Expressiveness of Deep Learning

Alexander Bentkamp

December 7, 2022

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

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 *dims A* = *dims B* **and** *vec A* = *vec B*
shows *A*=*B*
by (*metis assms tensor-from-vec-simp*)

abbreviation *order*::$'a$ *tensor* ⇒ *nat* **where**
order t == *length* (*dims t*)

inductive *valid-index*::*nat list* ⇒ *nat list* ⇒ *bool* (**infix** ⊲ *50*) **where**
Nil: [] ⊲ [] |
Cons: *is* ⊲ *ds* ⟹ *i*<*d* ⟹ *i*#*is* ⊲ *d*#*ds*

inductive-cases *valid-indexE*[*elim*]: *is* ⊲ *ds*
inductive-cases *valid-index-dimsE*[*elim*]: *is* ⊲ *dims A*

lemma *valid-index-length*: *is* ⊲ *ds* ⟹ *length is* = *length ds*
by (*induction rule*:*valid-index.induct*; *auto*)

lemma *valid-index-lt*: *is* ⊲ *ds* ⟹ *m*<*length ds* ⟹ *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 *length is* = *length ds* **and** ⋀*m*. *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*
shows *is* ⊲ *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* ⊲ *ds′* **using** *Cons* **by** (*metis length-Cons less-irrefl linorder-neqE-nat not-less-eq nth-Cons-Suc*)
then show *?case* **using** *Cons.prems*(*2*) ‹*ds* = *d* # *ds′*› *valid-index.Cons* **by** *fastforce*
qed

lemma *valid-index-append*:
assumes *is1-valid*:*is1* ⊲ *ds1* **and** *is2-valid*:*is2* ⊲ *ds2*
shows *is1* @ *is2* ⊲ *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* ⟷ *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* ⇒ *nat* ⇒ *nat* ⇒ ′*a list* **where**
fixed-length-sublist xs l i = (*take l* (*drop* (*l*∗*i*) *xs*))

fun *lookup-base*::*nat list* ⇒ ′*a list* ⇒ *nat list* ⇒ ′*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* ⇒ *nat list* ⇒ ′*a* **where**
lookup A = *lookup-base* (*dims A*) (*vec A*)

fun *tensor-vec-from-lookup*::*nat list* ⇒ (*nat list* ⇒ ′*a*) ⇒ ′*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* (λ*i*. *tensor-vec-from-lookup ds* (λ*is*. *e* (*i* # *is*))) [*0*..<*d*])

definition *tensor-from-lookup*::*nat list* ⇒ (*nat list* ⇒ ′*a*) ⇒ ′*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* ≤ *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* $\lhd$ *ds* $\Longrightarrow$ *lookup-base ds v is* = *e is*
shows *tensor-vec-from-lookup ds e* = *v*
using *assms* **proof** (*induction ds arbitrary*:*v e*)
  case *Nil*
  then show *?case* **unfolding** *tensor-vec-from-lookup.simps*
    by (*metis One-nat-def Tensor.lookup-base-Nil length-0-conv length-Suc-conv list.sel*(*1*) *prod-list.Nil valid-index.Nil*)
next
  case (*Cons a ds*)
  then have *a* ∗ *prod-list ds* = *length v* **by** *auto*
  {
   fix *i* **assume** *i*<*a*
   then have *prod-list ds* ∗ (*i*+*1*) ≤ *length v* **using** ‹*a* ∗ *prod-list ds* = *length v*› **using** *discrete mult.commute mult-le-mono1* **by** *metis*
    have $\bigwedge$*is'*. *is'* $\lhd$ *ds* $\Longrightarrow$ *e* (*i* # *is'*) = *lookup-base ds* (*fixed-length-sublist v* (*prod-list ds*) *i*) *is'*
    using ‹*i*<*a*› **by** (*metis Cons.prems*(*2*) *Tensor.lookup-base-Cons valid-index.simps*)
   then have *tensor-vec-from-lookup ds* (λ*is'*. *e* (*i* # *is'*)) = *fixed-length-sublist v* (*prod-list ds*) *i*
     using *Cons* **using** ‹*prod-list ds* ∗ (*i* + *1*) ≤ *length v*› **by** (*simp add*: *Cons.IH fixed-length-sublist-def*)
  }
  then show *?case* **unfolding** *tensor-vec-from-lookup-Cons lookup-base-Cons*
   using *concat-parts-eq*[*OF* ‹*a* ∗ *prod-list ds* = *length v*›]
    atLeastLessThan-iff map-eq-conv set-upt Cons **by** (*metis* (*no-types*, *lifting*))
qed

lemma *tensor-lookup*:
assumes $\bigwedge$*is. is* $\lhd$ *dims A* $\Longrightarrow$ *lookup A is* = *e is*
shows *tensor-from-lookup* (*dims A*) *e* = *A*
using *tensor-lookup-base lookup-def length-vec tensor-from-lookup-def* **by** (*metis assms tensor-from-vec-simp*)

lemma *concat-equal-length*:
assumes $\bigwedge$*xs. xs*∈*set xss* $\Longrightarrow$ *length xs* = *l*
shows *length* (*concat xss*) = *length xss*∗*l*
using *assms* **by** (*induction xss*;*auto*)

lemma *concat-equal-length-map*:
assumes $\bigwedge$*i. i*<*a* $\Longrightarrow$ *length* (*f i*) = *d*
shows *length* (*concat* (*map* (λ*i. f i*) [*0*..<*a*])) = *a*∗*d*
using *assms* **by** (*induction a*;*auto*)

lemma *concat-parts*:
assumes $\bigwedge$*xs. xs*∈*set xss* $\Longrightarrow$ *length xs* = *d* **and** *i*<*length xss*
shows *fixed-length-sublist* (*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.prems*(*1*) *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.prems*(*1*))
    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.prems*(*1*) **by** *auto*
  qed
qed

lemma *concat-parts'*:
assumes ⋀*i*. *i*<*a* ⟹ *length* (*f i*) = *d*
and *i*<*a*
shows *fixed-length-sublist* (*concat* (*map* (λ*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 (⋀*i*. *i* < *a* ⟹ *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.prems*(*1*) ‹*length* (*concat* (*map f* [*0*..<*a*])) = *a* ∗ *d*› *fixed-length-sublist-def*)
    then show *?case* **using** ‹*i*=*a*› **by** *auto*
  next
    assume *i*≠*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** ‹*concat* (*map f* [*0*..<*Suc a*]) = *concat* (*map f* [*0*..<*a*]) @ *f a*›
      unfolding *fixed-length-sublist-def drop-append*
      using ‹*length* (*concat* (*map f* [*0*..<*a*])) = *a* ∗ *d*› ‹*fixed-length-sublist* (*concat* (*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*:
length (*tensor-vec-from-lookup ds e*) = *prod-list ds*
by (*induction ds arbitrary*:*e*; *auto simp add*: *concat-equal-length-map*)

lemma *lookup-tensor-vec*:
assumes *is*⊲*ds*
shows *lookup-base ds* (*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* λ*i. tensor-vec-from-lookup ds* (λ*is. e* (*i* # *is*)) *prod-list ds i*] ‹*i* < *d*›)
qed

lemma *lookup-tensor-from-lookup*:
assumes *is*⊲*ds*
shows *lookup* (*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*: *dims* (*tensor-from-lookup ds e*) = *ds*
unfolding *tensor-from-lookup-def*
by (*simp add*: *length-tensor-vec-from-lookup*)

