j < nr \rangle$, of $evaluate-net\ m\ input$, $unfolded\ dim-vec-of-list$]
by $metis$

lemma $tensors-from-net-Conv-id$:
assumes $valid-net'\ m$
and $i < nr$
shows $tensors-from-net$ ($Conv$ ($id-matrix\ nr$ ($output-size'\ m$)) m) $\$ i$
= ($if\ i < output-size'\ m$ then $tensors-from-net\ m\ \$\ i$ else $tensor0$ ($input-sizes\ m$))
($is\ ?a\ \$\ i = ?b$)
proof ($rule$ $tensor-lookup-eqI$)
have $Tensor.dims$ ($?a\ \$\ i$) = $input-sizes\ m$ **by** ($metis$ $assms(1)$ $assms(2)$ $dims-tensors-from-net$ $id-matrix-dim(1)$ $id-matrix-dim(2)$ $input-sizes.simps(2)$ $output-size.simps(2)$)

output-size-correct-tensors remove-weights.simps(2) valid-net.intros(2) vec-setI
moreover have *Tensor.dims (?b) = input-sizes m using dims-tensors-from-net*
output-size-correct-tensors[OF assms(1)] dims-tensor0 by (simp add: vec-setI)
ultimately show *Tensor.dims (?a \$ i) = Tensor.dims (?b) by auto*

define *Conv_m* **where** *Conv_m = Conv (id-matrix nr (output-size' m)) m*
fix *is*
assume *is < Tensor.dims (?a \$ i)*
then have *is < input-sizes m using <Tensor.dims (?a \$ i) = input-sizes m> by*
auto
have *valid-net' Conv_m by (simp add: assms id-matrix-dim valid-net.intros(2)*
Conv_m-def)
have *base-input m is = base-input Conv_m is by (simp add: Conv_m-def base-input-def)*
have *i < output-size' Conv_m unfolding Conv_m-def remove-weights.simps out-*
put-size.simps
id-matrix-dim using assms by metis
have *is < input-sizes (Conv (id-matrix nr (output-size' m)) m)*
by (metis <is < input-sizes m> input-sizes.simps(2))
then have *f1: lookup (tensors-from-net (Conv (id-matrix nr (output-size' m))*
m) \$ i) is = evaluate-net (Conv (id-matrix nr (output-size' m)) m) (base-input
(Conv (id-matrix nr (output-size' m)) m) is) \$ i
using *Conv_m-def <i < output-size' Conv_m> <valid-net' Conv_m> lookup-tensors-from-net*
by blast
have *lookup (tensor0 (input-sizes m)) is = (0::real)*
by *(meson <is < input-sizes m> lookup-tensor0)*
then show *Tensor.lookup (?a \$ i) is = Tensor.lookup ?b is*
using *Conv_m-def <base-input m is = base-input Conv_m is> <is < input-sizes m>*
assms(1) assms(2)
base-input-length evaluate-net-Conv-id f1 lookup-tensors-from-net by auto
qed

lemma *evaluate-net-Conv-copy-first:*
assumes *valid-net' m*
and *input-sizes m = map dim-vec input*
and *j < nr*
and *output-size' m > 0*
shows *evaluate-net (Conv (copy-first-matrix nr (output-size' m)) m) input \$ j*
= evaluate-net m input \$ 0
unfolding *evaluate-net.simps output-size-correct[OF assms(1) assms(2)[symmetric]]*
using *mult-copy-first-matrix[OF <j < nr>, of evaluate-net m input, unfolded dim-vec-of-list]*
assms(3) copy-first-matrix-dim(1) by (metis <output-size' m = dim-vec (evaluate-net
m input)> assms(4))

lemma *tensors-from-net-Conv-copy-first:*
assumes *valid-net' m*
and *i < nr*
and *output-size' m > 0*
shows *tensors-from-net (Conv (copy-first-matrix nr (output-size' m)) m) \$ i =*
tensors-from-net m \$ 0

(is ?a \$ i = ?b)
proof (rule *tensor-lookup-eqI*)
have *Tensor.dims* (?a \$ i) = *input-sizes* m
by (*metis* *assms*(1) *assms*(2) *copy-first-matrix-dim*(1) *copy-first-matrix-dim*(2)
dims-tensors-from-net
input-sizes.simps(2) *output-size.simps*(2) *output-size-correct-tensors* *remove-weights.simps*(2)
valid-net.intros(2) *vec-setI*)
moreover **have** *Tensor.dims* (?b) = *input-sizes* m **using** *dims-tensors-from-net*
output-size-correct-tensors[OF *assms*(1)] **using** *assms*(3) **by** (*simp* *add*: *vec-setI*)
ultimately **show** *Tensor.dims* (?a \$ i) = *Tensor.dims* (?b) **by** *auto*

define *Conv* **where** *Conv* = *Conv* (*copy-first-matrix* nr (*output-size'* m)) m
fix *is*
assume *is* \triangleleft *Tensor.dims* (?a \$ i)
then **have** *is* \triangleleft *input-sizes* m **using** \langle *Tensor.dims* (?a \$ i) = *input-sizes* m \rangle **by**
auto
have *valid-net'* *Conv* **by** (*simp* *add*: *assms* *copy-first-matrix-dim* *valid-net.intros*(2)
Conv-def)
have *base-input* m *is* = *base-input* *Conv* *is* **by** (*simp* *add*: *Conv-def* *base-input-def*)
have *i* < *output-size'* *Conv* **unfolding** *Conv-def* *remove-weights.simps* *output-size.simps*
copy-first-matrix-dim **using** *assms* **by** *metis*
show *Tensor.lookup* (?a \$ i) *is* = *Tensor.lookup* ?b *is*
by (*metis* *Conv-def* \langle *base-input* m *is* = *base-input* *Conv* *is* \rangle \langle *i* < *output-size'*
Conv \rangle
 \langle *is* \triangleleft *input-sizes* m \rangle \langle *valid-net'* *Conv* \rangle *assms*(1) *assms*(2) *assms*(3) *base-input-length*
evaluate-net-Conv-copy-first *input-sizes.simps*(2) *lookup-tensors-from-net*)
qed

lemma *evaluate-net-Conv-all1*:
assumes *valid-net'* m
and *input-sizes* m = *map* *dim-vec* *input*
and *i* < nr
shows *evaluate-net* (*Conv* (*all1-matrix* nr (*output-size'* m)) m) *input* \$ i
= *Groups-List.sum-list* (*list-of-vec* (*evaluate-net* m *input*))
unfolding *evaluate-net.simps* *output-size-correct*[OF *assms*(1) *assms*(2)[*symmetric*]]
using *mult-all1-matrix*[OF \langle *i* < nr \rangle , of *evaluate-net* m *input*, *unfolded* *dim-vec-of-list*]
assms(3) *all1-matrix-dim*(1) **by** *metis*

lemma *tensors-from-net-Conv-all1*:
assumes *valid-net'* m
and *i* < nr
shows *tensors-from-net* (*Conv* (*all1-matrix* nr (*output-size'* m)) m) \$ i
= *listsum* (*input-sizes* m) (*list-of-vec* (*tensors-from-net* m))
(is ?a \$ i = ?b)
proof (rule *tensor-lookup-eqI*)
have *i* < *dim-vec* ?a **by** (*metis* *assms* *all1-matrix-dim* *output-size.simps*(2)
output-size-correct-tensors *remove-weights.simps*(2) *valid-net.intros*(2))
then **show** *Tensor.dims* (?a \$ i) = *Tensor.dims* (?b)

```

using dims-tensors-from-net input-sizes.simps(2) listsum-dims
by (metis index-vec-of-list in-set-conv-nth length-list-of-vec vec-list vec-setI)

define Convm where Convm = Conv (all1-matrix nr (output-size' m)) m
fix is assume is  $\triangleleft$  Tensor.dims (?a $ i)
then have is  $\triangleleft$  input-sizes m
  using  $\langle i < \text{dim-vec } ?a \rangle$  dims-tensors-from-net input-sizes.simps(2) by (metis
vec-setI)
  then have is  $\triangleleft$  input-sizes Convm by (simp add: Convm-def)
  have valid-net' Convm by (simp add: Convm-def assms all1-matrix-dim valid-net.intros(2))
  have  $i < \text{output-size}' \text{ Conv}_m$  using Convm-def  $\langle i < \text{dim-vec } ?a \rangle$   $\langle \text{valid-net}'$ 
Convm  $\rangle$ 
    output-size-correct-tensors by presburger
  have base-input Convm is = base-input m is unfolding base-input-def Convm-def
input-sizes.simps by metis
  have Tensor.lookup (?a $ i) is = evaluate-net Convm (base-input Convm is) $ i
    using lookup-tensors-from-net[OF  $\langle \text{valid-net}' \text{ Conv}_m \rangle$   $\langle is \triangleleft \text{input-sizes Conv}_m \rangle$ 
 $\langle i < \text{output-size}' \text{ Conv}_m \rangle$  ]
    by (metis Convm-def )
  also have  $\dots = \text{monoid-add-class.sum-list (list-of-vec (evaluate-net } m \text{ (base-input$ 
Convm is)))
    using evaluate-net-Conv-all1 Convm-def  $\langle is \triangleleft \text{input-sizes Conv}_m \rangle$  assms base-input-length
 $\langle i < nr \rangle$ 
    by simp
  also have  $\dots = \text{monoid-add-class.sum-list (list-of-vec (map-vec (\lambda A. lookup A$ 
is)(tensors-from-net m)))
    unfolding  $\langle \text{base-input Conv}_m \text{ is} = \text{base-input } m \text{ is} \rangle$ 
    using lookup-tensors-from-net[OF  $\langle \text{valid-net}' m \rangle$   $\langle is \triangleleft \text{input-sizes } m \rangle$  ]
    base-input-length[OF  $\langle is \triangleleft \text{input-sizes } m \rangle$  ] output-size-correct[OF assms(1)]
output-size-correct-tensors[OF assms(1)]
    eq-vecI[of evaluate-net m (base-input m is) map-vec (\lambda A. lookup A is) (tensors-from-net
m)] index-map-vec(1) index-map-vec(2)
    by force
  also have  $\dots = \text{monoid-add-class.sum-list (map (\lambda A. lookup A is) (list-of-vec$ 
(tensors-from-net m)))
    using eq-vecI[of vec-of-list (list-of-vec (map-vec (\lambda A. lookup A is)(tensors-from-net
m)))
    vec-of-list (map (\lambda A. lookup A is) (list-of-vec (tensors-from-net m)))] dim-vec-of-list
nth-list-of-vec length-map list-vec nth-map index-map-vec(1) index-map-vec(2)
vec-list
    by (metis (no-types, lifting))
  also have  $\dots = \text{Tensor.lookup } ?b \text{ is}$  using dims-tensors-from-net set-list-of-vec
    using lookup-listsum[OF  $\langle is \triangleleft \text{input-sizes } m \rangle$ , of list-of-vec (tensors-from-net
m)]
    by metis
  finally show Tensor.lookup (?a $ i) is = Tensor.lookup ?b is by blast
qed

fun witness and witness' where

```

witness' Y [] = Input Y |
witness' Y (r # rs) = Pool (witness Y r rs) (witness Y r rs) |
witness Y r rs = Conv ((if length rs = 0 then id-matrix else (if length rs = 1 then all1-matrix else copy-first-matrix)) Y r) (witness' r rs)

abbreviation *witness-l rs == witness (rs!0) (rs!1) (tl (tl rs))*

abbreviation *witness'-l rs == witness' (rs!0) (tl rs)*

lemma *witness-is-deep-model: remove-weights (witness Y r rs) = deep-model Y r rs*

proof (*induction rs arbitrary: Y r*)

case *Nil*

then show *?case unfolding witness.simps witness'.simps deep-model.simps deep-model'.simps by (simp add: id-matrix-dim)*

next

case (*Cons r' rs Y r*)

have *dim-row ((if length (r' # rs) = 0 then id-matrix else (if length (r' # rs) = 1 then all1-matrix else copy-first-matrix)) Y r) = Y*

dim-col ((if length (r' # rs) = 0 then id-matrix else (if length (r' # rs) = 1 then all1-matrix else copy-first-matrix)) Y r) = r

by (*simp-all add: all1-matrix-dim copy-first-matrix-dim*)

then show *?case unfolding witness.simps unfolding witness'.simps unfolding remove-weights.simps*

using *Cons by simp*

qed

lemma *witness'-is-deep-model: remove-weights (witness' Y rs) = deep-model' Y rs*

proof (*induction rs arbitrary: Y*)

case *Nil*

then show *?case unfolding witness.simps witness'.simps deep-model.simps deep-model'.simps by (simp add: id-matrix-dim)*

next

case (*Cons r rs Y*)

have *dim-row ((if length rs = 0 then id-matrix else (if length rs = 1 then all1-matrix else copy-first-matrix)) Y r) = Y*

dim-col ((if length rs = 0 then id-matrix else (if length rs = 1 then all1-matrix else copy-first-matrix)) Y r) = r

by (*simp-all add: all1-matrix-dim copy-first-matrix-dim id-matrix-dim*)

then show *?case unfolding witness'.simps unfolding witness.simps unfolding remove-weights.simps*

using *Cons by simp*

qed

lemma *witness-valid: valid-net' (witness Y r rs)*

using *valid-deep-model witness-is-deep-model by auto*

lemma *witness'-valid: valid-net' (witness' Y rs)*

using *valid-deep-model' witness'-is-deep-model by auto*

lemma *shared-weight-net-witness*: *shared-weight-net* (*witness* Y r rs)
proof (*induction* rs *arbitrary*: Y r)
case *Nil*
 then show *?case unfolding* *witness.simps* *witness'.simps* **by** (*simp add*: *shared-weight-net-Conv*
shared-weight-net-Input)
next
 case (*Cons* a rs)
 then show *?case unfolding* *witness.simps* *witness'.simps*
 by (*simp add*: *shared-weight-net-Conv* *shared-weight-net-Input* *shared-weight-net-Pool*)
qed

lemma *witness-l0'*: *witness'* Y $[M]$ =
 (*Pool*
 (*Conv* (*id-matrix* Y M) (*Input* M))
 (*Conv* (*id-matrix* Y M) (*Input* M))
)
unfolding *witness'.simps* *witness.simps* **by** *simp*

lemma *witness-l1*: *witness* Y $r0$ $[M]$ =
 Conv (*all1-matrix* Y $r0$) (*witness'* $r0$ $[M]$)
unfolding *witness'.simps* **by** *simp*

lemma *tensors-ht-l0*:
assumes $j < r0$
shows *tensors-from-net* (*Conv* (*id-matrix* $r0$ M) (*Input* M)) $\$ j$
 = (*if* $j < M$ *then* *unit-vec* M j *else* *tensor0* $[M]$)
 by (*metis* *assms* *input-sizes.simps(1)* *output-size.simps(1)* *remove-weights.simps(1)*
tensors-from-net.simps(1)
 tensors-from-net-Conv-id *valid-net.intros(1)* *index-vec*)

lemma *tensor-prod-unit-vec*:
unit-vec M j \otimes *unit-vec* M j = *tensor-from-lookup* $[M, M]$ (*λis. if* $is = [j, j]$ *then* 1
else 0) (*is* $?A = ?B$)
proof (*rule* *tensor-lookup-eqI*)
 show *Tensor.dims* $?A$ = *Tensor.dims* $?B$
 by (*metis* *append-Cons* *self-append-conv2* *dims-unit-vec* *dims-tensor-prod* *dims-tensor-from-lookup*)
 fix is **assume** *is-valid*: $is \triangleleft$ *Tensor.dims* (*unit-vec* M j \otimes *unit-vec* M j)
 then have $is \triangleleft [M, M]$ **by** (*metis* *append-Cons* *self-append-conv2* *dims-unit-vec*
dims-tensor-prod)
 then obtain $i1$ $i2$ **where** *is-split*: $is = [i1, i2]$ $i1 < M$ $i2 < M$ **using** *list.distinct(1)*
by *blast*
 then have $[i1] \triangleleft$ *Tensor.dims* (*unit-vec* M j) $[i2] \triangleleft$ *Tensor.dims* (*unit-vec* M j)
 by (*simp-all* *add*: *valid-index.Cons* *valid-index.Nil* *dims-unit-vec*)
 have $is = [i1] @ [i2]$ **by** (*simp* *add*: *is-split(1)*)
 show *Tensor.lookup* $?A$ is = *Tensor.lookup* $?B$ is
 unfolding $\langle is = [i1] @ [i2] \rangle$
 lookup-tensor-prod[OF $\langle [i1] \triangleleft$ *Tensor.dims* (*unit-vec* M j) \rangle $\langle [i2] \triangleleft$ *Tensor.dims*
 (*unit-vec* M j) \rangle
 lookup-tensor-from-lookup[OF $\langle is \triangleleft [M, M] \rangle$, *unfolded* $\langle is = [i1] @ [i2] \rangle$)

lookup-unit-vec[*OF* $\langle i1 < M \rangle$] *lookup-unit-vec*[*OF* $\langle i2 < M \rangle$] **by** *fastforce*
qed

lemma *tensors-ht-l0'*:

assumes $j < r0$

shows *tensors-from-net* (*witness'* $r0$ [M]) $\$ j$

= (*if* $j < M$ *then* *unit-vec* $M j \otimes$ *unit-vec* $M j$ *else* *tensor0* [M, M]) (**is** - = ?*b*)

proof -

have *valid-net'* (*Conv* (*id-matrix* $r0 M$) (*Input* M))

by (*metis* *convnet.inject*(3) *list.discI* *witness'.elims* *witness-l0'* *witness-valid*)

have $j\text{-le:} j < \text{dim-vec}$ (*tensors-from-net* (*Conv* (*id-matrix* $r0 M$) (*Input* M)))

using *output-size-correct-tensors*[*OF* $\langle \text{valid-net}'$ (*Conv* (*id-matrix* $r0 M$) (*Input* M)) \rangle ,

unfolded *remove-weights.simps* *output-size.simps* *id-matrix-dim*]

assms **by** *simp*

show ?*thesis*

unfolding *tensors-from-net.simps*(3) *witness-l0'* *index-component-mult*[*OF* $j\text{-le}$ *tensors-ht-l0*[*OF* *assms*]

by *auto*

qed

lemma *lookup-tensors-ht-l0'*:

assumes $j < r0$

and $is \triangleleft [M, M]$

shows (*Tensor.lookup* (*tensors-from-net* (*witness'* $r0$ [M]) $\$ j$)) *is* = (*if* $is = [j, j]$ *then* 1 *else* 0)

proof (*cases* $j < M$)

assume $j < M$

show ?*thesis* **unfolding** *tensors-ht-l0'*[*OF* *assms*(1)] *tensor-prod-unit-vec*

apply (*cases* $is = [j, j]$) **using** $\langle j < M \rangle$ *assms*(2)

by (*simp-all* *add:lookup-tensor-from-lookup*)

next

assume $\neg j < M$

then **have** $is \neq [j, j]$ **using** *assms*(2) **using** *list.distinct*(1) *nth-Cons-0* *valid-index.simps*
by *blast*

show ?*thesis* **unfolding** *tensors-ht-l0'*[*OF* *assms*(1)] *tensor-prod-unit-vec*

using $\langle \neg j < M \rangle$ **by** (*simp* *add:lookup-tensor0*[*OF* *assms*(2)] $\langle is \neq [j, j] \rangle$)

qed

lemma *lookup-tensors-ht-l1*:

assumes $j < r1$

and $is \triangleleft [M, M]$

shows *Tensor.lookup* (*tensors-from-net* (*witness* $r1 r0$ [M]) $\$ j$) *is*

= (*if* $is!0 = is!1 \wedge is!0 < r0$ *then* 1 *else* 0)

proof -

have *witness-l0'-valid: valid-net'* (*witness'* $r0$ [M]) **unfolding** *witness-l0'*

by (*simp* *add: id-matrix-dim* *valid-net.intros*)

have *input-sizes* (*witness'* $r0$ [M]) = [M, M] **unfolding** *witness-l0'* **by** *simp*

have $\text{output-size}' (witness' r0 [M]) = r0$ **unfolding** $witness-l0'$ **using** $witness-l0'-valid$
by ($\text{simp add: id-matrix-dim}$)
have $\text{dim-vec} (\text{tensors-from-net} (witness' r0 [M])) = r0$
using $\langle \text{output-size}' (witness' r0 [M]) = r0 \rangle witness-l0'-valid \text{output-size-correct-tensors}$
by fastforce
have $\text{all0-but1} : \bigwedge i. i \neq is!0 \implies i < r0 \implies \text{Tensor.lookup} (\text{tensors-from-net} (witness' r0 [M]) \$ i) is = 0$
using $\text{lookup-tensors-ht-l0}' \langle is \triangleleft [M, M] \rangle$ **by** auto

have $\text{tensors-from-net} (witness r1 r0 [M]) \$ j =$
 $\text{Tensor-Plus.listsum} [M, M] (\text{list-of-vec} (\text{tensors-from-net} (witness' r0 [M])))$
unfolding $witness-l1$ **using** $\text{tensors-from-net-Conv-all1} [OF witness-l0'-valid \text{assms}(1)]$
 $witness-l0' \langle \text{output-size}' (witness' r0 [M]) = r0 \rangle$ **by** simp
then have $\text{Tensor.lookup} (\text{tensors-from-net} (witness r1 r0 [M]) \$ j) is$
 $= \text{monoid-add-class.sum-list} (\text{map} (\lambda A. \text{Tensor.lookup} A is) (\text{list-of-vec} (\text{tensors-from-net} (witness' r0 [M])))$
using $\text{lookup-listsum} [OF \langle is \triangleleft [M, M] \rangle] \langle \text{input-sizes} (witness' r0 [M]) = [M, M] \rangle$
 $\text{dims-tensors-from-net}$ **by** ($\text{metis set-list-of-vec}$)
also have $\dots = \text{monoid-add-class.sum-list} (\text{map} (\lambda i. \text{lookup} (\text{tensors-from-net} (witness' r0 [M]) \$ i) is) [0..<r0])$
using $\text{map-map} [of (\lambda A. \text{Tensor.lookup} A is) \lambda i. (\text{tensors-from-net} (witness' r0 [M]) \$ i) [0..<r0]]$
using $\text{list-of-vec-map} \langle \text{dim-vec} (\text{tensors-from-net} (witness' r0 [M])) = r0 \rangle$ **by**
($\text{metis} (\text{mono-tags, lifting}) \text{comp-apply map-eq-conv}$)
also have $\dots = (\sum i < r0. \text{Tensor.lookup} ((\text{tensors-from-net} (witness' r0 [M]) \$ i) is)$
 $i) is$
using $\text{sum-set-upt-conv-sum-list-nat atLeast0LessThan}$ **by** (metis atLeast-upt)
also have $\dots = (\text{if } is!0 = is!1 \wedge is!0 < r0 \text{ then } 1 \text{ else } 0)$
proof ($\text{cases } is!0 < r0$)
case True
have $\text{finite} \{0..<r0\}$ **by** auto
have $is!0 \in \{0..<r0\}$ **using** True **by** auto
have $(\sum i < r0. \text{Tensor.lookup} ((\text{tensors-from-net} (witness' r0 [M]) \$ i) is)$
 $= \text{Tensor.lookup} (\text{tensors-from-net} (witness' r0 [M]) \$ (is!0)) is$
using $\langle \text{dim-vec} (\text{tensors-from-net} (witness' r0 [M])) = r0 \rangle$
using $\text{sum.remove} [OF \langle \text{finite} \{0..<r0\} \rangle \langle is!0 \in \{0..<r0\} \rangle,$
 $\text{of } \lambda i. (\text{Tensor.lookup} (\text{tensors-from-net} (witness' r0 [M]) \$ i) is)]$
using $\text{all0-but1 atLeast0LessThan}$ **by** force
then show $?thesis$ **using** $\text{lookup-tensors-ht-l0}' \langle is ! 0 < r0 \rangle \langle is \triangleleft [M, M] \rangle$ **by**
 fastforce
next
case False
then show $?thesis$ **using** $\text{all0-but1 atLeast0LessThan sum.neutral}$ **by** force
qed

finally show *?thesis* **by** *auto*
qed

lemma *length-output-deep-model*:
assumes *remove-weights* $m = \text{deep-model-l } rs$
shows $\text{dim-vec } (\text{tensors-from-net } m) = rs \neq 0$
using *output-size-correct-tensors valid-deep-model*
deep-model.elims output-size.simps(2) **by** (*metis assms*)

lemma *length-output-deep-model'*:
assumes *remove-weights* $m = \text{deep-model'-l } rs$
shows $\text{dim-vec } (\text{tensors-from-net } m) = rs \neq 0$
using *output-size-correct-tensors valid-deep-model'*
deep-model'.elims output-size.simps **by** (*metis assms deep-model.elims*)

lemma *length-output-witness*:
 $\text{dim-vec } (\text{tensors-from-net } (\text{witness-l } rs)) = rs \neq 0$
using *length-output-deep-model witness-is-deep-model* **by** *blast*

lemma *length-output-witness'*:
 $\text{dim-vec } (\text{tensors-from-net } (\text{witness'-l } rs)) = rs \neq 0$
using *length-output-deep-model' witness'-is-deep-model* **by** *blast*

lemma *dims-output-deep-model*:
assumes $\text{length } rs \geq 2$
and $\bigwedge r. r \in \text{set } rs \implies r > 0$
and $j < \text{rs!}0$
and *remove-weights* $m = \text{deep-model-l } rs$
shows $\text{Tensor.dims } (\text{tensors-from-net } m \$ j) = \text{replicate } (2^{\text{length } rs - 2}) (\text{last } rs)$
using *dims-tensors-from-net input-sizes-deep-model[OF assms(1)] output-size-correct-tensors*
valid-deep-model
assms(3) assms(4) input-sizes-remove-weights length-output-witness witness-is-deep-model
by (*metis vec-setI*)

lemma *dims-output-witness*:
assumes $\text{length } rs \geq 2$
and $\bigwedge r. r \in \text{set } rs \implies r > 0$
and $j < \text{rs!}0$
shows $\text{Tensor.dims } (\text{tensors-from-net } (\text{witness-l } rs) \$ j) = \text{replicate } (2^{\text{length } rs - 2}) (\text{last } rs)$
using *dims-output-deep-model witness-is-deep-model assms* **by** *blast*

lemma *dims-output-deep-model'*:
assumes $\text{length } rs \geq 1$
and $\bigwedge r. r \in \text{set } rs \implies r > 0$
and $j < \text{rs!}0$
and *remove-weights* $m = \text{deep-model'-l } rs$
shows $\text{Tensor.dims } (\text{tensors-from-net } m \$ j) = \text{replicate } (2^{\text{length } rs - 1}) (\text{last } rs)$

```

rs)
proof –
  have dim-vec (tensors-from-net m) > j
    using length-output-deep-model' ⟨remove-weights m = deep-model'-l rs⟩ ⟨j <
rs!0⟩ by auto
  then have Tensor.dims (tensors-from-net m $ j) = input-sizes m
    using dims-tensors-from-net[of - m] output-size-correct-tensors
    vec-setI by metis
  then show ?thesis
    using assms(1) input-sizes-deep-model'
    input-sizes-remove-weights[of m, unfolded ⟨remove-weights m = deep-model'-l
rs⟩] by auto
qed

```

```

lemma dims-output-witness':
assumes length rs ≥ 1
and  $\bigwedge r. r \in \text{set } rs \implies r > 0$ 
and j < rs!0
shows Tensor.dims (tensors-from-net (witness'-l rs) $ j) = replicate ( $2^{\text{length } rs - 1}$ ) (last rs)
using dims-output-deep-model' assms witness'-is-deep-model by blast

```

```

abbreviation ten2mat == matricize {n. even n}
abbreviation mat2ten == dematricize {n. even n}

```

```

locale deep-model-correct-params =
fixes shared-weights::bool
fixes rs::nat list
assumes deep:length rs ≥ 3
and no-zeros: $\bigwedge r. r \in \text{set } rs \implies 0 < r$ 
begin

```

```

definition r = min (last rs) (last (butlast rs))
definition N-half =  $2^{\text{length } rs - 3}$ 
definition weight-space-dim = count-weights shared-weights (deep-model-l rs)

```

end

```

locale deep-model-correct-params-y = deep-model-correct-params +
fixes y::nat
assumes y-valid:y < rs ! 0
begin

```

```

definition A :: (nat  $\Rightarrow$  real)  $\Rightarrow$  real tensor
  where A ws = tensors-from-net (insert-weights shared-weights (deep-model-l rs)
ws) $ y
definition A' :: (nat  $\Rightarrow$  real)  $\Rightarrow$  real mat
  where A' ws = ten2mat (A ws)

```

lemma *dims-tensor-deep-model*:
assumes *remove-weights* $m = \text{deep-model-l } rs$
shows $\text{dims } (\text{tensors-from-net } m \ \$ \ y) = \text{replicate } (2 * N\text{-half}) \ (\text{last } rs)$
proof –
 have $\text{dims } (\text{tensors-from-net } m \ \$ \ y) = \text{replicate } (2 \wedge (\text{length } rs - 2)) \ (\text{last } rs)$
 using *dims-output-deep-model*[*OF - no-zeros y-valid assms*] **using** *less-imp-le-nat*
Suc-le-lessD *deep numeral-3-eq-3*
 by *auto*
 then show *?thesis* **using** *N-half-def* **by** (*metis One-nat-def Suc-1 Suc-eq-plus1*
Suc-le-lessD *deep*
diff-diff-left less-numeral-extra(3) numeral-3-eq-3 power-eq-if zero-less-diff)
qed

lemma *order-tensor-deep-model*:
assumes *remove-weights* $m = \text{deep-model-l } rs$
shows $\text{order } (\text{tensors-from-net } m \ \$ \ y) = 2 * N\text{-half}$
 using *dims-tensor-deep-model* **by** (*simp add: assms*)

lemma *dims-A*:
shows $\text{Tensor.dims } (A \ ws) = \text{replicate } (2 * N\text{-half}) \ (\text{last } rs)$
 unfolding *A-def*
 using *dims-tensor-deep-model* *remove-insert-weights* **by** *blast*

lemma *order-A*:
shows $\text{order } (A \ ws) = 2 * N\text{-half}$ **using** *dims-A* *length-replicate* **by** *auto*

lemma *dims-A'*:
shows $\text{dim-row } (A' \ ws) = \text{prod-list } (\text{nths } (\text{Tensor.dims } (A \ ws)) \ \{n. \text{even } n\})$
and $\text{dim-col } (A' \ ws) = \text{prod-list } (\text{nths } (\text{Tensor.dims } (A \ ws)) \ \{n. \text{odd } n\})$
 unfolding *A'-def* *matricize-def* **by** (*simp-all add: A-def Collect-neg-eq*)

lemma *dims-A'-pow*:
shows $\text{dim-row } (A' \ ws) = (\text{last } rs) \wedge N\text{-half}$ $\text{dim-col } (A' \ ws) = (\text{last } rs) \wedge N\text{-half}$
 unfolding *dims-A'* *dims-A* *nths-replicate* *set-le-in* *card-even* *card-odd* *prod-list-replicate*
 by *simp-all*

definition $Aw = \text{tensors-from-net } (\text{witness-l } rs) \ \$ \ y$

definition $Aw' = \text{ten2mat } Aw$

definition $\text{witness-weights} = \text{extract-weights } \text{shared-weights } (\text{witness-l } rs)$

lemma *witness-weights:witness-l* $rs = \text{insert-weights } \text{shared-weights } (\text{deep-model-l } rs) \ \text{witness-weights}$
 by (*metis* (*full-types*) *insert-extract-weights-cong-shared* *insert-extract-weights-cong-unshared*
shared-weight-net-witness *witness-is-deep-model* *witness-weights-def*)

lemma *Aw-def'*: *Aw* = *A* witness-weights **unfolding** *Aw-def* *A-def* **using** witness-weights **by** *auto*

lemma *Aw'-def'*: *Aw'* = *A'* witness-weights **unfolding** *Aw'-def* *A'-def* *Aw-def'* **by** *auto*

lemma *dims-Aw*: *Tensor.dims Aw* = *replicate* (*2 * N-half*) (*last rs*)
unfolding *Aw-def'* **using** *dims-A* **by** *auto*

lemma *order-Aw*: *order Aw* = *2 * N-half*
unfolding *Aw-def'* **using** *order-A* **by** *auto*

lemma *dims-Aw'*:
dim-row Aw' = *prod-list* (*nths* (*Tensor.dims Aw*) {*n. even n*})
dim-col Aw' = *prod-list* (*nths* (*Tensor.dims Aw*) {*n. odd n*})
unfolding *Aw'-def'* *Aw-def'* **using** *dims-A'* **by** *auto*

lemma *dims-Aw'-pow*: *dim-row Aw'* = (*last rs*) \wedge *N-half* *dim-col Aw'* = (*last rs*)
 \wedge *N-half*
unfolding *Aw'-def'* *Aw-def'* **using** *dims-A'-pow* **by** *auto*

lemma *witness-tensor*:

assumes *is* \triangleleft *Tensor.dims Aw*

shows *Tensor.lookup Aw is*

= (*if* *nths is* {*n. even n*} = *nths is* {*n. odd n*} \wedge ($\forall i \in \text{set } is. i < \text{last } (\text{butlast } rs)$) *then* 1 *else* 0)

using *assms deep no-zeros y-valid* **unfolding** *Aw-def* **proof** (*induction butlast* (*butlast* (*butlast rs*)) *arbitrary:rs is y*)

case *Nil*

have *length rs* = 3

by (*rule antisym, metis Nil.hyps One-nat-def Suc-1 Suc-eq-plus1 add-2-eq-Suc'* *diff-diff-left*

length-butlast less-numeral-extra(3) *list.size*(3) *not-le numeral-3-eq-3 zero-less-diff*,
metis $\langle 3 \leq \text{length } rs \rangle$)

then have *rs* = [*rs!0, rs!1, rs!2*] **by** (*metis* (*no-types, lifting*) *Cons-nth-drop-Suc* *One-nat-def Suc-eq-plus1*

append-Nil id-take-nth-drop length-0-conv length-tl lessI list.sel(3) *list.size*(4)
not-le numeral-3-eq-3

numeral-le-one-iff one-add-one semiring-norm(70) *take-0 zero-less-Suc*)

have *input-sizes* (*witness-l* [*rs ! 0, rs ! 1, rs ! 2*]) = [*rs!2, rs!2*]

using *witness.simps witness'.simps input-sizes.simps* **by** *auto*

then have *Tensor.dims* (*tensors-from-net* (*witness-l rs*) \$ *y*) = [*rs!2, rs!2*]

using *dims-tensors-from-net*[*of tensors-from-net* (*witness-l rs*) \$ *y witness-l rs*]