lemma *tensor-lookup-cong*:
assumes *tensor-from-lookup ds* e_1 = *tensor-from-lookup ds* e_2
and *is*⊲*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*⊲*ds* $\Longrightarrow$ e_1 *is* = e_2 *is*
shows *tensor-from-lookup ds* e_1 = *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 *dims A* = *dims B* **and** $\bigwedge$*is. is*⊲(*dims A*) $\Longrightarrow$ *lookup A is* = *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. dims A* = [] $\Longrightarrow$ *P A*
and *order-step*:$\bigwedge$*As ds.* ($\bigwedge$*A. A*$\in$*set As* $\Longrightarrow$ *P A*) $\Longrightarrow$ ($\bigwedge$*A. A*$\in$*set As* $\Longrightarrow$ *dims A* = *ds*) $\Longrightarrow$ *P* (*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* (*subtensor-combine* (*tl* (*dims A*)) (*map* (*subtensor A*) [*0*..<*hd* (*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* $\lhd$ *dims A*
shows *lookup* (*subtensor A i*) *is* = *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*:*subtensor* (*subtensor-combine ds As*) *i* = *As* ! *i*
by (*metis list.discI list.sel*(*1*) *order-step.hyps order-step.prems subtensor-combine-dims subtensor-subtensor-combine valid-index-dimsE*)
have *1*:*dims* (*subtensor-combine ds As*) = *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.prems*
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* $\lhd$ *dims A*
shows *lookup A is* = *hd* (*vec* (*fold* (λ*i A. 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 $dims\ A \neq []$
and *dims-eq*:$dims\ A = dims\ B$
and $\bigwedge i.\ i < hd\ (dims\ A) \Longrightarrow subtensor\ A\ i = subtensor\ B\ i$
shows $A=B$
proof −
{
fix *is* **assume** $is \lhd dims\ A$
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* ‹$is \lhd dims\ A$› *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 $vec\text{-}plus\ a\ b = map\ (\lambda(x,y).\ plus\ x\ y)\ (zip\ a\ b)$

definition *plus-base*::$'a$::*semigroup-add tensor* ⇒ $'a$ *tensor* ⇒ $'a$ *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*]: $dims\ A = dims\ B \Longrightarrow dims\ (A + B) = dims\ A$ **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*]: $dims\ A = dims\ B \Longrightarrow dims\ (A + B) = dims\ B$ **using**

plus-dim1 **by** *metis*
lemma *plus-base*: *dims A* = *dims B* ⟹ *A* + *B* = *plus-base A B* **unfolding** *plus-def* **by** *metis*

lemma *fixed-length-sublist-plus*:
assumes *length xs1* = *c* ∗ *l* *length xs2* = *c* ∗ *l* *i* < *c*
shows *fixed-length-sublist* (*vec-plus xs1 xs2*) *l i*
= *vec-plus* (*fixed-length-sublist xs1 l i*) (*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 *dims A* = *dims B*
shows *vec* (*A*+*B*) = *vec-plus* (*vec A*) (*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* < *hd* (*dims A*)
and *dims A* = *dims B*
and *dims A* ≠ []
shows *subtensor* (*A* + *B*) *i* = *subtensor A i* + *subtensor B i*
proof −
have *length* (*vec A*) = *hd* (*dims A*) ∗ *prod-list* (*tl* (*dims A*))
length (*Tensor.vec B*) = *hd* (*dims A*) ∗ *prod-list* (*tl* (*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 tensor-eqI*
dims-subtensor plus-dim1 **by** *fastforce*
qed

lemma *lookup-plus*[*simp*]:
assumes *dims A* = *dims B*
and *is* ◁ *dims A*
shows *lookup* (*A* + *B*) *is* = *lookup A is* + *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*:[] ◁ *dims A* **using** *order-0* ‹*is* = []› **by** *auto*
have *2*:[] ◁ *dims B* **using** *order-0* ‹*is* = []› **by** *auto*
have *3*:[] ◁ *dims* (*A* + *B*) **using** *order-0* ‹*is* = []› **by** *auto*
have *length* (*vec A*) = *1* *length* (*vec B*) = *1*
by (*metis length-vec prod-list.Nil order-0.hyps order-0.prems*(*1*) *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.prems*(*1*)] **unfolding** *vec-plus-def* **using** *or-*

der-0.prems 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 order-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.prems*(*1*) *subtensor-plus valid-index.cases*)
using *1* ‹*is* = *i* # *is′*› *order-step.prems*(*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*) *subtensor-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* ⇒ ′*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* ◁ (*dims A*) ⟹ *lookup A is* ∈ *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.prems* **by** *auto*
have *2*:*is′* ◁ *dims* (*subtensor A i*) **using** ‹*is* = *i* # *is′*› *dims-subtensor order-step.prems* **by** *auto*
have *lookup A is* ∈ *set* (*Tensor.vec* (*subtensor A i*))
using *order-step.IH* [*OF 1 2*] *lookup-subtensor1* ‹*is* = *i* # *is′*› *order-step.prems* **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* ◁ *ds*
shows *lookup* (*tensor0 ds*) *is* = *0*
proof −
have *lookup* (*tensor0 ds*) *is* ∈ *set* (*vec* (*tensor0 ds*)) **using** *lookup-is-in-vec assms* **by** (*metis dims-tensor0*)
moreover have *set* (*vec* (*tensor0 ds*)) ⊆ {*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*]: *tensor0* (*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*::*nat list* ⇒ *'a*::*monoid-add tensor list* ⇒ *'a tensor* **where**
listsum ds As = *foldr* (+) *As* (*tensor0 ds*)

definition *listsum'*::*'a*::*monoid-add tensor list* ⇒ *'a tensor* **where**
listsum' As = *listsum* (*dims* (*hd As*)) *As*

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

lemma *listsum-one*: *listsum* (*dims A*) [*A*] = *A* **unfolding** *listsum-def* **by** *simp*

lemma *listsum-Cons*: *listsum ds* (*A* # *As*) = *A* + *listsum ds As*
unfolding *listsum-def* **by** *auto*

lemma *listsum-dims*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
shows *dims* (*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* ≠ [] **and** *i*<*hd ds*
shows *subtensor* (*tensor0 ds*) *i* = *tensor0* (*tl ds*)
proof (*rule tensor-lookup-eqI*)
show *1*:*dims* (*subtensor* (*tensor0 ds*) *i*) = *dims* (*tensor0* (*tl ds*)) **by** (*simp add*: *assms*(*1*) *assms*(*2*))
fix *is* **assume** *is* ◁ *dims* (*subtensor* (*tensor0 ds*) *i*)
then have *i* # *is* ◁ *dims* (*tensor0 ds*) **using** *assms*(*1*) *assms*(*2*) *valid-index.Cons* **by** *fastforce*
then show *lookup* (*subtensor* (*tensor0 ds*) *i*) *is* = *lookup* (*tensor0* (*tl ds*)) *is*
using *lookup-subtensor1 1* ‹*is* ◁ *dims* (*subtensor* (*tensor0 ds*) *i*)› *dims-tensor0 lookup-tensor0*
by *metis*
qed

lemma *subtensor-listsum*:

assumes $\bigwedge A.\ A{\in}set\ As \Longrightarrow dims\ A = ds$
and $ds \neq []$ **and** $i{<}hd\ ds$
shows $subtensor\ (listsum\ ds\ As)\ i = listsum\ (tl\ ds)\ (map\ (\lambda A.\ subtensor\ A\ i)\ As)$
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 $\bigwedge A.\ A{\in}set\ As \Longrightarrow A = tensor0\ ds$
shows *listsum ds As* = *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 $\bigwedge i.\ i{\neq}j \Longrightarrow i{<}length\ As \Longrightarrow As!i = tensor0\ ds$
and *dims* (*As*!*j*) = *ds*
and $j < length\ As$
shows *listsum ds 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 $\bigwedge i.\ i < length\ As \Longrightarrow As\ !\ i = tensor0\ ds$ **using** *Cons* **using** *Suc-less-eq length-Cons list.sel*(*3*) *nat.simps*(*3*) *nth-tl* **by** *fastforce*
   then have *listsum ds As* = *tensor0 ds* **using** *listsum0* **by** (*metis in-set-conv-nth*)
    then show *?thesis* **by** (*metis 0 Cons.prems*(*2*) *listsum-Cons nth-Cons-0 tensor-add-0-right*)
  next
    case (*Suc j'*)
   then have *listsum ds As* = *As*!*j'* **by** (*metis* (*no-types*, *lifting*) *Cons.IH Cons.prems*(*1*) *Cons.prems*(*2*) *Cons.prems*(*3*) *Suc-less-eq length-Cons less-Suc-eq list.sel*(*3*) *not-less-eq nth-tl*)
   then show *?thesis* **by** (*metis Cons.prems*(*1*) *Cons.prems*(*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* $\lhd$ *ds*
and $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
shows *lookup* (*listsum ds As*) *is* = ($\sum A \leftarrow As.\ lookup\ A\ is$)
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* (α + β) *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*) = (α ∗ *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*:*vec-smult α* (*vec-plus* (*a # as'*) *bs*) = (*α*∗(*a*+*b*)) # *vec-smult α* (*vec-plus as' bs'*)
    unfolding *vec-smult-def vec-plus-def* **using** *Cons.IH*[*of bs'*] **by** *simp*
  have *length bs' = length as'* **using** *Cons.prems* ‹*bs = b # bs'*› **by** *auto*
  then show *?case* **unfolding** *0* **unfolding** ‹*bs = b # bs'*› *vec-smult-Cons vec-plus-Cons*
    by (*simp add*: *Cons.IH distrib-left*)
qed

lemma *length-vec-smult*: *length* (*vec-smult α v*) = *length v* **unfolding** *vec-smult-def* **by** *simp*

definition *smult*::*'a*::*ring* ⇒ *'a tensor* ⇒ *'a tensor* (**infixl** · *70*) **where**
smult α A = (*tensor-from-vec* (*dims A*) (*vec-smult α* (*vec A*)))

lemma *tensor-smult0*: **fixes** *A*::*'a*::*ring tensor*
shows *0* · *A* = *tensor0* (*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*]:*dims* (*α* · *A*) = *dims A*
and *vec-smult*[*simp*]: *vec* (*α* · *A*) = *map* ((∗) *α*) (*vec A*)
  unfolding *smult-def vec-smult-def* **by** (*simp add*: *length-vec*)+

lemma *tensor-smult-distr-right*: (*α* + *β*) · *A* = *α* · *A* + *β* · *A*
  unfolding *plus-def plus-base-def*
  by (*auto*; *metis smult-def vec-smult-def vec-smult-distr-right*)

lemma *tensor-smult-distr-left*: *dims A = dims B* ⟹ *α* · (*A* + *B*) = *α* · *A* + *α* · *B*
proof −
  assume *a1*: *dims A = dims B*
  then have *f2*: *length* (*vec-plus* (*vec A*) (*vec B*)) = *length* (*vec A*)
    by (*simp add*: *length-vec vec-plus-def*)
  have *f3*: *dims* (*tensor-from-vec* (*dims B*) (*vec-smult α* (*vec A*))) = *dims B*
    using *a1* **by** (*simp add*: *length-vec vec-smult-def*)
  have *f4*: *vec* (*α* · *A*) = *vec-smult α* (*vec A*)
    by (*simp add*: *vec-smult-def*)
  have *length* (*vec-smult α* (*vec B*)) = *length* (*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 length xs = l ∗ c i<c
shows fixed-length-sublist (vec-smult α xs) l i = vec-smult α (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 dims A ≠ [] i < hd (dims A)
shows α · subtensor A i = subtensor (α · A) i
proof (rule tensor-eqI)
  show dims (α · subtensor A i) = dims (subtensor (α · A) i)
    using dims-smult dims-subtensor assms(1) assms(2) by simp
  show vec (α · subtensor A i) = vec (subtensor (α · 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 ⊲ dims A
shows lookup (α · A) is = α ∗ lookup A is
using assms proof (induction A arbitrary:is rule:subtensor-induct)
  case (order-0 A is)
  then have length (vec A) = 1 by (simp add: length-vec)
  then have hd (vec-smult α (vec A)) = α ∗ hd (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 lookup (α · subtensor A i) is′ = α ∗ lookup (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.prems valid-index-dimsE)
  then show ?case using smult-subtensor ‹is = i # is′› dims-smult lookup-subtensor1
    list.sel(1) order-step.hyps order-step.prems valid-index-dimsE
    by metis
qed

lemma tensor-smult-assoc:
fixes A::′a::ring tensor
shows α · (β · A) = (α ∗ β) · 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*:$A * B$ = *tensor-from-vec* (*dims A* @ *dims B*) (*concat* (*map* (λa. *vec-smult a* (*vec B*)) (*vec A*)))

abbreviation *tensor-prod-otimes* :: *'a tensor* $\Rightarrow$ *'a tensor* $\Rightarrow$ *'a tensor* (**infixl** $\otimes$ *70*)
where $A \otimes B \equiv A * B$

lemma *vec-tensor-prod*[*simp*]: *vec* ($A \otimes B$) = *concat* (*map* (λa. *vec-smult a* (*vec B*)) (*vec A*)) (**is** *?V*)
and *dims-tensor-prod*[*simp*]: *dims* ($A \otimes B$) = *dims A* @ *dims B* (**is** *?D*)
proof −
have *length* (*concat* (*map* (λa. *vec-smult a* (*vec B*)) (*vec A*))) = *prod-list* (*dims A* @ *dims B*)
proof −
have $\bigwedge$*xs*. *xs* $\in$ *set* (*map* (λa. *vec-smult a* (*vec B*)) (*vec A*)) $\Longrightarrow$ *length xs* = *length* (*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 *concat* (*map f* (*concat xss*)) = *concat* (*map* (λ*xs*. *concat* (*map f xs*)) *xss*)
by (*induction xss*; *auto*)

lemma *subtensor-combine-tensor-prod*:
assumes $\bigwedge$*A*. *A*$\in$*set As* $\Longrightarrow$ *dims A* = *ds*
shows *subtensor-combine ds As* $\otimes$ *B* = *subtensor-combine* (*ds* @ *dims B*) (*map* (λA. $A \otimes B$) *As*)
proof −
let *?f* = λa. *vec-smult a* (*Tensor.vec B*)
let *?xss* = *map Tensor.vec As*
have *1*:*prod-list* (*length As* # *ds*) = *length* (*concat ?xss*) **by** (*metis assms length-vec subtensor-combine-dims subtensor-combine-vec*)
have *2*:$\bigwedge$*A*. *A*$\in$*set As* $\Longrightarrow$ *prod-list* (*dims A* @ *dims B*) = *length* (*concat* (*map ?f* (*Tensor.vec A*)))
by (*metis dims-tensor-prod length-vec vec-tensor-prod*)
have *3*: *length As* # *ds* @ *dims B* = (*length* (*map* (λA. *tensor-from-vec* (*dims*