*Nil.prem*s(4) *length-output-witness* $\langle rs = [rs ! 0, rs ! 1, rs ! 2] \rangle$ *vec-setI* **by** *metis*

then have *is* \triangleleft [*rs!2, rs!2*] **using** *Nil.prem*s **by** *metis*

then have *Tensor.lookup* ((*tensors-from-net* (*witness-l rs*))\$*y*) *is*

= (*if* *is ! 0* = *is ! 1* \wedge *is ! 0* < *rs ! 1* *then* 1 *else* 0)

```

using Nil.prems(4) ⟨rs = [rs ! 0, rs ! 1, rs ! 2]⟩ by (metis list.sel(3) lookup-tensors-ht-l1)
have is ! 0 = is ! 1 ∧ is ! 0 < rs ! 1
  ↔ nths is {n. even n} = nths is {n. odd n} ∧ (∀ i ∈ set is. i < last (butlast
rs))
proof -
  have length is = 2 by (metis One-nat-def Suc-eq-plus1 ⟨is < [rs ! 2, rs ! 2]⟩
list.size(3) list.size(4) numeral-2-eq-2 valid-index-length)
  have nths is {n. even n} = [is!0]
  apply (rule nths-only-one)
  using subset-antisym less-2-cases ⟨length is = 2⟩ by fastforce
  have nths is {n. odd n} = [is!1]
  apply (rule nths-only-one)
  using subset-antisym less-2-cases ⟨length is = 2⟩ by fastforce
  have last (butlast rs) = rs!1 by (metis One-nat-def Suc-eq-plus1 ⟨rs = [rs ! 0,
rs ! 1, rs ! 2]⟩
append-butlast-last-id last-conv-nth length-butlast length-tl lessI list.sel(3)
list.simps(3)
list.size(3) list.size(4) nat.simps(3) nth-append)
  show ?thesis unfolding ⟨last (butlast rs) = rs!1⟩
  apply (rule iffI; rule conjI)
  apply (simp add: ⟨nths is (Collect even) = [is ! 0]⟩ ⟨nths is {n. odd n} =
[is ! 1]⟩)
  apply (metis ⟨length is = 2⟩ One-nat-def in-set-conv-nth less-2-cases)
  apply (simp add: ⟨nths is (Collect even) = [is ! 0]⟩ ⟨nths is {n. odd n} = [is
! 1]⟩)
  apply (simp add: ⟨length is = 2⟩)
done
qed
then show ?case unfolding ⟨Tensor.lookup (tensors-from-net (witness-l rs) $
y) is = (if is ! 0 = is ! 1 ∧ is ! 0 < rs ! 1 then 1 else 0)⟩
using witness-is-deep-model witness-valid ⟨rs = [rs ! 0, rs ! 1, rs ! 2]⟩ by auto
next
case (Cons r rs' rs is j)

```

We prove the Induction Hypothesis for "tl rs" and j=0:

```

have rs = r # tl rs by (metis Cons.hyps(2) append-butlast-last-id butlast.simps(1)
hd-append2 list.collapse list.discI list.sel(1))
  have 1:rs' = butlast (butlast (butlast (tl rs))) by (metis Cons.hyps(2) butlast-tl
list.sel(3))
  have 2:3 ≤ length (tl rs) by (metis (no-types, lifting) Cons.hyps(2) Cons.prems(2)
Nitpick.size-list-simp(2) One-nat-def Suc-eq-plus1 ⟨rs = r # tl rs⟩ ⟨rs' = butlast
(butlast (butlast (tl rs)))⟩
diff-diff-left diff-self-eq-0 gr0-conv-Suc le-Suc-eq length-butlast length-tl less-numeral-extra(3)
list.simps(3) numeral-3-eq-3)
  have 3:∧r. r ∈ set (tl rs) ⇒ 0 < r by (metis Cons.prems(3) list.sel(2)
list.set-sel(2))
  have 4:0 < (tl rs) ! 0 using 2 3 by auto
  have IH: ∧is'. is' < Tensor.dims (tensors-from-net (witness-l (tl rs)) $ 0)
  ⇒ Tensor.lookup (tensors-from-net (witness-l (tl rs)) $ 0) is' =

```

(if $nths\ is' (Collect\ even) = nths\ is' \{n.\ odd\ n\} \wedge (\forall i \in set\ is'.\ i < last\ (butlast\ (tl\ rs)))$ then 1 else 0)

using 1 2 3 4 *Cons.hyps(1)* **by** *blast*

The list "is" can be split in two parts:

have $is \triangleleft replicate\ (2^{length\ rs - 2})\ (last\ rs)$

using *Cons.prem(3)* *dims-output-witness 2* **by** (*metis (no-types, lifting) Cons.prem(1) Cons.prem(3)*)

Cons.prem(4) *Nitpick.size-list-simp(2)* *One-nat-def diff-diff-left diff-is-0-eq length-tl*

nat-le-linear not-numeral-le-zero numeral-le-one-iff one-add-one semiring-norm(70)

then have $is \triangleleft replicate\ (2^{length\ (tl\ rs) - 2})\ (last\ rs) @ replicate\ (2^{length\ (tl\ rs) - 2})\ (last\ rs)$

using *Cons.prem dims-output-witness* **by** (*metis 2 Nitpick.size-list-simp(2) One-nat-def*)

diff-diff-left length-tl mult-2 not-numeral-le-zero numeral-le-one-iff one-add-one power.simps(2) replicate-add semiring-norm(70)

then obtain $is1\ is2$ **where** $is = is1 @ is2$ **and**

is1-replicate: is1 $\triangleleft replicate\ (2^{length\ (tl\ rs) - 2})\ (last\ rs)$ **and**

is2-replicate: is2 $\triangleleft replicate\ (2^{length\ (tl\ rs) - 2})\ (last\ rs)$ **by** (*metis valid-index-split*)

then have

is1-valid: is1 $\triangleleft Tensor.dims\ (tensors-from-net\ (witness-l\ (tl\ rs))\ \$\ 0)$ (**is** *?is1*)

and

is2-valid: is2 $\triangleleft Tensor.dims\ (tensors-from-net\ (witness-l\ (tl\ rs))\ \$\ 0)$ (**is** *?is2*)

proof –

have $last\ (tl\ rs) = last\ rs$ **by** (*metis 2* $\langle rs = r \# tl\ rs \rangle$ *last-ConsR list.size(3) not-numeral-le-zero*)

then show *?is1 ?is2* **using** *dims-output-witness[of tl rs]*

using *dims-output-witness[of tl rs] 2 3 is1-replicate is2-replicate* $\langle last\ (tl\ rs) = last\ rs \rangle$ **by** *auto*

qed

A shorthand for the condition to find a "1" in the tensor:

let $?cond = \lambda is\ rs.\ nths\ is\ \{n.\ even\ n\} = nths\ is\ \{n.\ odd\ n\} \wedge (\forall i \in set\ is.\ i < last\ (butlast\ rs))$

We can use the IH on our newly created *is1* and *is2*:

have *IH-is12*:

Tensor.lookup (tensors-from-net (witness-l (tl rs)) \$ 0) is1 =

(if (?cond is1 (tl rs)) then 1 else 0)

Tensor.lookup (tensors-from-net (witness-l (tl rs)) \$ 0) is2 =

(if (?cond is2 (tl rs)) then 1 else 0)

using *IH is1-valid is2-valid* **by** *fast+*

In the induction step we have to add two layers: first the Pool layer, then the Conv layer.

The Pool layer connects the two subtrees. Therefore the two conditions on *is1* and *is2* become one, and we have to prove that they are equivalent:

```

have ?cond is1 (tl rs) ∧ ?cond is2 (tl rs)  $\longleftrightarrow$  ?cond is rs
proof –
  have length is1 = 2  $\wedge$  (length (tl rs) – 2)
    length is2 = 2  $\wedge$  (length (tl rs) – 2)
  using is1-replicate is2-replicate by (simp-all add: valid-index-length)
  then have even (length is1) even (length is2)
    by (metis Cons.hyps(2) One-nat-def add-gr-0 diff-diff-left even-numeral
even-power
length-butlast length-tl list.size(4) one-add-one zero-less-Suc)+
  then have {j. j + length is1 ∈ {n. even n}} = {n. even n}
    {j. j + length is1 ∈ {n. odd n}} = {n. odd n} by simp-all
  have length (nth is2 (Collect even)) = length (nth is2 (Collect odd))
  using length-nths-even  $\langle$ even (length is2) $\rangle$  by blast
  have cond1-iff: (nth is1 (Collect even) = nth is1 {n. odd n} ∧ nth is2
(Collect even) = nth is2 {n. odd n})
    = (nth is (Collect even) = nth is {n. odd n})
  unfolding  $\langle$ is = is1 @ is2 $\rangle$  nths-append
     $\langle$ {j. j + length is1 ∈ {n. odd n}} = {n. odd n} $\rangle$   $\langle$ {j. j + length is1 ∈ {n.
even n}} = {n. even n} $\rangle$ 
  by (simp add:  $\langle$ length (nth is2 (Collect even)) = length (nth is2 (Collect
odd)) $\rangle$ )
  have last (butlast (tl rs)) = last (butlast rs) using Nitpick.size-list-simp(2)
 $\langle$ even (length is1) $\rangle$ 
     $\langle$ length is1 = 2  $\wedge$  (length (tl rs) – 2) $\rangle$  butlast-tl last-tl length-butlast length-tl
not-less-eq zero-less-diff
  by (metis (full-types) Cons.hyps(2) length-Cons less-nat-zero-code)
  have cond2-iff: ( $\forall i \in \text{set } is1. i < \text{last } (\text{butlast } (tl \text{ rs}))$ ) ∧ ( $\forall i \in \text{set } is2. i < \text{last }
(\text{butlast } (tl \text{ rs}))$ )  $\longleftrightarrow$  ( $\forall i \in \text{set } is. i < \text{last } (\text{butlast } rs)$ )
  unfolding  $\langle$ last (butlast (tl rs)) = last (butlast rs) $\rangle$   $\langle$ is = is1 @ is2 $\rangle$  set-append
by blast
  then show ?thesis using cond1-iff cond2-iff by blast
qed

```

Now we can make the Pool layer step:

```

have lookup-witness': Tensor.lookup ((tensors-from-net (witness' (rs ! 1) (tl (tl
rs)))) $ 0) is =
  (if ?cond is rs then 1 else 0)
proof –
  have lookup-prod: Tensor.lookup ((tensors-from-net (witness-l (tl rs)) $ 0)  $\otimes$ 
(tensors-from-net (witness-l (tl rs))) $ 0) is =
  (if ?cond is rs then 1 else 0)
  using  $\langle$ ?cond is1 (tl rs) ∧ ?cond is2 (tl rs)  $\longleftrightarrow$  ?cond is rs $\rangle$ 
  unfolding  $\langle$ is = is1 @ is2 $\rangle$  lookup-tensor-prod[OF is1-valid is2-valid] IH-is12
  by auto
  have witness-l-tl: witness-l (tl rs) = witness (rs ! 1) (rs ! 2) (tl (tl (tl rs)))
  by (metis One-nat-def Suc-1  $\langle$ rs = r # tl rs $\rangle$  nth-Cons-Suc)
  have tl-tl:(tl (tl rs)) = ((rs ! 2) # tl (tl (tl rs)))
proof –
  have length (tl (tl rs))  $\neq$  0

```

```

by (metis One-nat-def Suc-eq-plus1 diff-diff-left diff-is-0-eq length-tl not-less-eq-eq
  Cons.prem1(2) numeral-3-eq-3)
then have tl (tl rs) ≠ []
  by fastforce
then show ?thesis
  by (metis list.exhaust-sel nth-Cons-0 nth-Cons-Suc numeral-2-eq-2 tl-Nil)
qed
have length-gt0:dim-vec (tensors-from-net (witness (rs ! 1) (rs ! 2) (tl (tl (tl
rs)))))) > 0
  using output-size-correct-tensors[of witness (rs ! 1) (rs ! 2) (tl (tl (tl rs)))]
  witness-is-deep-model[of rs ! 1 rs ! 2 tl (tl (tl rs))]
  valid-deep-model[of rs ! 1 rs ! 2 tl (tl (tl rs))] output-size.simps witness.simps
  by (metis 2 3 One-nat-def ⟨rs = r # tl rs⟩ deep-model.elims length-greater-0-conv
list.size(3)
  not-numeral-le-zero nth-Cons-Suc nth-mem)
then have tensors-from-net (witness' (rs ! 1) ((rs ! 2) # tl (tl (tl rs)))) $ 0
  = (tensors-from-net (witness-l (tl rs)) $ 0) ⊗ (tensors-from-net (witness-l (tl
rs)) $ 0)
  unfolding witness'.simps tensors-from-net.simps witness-l-tl using index-component-mult
by blast
then show ?thesis using lookup-prod tl-tl by simp
qed

```

Then we can make the Conv layer step:

```

show ?case
proof -
  have valid-net' (witness' (rs ! 1) (tl (tl rs))) by (simp add: witness'-valid)
  have output-size' (witness' (rs ! 1) (tl (tl rs))) = rs ! 1
  by (metis 2 Nitpick.size-list-simp(2) diff-diff-left diff-is-0-eq hd-Cons-tl deep-model'.simps(2)
deep-model.elims length-tl not-less-eq-eq numeral-2-eq-2 numeral-3-eq-3 one-add-one
output-size.simps(2) output-size.simps(3) tl-Nil witness'-is-deep-model)
  have if-resolve:(if length (tl (tl rs)) = 0 then id-matrix else if length (tl (tl rs))
= 1 then all1-matrix else copy-first-matrix) = copy-first-matrix
  by (metis 2 Cons.prem1(2) Nitpick.size-list-simp(2) One-nat-def Suc-n-not-le-n
not-numeral-le-zero numeral-3-eq-3)
  have tensors-from-net (Conv (copy-first-matrix (rs ! 0) (rs ! 1)) (witness' (rs
! 1) (tl (tl rs)))) $ j =
    tensors-from-net (witness' (rs ! 1) (tl (tl rs))) $ 0
  using tensors-from-net-Conv-copy-first[OF ⟨valid-net' (witness' (rs ! 1) (tl (tl
rs)))⟩ ⟨j < rs ! 0⟩, unfolded ⟨output-size' (witness' (rs ! 1) (tl (tl rs))) = rs ! 1⟩]
  using 4 One-nat-def ⟨rs = r # tl rs⟩ nth-Cons-Suc by metis
  then show ?thesis unfolding witness.simps if-resolve ⟨output-size' (witness'
(rs ! 1) (tl (tl rs))) = rs ! 1⟩
  using lookup-witness' ⟨valid-net' (witness' (rs ! 1) (tl (tl rs)))⟩ hd-conv-nth
output-size-correct-tensors
  by fastforce
qed
qed

```


lemma *witness-matricization*:

assumes $i < \dim\text{-row } Aw'$ **and** $j < \dim\text{-col } Aw'$

shows $Aw' \text{ \#\# } (i, j)$

= (if $i=j \wedge (\forall i0 \in \text{set } (\text{digit-encode } (\text{nths } (\text{Tensor.dims } Aw) \{n. \text{ even } n\}) i). i0 < \text{last } (\text{butlast } rs))$ then 1 else 0)

proof –

define *is* **where** $is = \text{weave } \{n. \text{ even } n\}$

($\text{digit-encode } (\text{nths } (\text{Tensor.dims } Aw) \{n. \text{ even } n\}) i$)

($\text{digit-encode } (\text{nths } (\text{Tensor.dims } Aw) \{n. \text{ odd } n\}) j$)

have *lookup-eq*: $Aw' \text{ \#\# } (i, j) = \text{Tensor.lookup } Aw \text{ is}$

using $Aw'\text{-def } \text{matricize-def } \text{dims-Aw}'(1)[\text{symmetric}, \text{unfolded } A\text{-def}] \text{dims-Aw}'(2)[\text{symmetric}, \text{unfolded } A\text{-def } \text{Collect-neg-eq}]$

$\text{index-mat}(1)[OF \langle i < \dim\text{-row } Aw' \rangle \langle j < \dim\text{-col } Aw' \rangle]$ *is-def* *Collect-neg-eq case-prod-conv*

by (*metis* (*no-types*) $Aw'\text{-def } \text{Collect-neg-eq case-prod-conv is-def } \text{matricize-def}$)

have $is \triangleleft \text{Tensor.dims } Aw$

using $is\text{-def } \text{valid-index-weave } A\text{-def } \text{Collect-neg-eq assms } \text{digit-encode-valid-index } \text{dims-Aw}'$ **by** *metis*

have *even* (*order* Aw)

unfolding $Aw\text{-def}$ **using** *assms* *dims-output-witness even-numeral le-eq-less-or-eq numeral-2-eq-2 numeral-3-eq-3 deep no-zeros y-valid* **by** *fastforce*

have *nths-dimsAw*: $\text{nths } (\text{Tensor.dims } Aw) (\text{Collect even}) = \text{nths } (\text{Tensor.dims } Aw) \{n. \text{ odd } n\}$

proof –

have $0: \text{Tensor.dims } (\text{tensors-from-net } (\text{witness-l } rs) \$ y) = \text{replicate } (2 \wedge (\text{length } rs - 2)) (\text{last } rs)$

using *dims-output-witness*[*OF - no-zeros y-valid*] **using** *deep* **by** *linarith*

show *?thesis unfolding* $A\text{-def}$

using *nths-replicate*

by (*metis* (*no-types, lifting*) $0 Aw\text{-def } \langle \text{even } (\text{order } Aw) \rangle \text{length-replicate length-nths-even}$)

qed

have $i = j \iff \text{nths is } (\text{Collect even}) = \text{nths is } \{n. \text{ odd } n\}$

proof

have *eq-lengths*: $\text{length } (\text{digit-encode } (\text{nths } (\text{Tensor.dims } Aw) (\text{Collect even})) i)$

= $\text{length } (\text{digit-encode } (\text{nths } (\text{Tensor.dims } Aw) \{n. \text{ odd } n\}) j)$

unfolding *length-digit-encode* **by** (*metis* $\langle \text{even } (\text{order } Aw) \rangle \text{length-nths-even}$)

then show $i = j \implies \text{nths is } (\text{Collect even}) = \text{nths is } \{n. \text{ odd } n\}$ **unfolding** *is-def*

using *nths-weave*[*of digit-encode* ($\text{nths } (\text{Tensor.dims } Aw) (\text{Collect even})$) i