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

lemma *subtensor-tensor-prod*:
assumes *dims A* ≠ [] **and** *i* < *hd* (*dims A*)
shows *subtensor* (*A* ⊗ *B*) *i* = *subtensor A i* ⊗ *B*
using *assms* **proof** (*induction A rule*:*subtensor-combine-induct*)
case *order-0*
then show *?case* **by** *auto*
next
case (*order-step As ds*)
have *1*:*i* < *length* (*map* (λ*A*. *A* ⊗ *B*) *As*) **using** *order-step* **by** (*simp add*: *order-step.hyps order-step.prems*(*1*))
have *2*:(⋀*A*. *A* ∈ *set* (*map* (λ*A*. *A* ⊗ *B*) *As*) ⟹ *dims A* = *ds* @ *dims B*) **using** *order-step* **by** *auto*
have *subtensor* (*subtensor-combine ds As* ⊗ *B*) *i* = *subtensor* (*subtensor-combine* (*ds* @ *dims B*) (*map* (λ*A*. *A* ⊗ *B*) *As*)) *i*
using *subtensor-combine-tensor-prod order-step* **by** *metis*
also have ... = *As* ! *i* ⊗ *B*
using *order-step subtensor-subtensor-combine*[*of* (*map* (λ*A*. *A* ⊗ *B*) *As*) *ds* @ *dims B i*] *1 2* **by** *auto*
also have ... = *subtensor* (*subtensor-combine ds As*) *i* ⊗ *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* ⊲ *dims A* **and** *is2-valid*:*is2* ⊲ *dims B*
shows *lookup* (*A* ⊗ *B*) (*is1* @ *is2*) = *lookup A is1* ∗ *lookup B is2*
using *assms* **proof** (*induction A arbitrary*:*is1 rule*:*subtensor-induct*)
case (*order-0 A is1*)

then obtain *a* **where** *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* ⊗ *B* = *a* · *B* **unfolding** *tensor-prod-def smult-def* **using** *order-0* **by** *simp*
moreover have *lookup A* [] = *a* **by** (*simp add*: ‹*Tensor.vec A* = [*a*]› *lookup-def order-0.hyps*)
ultimately have *lookup* (*A* ⊗ *B*) (*is2*) = *a* ∗ *lookup B is2* **by** (*simp add*: *lookup-smult is2-valid*)
then show *?case* **using** ‹*lookup A* [] = *a*› *null-rec*(*1*) *order-0.hyps order-0.prems*(*1*) **by** *auto*
next
case (*order-step A is1*)
then obtain *i is1′* **where** *i* # *is1′* = *is1* **by** *blast*
have *lookup* (*subtensor A i* ⊗ *B*) (*is1′* @ *is2*) = *lookup* (*subtensor A i*) *is1′* ∗ *lookup B is2* **using** *order-step*
by (*metis* ‹*i* # *is1′* = *is1*› *dims-subtensor list.sel*(*1*) *list.sel*(*3*) *valid-index-dimsE*)
then show *lookup* (*A* ⊗ *B*) (*is1* @ *is2*) = *lookup A is1* ∗ *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.prems*(*1*) *valid-index-append valid-index-dimsE*
by *metis*
qed

lemma *valid-index-split*:
assumes *is* ⊲ *ds1* @ *ds2*
obtains *is1 is2* **where** *is1* @ *is2* = *is is1* ⊲ *ds1 is2* ⊲ *ds2*
proof
assume *a*: ⋀*is1 is2*. *is1* @ *is2* = *is* ⟹ *is1* ⊲ *ds1* ⟹ *is2* ⊲ *ds2* ⟹ *thesis*
have *length-is*:*length is* = *length ds1* + *length ds2* **using** *valid-index-length* **using** *assms* **by** *auto*
show *take* (*length ds1*) *is* ⊲ *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 *drop* (*length ds1*) *is* ⊲ *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 *take* (*length ds1*) *is* @ *drop* (*length ds1*) *is* = *is* **using** *length-is* **by** *auto*
qed

instance proof
fix *A B C*::′*a*::*ring tensor*
show (*A* ⊗ *B*) ⊗ *C* = *A* ⊗ (*B* ⊗ *C*)
proof (*rule tensor-lookup-eqI*, *simp*)

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

end

lemma *tensor-prod-distr-left*:
assumes *dims A* = *dims B*
shows (*A* + *B*) ⊗ *C* = (*A* ⊗ *C*) + (*B* ⊗ *C*)
proof −
have ⋀*is*. *is* ⊲ *dims A* @ *dims C* ⟹ *lookup* ((*A* + *B*) ⊗ *C*) *is* = *lookup* (*A* ⊗ *C* + *B* ⊗ *C*) *is*
proof −
fix *is* **assume** *is* ⊲ *dims A* @ *dims C*
obtain *is1 is2* **where** *is* = *is1* @ *is2 is1* ⊲ *dims A is2* ⊲ *dims C* **using** *valid-index-split* **using** ‹*is* ⊲ *dims A* @ *dims C*› **by** *blast*
then show *lookup* ((*A* + *B*) ⊗ *C*) *is* = *lookup* ((*A* ⊗ *C*) + (*B* ⊗ *C*)) *is*
using *lookup-plus*
‹*is1* ⊲ *dims A*› ‹*is2* ⊲ *dims C*› *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distribs*(*2*) *valid-index-append*
by *fastforce*
qed
moreover have *tensor-from-lookup* (*dims A* @ *dims C*) (*lookup* ((*A* + *B*) ⊗ *C*)) = (*A* + *B*) ⊗ *C*
tensor-from-lookup (*dims A* @ *dims C*) (*lookup* ((*A* ⊗ *C*) + (*B* ⊗ *C*))) = (*A* ⊗ *C*) + (*B* ⊗ *C*)
by (*metis* (*no-types*, *lifting*) *assms plus-dim1 dims-tensor-prod tensor-lookup*)+
ultimately show *?thesis* **using** *tensor-from-lookup-eqI*
by (*metis* ‹⋀*is*. *is* ⊲ *dims A* @ *dims C* ⟹ *lookup* ((*A* + *B*) ⊗ *C*) *is* = *lookup* (*A* ⊗ *C* + *B* ⊗ *C*) *is*›)
qed