$\text{Collect even digit-encode } (\text{nths } (\text{Tensor.dims } Aw) \{n. \text{ odd } n\}) j, \text{unfolded } \text{eq-lengths}, \text{unfolded } \text{Collect-neg-eq}[\text{symmetric}] \text{card-even mult-2}[\text{symmetric}] \text{card-odd}$

nths-dimsAw **by** *simp*

show $\text{nths is } (\text{Collect even}) = \text{nths is } \{n. \text{ odd } n\} \implies i = j$ **unfolding** *is-def*

using *nths-weave*[*of digit-encode* ($\text{nths } (\text{Tensor.dims } Aw) (\text{Collect even})$) i

Collect even digit-encode (nth (Tensor.dims Aw) {n. odd n}) j, unfolded
 eq-lengths, unfolded Collect-neg-eq[symmetric] card-even mult-2[symmetric] card-odd
using ⟨nth (Tensor.dims Aw) (Collect even) = nth (Tensor.dims Aw) {n.
 odd n}⟩
 deep no-zeros y-valid assms digit-decode-encode dims-Aw'
by auto (metis digit-decode-encode-lt)
qed

have $i=j \implies \text{set } (\text{digit-encode } (\text{nth } (\text{Tensor.dims } Aw) \{n. \text{even } n\}) i) = \text{set } is$
unfolding is-def nth-dimsAw
using set-weave[of (digit-encode (nth (Tensor.dims Aw) {n. odd n}) j) Collect
 even
 (digit-encode (nth (Tensor.dims Aw) {n. odd n}) j),
 unfolded mult-2[symmetric] card-even Collect-neg-eq[symmetric]
 card-odd]
 Un-absorb card-even card-odd mult-2 **by** blast
then show ?thesis **unfolding** lookup-eq
using witness-tensor[OF ⟨is < Tensor.dims Aw⟩]
by (simp add: A-def ⟨(i = j) = (nth is (Collect even) = nth is {n. odd n})⟩)
qed

definition rows-with-1 = {i. (∀ i0 ∈ set (digit-encode (nth (Tensor.dims Aw) {n.
 even n}) i). i0 < last (butlast rs))}

lemma card-low-digits:

assumes $m > 0 \wedge d. d \in \text{set } ds \implies m \leq d$

shows $\text{card } \{i. i < \text{prod-list } ds \wedge (\forall i0 \in \text{set } (\text{digit-encode } ds \ i). i0 < m)\} = m \wedge$
 (length ds)

using assms **proof** (induction ds)

case Nil

then show ?case **using** prod-list.Nil **by** simp

next

case (Cons d ds)

define low-digits

where low-digits ds i $\longleftrightarrow i < \text{prod-list } ds \wedge (\forall i0 \in \text{set } (\text{digit-encode } ds \ i). i0$
 < m) **for** ds i

have $\text{card } \{i. \text{low-digits } ds \ i\} = m \wedge (\text{length } ds)$ **unfolding** low-digits-def

by (simp add: Cons.IH Cons.prem1 Cons.prem2)

have $\text{card } \{i. \text{low-digits } (d \# ds) \ i\} = \text{card } (\{..<m\} \times \{i. \text{low-digits } ds \ i\})$

proof –

define f **where** $f \ p = \text{fst } p + d * \text{snd } p$ **for** p

have inj-on f ($\{..<m\} \times \{i. \text{low-digits } ds \ i\}$)

proof (rule inj-onI)

fix x y **assume** $x \in \{..<m\} \times \{i. \text{low-digits } ds \ i\} \ y \in \{..<m\} \times \{i. \text{low-digits}$
 ds i} $f \ x = f \ y$

then have $\text{fst } x < m \ \text{fst } y < m$ **by** auto

then have $\text{fst } x < d \ \text{fst } y < d$ **using** Cons(3) **by** (meson list.set-intros(1) not-le
 order-trans)+

```

then have  $f x \bmod d = \text{fst } x \text{ } f y \bmod d = \text{fst } y$  unfolding  $f\text{-def}$  by  $\text{simp-all}$ 
have  $f x \text{ div } d = \text{snd } x \text{ } f y \text{ div } d = \text{snd } y$  using  $\langle f x = f y \rangle \langle f x \bmod d = \text{fst } x \rangle \langle \text{fst } y < d \rangle$   $f\text{-def}$  by  $\text{auto}$ 
show  $x = y$  using  $\langle f x = f y \rangle \langle f x \text{ div } d = \text{snd } x \rangle \langle f x \bmod d = \text{fst } x \rangle \langle f y \text{ div } d = \text{snd } y \rangle \langle f y \bmod d = \text{fst } y \rangle$   $\text{prod-eqI}$  by  $\text{fastforce}$ 
qed
have  $f' (\{..<m\} \times \{i. \text{low-digits } ds \ i\}) = \{i. \text{low-digits } (d \# ds) \ i\}$ 
proof ( $\text{rule subset-antisym}; \text{rule subsetI}$ )
fix  $x$  assume  $x \in f' (\{..<m\} \times \{i. \text{low-digits } ds \ i\})$ 
then obtain  $i0 \ i1$  where  $x = i0 + d * i1 \ i0 < m \ \text{low-digits } ds \ i1$  using  $f\text{-def}$  by  $\text{force}$ 
then have  $i0 < d$  using  $\text{Cons}(3)$  by ( $\text{meson list.set-intros}(1) \ \text{not-le order-trans}$ )
show  $x \in \{i. \text{low-digits } (d \# ds) \ i\}$  unfolding  $\text{low-digits-def}$ 
proof ( $\text{rule}; \text{rule conjI}$ )
have  $i1 < \text{prod-list } ds \ \forall i0 \in \text{set } (\text{digit-encode } ds \ i1). \ i0 < m$ 
using  $\langle \text{low-digits } ds \ i1 \rangle \text{low-digits-def}$  by  $\text{auto}$ 
show  $x < \text{prod-list } (d \# ds)$  unfolding  $\text{prod-list.Cons}$   $\langle x = i0 + d * i1 \rangle$ 
using  $\langle i0 < d \rangle \langle i1 < \text{prod-list } ds \rangle$ 
proof  $-$ 
have  $d \neq 0$ 
by ( $\text{metis } \langle i0 < d \rangle \ \text{gr-implies-not0}$ )
then have  $(i0 + d * i1) \text{ div } (d * \text{prod-list } ds) = 0$ 
by ( $\text{simp add: Divides.div-mult2-eq } \langle i0 < d \rangle \langle i1 < \text{prod-list } ds \rangle$ )
then show  $i0 + d * i1 < d * \text{prod-list } ds$ 
by ( $\text{metis } (\text{no-types}) \langle i0 < d \rangle \langle i1 < \text{prod-list } ds \rangle \ \text{div-eq-0-iff gr-implies-not0 no-zero-divisors}$ )
qed
show  $\forall i0 \in \text{set } (\text{digit-encode } (d \# ds) \ x). \ i0 < m$ 
using  $\langle \forall i0 \in \text{set } (\text{digit-encode } ds \ i1). \ i0 < m \rangle \langle i0 < d \rangle \langle i0 < m \rangle \langle x = i0 + d * i1 \rangle$  by  $\text{auto}$ 
qed
next
fix  $x$  assume  $x \in \{i. \text{low-digits } (d \# ds) \ i\}$ 
then have  $x < \text{prod-list } (d \# ds) \ \forall i0 \in \text{set } (\text{digit-encode } (d \# ds) \ x). \ i0 < m$ 
using  $\text{low-digits-def}$  by  $\text{auto}$ 
have  $x \bmod d < m$  using  $\langle \forall i0 \in \text{set } (\text{digit-encode } (d \# ds) \ x). \ i0 < m \rangle$  [ $\text{unfolded digit-encode.simps}$ ] by  $\text{simp}$ 
have  $x \text{ div } d < \text{prod-list } ds$  using  $\langle x < \text{prod-list } (d \# ds) \rangle$  [ $\text{unfolded prod-list.Cons}$ ]
by ( $\text{metis div-eq-0-iff div-mult2-eq mult-0-right not-less0}$ )
have  $\forall i0 \in \text{set } (\text{digit-encode } ds \ (x \text{ div } d)). \ i0 < m$  by ( $\text{simp add: } \langle \forall i0 \in \text{set } (\text{digit-encode } (d \# ds) \ x). \ i0 < m \rangle$ )
have  $f ((x \bmod d), (x \text{ div } d)) = x$  by ( $\text{simp add: } f\text{-def}$ )
show  $x \in f' (\{..<m\} \times \{i. \text{low-digits } ds \ i\})$  by ( $\text{metis SigmaI } \langle \forall i0 \in \text{set } (\text{digit-encode } ds \ (x \text{ div } d)). \ i0 < m \rangle \langle f ((x \bmod d), (x \text{ div } d)) = x \rangle \langle x \text{ div } d < \text{prod-list } ds \rangle \langle x \bmod d < m \rangle \ \text{image-eqI lessThan-iff low-digits-def mem-Collect-eq}$ )
qed
then have  $\text{bij-betw } f (\{..<m\} \times \{i. \text{low-digits } ds \ i\}) \ \{i. \text{low-digits } (d \# ds) \ i\}$ 
by ( $\text{simp add: } \langle \text{inj-on } f (\{..<m\} \times \{i. \text{low-digits } ds \ i\}) \rangle \ \text{bij-betw-def}$ )

```

```

    then show ?thesis by (simp add: bij-betw-same-card)
  qed
  then show ?case unfolding ⟨card {i. low-digits ds i} = m ^ (length ds)⟩
  card-cartesian-product using low-digits-def by simp
  qed

lemma card-rows-with-1: card {i∈rows-with-1. i<dim-row Aw'} = r ^ N-half
proof -
  have 1:{i∈rows-with-1. i<dim-row Aw'} = {i. i < prod-list (nth (Tensor.dims
  Aw) (Collect even)) ∧
    (∀i0∈set (digit-encode (nth (Tensor.dims Aw) (Collect even)) i). i0 <
  r)} (is ?A = ?B)
  proof (rule subset-antisym; rule subsetI)
    fix i assume i ∈ ?A
    then have i < dim-row Aw' ∀i0∈set (digit-encode (nth (Tensor.dims Aw) {n.
  even n}) i). i0 < last (butlast rs)
      using rows-with-1-def by auto
    then have i < prod-list (nth (dims Aw) (Collect even)) using dims-Aw' by
  linarith
    then have digit-encode (nth (dims Aw) (Collect even)) i < nth (dims Aw)
  (Collect even)
      using digit-encode-valid-index by auto
    have ∀i0∈set (digit-encode (nth (Tensor.dims Aw) {n. even n}) i). i0 < r
  proof
    fix i0 assume 1:i0 ∈ set (digit-encode (nth (dims Aw) (Collect even)) i)
    then obtain k where k < length (digit-encode (nth (dims Aw) (Collect
  even)) i)
      digit-encode (nth (dims Aw) (Collect even)) i ! k = i0 by (meson
  in-set-conv-nth)
    have i0 < last (butlast rs)
      using ⟨∀i0∈set (digit-encode (nth (dims Aw) (Collect even)) i). i0 < last
  (butlast rs)⟩ 1 by blast
    have set (nth (dims Aw) (Collect even)) ⊆ {last rs} unfolding dims-Aw
  using subset-eq by fastforce
    then have nth (dims Aw) (Collect even) ! k = last rs
      using ⟨digit-encode (nth (dims Aw) (Collect even)) i < nth (dims Aw)
  (Collect even)⟩
      ⟨k < length (digit-encode (nth (dims Aw) (Collect even)) i)⟩
      nth-mem valid-index-length by auto
    then have i0 < last rs
      using valid-index-lt ⟨digit-encode (nth (dims Aw) (Collect even)) i ! k =
  i0⟩
      ⟨digit-encode (nth (dims Aw) (Collect even)) i < nth (dims Aw) (Collect
  even)⟩
      ⟨k < length (digit-encode (nth (dims Aw) (Collect even)) i)⟩ valid-index-length
  by fastforce
    then show i0 < r unfolding r-def by (simp add: ⟨i0 < last (butlast rs)⟩)
  qed
  then show i ∈ ?B using ⟨i < prod-list (nth (dims Aw) (Collect even))⟩ by

```

```

blast
next
  fix i assume i ∈ ?B
  then show i ∈ ?A by (simp add: dims-Aw' r-def rows-with-1-def)
qed
have 2:  $\bigwedge d. d \in \text{set } (\text{nths } (\text{Tensor.dims } Aw) (\text{Collect even})) \implies r \leq d$ 
proof -
  fix d assume d ∈ set (nths (Tensor.dims Aw) (Collect even))
  then have d ∈ set (Tensor.dims Aw) using in-set-nthsD by fast
  then have d = last rs using dims-Aw by simp
  then show r ≤ d by (simp add: r-def)
qed
have 3: 0 < r unfolding r-def by (metis deep diff-diff-cancel diff-zero dual-order.trans
in-set-butlastD last-in-set length-butlast list.size(3) min-def nat-le-linear no-zeros
not-numeral-le-zero numeral-le-one-iff rel-simps(3))
  have 4: length (nths (Tensor.dims Aw) (Collect even)) = N-half
  unfolding length-nths order-Aw using card-even[of N-half]
  by (metis (mono-tags, lifting) Collect-cong)
  then show ?thesis using card-low-digits[of r nths (Tensor.dims Aw) (Collect
even)] 1 2 3 4 by metis
qed

```

lemma infinite-rows-with-1: infinite rows-with-1

```

proof -
  define listpr where listpr = prod-list (nths (Tensor.dims Aw) {n. even n})
  have  $\bigwedge i. \text{listpr } \text{dvd } i \implies i \in \text{rows-with-1}$ 
  proof -
    fix i assume dvd-i: listpr dvd i
    {
      fix i0::nat
      assume i0 ∈ set (digit-encode (nths (Tensor.dims Aw) {n. even n}) i)
      then have i0=0 using digit-encode-0 dvd-i listpr-def by auto
      then have i0 < last (butlast rs) using deep no-zeros
      by (metis Nitpick.size-list-simp(2) One-nat-def Suc-le-lessD in-set-butlastD
last-in-set length-butlast length-tl not-numeral-less-zero numeral-2-eq-2 numeral-3-eq-3
numeral-le-one-iff semiring-norm(70))
    }
    then show i ∈ rows-with-1 by (simp add: rows-with-1-def)
  qed
  have 0: Tensor.dims Aw = replicate (2 ^ (length rs - 2)) (last rs) unfolding
Aw-def
  using dims-output-witness[OF - no-zeros y-valid] using deep by linarith
  then have listpr > 0 unfolding listpr-def 0
  by (metis 0 deep last-in-set length-greater-0-conv less-le-trans no-zeros dims-Aw'-pow(1)
dims-Aw'(1)
zero-less-numeral zero-less-power)
  then have inj ((* listpr) by (metis injI mult-left-cancel neq0-conv)
  then show ?thesis using  $\langle \bigwedge i. \text{listpr } \text{dvd } i \implies i \in \text{rows-with-1} \rangle$ 

```

by (meson dvd-triv-left image-subset-iff infinite-iff-countable-subset)
qed

lemma witness-submatrix: submatrix Aw' rows-with-1 rows-with-1 = $1_m (r \hat{=} N\text{-half})$

proof

show dim-row (submatrix Aw' rows-with-1 rows-with-1) = dim-row ($1_m (r \hat{=} N\text{-half})$)

unfolding index-one-mat(2) dim-submatrix(1)

by (metis (full-types) set-le-in card-rows-with-1)

show dim-col (submatrix Aw' rows-with-1 rows-with-1) = dim-col ($1_m (r \hat{=} N\text{-half})$)

by (metis ⟨dim-row (submatrix Aw' rows-with-1 rows-with-1) = dim-row ($1_m (r \hat{=} N\text{-half})$)⟩ dim-submatrix(1) dim-submatrix(2) index-one-mat(2) index-one-mat(3) dims-Aw'-pow)

show $\bigwedge i j. i < \text{dim-row } (1_m (r \hat{=} N\text{-half})) \implies$

$j < \text{dim-col } (1_m (r \hat{=} N\text{-half})) \implies \text{submatrix Aw' rows-with-1 rows-with-1}$

$\$ \$ (i, j) = 1_m (r \hat{=} N\text{-half}) \$ \$ (i, j)$

proof –

fix i j assume $i < \text{dim-row } (1_m (r \hat{=} N\text{-half}))$ $j < \text{dim-col } (1_m (r \hat{=} N\text{-half}))$

then have $i < r \hat{=} N\text{-half}$ $j < r \hat{=} N\text{-half}$ by auto

then have $i < \text{card } \{i \in \text{rows-with-1}. i < \text{dim-row Aw}'\}$ $j < \text{card } \{i \in \text{rows-with-1}. i < \text{dim-col Aw}'\}$

using card-rows-with-1 dims-Aw'-pow by auto

then have pick rows-with-1 $i < \text{dim-row Aw}'$ pick rows-with-1 $j < \text{dim-col Aw}'$

using card-le-pick-inf[OF infinite-rows-with-1, of dim-row Aw' i]

using card-le-pick-inf[OF infinite-rows-with-1, of dim-col Aw' j] by force+

have $\forall i0 \in \text{set } (\text{digit-encode } (nths (\text{dims Aw}) (\text{Collect even})) (\text{pick rows-with-1 } i)). i0 < \text{last } (\text{butlast } rs)$

using infinite-rows-with-1 pick-in-set-inf rows-with-1-def by auto

then have Aw' $\$ \$ (\text{pick rows-with-1 } i, \text{pick rows-with-1 } j) = (\text{if pick rows-with-1 } i = \text{pick rows-with-1 } j \text{ then } 1 \text{ else } 0)$

using witness-matricization[OF ⟨pick rows-with-1 $i < \text{dim-row Aw}'$ ⟩ ⟨pick rows-with-1 $j < \text{dim-col Aw}'$ ⟩] by simp

then have submatrix Aw' rows-with-1 rows-with-1 $\$ \$ (i, j) = (\text{if pick rows-with-1 } i = \text{pick rows-with-1 } j \text{ then } 1 \text{ else } 0)$

using submatrix-index by (metis (no-types, lifting)

⟨dim-col (submatrix Aw' rows-with-1 rows-with-1) = dim-col ($1_m (r \hat{=} N\text{-half})$)⟩

⟨dim-row (submatrix Aw' rows-with-1 rows-with-1) = dim-row ($1_m (r \hat{=} N\text{-half})$)⟩

⟨ $i < \text{dim-row } (1_m (r \hat{=} N\text{-half}))$ ⟩ ⟨ $j < r \hat{=} N\text{-half}$ ⟩ dim-submatrix(1) dim-submatrix(2) index-one-mat(3))

then have submatrix Aw' rows-with-1 rows-with-1 $\$ \$ (i, j) = (\text{if } i = j \text{ then } 1 \text{ else } 0)$

using pick-eq-iff-inf[OF infinite-rows-with-1] by auto

then show submatrix Aw' rows-with-1 rows-with-1 $\$ \$ (i, j) = 1_m (r \hat{=} N\text{-half})$

$\$ \$ (i, j)$

by (simp add: ⟨ $i < r \hat{=} N\text{-half}$ ⟩ ⟨ $j < r \hat{=} N\text{-half}$ ⟩)

qed

qed

lemma *witness-det*: \det (submatrix Aw' rows-with-1 rows-with-1) $\neq 0$ **unfolding**
witness-submatrix **by** *simp*

end

interpretation *example* : *deep-model-correct-params* *False* [10,10,10]
unfolding *deep-model-correct-params-def* **by** *simp*

interpretation *example* : *deep-model-correct-params-y* *False* [10,10,10] 1
unfolding *deep-model-correct-params-y-def* *deep-model-correct-params-y-axioms-def*

deep-model-correct-params-def **by** *simp*

end

15 Polynomials representing the Deep Network Model

theory *DL-Deep-Model-Poly*

imports *DL-Deep-Model* *Polynomials*.*More-MPoly-Type* *Jordan-Normal-Form*.*Determinant*
begin

lemma *polyfun-det*:

assumes $\bigwedge x. (A\ x) \in \text{carrier-mat } n\ n$

assumes $\bigwedge x\ i\ j. i < n \implies j < n \implies \text{polyfun } N\ (\lambda x. (A\ x)\ \S\S\ (i,j))$

shows $\text{polyfun } N\ (\lambda x. \det (A\ x))$

proof –

{
 fix p **assume** $p \in \{p. p \text{ permutes } \{0..<n\}\}$
 then have $p \text{ permutes } \{0..<n\}$ **by** *auto*
 then have $\bigwedge x. x < n \implies p\ x < n$ **using** *permutes-in-image* **by** *auto*
 then have $\text{polyfun } N\ (\lambda x. \prod i = 0..<n. A\ x\ \S\S\ (i, p\ i))$
 using *polyfun-Prod*[of $\{0..<n\}$ $N\ \lambda i\ x. A\ x\ \S\S\ (i, p\ i)$] *assms* **by** *simp*
 then have $\text{polyfun } N\ (\lambda x. \text{signof } p * (\prod i = 0..<n. A\ x\ \S\S\ (i, p\ i)))$ **using**
polyfun-const *polyfun-mult* **by** *blast*
}

moreover have *finite* $\{i. i \text{ permutes } \{0..<n\}\}$ **by** (*simp add: finite-permutations*)
ultimately show *?thesis* **unfolding** *det-def'*[OF *assms*(1)]
using *polyfun-Sum*[OF $\langle \text{finite } \{i. i \text{ permutes } \{0..<n\}\} \rangle$, of $N\ \lambda p\ x. \text{signof } p * (\prod i = 0..<n. A\ x\ \S\S\ (i, p\ i))$]
by *blast*

qed

lemma *polyfun-extract-matrix*:

assumes $i < m\ j < n$

shows $\text{polyfun } \{..<a + (m * n + c)\} (\lambda f. \text{extract-matrix } (\lambda i. f\ (i + a))\ m\ n\ \S\S\ (i,j))$

unfolding *index-extract-matrix*[*OF* *assms*] **apply** (*rule polyfun-single*) **using** *two-digit-le*[*OF* *assms*] **by** *simp*

lemma *polyfun-mult-mat-vec*:

assumes $\bigwedge x. v\ x \in \text{carrier-vec}\ n$

assumes $\bigwedge j. j < n \implies \text{polyfun}\ N\ (\lambda x. v\ x\ \$\ j)$

assumes $\bigwedge x. A\ x \in \text{carrier-mat}\ m\ n$

assumes $\bigwedge i\ j. i < m \implies j < n \implies \text{polyfun}\ N\ (\lambda x. A\ x\ \$\$ (i,j))$

assumes $j < m$

shows $\text{polyfun}\ N\ (\lambda x. ((A\ x) *_v (v\ x))\ \$\ j)$

proof –

have $\bigwedge x. j < \text{dim-row}\ (A\ x)$ **using** $\langle j < m \rangle$ *assms*(3) *carrier-matD*(1) **by** *force*

have $\bigwedge x. n = \text{dim-vec}\ (v\ x)$ **using** *assms*(1) *carrier-vecD* **by** *fastforce*

{
 fix *i* **assume** $i \in \{0..<n\}$
 then have $i < n$ **by** *auto*

{
 fix *x*
 have $i < \text{dim-vec}\ (v\ x)$ **using** *assms*(1) *carrier-vecD* $\langle i < n \rangle$ **by** *fastforce*
 have $j < \text{dim-row}\ (A\ x)$ **using** $\langle j < m \rangle$ *assms*(3) *carrier-matD*(1) **by** *force*
 have $\text{dim-col}\ (A\ x) = \text{dim-vec}\ (v\ x)$ **by** (*metis* *assms*(1) *assms*(3) *carrier-matD*(2) *carrier-vecD*)

then have $\text{row}\ (A\ x)\ j\ \$\ i = A\ x\ \$\$ (j,i)$ $i < n$ **using** $\langle j < \text{dim-row}\ (A\ x) \rangle$ $\langle i < n \rangle$ **by** (*simp-all* *add*: $\langle i < \text{dim-vec}\ (v\ x) \rangle$)

then have $\text{polyfun}\ N\ (\lambda x. \text{row}\ (A\ x)\ j\ \$\ i * v\ x\ \$\ i)$

using *polyfun-mult* *assms*(4)[*OF* $\langle j < m \rangle$] *assms*(2) **by** *fastforce*

then show *?thesis* **unfolding** *index-mult-mat-vec*[*OF* $\langle \bigwedge x. j < \text{dim-row}\ (A\ x) \rangle$] *scalar-prod-def*

using *polyfun-Sum*[*of* $\{0..<n\}$ $N\ \lambda i\ x. \text{row}\ (A\ x)\ j\ \$\ i * v\ x\ \$\ i$] *finite-atLeastLessThan*[*of* $0\ n$] $\langle \bigwedge x. n = \text{dim-vec}\ (v\ x) \rangle$

by *simp*

qed

lemma *polyfun-evaluate-net-plus-a*:

assumes *map dim-vec inputs = input-sizes* *m*

assumes *valid-net* *m*

assumes $j < \text{output-size}\ m$

shows $\text{polyfun}\ \{..<a + \text{count-weights}\ s\ m\}\ (\lambda f. \text{evaluate-net}\ (\text{insert-weights}\ s\ m\ (\lambda i. f\ (i + a))))\ \text{inputs}\ \$\ j$

using *assms* **proof** (*induction* *m* *arbitrary:inputs* *j* *a*)

case (*Input*)

then show *?case* **unfolding** *insert-weights.simps* *evaluate-net.simps* **using** *polyfun-const* **by** *metis*

next

case (*Conv* *x* *m*)

then obtain *x1* *x2* **where** $x = (x1, x2)$ **by** *fastforce*


```

show ?case unfolding ⟨ $x=(x1,x2)$ ⟩ insert-weights.simps evaluate-net.simps drop-map
unfolding list-of-vec-index
proof (rule polyfun-mult-mat-vec)
  {
    fix f
    have 1:valid-net' (insert-weights s m ( $\lambda i. f (i + x1 * x2)$ ))
      using ⟨valid-net (Conv x m)⟩ valid-net.simps by (metis
        convnet.distinct(1) convnet.distinct(5) convnet.inject(2) remove-insert-weights)
      have 2:map dim-vec inputs = input-sizes (insert-weights s m ( $\lambda i. f (i + x1$ 
        *  $x2)$ ))
      using input-sizes-remove-weights remove-insert-weights
      by (simp add: Conv.prem(1))
      have dim-vec (evaluate-net (insert-weights s m ( $\lambda i. f (i + x1 * x2)$ )) inputs)
    = output-size m
      using output-size-correct[OF 1 2] using remove-insert-weights by auto
      then show evaluate-net (insert-weights s m ( $\lambda i. f (i + x1 * x2)$ )) inputs ∈
    carrier-vec (output-size m)
      using carrier-vec-def by (metis (full-types) mem-Collect-eq)
    }

    have map dim-vec inputs = input-sizes m by (simp add: Conv.prem(1))
    have valid-net m using Conv.prem(2) valid-net.cases by fastforce
    show  $\bigwedge j. j < \text{output-size } m \implies \text{polyfun } \{..<a + \text{count-weights } s (\text{Conv } (x1,$ 
     $x2) m)\}$ 
      ( $\lambda f. \text{evaluate-net} (\text{insert-weights } s m (\lambda i. f (i + x1 * x2 + a))) \text{inputs } \$ j$ )
      unfolding vec-of-list-index count-weights.simps
      using Conv(1)[OF <map dim-vec inputs = input-sizes m> <valid-net m>, of -
     $x1 * x2 + a]$ 
      unfolding semigroup-add-class.add.assoc ab-semigroup-add-class.add.commute[of
     $x1 * x2 a]$ 
      by blast

    have output-size m = x2 using Conv.prem(2) <x = (x1, x2)> valid-net.cases
by fastforce
    show  $\bigwedge f. \text{extract-matrix} (\lambda i. f (i + a)) x1 x2 \in \text{carrier-mat } x1 (\text{output-size}$ 
     $m)$  unfolding ⟨output-size m = x2⟩ using dim-extract-matrix
      using carrier-matI by (metis (no-types, lifting))

    show  $\bigwedge i j. i < x1 \implies j < \text{output-size } m \implies \text{polyfun } \{..<a + \text{count-weights } s$ 
    ( $\text{Conv } (x1, x2) m)\}$  ( $\lambda f. \text{extract-matrix} (\lambda i. f (i + a)) x1 x2 \$ \$ (i, j)$ )
      unfolding ⟨output-size m = x2⟩ count-weights.simps using polyfun-extract-matrix[of
     $- x1 - x2 a \text{count-weights } s m]$  by blast

    show  $j < x1$  using Conv.prem(3) <x = (x1, x2)> by auto
    qed
next
    case (Pool m1 m2 inputs j a)
    have A2: $\bigwedge f. \text{map dim-vec} (\text{take } (\text{length } (\text{input-sizes } (\text{insert-weights } s m1 (\lambda i. f$ 
    ( $i + a)))) \text{inputs}) = \text{input-sizes } m1$ 

```

```

  by (metis Pool.premis(1) append-eq-conv-conj input-sizes.simps(3) input-sizes-remove-weights
remove-insert-weights take-map)
  have B2:  $\bigwedge f$ . map dim-vec (drop (length (input-sizes (insert-weights s m1 ( $\lambda i$ . f
(i + a)))))) inputs) = input-sizes m2
  using Pool.premis(1) append-eq-conv-conj input-sizes.simps(3) input-sizes-remove-weights
remove-insert-weights by (metis drop-map)
  have A3: valid-net m1 and B3: valid-net m2 using  $\langle$ valid-net (Pool m1 m2) $\rangle$ 
valid-net.simps by blast+
  have output-size (Pool m1 m2) = output-size m2 unfolding output-size.simps
  using  $\langle$ valid-net (Pool m1 m2) $\rangle$  valid-net.cases by fastforce
  then have A4:  $j < \text{output-size } m1$  and B4:  $j < \text{output-size } m2$  using  $\langle j <
\text{output-size } (Pool m1 m2) \rangle$  by simp-all

let ?net1 =  $\lambda f$ . evaluate-net (insert-weights s m1 ( $\lambda i$ . f (i + a)))
  (take (length (input-sizes (insert-weights s m1 ( $\lambda i$ . f (i + a)))))) inputs)
let ?net2 =  $\lambda f$ . evaluate-net (insert-weights s m2 (if s then  $\lambda i$ . f (i + a) else
( $\lambda i$ . f (i + count-weights s m1 + a))))
  (drop (length (input-sizes (insert-weights s m1 ( $\lambda i$ . f (i + a)))))) inputs)
have length1:  $\bigwedge f$ . output-size m1 = dim-vec (?net1 f)
  by (metis A2 A3 input-sizes-remove-weights output-size-correct remove-insert-weights)
then have jlength1:  $\bigwedge f$ .  $j < \text{dim-vec } (?net1 f)$  using A4 by metis
have length2:  $\bigwedge f$ . output-size m2 = dim-vec (?net2 f)
  by (metis B2 B3 input-sizes-remove-weights output-size-correct remove-insert-weights)
then have jlength2:  $\bigwedge f$ .  $j < \text{dim-vec } (?net2 f)$  using B4 by metis
have cong1:  $\bigwedge x f$ . ( $\lambda f$ . evaluate-net (insert-weights s m1 ( $\lambda i$ . f (i + a)))
  (take (length (input-sizes (insert-weights s m1 ( $\lambda i$ . x f (i + a)))))) inputs) $
j)
  = ( $\lambda f$ . ?net1 f $ j)
  using input-sizes-remove-weights remove-insert-weights by auto
have cong2:  $\bigwedge x f$ . ( $\lambda f$ . evaluate-net (insert-weights s m2 ( $\lambda i$ . f (i + (a + (if s
then 0 else count-weights s m1))))))
  (drop (length (input-sizes (insert-weights s m1 ( $\lambda i$ . x f (i + a)))))) inputs) $
j)
  = ( $\lambda f$ . ?net2 f $ j)
unfolding semigroup-add-class.add.assoc[symmetric] ab-semigroup-add-class.add.commute[of
a if s then 0 else count-weights s m1]
  using input-sizes-remove-weights remove-insert-weights by auto

show ?case unfolding insert-weights.simps evaluate-net.simps count-weights.simps
unfolding index-component-mult[OF jlength1 jlength2]
  apply (rule polyfun-mult)
  using Pool.IH(1)[OF A2 A3 A4, of a, unfolded cong1]
  apply (simp add: polyfun-subset[of  $\{.. < a + \text{count-weights } s m1\}$   $\{.. < a + (if
s \text{ then } \text{max} (\text{count-weights } s m1) (\text{count-weights } s m2) \text{ else } \text{count-weights } s m1 +
\text{count-weights } s m2)\}$ ])
  using Pool.IH(2)[OF B2 B3 B4, of a + (if s then 0 else count-weights s m1),
unfolded cong2 semigroup-add-class.add.assoc[of a]]
  by (simp add: polyfun-subset[of  $\{.. < a + ((if s \text{ then } 0 \text{ else } \text{count-weights } s m1) +
\text{count-weights } s m2)\}$   $\{.. < a + (if s \text{ then } \text{max} (\text{count-weights } s m1) (\text{count-weights }
s m2)\}$ ])