lemma *tensor-prod-distr-right*:
assumes *dims A* = *dims B*
shows $C \otimes (A + B) = (C \otimes A) + (C \otimes B)$
proof −
  have $\bigwedge$*is*. *is* ◁ *dims C* @ *dims A* ⟹ *lookup* $(C \otimes (A + B))$ *is* = *lookup* $(C \otimes A + C \otimes B)$ *is*
  proof −
    fix *is* **assume** *is* ◁ *dims C* @ *dims A*
    obtain *is1 is2* **where** *is* = *is1* @ *is2 is1* ◁ *dims C is2* ◁ *dims A* **using** *valid-index-split* **using** ‹*is* ◁ *dims C* @ *dims A*› **by** *blast*
    then show *lookup* $(C \otimes (A + B))$ *is* = *lookup* $((C \otimes A) + (C \otimes B))$ *is*
     using *lookup-plus*
      using ‹*is2* ◁ *dims A*› ‹*is1* ◁ *dims C*› *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distribs*(*1*) *valid-index-append*
     by *fastforce*
  qed
  moreover have *tensor-from-lookup* (*dims C* @ *dims A*) (*lookup* $(C \otimes (A + B))$) = $C \otimes (A + B)$
      tensor-from-lookup (*dims C* @ *dims A*) (*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* ‹$\bigwedge$*is*. *is* ◁ *dims C* @ *dims A* ⟹ *lookup* $(C \otimes (A + B))$ *is* = *lookup* $(C \otimes A + C \otimes B)$ *is*›)
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* [] (λ-. *1*)
   unfolding *tensor-one-def* **by** (*rule tensor-eqI*; *simp-all add*: *tensor-from-lookup-def*)

  instance proof
    fix *A*::′*a*::*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* [] (λ-. *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*::*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* [] (λ-. *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 $a::'a::comm\text{-}ring\text{-}1$
shows $a \cdot (A \otimes B) = (a \cdot A) \otimes B$
proof (*rule tensor-lookup-eqI*)
show *dims* $(a \cdot (A \otimes B))$ = *dims* $((a \cdot A) \otimes B)$ **by** *simp*
fix *is* **assume** *is* ⊲ *dims* $(a \cdot (A \otimes B))$
then have *is* ⊲ *dims* $(A \otimes B)$ **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* $(a \cdot A)$ **by** *auto*
show *lookup* $(a \cdot (A \otimes B))$ *is* = *lookup* $(a \cdot A \otimes B)$ *is*
using *lookup-tensor-prod*[*OF* ‹*is1* ⊲ *dims A*› ‹*is2* ⊲ *dims B*›] *lookup-tensor-prod*[*OF* ‹*is1* ⊲ *dims* $(a \cdot A)$› ‹*is2* ⊲ *dims B*›]
lookup-smult[*OF* ‹*is* ⊲ *dims* $(A \otimes B)$›] *lookup-smult*[*OF* ‹*is1* ⊲ *dims A*›] ‹*is* = *is1* @ *is2*› **by** *simp*
qed

lemma *smult-prod-extract2*:
fixes $a::'a::comm\text{-}ring\text{-}1$
shows $a \cdot (A \otimes B) = A \otimes (a \cdot B)$
proof (*rule tensor-lookup-eqI*)
show *dims* $(a \cdot (A \otimes B))$ = *dims* $(A \otimes (a \cdot B))$ **by** *simp*
fix *is* **assume** *is* ⊲ *dims* $(a \cdot (A \otimes B))$
then have *is* ⊲ *dims* $(A \otimes B)$ **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* $(a \cdot B)$ **by** *auto*
show *lookup* $(a \cdot (A \otimes B))$ *is* = *lookup* $(A \otimes (a \cdot B))$ *is*
using *lookup-tensor-prod*[*OF* ‹*is1* ⊲ *dims A*› ‹*is2* ⊲ *dims B*›] *lookup-tensor-prod*[*OF* ‹*is1* ⊲ *dims A*› ‹*is2* ⊲ *dims* $(a \cdot B)$›]
lookup-smult[*OF* ‹*is* ⊲ *dims* $(A \otimes B)$›] *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** $A = a \cdot 1$
proof
assume $(\bigwedge a.\ A = a \cdot 1 \Longrightarrow thesis)$
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 *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 $A = hd\ (vec\ A) \cdot 1$ **using** ‹*vec* $A = [a]$› **by** *auto*
qed

lemma *smult-1*:
fixes A::$'a$::*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*]: $A \otimes$ *tensor0 ds* = *tensor0* (*dims* A @ *ds*)
proof (*rule tensor-lookup-eqI*,*simp*)
fix *is* **assume** *is* ⊲ *dims* ($A \otimes$ *tensor0 ds*)
then obtain *is1 is2* **where** *is1* ⊲ *dims* A *is2* ⊲ *dims* (*tensor0 ds*) *is* = *is1* @ *is2*
by (*metis dims-tensor0 dims-tensor-prod valid-index-split*)
then show *lookup* ($A \otimes$ *tensor0 ds*) *is* = *lookup* (*tensor0* (*dims* A @ *ds*)) *is*
by (*metis* (*no-types*, *lifting*) ‹*is* ⊲ *dims* ($A \otimes$ *tensor0 ds*)› *dims-tensor0 dims-tensor-prod lookup-tensor0 lookup-tensor-prod mult-zero-right*)
qed

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

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

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

lemma *subtensor-decomposition*:
assumes *dims A* ≠ []
shows *listsum* (*dims A*) (*map* (λ*i*. *unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*) [*0*..<*hd* (*dims A*)]) = *A* (**is** *?LS* = *A*)
proof −
  let *?f* = λ*i*. *unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*
  have *correct-dims*:⋀*B*. *B* ∈ *set* (*map ?f* [*0*..<*hd* (*dims A*)]) ⟹ *dims B* = *dims A*
  proof−
    fix *B*
    assume *B* ∈ *set* (*map ?f* [*0*..<*hd* (*dims A*)])
    then obtain *i* **where** *B*:*B* = *?f i* **and** *i*<*hd* (*dims A*) **by** *auto*
     then have *dims* (*subtensor A i*) = *tl* (*dims A*) **using** *dims-subtensor* **using** *assms* **by** *blast*
    then show *dims B* = *dims A* **unfolding** *B*
      by (*metis append-Cons assms dims-tensor-prod dims-unit-vec list.exhaust-sel self-append-conv2*)
  qed
  have ⋀*j*. *j* < *hd* (*dims A*) ⟹ *subtensor ?LS j* = *subtensor A j*
  proof −
    fix *j*
    assume *j* < *hd* (*dims A*)
   have *1*:*subtensor ?LS j* = *listsum* (*tl* (*dims A*)) (*map* (λ*A*. *subtensor A j*) (*map ?f* [*0*..<*hd* (*dims A*)]))
      using *subtensor-listsum*[*of* (*map* (λ*i*. *?f i*) [*0*..<*hd* (*dims A*)]) *dims A j*, *OF correct-dims assms* ‹*j* < *hd* (*dims A*)›]
      by *linarith*
     also have ... = *listsum* (*tl* (*dims A*)) (*map* (λ*i*. *subtensor* (*?f i*) *j*) [*0*..<*hd* (*dims A*)])
    proof −
       have *map* (λ*A*. *subtensor A j*) (*map ?f* [*0*..<*hd* (*dims A*)]) = *map* (λ*i*. *subtensor* (*?f i*) *j*) [*0*..<*hd* (*dims A*)]
        unfolding *map-map*[*of* (λ*A*. *subtensor A j*) *?f* [*0*..<*hd* (*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*∈*set Bs* ⟹ *order B* = *1 a* · *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* · *prod-list* [] **using** *order-0-multiple-of-one* **using** *prod-list.Nil* **by** *auto*
then show *?thesis* **using** *length-pos-if-in-set order1.prems* **by** *fastforce*
next
case *False*
then have *order A* = *1* **using** *order1* **by** *linarith*
then have *A* = *1* · *prod-list* [*A*] **by** (*simp add*: *smult-1*)
then show *?thesis* **by** (*metis* ‹*order A* = *1*› *length-greater-0-conv length-pos-if-in-set order1.prems set-ConsD*)
qed
next
case (*higher-order A B*)
then obtain *Bs b* **where** ($\bigwedge B'$. *B'* ∈ *set Bs* ⟹ *order B'* = *1*) *b* · *prod-list Bs* = *B* **by** *metis*
then have ($\bigwedge B$. *B* ∈ *set* (*A* # *Bs*) ⟹ *order B* = *1*) **using** *higher-order* **by** *auto*
have *A* ⊗ *B* = *b* · (*A* ⊗ *prod-list Bs*) **using** *smult-prod-extract2* ‹*b* · *prod-list Bs* = *B*› **by** *metis*
then show *?case* **by** (*metis* ‹$\bigwedge$*Ba*. *Ba* ∈ *set* (*A* # *Bs*) ⟹ *order Ba* = *1*› *higher-order.prems prod-list.Cons*)
qed

inductive *cprank-max* :: *nat* ⇒ *'a*::*ring-1 tensor* ⇒ *bool* **where**
cprank-max0: *cprank-max 0* (*tensor0 ds*) |
cprank-max-Suc: *dims A* = *dims B* ⟹ *cprank-max1 A* ⟹ *cprank-max j B* ⟹ *cprank-max* (*Suc j*) (*A*+*B*)

lemma *cprank-max1*: *cprank-max1 A* ⟹ *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* ⟹ *cprank-max j B* ⟹ *dims A* = *dims B* ⟹ *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 $\bigwedge A.\ A{\in}set\ As \Longrightarrow dims\ A = ds$
and $\bigwedge A.\ A{\in}set\ As \Longrightarrow cprank\text{-}max\ n\ A$
shows *cprank-max* (*n*∗*length As*) (*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 *cprank-max n A*
obtains *BS* **where** $(\bigwedge B.\ B{\in}set\ BS \Longrightarrow cprank\text{-}max1\ B)$ **and** $(\bigwedge B.\ B{\in}set\ BS \Longrightarrow dims\ A = dims\ B)$ **and** *listsum* (*dims A*) *BS* = *A* **and** *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.prems* **by** *fastforce*
next
 case (*cprank-max-Suc A B j*)
 then obtain *BS* **where** *BS-def*:$(\bigwedge B.\ B \in set\ BS \Longrightarrow cprank\text{-}max1\ B)$ $(\bigwedge B'.\ B' \in set\ BS \Longrightarrow dims\ B' = dims\ B)$
 listsum (*dims B*) *BS* = *B length BS* = *j* **by** *metis*
 then have *listsum* (*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.prems set-ConsD*
 by (*metis plus-dim1 cprank-max-Suc.hyps*(*1*))
qed

lemma *cprank-maxI*:
assumes $\bigwedge B.\ B{\in}set\ BS \Longrightarrow cprank\text{-}max1\ B$
and $\bigwedge B.\ B{\in}set\ BS \Longrightarrow dims\ B = ds$
shows *cprank-max* (*length BS*) (*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*: $cprank\text{-}max\ 0\ A \Longrightarrow A = tensor0\ (dims\ A)$ **by** (*metis* *dims-tensor0 length-0-conv cprank-max0 cprank-maxE*)

lemma *listsum-prod-distr-right*:
assumes ($\bigwedge C$. $C \in set\ CS \Longrightarrow dims\ C = ds$)
shows $A \otimes listsum\ ds\ CS = listsum\ (dims\ A\ @\ ds)\ (map\ (\lambda C.\ A \otimes C)\ CS)$
using *assms* **proof** (*induction CS*)
  case *Nil*
  then show *?case* **by** (*simp add*:*listsum-Nil*)
next
  case (*Cons C CS*)
  then have $dims\ C = dims\ (listsum\ ds\ CS)$ **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.prems list.set-intros*(*2*) **by** *fastforce*
qed

lemma *cprank-max-prod-order1*:
assumes *order A* = *1*
and *cprank-max n B*
shows *cprank-max n* ($A \otimes B$)
proof −
  obtain *CS* **where** ($\bigwedge C$. $C \in set\ CS \Longrightarrow cprank\text{-}max1\ C$)
    and ($\bigwedge C$. $C \in set\ CS \Longrightarrow dims\ C = dims\ B$)
    and *listsum* (*dims B*) *CS* = *B*
    and *length CS* = *n*
    using *assms*(*2*) *cprank-maxE* **by** *metis*
  define CS' **where** $CS' = map\ (\lambda C.\ A \otimes C)\ CS$
  then have $\bigwedge C'$. $C' \in set\ CS' \Longrightarrow cprank\text{-}max1\ C'$
    using *assms*(*1*) *higher-order* ‹$\bigwedge C$. $C \in set\ CS \Longrightarrow cprank\text{-}max1\ C$› *imageE set-map* **by** *auto*
  have *listsum* (*dims A* @ *dims B*) $CS' = A \otimes B$ **using** CS'*-def* ‹*Tensor-Plus.listsum* (*dims B*) *CS* = *B*›
    using ‹$\bigwedge Ca$. $Ca \in set\ CS \Longrightarrow dims\ Ca = dims\ B$› *listsum-prod-distr-right* **by** *fastforce*
  then show *?thesis* **by** (*metis* (*mono-tags*, *lifting*) CS'*-def* ‹$\bigwedge C'$. $C' \in set\ CS' \Longrightarrow cprank\text{-}max1\ C'$› ‹$\bigwedge Ca$. $Ca \in set\ CS \Longrightarrow dims\ Ca = dims\ B$› ‹*length CS* = *n*› *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** $Bs = map\ (\lambda i.\ unit\text{-}vec\ (hd\ (dims\ A))\ i \otimes subtensor\ A\ i)$ [*0*..<*hd* (*dims A*)]

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

definition *cprank* :: *'a::ring-1 tensor* ⇒ *nat* **where**
$cprank\ A = (LEAST\ n.\ cprank\text{-}max\ n\ A)$

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

lemma *cprank-max-cprank*: $cprank\text{-}max\ (cprank\ A)\ A$
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* ⇒ *nat list* ⇒ *nat* **where**
digit-decode [] [] = *0* |
digit-decode (*d* # *ds*) (*i* # *is*) = *i* + *d* ∗ *digit-decode ds is*

fun *digit-encode* :: *nat list* ⇒ *nat* ⇒ *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* ⊲ *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* ⊲ *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* ⊲ *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*) ⊲ *ds*
    by (*rule Cons*)
  moreover have *d* > *0*