```

$s\ m2)$ else count-weights $s\ m1 + \text{count-weights } s\ m2\}}\})$
qed

lemma *polyfun-evaluate-net*:
assumes $\text{map dim-vec inputs} = \text{input-sizes } m$
assumes *valid-net* m
assumes $j < \text{output-size } m$
shows $\text{polyfun } \{..<\text{count-weights } s\ m\} (\lambda f. \text{evaluate-net } (\text{insert-weights } s\ m\ f)\ \text{inputs } \$ j)$
using *polyfun-evaluate-net-plus-a*[**where** $a=0$, *OF assms*] **by** *simp*

lemma *polyfun-tensors-from-net*:
assumes *valid-net* m
assumes $is \triangleleft \text{input-sizes } m$
assumes $j < \text{output-size } m$
shows $\text{polyfun } \{..<\text{count-weights } s\ m\} (\lambda f. \text{Tensor.lookup } (\text{tensors-from-net } (\text{insert-weights } s\ m\ f)\ \$ j)\ \text{is})$
proof –
have $1:\bigwedge f. \text{valid-net}' (\text{insert-weights } s\ m\ f)$ **by** (*simp add: assms(1) remove-insert-weights*)
have $\text{input-sizes}:\bigwedge f. \text{input-sizes } (\text{insert-weights } s\ m\ f) = \text{input-sizes } m$
unfolding *input-sizes-remove-weights* **by** (*simp add: remove-insert-weights*)
have $2:\bigwedge f. is \triangleleft \text{input-sizes } (\text{insert-weights } s\ m\ f)$
unfolding *input-sizes* **using** *assms(2)* **by** *blast*
have $3:\bigwedge f. j < \text{output-size}' (\text{insert-weights } s\ m\ f)$
by (*simp add: assms(3) remove-insert-weights*)
have $\bigwedge f1\ f2. \text{base-input } (\text{insert-weights } s\ m\ f1)\ \text{is} = \text{base-input } (\text{insert-weights } s\ m\ f2)\ \text{is}$
unfolding *base-input-def* **by** (*simp add: input-sizes*)
then have $\bigwedge x f. (\lambda f. \text{evaluate-net } (\text{insert-weights } s\ m\ f)\ (\text{base-input } (\text{insert-weights } s\ m\ x f)\ \text{is})\ \$ j)$
 $= (\lambda f. \text{evaluate-net } (\text{insert-weights } s\ m\ f)\ (\text{base-input } (\text{insert-weights } s\ m\ f)\ \text{is})\ \$ j)$
by *metis*
then show *?thesis* **unfolding** *lookup-tensors-from-net*[*OF 1 2 3*]
using *polyfun-evaluate-net*[*OF base-input-length*][*OF 2, unfolded input-sizes, symmetric*] *assms(1) assms(3), of s*
by *simp*
qed

lemma *polyfun-matricize*:
assumes $\bigwedge x. \text{dims } (T\ x) = ds$
assumes $\bigwedge is. is \triangleleft ds \implies \text{polyfun } N (\lambda x. \text{Tensor.lookup } (T\ x)\ \text{is})$
assumes $\bigwedge x. \text{dim-row } (\text{matricize } I\ (T\ x)) = nr$
assumes $\bigwedge x. \text{dim-col } (\text{matricize } I\ (T\ x)) = nc$
assumes $i < nr$
assumes $j < nc$
shows $\text{polyfun } N (\lambda x. \text{matricize } I\ (T\ x)\ \$\$ (i,j))$
proof –
let *?weave* = $\lambda x. (\text{weave } I$

```

    (digit-encode (nths ds I ) i)
    (digit-encode (nths ds (-I )) j))
  have 1:  $\bigwedge x. \text{matricize } I (T x) \text{ \}\} (i,j) = \text{Tensor.lookup } (T x) (?weave x) \text{ unfolding } \text{matricize-def}$ 
  by (metis (no-types, lifting) assms(1) assms(3) assms(4) assms(5) assms(6)
case-prod-conv
dim-col-mat(1) dim-row-mat(1) index-mat(1) matricize-def)
  have  $\bigwedge x. ?weave x \triangleleft ds$ 
  using valid-index-weave(1) assms(2) digit-encode-valid-index dim-row-mat(1)
matricize-def
  using assms digit-encode-valid-index matricize-def by (metis dim-col-mat(1))
  then have polyfun N ( $\lambda x. \text{Tensor.lookup } (T x) (?weave x)$ ) using assms(2) by
simp
  then show ?thesis unfolding 1 using assms(1) by blast
qed

```

```

lemma ( $\neg (a::nat) < b$ ) = ( $a \geq b$ )
by (metis not-le)

```

```

lemma polyfun-submatrix:
assumes  $\bigwedge x. (A x) \in \text{carrier-mat } m \ n$ 
assumes  $\bigwedge x \ i \ j. i < m \implies j < n \implies \text{polyfun } N (\lambda x. (A x) \text{ \}\} (i,j))$ 
assumes  $i < \text{card } \{i. i < m \wedge i \in I\}$ 
assumes  $j < \text{card } \{j. j < n \wedge j \in J\}$ 
assumes infinite I infinite J
shows polyfun N ( $\lambda x. (\text{submatrix } (A x) I J) \text{ \}\} (i,j)$ )
proof -
  have 1:  $\bigwedge x. (\text{submatrix } (A x) I J) \text{ \}\} (i,j) = (A x) \text{ \}\} (\text{pick } I \ i, \text{pick } J \ j)$ 
  using submatrix-index by (metis (no-types, lifting) Collect-cong assms(1)
assms(3) assms(4) carrier-matD(1) carrier-matD(2))
  have pick I i < m pick J j < n using card-le-pick-inf[OF  $\langle \text{infinite } I \rangle$ ] card-le-pick-inf[OF
 $\langle \text{infinite } J \rangle$ ]
  <math>\langle i < \text{card } \{i. i < m \wedge i \in I\} \rangle [\text{unfolded set-le-in}] \langle j < \text{card } \{j. j < n \wedge j \in J\} \rangle [\text{unfolded set-le-in}] \text{ not-less}</math> by metis+
  then show ?thesis unfolding 1 by (simp add: assms(2))
qed

```

```

context deep-model-correct-params-y
begin

```

```

definition witness-submatrix where
witness-submatrix f = submatrix (A' f) rows-with-1 rows-with-1

```

```

lemma polyfun-tensor-deep-model:
assumes  $is \triangleleft \text{input-sizes } (\text{deep-model-l } rs)$ 
shows polyfun  $\{.. < \text{weight-space-dim}\}$ 
( $\lambda f. \text{Tensor.lookup } (\text{tensors-from-net } (\text{insert-weights shared-weights } (\text{deep-model-l } rs) f) \text{ \}\} y) \text{ is}$ )

```

proof –
have $1:\bigwedge f. \text{remove-weights } (\text{insert-weights shared-weights } (\text{deep-model-l } rs) f) = \text{deep-model-l } rs$
using *remove-insert-weights* **by** *metis*
then have $y < \text{output-size } (\text{deep-model-l } rs)$ **using** *valid-deep-model y-valid length-output-deep-model* **by** *force*
have $0:\{..<\text{weight-space-dim}\} = \text{set } [0..<\text{weight-space-dim}]$ **by** *auto*
then show *?thesis unfolding weight-space-dim-def using polyfun-tensors-from-net assms(1) valid-deep-model*
 $\langle y < \text{output-size } (\text{deep-model-l } rs) \rangle$ **by** *metis*
qed

lemma *input-sizes-deep-model: input-sizes (deep-model-l rs) = replicate (2 * N-half) (last rs)*
unfolding *N-half-def* **using** *input-sizes-deep-model deep*
by (*metis (no-types, lifting) Nitpick.size-list-simp(2) One-nat-def Suc-1 Suc-le-lessD diff-Suc-Suc length-tl less-imp-le-nat list.size(3) not-less-eq numeral-3-eq-3 power-eq-if*)

lemma *polyfun-matrix-deep-model:*

assumes $i < (\text{last } rs) \wedge N\text{-half}$

assumes $j < (\text{last } rs) \wedge N\text{-half}$

shows *polyfun* $\{..<\text{weight-space-dim}\} (\lambda f. A' f \text{\$\$ } (i,j))$

proof –

have $0:y < \text{output-size } (\text{deep-model-l } rs)$ **using** *valid-deep-model y-valid length-output-deep-model* **by** *force*

have $1:\bigwedge f. \text{remove-weights } (\text{insert-weights shared-weights } (\text{deep-model-l } rs) f) = \text{deep-model-l } rs$

using *remove-insert-weights* **by** *metis*

have $2:(\bigwedge is. is \triangleleft \text{replicate } (2 * N\text{-half}) (\text{last } rs) \implies$

$\text{polyfun } \{..<\text{weight-space-dim}\} (\lambda x. \text{Tensor.lookup } (A x) is))$

unfolding *A-def* **using** *polyfun-tensor-deep-model[unfolded input-sizes-deep-model]*

0 **by** *blast*

show *?thesis*

unfolding *A'-def A-def* **apply** (*rule polyfun-matricize*)

using *dims-tensor-deep-model[OF 1] 2[unfolded A-def]*

using *dims-A'-pow[unfolded A'-def A-def] <i < (last rs) ^ N-half> <j < (last rs) ^*

N-half>

by *auto*

qed

lemma *polyfun-submatrix-deep-model:*

assumes $i < r \wedge N\text{-half}$

assumes $j < r \wedge N\text{-half}$

shows *polyfun* $\{..<\text{weight-space-dim}\} (\lambda f. \text{witness-submatrix } f \text{\$\$ } (i,j))$

unfolding *witness-submatrix-def*

proof (*rule polyfun-submatrix*)

have $1:\bigwedge f. \text{remove-weights } (\text{insert-weights shared-weights } (\text{deep-model-l } rs) f) = \text{deep-model-l } rs$

using *remove-insert-weights* **by** *metis*

```

show  $\bigwedge f. A' f \in \text{carrier-mat } ((\text{last } rs) \wedge N\text{-half}) ((\text{last } rs) \wedge N\text{-half})$ 
  using 1 dims-A'-pow using weight-space-dim-def by auto
show  $\bigwedge f i j. i < \text{last } rs \wedge N\text{-half} \implies j < \text{last } rs \wedge N\text{-half} \implies$ 
  polyfun  $\{..<\text{weight-space-dim}\} (\lambda f. A' f \text{ \&\amp; } (i, j))$ 
  using polyfun-matrix-deep-model weight-space-dim-def by force
show  $i < \text{card } \{i. i < \text{last } rs \wedge N\text{-half} \wedge i \in \text{rows-with-1}\}$ 
  using assms(1) card-rows-with-1 dims-Aw'-pow set-le-in by metis
show  $j < \text{card } \{i. i < \text{last } rs \wedge N\text{-half} \wedge i \in \text{rows-with-1}\}$ 
  using assms(2) card-rows-with-1 dims-Aw'-pow set-le-in by metis
show infinite rows-with-1 infinite rows-with-1 by (simp-all add: infinite-rows-with-1)
qed

lemma polyfun-det-deep-model:
shows polyfun  $\{..<\text{weight-space-dim}\} (\lambda f. \text{det } (\text{witness-submatrix } f))$ 
proof (rule polyfun-det)
  fix f
  have remove-weights (insert-weights shared-weights (deep-model-l rs) f) = deep-model-l
  rs
  using remove-insert-weights by metis

  show witness-submatrix f  $\in \text{carrier-mat } (r \wedge N\text{-half}) (r \wedge N\text{-half})$ 
  unfolding witness-submatrix-def apply (rule carrier-matI) unfolding dim-submatrix[unfolded
set-le-in]
  unfolding dims-A'-pow[unfolded weight-space-dim-def] using card-rows-with-1
dims-Aw'-pow by simp-all
  show  $\bigwedge i j. i < r \wedge N\text{-half} \implies j < r \wedge N\text{-half} \implies \text{polyfun } \{..<\text{weight-space-dim}\}$ 
   $(\lambda f. \text{witness-submatrix } f \text{ \&\amp; } (i, j))$ 
  using polyfun-submatrix-deep-model by blast
qed

end

end

```

16 Alternative Lebesgue Measure Definition

```

theory Lebesgue-Functional
imports HOL-Analysis.Lebesgue-Measure
begin

```

Lebesgue_Measure.lborel is defined on the typeclass euclidean_space, which does not allow the space dimension to be dependent on a variable. As the Lebesgue measure of higher dimensions is the product measure of the one dimensional Lebesgue measure, we can easily define a more flexible version of the Lebesgue measure as follows. This version of the Lebesgue measure measures sets of functions from nat to real whose values are undefined for arguments higher than n. These "Extensional Function Spaces" are defined in HOL/FuncSet.

definition $lborel-f :: nat \Rightarrow (nat \Rightarrow real)$ *measure* **where**
 $lborel-f\ n = (\Pi_M\ b \in \{..<n\}).\ lborel$

lemma *product-sigma-finite-interval: product-sigma-finite* ($\lambda b.$ *interval-measure* ($\lambda x.$ x))
unfolding *product-sigma-finite-def* **using** *sigma-finite-interval-measure* **by** *auto*

lemma *l-borel-f-1: distr* ($lborel-f\ 1$) *lborel* ($\lambda x.$ $x\ 0$) = *lborel*
unfolding *lborel-f-def*
using *product-sigma-finite.distr-singleton*[*OF product-sigma-finite-interval, of 0*]
lborel-eq-real lessThan-Suc **by** *auto*

lemma *space-lborel-f: space* ($lborel-f\ n$) = $Pi_E\ \{..<n\}$ ($\lambda.$ *UNIV*) **unfolding**
lborel-f-def
unfolding *space-PiM space-lborel space-borel* **by** *metis*

lemma *space-lborel-f-subset: space* ($lborel-f\ n$) \subseteq *space* ($lborel-f\ (Suc\ n)$)
unfolding *space-lborel-f* **by** (*rule subsetI, rule PiE-I, blast,*
metis PiE-E Suc-n-not-le-n le-cases lessThan-subset-iff subsetCE)

lemma *space-lborel-add-dim:*
assumes $f \in$ *space* ($lborel-f\ n$)
shows $f(n:=x) \in$ *space* ($lborel-f\ (Suc\ n)$)
unfolding *space-lborel-f*
using *assms lessThan-Suc space-lborel-f* **by** *auto*

lemma *integral-lborel-f:*
assumes $f \in$ *borel-measurable* ($lborel-f\ (Suc\ n)$)
shows $integral^N$ ($lborel-f\ (Suc\ n)$) $f = \int^+ y. \int^+ x. f\ (x(n := y))\ \partial lborel-f\ n$
 $\partial lborel$
unfolding *lborel-f-def*
using *product-sigma-finite.product-nn-integral-insert-rev*[*OF product-sigma-finite-interval,*
of \{..<n\}, OF finite-lessThan -]
assms[unfolded lborel-f-def] lborel-eq-real **by** (*simp add: lessThan-Suc*)

lemma *emeasure-lborel-f-Suc:*
assumes $A \in$ *sets* ($lborel-f\ (Suc\ n)$)
assumes $\bigwedge y. \{x \in$ *space* ($lborel-f\ n$). $x(n := y) \in A\} \in$ *sets* ($lborel-f\ n$)
shows $emeasure$ ($lborel-f\ (Suc\ n)$) $A = \int^+ y. emeasure$ ($lborel-f\ n$) $\{x \in$ *space*
($lborel-f\ n$). $x(n := y) \in A\}\ \partial lborel$
proof –
{
 fix $x\ y$ **assume** $x \in$ *space* ($lborel-f\ n$)
 then have (*indicator* A) ($x(n := y)$) = (*indicator* $\{x \in$ *space* ($lborel-f\ n$). $x(n$
:= y) $\in A\}$) x
 by (*simp add: indicator-def*)
}
then show *?thesis*
unfolding *nn-integral-indicator*[*OF assms(1), symmetric*] *nn-integral-indicator*[*OF*

```

assms(2), symmetric]
  integral-lborel-f[OF borel-measurable-indicator, OF assms(1)]
  using nn-integral-cong by (metis (no-types, lifting))
qed

lemma lborel-f-measurable-add-dim:  $(\lambda f. f(n := x)) \in \text{measurable } (\text{lborel-f } n) (\text{lborel-f } (\text{Suc } n))$ 
proof -
  have  $x \in \text{space } \text{lborel}$  by simp
  have  $0: (\lambda(f, y). f(n := y)) \circ (\lambda xa. (xa, x)) = (\lambda f. f(n := x))$  unfolding comp-def
using case-prod-conv by fast
  show ?thesis unfolding lborel-f-def
    using measurable-comp[OF measurable-Pair2'[of  $x \text{ lborel } \text{Pi}_M \{..<n\}$ ] ( $\lambda b. \text{lborel}$ ), OF  $\langle x \in \text{space } \text{lborel} \rangle$ ]
    measurable-add-dim[of  $n \{..<n\}$ ]  $\lambda b. \text{lborel}$ ], unfolded 0] lessThan-Suc by auto
qed

lemma sets-lborel-f-sub-dim:
assumes  $A \in \text{sets } (\text{lborel-f } (\text{Suc } n))$ 
shows  $\{x. x(n := y) \in A\} \cap \text{space } (\text{lborel-f } n) \in \text{sets } (\text{lborel-f } n)$ 
proof -
  have  $(\lambda f. f(n := y)) -' A \cap \text{space } (\text{lborel-f } n) \in \text{sets } (\text{lborel-f } n)$ 
    using measurable-sets[OF lborel-f-measurable-add-dim assms] by auto
  moreover have  $(\lambda f. f(n := y)) -' A = \{x. x(n := y) \in A\}$  by auto
  finally show ?thesis by metis
qed

lemma lborel-f-measurable-restrict:
assumes  $m \leq n$ 
shows  $(\lambda f. \text{restrict } f \{..<m\}) \in \text{measurable } (\text{lborel-f } n) (\text{lborel-f } m)$ 
using measurable-restrict-subset lborel-f-def assms by auto

lemma lborel-measurable-sub-dim:  $(\lambda f. \text{restrict } f \{..<n\}) \in \text{measurable } (\text{lborel-f } (\text{Suc } n)) (\text{lborel-f } n)$ 
using lborel-f-measurable-restrict[of  $n \text{ Suc } n$ ] by linarith

lemma measurable-lborel-component [measurable]:
assumes  $k < n$ 
shows  $(\lambda x. x k) \in \text{borel-measurable } (\text{lborel-f } n)$ 
  unfolding lborel-f-def using assms measurable-PiM-component-rev by simp-all
end

```

17 Lebesgue Measure of Polynomial Zero Sets

```

theory Lebesgue-Zero-Set
imports
  Polynomials.More-MPoly-Type
  Lebesgue-Functional