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

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

lemma *digit-encode-0*:
$prod\text{-}list\ ds\ dvd\ a \Longrightarrow digit\text{-}encode\ ds\ a = replicate\ (length\ ds)\ 0$
proof (*induction ds arbitrary*:*a*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons d ds a*)
then have $prod\text{-}list\ ds\ dvd\ (a\ div\ d)$ **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 $is1 \lhd (nths\ ds\ A)$
and $is2 \lhd (nths\ ds\ (-A))$
shows $weave\ A\ is1\ is2 \lhd ds$
and $nths\ (weave\ A\ is1\ is2)\ A = is1$
and $nths\ (weave\ A\ is1\ is2)\ (-A) = is2$
proof −
have *length-ds*: $length\ is1 + length\ is2 = length\ ds$
using *valid-index-length*[*OF assms*(*1*)] *valid-index-length*[*OF assms*(*2*)]
length-weave weave-complementary-nthss **by** *metis*
have *1*:$length\ is1 = card\ \{i \in A.\ i < length\ is1 + length\ is2\}$ **unfolding** *length-ds*
using *length-nths′ assms*(*1*) *valid-index-length* **by** *auto*
have *2*:$length\ is2 = card\ \{i \in -A.\ i < length\ is1 + length\ is2\}$ **unfolding** *length-ds*
using *length-nths′*[*of ds −A*] *assms*(*2*) *valid-index-length* **by** *auto*
show $nths\ (weave\ A\ is1\ is2)\ A = is1$ $nths\ (weave\ A\ is1\ is2)\ (-A) = is2$ **using** *nths-weave*[*OF 1 2*] **by** *blast+*
then have $nths\ (weave\ A\ is1\ is2)\ A \lhd (nths\ ds\ A)$
$nths\ (weave\ A\ is1\ is2)\ (-A) \lhd (nths\ ds\ (-A))$ **using** *assms* **by** *auto*
then show $weave\ A\ is1\ is2 \lhd ds$ **using** *list-all2-nths valid-index-list-all2-iff* **by** *blast*
qed

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

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

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

lemma *dims-matricize*:
dim-row (*matricize rmodes T*) = *prod-list* (*nths* (*Tensor.dims T*) *rmodes*)
dim-col (*matricize rmodes T*) = *prod-list* (*nths* (*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* = (*nths* (*Tensor.dims T*) *rmodes*)
  let *?cds* = (*nths* (*Tensor.dims T*) (−*rmodes*))

have *decode-r*: *digit-decode ?rds* (*nths is rmodes*) < *prod-list ?rds*
by (*simp add*: ‹*is* ◁ *Tensor.dims T*› *valid-index-nths digit-decode-lt*)
have *decode-c*: *digit-decode ?cds* (*nths is* (−*rmodes*)) < *prod-list ?cds*
by (*simp add*: ‹*is* ◁ *Tensor.dims T*› *valid-index-nths digit-decode-lt*)
have (*matricize rmodes T*) $$
(*digit-decode ?rds* (*nths is rmodes*),
digit-decode ?cds (*nths 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* (*nths ds rmodes*)
and *dim-col A* = *prod-list* (*nths 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* (*nths ds rmodes*) *r* ◁ *nths ds rmodes* **and**
valid2:*digit-encode* (*nths ds* (− *rmodes*)) *c* ◁ *nths 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* (*nths* (*Tensor.dims* (*dematricize rmodes A ds*)) *rmodes*) *r*)
(*digit-encode* (*nths* (*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* (*nths* (*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* (*nths* (*Tensor.dims* (*dematricize rmodes A ds*)) (− *rmodes*))

by (*metis* ‹*dim-col* (*matricize rmodes* (*dematricize rmodes A ds*)) = *dim-col A*› *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* (*nths* (*Tensor*.*dims A*) *I*) *j* < *prod-list* (*nths* (*Tensor*.*dims A*) (−*I*)) **and**
ij-le3:*i* < *prod-list* (*nths* (*Tensor*.*dims B*) *I*) *j* < *prod-list* (*nths* (*Tensor*.*dims B*) (−*I*)) **and**
ij-le4:*i* < *prod-list* (*nths* (*Tensor*.*dims* (*A* + *B*)) *I*) *j* < *prod-list* (*nths* (*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*) $$ (*i*, *j*) = *matricize I* (*A* + *B*) $$ (*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*))
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 index-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 ⋀*B*. *B*∈*set Bs* ⟹ *Tensor.order B* = *1*
assumes *is* ◁ *dims* (*prod-list Bs*)
shows *lookup* (*prod-list Bs*) *is* = *prod-list* (*map* (λ(*i*,*B*). *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 *valid1*:[*i*] ◁ *dims B*
using ‹*is′* ◁ *dims* (*prod-list* (*B* # *Bs*))›[*unfolded prod-list.Cons dims-tensor-prod* ‹*is′* = *i* # *is*›]
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 *valid2*:*is* ◁ *dims* (*prod-list Bs*)
using ‹*is′* ◁ *dims* (*prod-list* (*B* # *Bs*))›[*unfolded prod-list.Cons dims-tensor-prod* ‹*is′* = *i* # *is*›] ‹*Tensor.order B* = *1*›
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** ‹*is'* = *i* # *is*› *List.zip-Cons-Cons List.list.map*(*2*) *prod-list.Cons*
lookup-tensor-prod[*OF valid1 valid2*, *simplified*] **by** (*simp add*: *Cons.IH Cons.prems*(*1*) *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* ∈ *set Bs* ⟹ *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* (λ(*i*,*B*). *lookup B* [*i*]) (*zip ris* (*nths Bs I*)))
for *ris*
define *col-factor*
where *col-factor cis* = *prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*zip cis* (*nths Bs* (−*I*))))
for *cis*
have $\bigwedge is$. *is* ◁ *dims A* ⟹ *lookup A is* = *row-factor* (*nths is I*) ∗ *col-factor* (*nths is* (−*I*))
proof −
fix *is* **assume** *is* ◁ *dims A*
then have *lookup A is* = *a* ∗ (*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*zip is Bs*)))
using *lookup-order1-prod*[*OF* ‹$\bigwedge B$. *B* ∈ *set Bs* ⟹ *Tensor.order B* = *1*›] *lookup-smult*
using ‹*a* · *prod-list Bs* = *A*› *dims-smult* **by** *fastforce*
also have ... = *a* ∗ (*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*nths* (*zip is Bs*) *I*))) ∗
(*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*nths* (*zip is Bs*) (−*I*))))
using *prod-list-complementary-nthss* **by** *auto*
also have ... = *row-factor* (*nths is I*) ∗ *col-factor* (*nths is* (−*I*))
using *nths-zip row-factor-def col-factor-def* **by** *metis*
finally show *lookup A is* = *row-factor* (*nths is I*) ∗ *col-factor* (*nths is* (−*I*)) **.**
qed
define *row-factor'*
where *row-factor' r* = *row-factor* (*digit-encode* (*nths* (*Tensor.dims A*) *I*) *r*) **for** *r*
define *col-factor'*
where *col-factor' c* = *col-factor* (*digit-encode* (*nths* (*Tensor.dims A*) (−*I*)) *c*) **for** *c*
have $\bigwedge r\ c$. *r*<*dim-row* (*matricize I A*) ⟹ *c*<*dim-col* (*matricize I A*) ⟹ *matricize 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 (nths (Tensor.dims A) I) r)
    (digit-encode (nths (Tensor.dims A) (−I)) c)
  ) unfolding dims-matricize unfolding matricize-def by simp
  also have ... = row-factor' r ∗ col-factor' c
   using ‹⋀is. is ⊲ dims A ⟹ lookup A is = row-factor (nths is I) ∗ col-factor
(nths is (− I))›
    valid-index-weave[OF
   digit-encode-valid-index[OF ‹r < dim-row (matricize I A)›[unfolded dims-matricize]]
   digit-encode-valid-index[OF ‹c < dim-col (matricize I A)›[unfolded 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) tensor
assumes cprank-max r A
shows mrank (matricize I A) ≤ r
using assms
proof (induction rule:cprank-max.induct)
 fix ds :: nat list
 have matricize I (tensor0 ds) = 0m (dim-row (matricize I (tensor0 ds))) (dim-col
(matricize I (tensor0 ds)))
  using matricize-0 by auto
 then show mrank (matricize I (tensor0 ds)) ≤ 0
  using eq-imp-le vec-space.rank-0I by metis
next
 fix A B::'a tensor and j::nat
 assume dims A = dims B
 assume cprank-max1 A
 assume mrank (matricize I B) ≤ j
 have mrank (matricize I A) ≤ 1 using ‹cprank-max1 A› matricize-cprank-max1
by auto
 have mrank (matricize I (A + B)) ≤ mrank (matricize I A) + mrank (matricize
I B)
  using matricize-add vec-space.rank-subadditive dims-matricize
  carrier-matI index-add-mat(2) ‹dims A = dims B› by metis
 then show mrank (matricize I (A + B)) ≤ Suc j
  using ‹mrank (matricize I A) ≤ 1› ‹mrank (matricize I B) ≤ j› by linarith
qed