```



```

    Polynomials.MPoly-Type-Univariate
begin

lemma measurable-insertion [measurable]:
assumes vars p  $\subseteq$  {..n}
shows ( $\lambda f$ . insertion f p)  $\in$  borel-measurable (lborel-f n)
using assms proof (induction p rule:mpoly-induct)
  case (monom m a)
  then show ?case
  proof (cases a = 0)
    case True
    show ?thesis unfolding insertion-single  $\langle a = 0 \rangle$  MPoly-Type.monom.abs-eq
single-zero
      zero-mpoly.abs-eq[symmetric] insertion-zero by measurable
    next
    case False
    have Poly-Mapping.keys m  $\subseteq$  {..n} using monom by (simp add: False
vars-monom-keys)
    then show ?thesis using  $\langle a \neq 0 \rangle$ 
    proof (induction m arbitrary:a rule:poly-mapping-induct)
      case (single x i a)
      then show ?case
      proof (cases i = 0)
        case True
        show ?thesis unfolding insertion-single  $\langle i = 0 \rangle$  by simp
      next
        case False
        then show ?thesis unfolding insertion-single apply measurable
        using vars-monom-single-cases single False insert-subset lessThan-iff  $\langle a \neq 0 \rangle$ 
by fastforce
      qed
    next
    case (sum m1 m2 x i)
    then have Poly-Mapping.keys m1  $\cap$  Poly-Mapping.keys m2 = {} by simp
    then have Poly-Mapping.keys m1  $\cup$  Poly-Mapping.keys m2 = Poly-Mapping.keys
(m1 + m2) using keys-add by metis
    then have 1:Poly-Mapping.keys m1  $\subseteq$  {..n} and 2:Poly-Mapping.keys m2
 $\subseteq$  {..n} using sum.premis by auto
    show ?case unfolding MPoly-Type.mult-monom[of m1 a m2 1,simplified,symmetric]
insertion-mult using sum.IH(1)[OF 1  $\langle a \neq 0 \rangle$ ] sum.IH(2)[OF 2, of 1,
simplified] by measurable
      qed
    qed
  next
  case (sum p1 p2 m a)
  then have ( $\lambda f$ . insertion f p1)  $\in$  borel-measurable (lborel-f n)
    ( $\lambda f$ . insertion f p2)  $\in$  borel-measurable (lborel-f n)
    using vars-add-monom[OF sum.hyps] le-sup-iff by blast+
  then show ?case unfolding insertion-add by measurable

```

qed

This proof follows Richard Caron and Tim Traynor, "The zero set of a polynomial" <http://www1.uwindsor.ca/math/sites/uwindsor.ca.math/files/05-03.pdf>

```

lemma lebesgue-mpoly-zero-set:
fixes p::real mpoly
assumes p ≠ 0 vars p ⊆ {.. $n$ }
shows {f∈space (lborel-f n). insertion f p = 0} ∈ null-sets (lborel-f n)
using assms proof (induction n arbitrary:p)
  case 0
    then have vars p = {} by simp then have  $\bigwedge f$ . insertion f p = MPoly-Type.coeff
    p 0
      unfolding insertion-trivial[symmetric] using insertion-irrelevant-vars by
    blast
    have  $\bigwedge m$ . m≠0  $\implies$  MPoly-Type.coeff p m = 0
    proof (rule ccontr)
      fix m::nat  $\Rightarrow_0$  nat assume m≠0 MPoly-Type.coeff p m ≠ 0
      then obtain v where Poly-Mapping.lookup m v ≠ 0 using aux by auto
      then have v∈vars p unfolding More-MPoly-Type.vars-def using  $\langle$ MPoly-Type.coeff
    p m ≠ 0 $\rangle$ 
        by (meson UN-I coeff-keys lookup-not-eq-zero-eq-in-keys)
      then show False using  $\langle$ vars p = {} $\rangle$  by auto
    qed
    then have MPoly-Type.coeff p 0 ≠ 0 using  $\langle$ p ≠ 0 $\rangle$ 
      by (metis coeff-all-0)
    then have {f. insertion f p = 0} = {} using  $\langle$  $\bigwedge f$ . insertion f p = MPoly-Type.coeff
    p 0 $\rangle$  by auto
    then show ?case by auto
  next
    case (Suc n p)

    Show that N is finite:

    then have extract-var p n ≠ 0 using reduce-nested-mpoly-0
      by (metis reduce-nested-mpoly-extract-var)
    let ?q =  $\lambda j$ . MPoly-Type.coeff (extract-var p n) j
    obtain j where ?q j ≠ 0 using  $\langle$ extract-var p n ≠ 0 $\rangle$ 
      by (metis coeff-all-0)
    then have finite {x. insertion ( $\lambda$ -. x) (?q j) = 0}
      using univariate-mpoly-roots-finite[OF vars-coeff-extract-var] by metis
    then have finite ( $\bigcap j$ . {x. insertion ( $\lambda$ -. x) (?q j) = 0}) by auto
    moreover have {x.  $\forall j$ . insertion ( $\lambda$ -. x) (?q j) = 0} = ( $\bigcap j$ . {x. insertion ( $\lambda$ -.
    x) (?q j) = 0}) by blast
    ultimately have finite {x.  $\forall j$ . insertion ( $\lambda$ -. x) (?q j) = 0} by metis

    define p-fix1 where p-fix1 x1 = replace-coeff (insertion ( $\lambda$ -. x1)) (extract-var p
    n) for x1
    define N where N = {x1. p-fix1 x1 = 0}
    have N ⊆ {x.  $\forall j$ . insertion ( $\lambda$ -. x) (?q j) = 0}

```

```

proof
  fix  $x$  assume  $x \in N$ 
  then have  $p\text{-fix1 } x = 0$  using  $N\text{-def}$  by  $auto$ 
  then have  $\bigwedge m. MPoly\text{-Type.coeff } (p\text{-fix1 } x) m = 0$  by ( $metis$   $More\text{-MPoly-Type.coeff-monom}$ 
 $monom\text{-zero when-def}$ )
  have  $\bigwedge j. insertion (\lambda-. x) (?q j) = 0$ 
    using  $\langle \bigwedge m. MPoly\text{-Type.coeff } (p\text{-fix1 } x) m = 0 \rangle$  [ $unfolded$   $p\text{-fix1-def}$   $coeff\text{-replace-coeff}$ 
 $[of$   $insertion (\lambda-. x)$ ,  $OF$   $insertion\text{-zero}$ ]]
    by  $metis$ 
  then show  $x \in \{x. \forall j. insertion (\lambda-. x) (MPoly\text{-Type.coeff } (extract\text{-var } p n) j) = 0\}$  by  $blast$ 
qed
  then have  $finite\ N$  by ( $simp$   $add$ :  $\langle finite \{x. \forall j. insertion (\lambda-. x) (MPoly\text{-Type.coeff } (extract\text{-var } p n) j) = 0\} \rangle$ 
 $finite\text{-subset}$ )

  Use the IH:

  define  $A$  where  $A = \{f \in space (lborel\text{-f } (Suc\ n)). insertion\ f\ p = 0\}$ 

  have  $\bigwedge x1. vars (p\text{-fix1 } x1) \subseteq \{..<n\}$ 
proof -
  fix  $x1$ 
  have  $vars (extract\text{-var } p n) \subseteq \{..<n\}$ 
  using  $\langle vars\ p \subseteq \{..<Suc\ n\} \rangle$   $lessThan\text{-Suc } v\text{-not-in-vars-} extract\text{-var } vars\text{-extract-var-subset}$ 
by  $fastforce$ 
  then show  $vars (p\text{-fix1 } x1) \subseteq \{..<n\}$  unfolding  $p\text{-fix1-def}$ 
  using  $vars\text{-replace-coeff}$  [ $of$   $insertion (\lambda-. x1)$ ,  $OF$   $insertion\text{-zero}$ ] by  $blast$ 
qed
  have  $set\text{-eq}: \bigwedge x1. \{x \in space (lborel\text{-f } n). x(n := x1) \in A\} = \{f \in space (lborel\text{-f } n). insertion\ f (p\text{-fix1 } x1) = 0\}$ 
proof -
  fix  $x1$ 
  show  $\{x \in space (lborel\text{-f } n). x(n := x1) \in A\} = \{f \in space (lborel\text{-f } n). insertion\ f (p\text{-fix1 } x1) = 0\}$ 
proof ( $rule$   $subset\text{-antisym}; rule\ subsetI$ )
  fix  $x$  assume  $x \in \{x \in space (lborel\text{-f } n). x(n := x1) \in A\}$ 
  then have  $insertion (x(n := x1)) p = 0$   $x \in space (lborel\text{-f } n)$ 
  using  $A\text{-def}$  by  $auto$ 
  then have  $insertion\ x (p\text{-fix1 } x1) = 0$  unfolding  $p\text{-fix1-def}$ 
  unfolding  $replace\text{-coeff-extract-var-cong}$  [ $of$   $\lambda-. x1\ n\ x(n := x1)\ p$ ,  $OF$ 
 $fun\text{-upd-same}$  [ $symmetric$ ]]
  using  $insertion\text{-replace-coeff}$  [ $of$   $x(n := x1)$ ]
  using  $insertion\text{-irrelevant-vars}$  [ $of$   $replace\text{-coeff } (insertion (x(n := x1)))$ 
 $(extract\text{-var } p n)\ x\ x(n := x1)$ ]
   $vars\text{-replace-coeff } fun\text{-upd-other } insertion\text{-zero } reduce\text{-nested-mpoly-extract-var}$ 
 $subset\text{-eq}$ 
   $v\text{-not-in-vars-extract-var}$  by  $metis$ 
  then show  $x \in \{f \in space (lborel\text{-f } n). insertion\ f (p\text{-fix1 } x1) = 0\}$  using  $\langle x \in space (lborel\text{-f } n) \rangle$  by  $blast$ 
next

```

fix f **assume** $f \in \{f \in \text{space } (\text{lborel-}f \ n). \text{insertion } f \ (p\text{-fix1 } x1) = 0\}$
then have $f \in \text{space } (\text{lborel-}f \ n) \text{insertion } f \ (p\text{-fix1 } x1) = 0$ **by** *auto*
have $\text{insertion } (f(n := x1)) \ p = 0$ **using** $\langle \text{insertion } f \ (p\text{-fix1 } x1) = 0 \rangle$ [*unfolded*
p-fix1-def]
insertion-replace-coeff insertion-irrelevant-vars replace-coeff-extract-var-cong
by (*metis (no-types, lifting) \langle insertion f (p-fix1 x1) = 0 \rangle \langle vars (p-fix1 x1) \subseteq \{..<n\} \rangle*)
fun-upd-other fun-upd-same lessThan-iff order-less-irrefl p-fix1-def reduce-nested-mpoly-extract-var subsetCE)
then have $f(n := x1) \in A$ **unfolding** $A\text{-def}$ **using** *space-lborel-add-dim*
using $\langle f \in \text{space } (\text{lborel-}f \ n) \rangle$ *lborel-f-def mem-Collect-eq* **by** *blast*
then show $f \in \{f \in \text{space } (\text{lborel-}f \ n). f(n := x1) \in A\}$ **using** $\langle f \in \text{space } (\text{lborel-}f \ n) \rangle$ **by** *auto*
qed
qed

have $\bigwedge x1. x1 \in N \implies \{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-}f \ n)$
and *emeasure-in-N*: $\bigwedge x1. x1 \in N \implies \text{emeasure } (\text{lborel-}f \ n) \ \{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} = \text{emeasure } (\text{lborel-}f \ n) \ (\text{space } (\text{lborel-}f \ n))$
proof –
fix $x1$ **assume** $x1 \in N$
then have $p\text{-fix1 } x1 = 0$ **using** $N\text{-def}$ **by** *auto*
then have $\bigwedge f. \text{insertion } f \ (p\text{-fix1 } x1) = 0$ **using** *insertion-zero* **by** *auto*
then have $\{f \in \text{space } (\text{lborel-}f \ n). \text{insertion } f \ (p\text{-fix1 } x1) = 0\} = \text{space } (\text{lborel-}f \ n)$ **by** *simp*
show $\{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-}f \ n)$ **unfolding** *set-eq*
by (*simp add: \langle f \in \text{space } (\text{lborel-}f \ n). \text{insertion } f \ (p\text{-fix1 } x1) = 0 \rangle = \text{space } (\text{lborel-}f \ n) \rangle*)
show $\text{emeasure } (\text{lborel-}f \ n) \ \{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} = \text{emeasure } (\text{lborel-}f \ n) \ (\text{space } (\text{lborel-}f \ n))$
unfolding *set-eq*
by (*simp add: \langle f \in \text{space } (\text{lborel-}f \ n). \text{insertion } f \ (p\text{-fix1 } x1) = 0 \rangle = \text{space } (\text{lborel-}f \ n) \rangle*)
qed

have *emeasure-not-in-N*: $\bigwedge x1. x1 \notin N \implies \text{emeasure } (\text{lborel-}f \ n) \ \{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} = 0$
and $\bigwedge x1. x1 \notin N \implies \{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-}f \ n)$
proof –
fix $x1$ **assume** $x1 \notin N$
then have $p\text{-fix1 } x1 \neq 0$ **using** $p\text{-fix1-def } N\text{-def}$ **by** *auto*
then have $\text{emeasure } (\text{lborel-}f \ n) \ \{f \in \text{space } (\text{lborel-}f \ n). \text{insertion } f \ (p\text{-fix1 } x1) = 0\} = 0$
 $\{f \in \text{space } (\text{lborel-}f \ n). \text{insertion } f \ (p\text{-fix1 } x1) = 0\} \in \text{sets } (\text{lborel-}f \ n)$
using *Suc.IH[OF \langle p-fix1 x1 \neq 0 \rangle] \langle \bigwedge x1. \text{vars } (p\text{-fix1 } x1) \subseteq \{..<n\} \rangle* **by** *auto*
then show $\text{emeasure } (\text{lborel-}f \ n) \ \{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} = 0$
 $\{x \in \text{space } (\text{lborel-}f \ n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-}f \ n)$

using $\langle f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0 \rangle \in \text{sets } (\text{lborel-f } n) \rangle$
 $\langle \text{emeasure } (\text{lborel-f } n) \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} = 0 \rangle$
using *set-eq*
by *auto*
qed

have $N \in \text{null-sets lborel}$ **using** $\langle \text{finite } N \rangle$ *finite-imp-null-set-lborel* **by** *blast*
have *ae-zero*: $AE \ x1 \text{ in } \text{lborel}. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = 0$
apply (*rule* *AE-I*[*OF* $\langle N \in \text{null-sets lborel} \rangle$])
using $\langle \bigwedge x1. x1 \notin N \implies \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = 0 \rangle$
by *force*

have *measurable*: $(\lambda x1. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\}) \in \text{borel-measurable lborel}$
proof (*rule* *borel-measurableI*)
let $?f = (\lambda x1. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\})$
fix $S :: \text{ennreal set}$ **assume** *open S*
have $0: 0 \in S \implies - N \subseteq ?f - ' S$
using *emeasure-not-in-N* **by** *auto*
have $1: \text{emeasure } (\text{lborel-f } n) (\text{space } (\text{lborel-f } n)) \in S \implies N \subseteq ?f - ' S$
using *emeasure-in-N* **by** *auto*
have $2: 0 \notin S \implies ?f - ' S \subseteq N$ **using** *emeasure-not-in-N* **by** *fastforce*
have $3: \text{emeasure } (\text{lborel-f } n) (\text{space } (\text{lborel-f } n)) \notin S \implies ?f - ' S \subseteq -N$ **using** *emeasure-in-N* **by** *auto*
have $?f - ' S = \{\} \vee ?f - ' S = N \vee ?f - ' S = \text{UNIV} \vee ?f - ' S = -N$
apply (*cases* $0 \in S$; *cases* *emeasure* $(\text{lborel-f } n) (\text{space } (\text{lborel-f } n)) \notin S$)
using $0\ 1\ 2\ 3$ **by** *auto*
then show $?f - ' S \cap \text{space lborel} \in \text{sets lborel}$
using $\langle \text{finite } N \rangle$ *finite-imp-null-set-lborel* *borel-comp null-setsD2 sets-lborel* **by** *fastforce*
qed

have $A \in \text{sets } (\text{lborel-f } (\text{Suc } n))$ **unfolding** *A-def*
using *pred-eq-const1*[*OF* *measurable-insertion*[*OF* $\langle \text{vars } p \subseteq \{..<\text{Suc } n\} \rangle$]]
pred-def **by** *force*
then have *in-sets*: $\{f \in \text{space } (\text{lborel-f } (\text{Suc } n)). \text{insertion } f p = 0\} \in \text{sets } (\text{lborel-f } (\text{Suc } n))$ **using** *A-def* **by** *metis*
have $\bigwedge x1. \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n)$
using $\langle \bigwedge x1. x1 \in N \implies \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n) \rangle$
 $\langle \bigwedge x1. x1 \notin N \implies \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n) \rangle$ **by** *auto*
have *emeasure* $(\text{lborel-f } (\text{Suc } n)) A = \int^+ y. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := y) \in A\} \partial \text{lborel}$
using *emeasure-lborel-f-Suc*[*OF* $\langle A \in \text{sets } (\text{lborel-f } (\text{Suc } n)) \rangle$]
 $\langle \bigwedge x1. \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n) \rangle$ **by** *blast*

```

also have ... = 0
  using nn-integral-0-iff-AE[OF measurable] ae-zero by blast
  finally have emeasure (lborel-f (Suc n)) A = 0 by auto
  then show ?case unfolding null-sets-def using in-sets A-def by blast
qed

end

```

18 Shallow Network Model

```

theory DL-Shallow-Model
imports DL-Network Tensor-Rank
begin

```

```

fun shallow-model' where
  shallow-model' Z M 0 = Conv (Z,M) (Input M) |
  shallow-model' Z M (Suc N) = Pool (shallow-model' Z M 0) (shallow-model' Z M
N)

```

```

definition shallow-model where
  shallow-model Y Z M N = Conv (Y,Z) (shallow-model' Z M N)

```

```

lemma valid-shallow-model': valid-net (shallow-model' Z M N)
  apply (induction N) unfolding shallow-model'.simps
  by (simp add: valid-net.intros, metis shallow-model'.elims shallow-model'.simps(1)
valid-net.intros output-size.simps)

```

```

lemma output-size-shallow-model': output-size (shallow-model' Z M N) = Z
  apply (induction N) unfolding shallow-model'.simps using output-size.simps
by simp-all

```

```

lemma valid-shallow-model: valid-net (shallow-model Y Z M N)
  unfolding shallow-model-def using valid-shallow-model' valid-net.intros out-
put-size.simps output-size-shallow-model' by metis

```

```

lemma output-size-shallow-model: output-size (shallow-model Y Z M N) = Y
  unfolding shallow-model-def using output-size-shallow-model' output-size.simps
by simp

```

```

lemma input-sizes-shallow-model: input-sizes (shallow-model Y Z M N) = replicate
(Suc N) M
  apply (induction N) unfolding shallow-model-def input-sizes.simps by simp-all

```

```

lemma balanced-net-shallow-model': balanced-net (shallow-model' Z M N)
proof(induction N)
case 0
  then show ?case
  by (metis balanced-net.simps shallow-model'.simps(1))
next

```

```

case (Suc N)
have count-weights True (Conv (Z, M) (Input M)) = count-weights True (shallow-model' Z M N)
  by (induction N; simp)
then show ?case unfolding shallow-model'.simps
by (simp add: Suc.IH balanced-net-Conv balanced-net-Input balanced-net-Pool)
qed

```

```

lemma balanced-net-shallow-model: balanced-net (shallow-model Y Z M N)
  unfolding shallow-model-def
  by (simp add: balanced-net-Conv balanced-net-shallow-model')

```

```

lemma cprank-max1-shallow-model':
assumes y < output-size (shallow-model' Z M N)
shows cprank-max1 (tensors-from-net (insert-weights s (shallow-model' Z M N) w) $ y)
  using assms proof (induction N arbitrary:w)
  case 0
  then have input-sizes (insert-weights s (shallow-model' Z M 0) w) = [M]
    unfolding shallow-model-def shallow-model'.simps insert-weights.simps input-sizes.simps by metis
  then have dims (tensors-from-net (insert-weights s (shallow-model' Z M 0) w) $ y) = [M]
    using dims-tensors-from-net[OF vec-setI] 0.prem1 output-size-correct-tensors remove-insert-weights valid-shallow-model' by metis
  then show ?case
    using order1 by (metis One-nat-def eq-imp-le length-Cons list.size(3))
next
  case (Suc N)
  have y-le-IH:y < dim-vec (tensors-from-net (insert-weights s (shallow-model' Z M N) (λi. w (i + (count-weights s (shallow-model' Z M 0))))))
    using output-size-correct-tensors[of insert-weights s (shallow-model' Z M N) (λi. w (i + (count-weights s (shallow-model' Z M 0))))],
    unfolded remove-insert-weights, OF valid-shallow-model'
    using Suc.prem1 output-size-shallow-model' by auto
  have cprank-max1-IH:cprank-max1 (tensors-from-net (insert-weights s (shallow-model' Z M N) (λi. w (i + (count-weights s (shallow-model' Z M 0)))))) $ y
    using Suc.IH Suc.prem1 output-size-shallow-model' by auto
  have y-le-0:y < dim-vec (tensors-from-net (insert-weights s (shallow-model' Z M 0) w))
    by (metis assms output-size-correct-tensors output-size-shallow-model' remove-insert-weights valid-shallow-model')
  have cprank-max1-0:cprank-max1 (tensors-from-net (insert-weights s (shallow-model' Z M 0) w) $ y)
  proof –
    have input-sizes (insert-weights s (shallow-model' Z M 0) w) = [M]
      unfolding shallow-model-def shallow-model'.simps insert-weights.simps input-sizes.simps by metis
    then show ?thesis using order1 dims-tensors-from-net[OF vec-setI] One-nat-def

```

```

eq-imp-le length-Cons list.size(3) y-le-0 by metis
qed
then show ?case unfolding shallow-model'.simps(2) insert-weights.simps ten-
sors-from-net.simps
using cprank-max1-IH cprank-max1-0 cprank-max1-prod index-component-mult
y-le-0 y-le-IH
by (metis Suc.IH output-size-correct-tensors remove-insert-weights valid-shallow-model')
qed

```

lemma *cprank-shallow-model*:

assumes $m = \text{insert-weights } s \text{ (shallow-model } Y \ Z \ M \ N) \ w$

assumes $y < Y$

shows $\text{cprank (tensors-from-net } m \ \$ \ y) \leq Z$

proof –

have $s \implies \text{shared-weight-net } m$

by (*simp add: assms(1) balanced-net-shallow-model shared-weight-net-insert-weights*)

have $\text{cprank-max } Z \text{ (tensors-from-net } m \ \$ \ y)$

proof –

have *dim-extract*: $\text{dim-row (extract-matrix } w \ Y \ Z) = Y$

using *dim-extract-matrix(1)* **by force**

have *dimc-extract-matrix*: $\text{dim-col (extract-matrix } w \ Y \ Z) = Z$

using *dim-extract-matrix(2)* **by force**

have *input-sizes*: $(\text{input-sizes (insert-weights } s \text{ (shallow-model' } Z \ M \ N) \ (\lambda i. w \ (i + Y * Z)))) = (\text{input-sizes (shallow-model' } Z \ M \ N))$

using *input-sizes-remove-weights remove-insert-weights* **by auto**

have $0 : \text{tensors-from-net } m \ \$ \ y = \text{Tensor-Plus.listsum (input-sizes (shallow-model' } Z \ M \ N))$

$(\text{map } (\lambda j. (\text{extract-matrix } w \ Y \ Z) \ \$ \ \$ (y, j) \cdot (\text{tensors-from-net (insert-weights } s \text{ (shallow-model' } Z \ M \ N) \ (\lambda i. w \ (i + Y * Z)))) \ \$ \ j) \ [0..<Z])$

unfolding $\langle m = \text{insert-weights } s \text{ (shallow-model } Y \ Z \ M \ N) \ w \rangle$ *shallow-model-def insert-weights.simps tensors-from-net.simps*

using *nth-mat-tensorlist-mult dims-tensors-from-net assms(2) dim-extract output-size-correct-tensors[of insert-weights s (shallow-model' Z M N) (λi. w (i + Y * Z)), unfolded remove-insert-weights, OF valid-shallow-model']*

dimc-extract-matrix output-size-shallow-model' input-sizes **by auto**

define Bs **where** $Bs = \text{map } (\lambda j. \text{extract-matrix } w \ Y \ Z \ \$ \ \$ (y, j) \cdot \text{tensors-from-net (insert-weights } s \text{ (shallow-model' } Z \ M \ N) \ (\lambda i. w \ (i + Y * Z)))) \ \$ \ j) \ [0..<Z]$

have $\bigwedge B. B \in \text{set } Bs \implies \text{cprank-max1 } B \ \bigwedge B. B \in \text{set } Bs \implies \text{dims } B = \text{input-sizes (shallow-model' } Z \ M \ N)$

proof –

fix B **assume** $B \in \text{set } Bs$

then obtain j **where** $B = Bs ! j \ j < \text{length } Bs$ **by** (*metis in-set-conv-nth*)

then have $j < Z$ **using** *length-map Bs-def* **by simp**

have $1 : \text{cprank-max1 (tensors-from-net (insert-weights } s \text{ (shallow-model' } Z \ M \ N) \ (\lambda i. w \ (i + Y * Z))) \ \$ \ j)$


```

    using ⟨j < Z⟩ output-size-shallow-model' cprank-max1-shallow-model' by
  auto
  then have cprank-max1 (extract-matrix w Y Z $$ (y, j) · tensors-from-net
    (insert-weights s (shallow-model' Z M N) (λi. w (i + Y * Z))) $ j)
    using smult-prod-extract1 cprank-max1-order0[OF 1, of extract-matrix w Y
      Z $$ (y, j) · 1]
    by (metis dims-smult mult.left-neutral order-tensor-one)
  then show cprank-max1 B by (simp add: Bs-def ⟨B = Bs ! j⟩ ⟨j < Z⟩)
  show dims B = input-sizes (shallow-model' Z M N) unfolding ⟨B = Bs ! j⟩
  Bs-def
  nth-map[of j [0..<Z], unfolded length-upt Nat.diff-0, OF ⟨j < Z⟩] dims-smult
  input-sizes[symmetric]
  by (rule dims-tensors-from-net; rule vec-setI[where i=j], simp add:⟨j <
    Z⟩, metis (no-types) ⟨j < Z⟩ output-size-correct-tensors output-size-shallow-model'
    remove-insert-weights valid-shallow-model')
  qed
  then show ?thesis unfolding 0 using cprank-max1 length-map Bs-def by
  (metis (no-types, lifting) diff-zero length-upt)
  qed
  then show ?thesis unfolding cprank-def by (simp add: Least-le)
  qed
end

```

19 Fundamental Theorem of Network Capacity

theory *DL-Fundamental-Theorem-Network-Capacity*

imports *DL-Rank-CP-Rank DL-Deep-Model-Poly Lebesgue-Zero-Set*

Jordan-Normal-Form.DL-Rank-Submatrix HOL-Analysis.Complete-Measure DL-Shallow-Model

begin

context *deep-model-correct-params-y*

begin

definition *polynomial-f* $w = \det (\text{submatrix} (\text{matricize} \{n. \text{even } n\} (A \ w)) \text{rows-with-1}$
 $\text{rows-with-1})$

lemma *polyfun-polynomial:*

shows *polyfun* $\{..<\text{weight-space-dim}\}$ *polynomial-f*

unfolding *polynomial-f-def* **using** *polyfun-det-deep-model* **unfolding** *witness-submatrix-def*
A'-def .

definition *polynomial-p* $= (\text{SOME } p. \text{vars } p \subseteq \{..<\text{weight-space-dim}\} \wedge (\forall x. \text{in-}$
 $\text{sertion } x \ p = \text{polynomial-f } x))$

lemma

polynomial-p-not-0: polynomial-p ≠ 0 **and**

vars-polynomial-p: *vars polynomial-p* $\subseteq \{..<weight-space-dim\}$ **and**
polynomial-pf: $\bigwedge w. \text{insertion } w \text{ polynomial-p} = \text{polynomial-f } w$
proof –
have *vars polynomial-p* $\subseteq \{..<weight-space-dim\} \wedge (\forall x. \text{insertion } x \text{ polynomial-p} = \text{polynomial-f } x)$ **unfolding** *polynomial-p-def*
using *someI-ex*[*OF polyfun-polynomial*[*unfolded polyfun-def*]] .
then show *vars polynomial-p* $\subseteq \{..<weight-space-dim\} \bigwedge w. \text{insertion } w \text{ polynomial-p} = \text{polynomial-f } w$ **by auto**
show *polynomial-p* $\neq 0$ **using** *A'-def Aw'-def'* $\langle \bigwedge w. \text{insertion } w \text{ polynomial-p} = \text{polynomial-f } w \rangle$ *polynomial-f-def witness-det* **by auto**
qed

lemma *if-polynomial-0-rank*:
assumes *polynomial-f w* $\neq 0$
shows $r \wedge N\text{-half} \leq \text{cprank } (A \ w)$
proof –
have $r \wedge N\text{-half} = \text{dim-row } (\text{submatrix } (\text{matricize } \{n. \text{even } n\} (A \ w)) \ \text{rows-with-1} \ \text{rows-with-1})$
by (*metis* (*full-types*) *Aw'-def card-rows-with-1 dim-submatrix*(1) *dims-A dims-Aw dims-matricize*(1) *set-le-in*)
also have $\dots \leq \text{mrank } (\text{matricize } \{n. \text{even } n\} (A \ w))$
using *assms vec-space.rank-gt-minor*[*OF carrier-matI*[*OF dims-A'-pow, unfolded weight-space-dim-def*]]
by (*metis* (*full-types*) *A'-def dim-submatrix*(1) *dims-A'-pow*(1) *polynomial-f-def*)
also have $\dots \leq \text{cprank } (A \ w)$ **using** *matrix-rank-le-cp-rank* **by blast**
finally show *?thesis* .
qed

lemma *if-polynomial-0-evaluate*:
assumes *polynomial-f wd* $\neq 0$
assumes $\forall \text{inputs}. \text{input-sizes } (\text{deep-model-l } rs) = \text{map dim-vec inputs} \longrightarrow \text{evaluate-net } (\text{insert-weights shared-weights } (\text{deep-model-l } rs) \ wd) \ \text{inputs}$
 $= \text{evaluate-net } (\text{insert-weights shared-weights } (\text{shallow-model } (rs \ ! \ 0) \ Z \ (\text{last } rs) \ (2 * N\text{-half} - 1)) \ ws) \ \text{inputs}$
shows $Z \geq r \wedge N\text{-half}$
proof –
have *valid1:valid-net'* (*insert-weights shared-weights* (*deep-model-l rs*) *wd*)
using *remove-insert-weights valid-deep-model* **by presburger**
have *valid2:valid-net'* (*insert-weights shared-weights* (*shallow-model* (*rs ! 0*) *Z* (*last rs*) (*2 * N-half - 1*)) *ws*)
by (*simp add: remove-insert-weights valid-shallow-model*)
have *input-sizes: input-sizes* (*insert-weights shared-weights* (*deep-model-l rs*) *wd*)
 $= \text{input-sizes } (\text{insert-weights shared-weights } (\text{shallow-model } (rs \ ! \ 0) \ Z \ (\text{last } rs) \ (2 * N\text{-half} - 1)) \ ws)$
using *input-sizes-remove-weights input-sizes-deep-model remove-insert-weights*
by (*simp add: N-half-def input-sizes-shallow-model*)
have *0:tensors-from-net* (*insert-weights shared-weights* (*deep-model-l rs*) *wd*)
 $= \text{tensors-from-net } (\text{insert-weights shared-weights } (\text{shallow-model } (rs \ ! \ 0) \ Z \ (\text{last } rs) \ (2 * N\text{-half} - 1)) \ ws)$

using *tensors-from-net-eqI*[*OF valid1 valid2 input-sizes, unfolded input-sizes-remove-weights remove-insert-weights*]
using *assms* **by** *blast*
have *cprank* (*tensors-from-net* (*insert-weights shared-weights* (*deep-model-l rs*) *wd*) \$ *y*) $\leq Z$
unfolding *0* **using** *y-valid cprank-shallow-model* **by** *blast*
then show *?thesis*
using *if-polynomial-0-rank assms*
using *A-def assms(1) less-le-trans not-le remove-insert-weights*
by *fastforce*
qed

lemma *if-polynomial-0-evaluate-notex*:
assumes *polynomial-f wd $\neq 0$*
shows $\neg(\exists \text{weights-shallow } Z. Z < r \wedge N\text{-half} \wedge (\forall \text{inputs. input-sizes (deep-model-l rs)} = \text{map dim-vec inputs} \longrightarrow \text{evaluate-net (insert-weights shared-weights (deep-model-l rs) wd) inputs} = \text{evaluate-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N\text{-half}-1)) ws) inputs}))$
using *assms if-polynomial-0-evaluate not-le* **by** *blast*

theorem *fundamental-theorem-network-capacity*:
AE x in lborel-f weight-space-dim. $r \wedge N\text{-half} \leq \text{cprank} (A x)$
using *AE-I'[OF lebesgue-mpoly-zero-set[OF polynomial-p-not-0 vars-polynomial-p]]*
by (*metis (mono-tags, lifting) Collect-mono if-polynomial-0-rank polynomial-pf*)

theorem *fundamental-theorem-network-capacity-v2*:
shows *AE wd in lborel-f weight-space-dim.*
 $\neg(\exists \text{ws } Z. Z < r \wedge N\text{-half} \wedge (\forall \text{inputs. input-sizes (deep-model-l rs)} = \text{map dim-vec inputs} \longrightarrow \text{evaluate-net (insert-weights shared-weights (deep-model-l rs) wd) inputs} = \text{evaluate-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N\text{-half}-1)) ws) inputs}))$
apply (*rule AE-I'[OF lebesgue-mpoly-zero-set[OF polynomial-p-not-0 vars-polynomial-p], unfolded polynomial-pf]*)
apply (*rule subsetI*) **unfolding** *mem-Collect-eq*
using *if-polynomial-0-evaluate-notex* **by** *metis*

abbreviation *lebesgue-f where lebesgue-f n \equiv completion (lborel-f n)*

lemma *space-lebesgue-f: space (lebesgue-f n) = $Pi_E \{..<n\} (\lambda-. UNIV)$*
by (*simp add: space-lborel-f*)

theorem *fundamental-theorem-network-capacity-v3*:
assumes
 $S = \{wd \in \text{space (lebesgue-f weight-space-dim)}. \exists \text{ws } Z. Z < r \wedge N\text{-half} \wedge (\forall \text{inputs. input-sizes (deep-model-l rs)} = \text{map dim-vec inputs} \longrightarrow \text{evaluate-net (insert-weights shared-weights (deep-model-l rs) wd) inputs})\}$

```

    = evaluate-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last
rs) (2*N-half-1)) ws) inputs)
  shows S ∈ null-sets (completion (lborel-f weight-space-dim))
  unfolding assms
  using fundamental-theorem-network-capacity-v2[unfolded completion.AE-iff-null-sets[unfolded
AE-completion-iff], unfolded not-not]
  by blast

end
end

```

References

- [1] A. Bentkamp. An Isabelle Formalization of the Expressiveness of Deep Learning. Master’s thesis, Universität des Saarlandes, 2016. http://matryoshka.gforge.inria.fr/bentkamp_msc_thesis.pdf.
- [2] N. Cohen, O. Sharir, and A. Shashua. On the expressive power of deep learning: A tensor analysis. In V. Feldman, A. Rakhlin, and O. Shamir, editors, *Conference on Learning Theory (COLT 2016)*, volume 49 of *JMLR Workshop and Conference Proceedings*, pages 698–728. JMLR.org, 2016.