lemma matrix-rank-le-cp-rank:
fixes A :: ('a::field) tensor
shows mrank (matricize I A) ≤ 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 $\bigwedge i.\ i < m * n \Longrightarrow a\ i = b\ i$
shows *extract-matrix a m n* = *extract-matrix b m n*
proof −
  have $\bigwedge i\ j.\ i < m \Longrightarrow j < n \Longrightarrow a\ (i*n + j) = b\ (i*n + j)$ **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 $\bigwedge x.\ x <$ *dim-row A* $*$ *dim-col A* $\Longrightarrow$ *f x* = *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 neq0-conv*)
  moreover have *k mod n* $< n$
    using ‹$n > 0$› **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 as m n*) = *m*
and *dim-col* (*extract-matrix as 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* ⇒ *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* ⇒ (*nat* × *nat*) *convnet* ⇒ *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* × *nat*) *convnet* ⇒ *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* (λ*i*. *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* (λ*i*. *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′* == (λ*m*. *output-size* (*remove-weights m*))
abbreviation *valid-net′* == (λ*m*. *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*) (λ*j*. *tensor-from-lookup ds* (λ*is*. (*A* $*_v$ (*map-vec* (λ*T*. *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* ($\lambda i.$ *w1* (i + *r0* ∗ *r1*)) = *insert-weights s m* ($\lambda i.$ *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.prems* ‹*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* ($\lambda i.$ *w1* (i + *count-weights s m1*)) = *insert-weights s m2* ($\lambda i.$ *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 $T \in set_v$ (*mat-tensorlist-mult A Ts ds*)
shows *Tensor.dims T* = *ds*
proof −
obtain *j* **where** *T* = *tensor-from-lookup ds* ($\lambda is.$ ($A *_v$ (*map-vec* ($\lambda T.$ *Tensor.lookup T is*) *Ts*)) $*j*)
using *vec-setE*[*OF assms*, *unfolded 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* ($\lambda i.$ *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*) = *input-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*) *tensors-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 $T \in set_v$ (*tensors-from-net m*)
shows *Tensor.dims T* = *input-sizes m*
using *assms* **proof** (*induction m arbitrary*: *T*)
  case (*Input M*)
  then obtain *j* **where** *T* = *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 < *dim-vec* (*tensors-from-net m1*) *i* < *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** $T = T1 \otimes T2$ $T1 \in set_v$ (*tensors-from-net m1*) *T2* $\in set_v$ (*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* ⇒ *nat list* ⇒ *real vec list* **where**
base-input m is = (*map* (λ(*n*, *i*). $unit_v$ *n i*) (*zip* (*input-sizes m*) *is*))

lemma *base-input-length*:
assumes *is* ⊲ *input-sizes m*
shows *input-sizes m* = *map dim-vec* (*base-input m is*)
proof (*rule nth-equalityI*)
  have *length* (*input-sizes m*) = *length is* **using** *assms valid-index-length* **by** *auto*
  then show *length* (*input-sizes m*) = *length* (*map dim-vec* (*base-input m is*)) **unfolding** *base-input-def* **by** *auto*
  {
    fix *i*
    assume *i*<*length* (*input-sizes m*)
    then have *map* (λ(*n*, *i*). $unit_v$ *n i*) (*zip* (*input-sizes m*) *is*) ! *i* = $unit_v$ (*input-sizes m* ! *i*) (*is* ! *i*)
      using ‹*length* (*input-sizes m*) = *length is*› **by** *auto*
    then have *input-sizes m* ! *i* = *map dim-vec* (*base-input m is*) ! *i* **unfolding** *base-input-def* **using** *index-unit-vec*(*3*)
      using ‹*i* < *length* (*input-sizes m*)› ‹*length* (*input-sizes m*) = *length* (*map dim-vec* (*base-input m is*))›
        base-input-def assms length-map nth-map valid-index-lt **by** (*simp add*: *input-sizes-remove-weights*)
  }
  then show ⋀*i*. *i* < *length* (*input-sizes m*) ⟹ *input-sizes m* ! *i* = *map dim-vec* (*base-input m is*) ! *i* **by** *auto*

qed

lemma *nth-mat-tensorlist-mult*:
assumes $\bigwedge A.\ A \in set_v\ Ts \Longrightarrow dims\ A = ds$
assumes $i < dim\text{-}row\ A$
assumes $dim\text{-}vec\ Ts = dim\text{-}col\ A$
shows $mat\text{-}tensorlist\text{-}mult\ A\ Ts\ ds\ \$\ i = listsum\ ds\ (map\ (\lambda j.\ (A\ \$\$\ (i,j)) \cdot Ts\ \$\ j)\ [0..<dim\text{-}vec\ Ts])$
(**is** - = *listsum ds ?Ts′*)
proof (*rule tensor-lookup-eqI*)
have *dims-Ts′*: $\bigwedge T.\ T \in set\ ?Ts' \Longrightarrow dims\ T = ds$
proof −
fix *T* **assume** *T∈set ?Ts′*
then obtain *k* **where** $T = ?Ts'\ !\ k$ **and** $k < length\ ?Ts'\ k < dim\text{-}vec\ Ts$ **using** *in-set-conv-nth* **by** *force*
show *dims T = ds* **unfolding** ‹*T = ?Ts′ ! k*› *nth-map*[*OF* ‹*k < length ?Ts′*›[*unfolded length-map*]]
using *assms*(*1*) ‹*k < dim-vec Ts*›
by (*simp add*: ‹$k < length\ (map\ (\lambda j.\ A\ \$\$\ (i,\ j) \cdot Ts\ \$\ j)\ [0..<dim\text{-}vec\ Ts])$› *vec-setI*)
qed
then show *dims-eq*: $dims\ (mat\text{-}tensorlist\text{-}mult\ A\ Ts\ ds\ \$\ i) = dims\ (Tensor\text{-}Plus.listsum\ ds\ (map\ (\lambda j.\ A\ \$\$\ (i,\ j) \cdot Ts\ \$\ j)\ [0..<dim\text{-}vec\ Ts]))$
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-valid*: $is \lhd dims\ (mat\text{-}tensorlist\text{-}mult\ A\ Ts\ ds\ \$\ i)$
then have $is \lhd ds$ **using** *dims-eq dims-Ts′ listsum-dims* **by** (*metis* (*no-types*, *lifting*))

have *summand-eq*: $\bigwedge j.\ j \in \{0\ ..<\ dim\text{-}vec\ Ts\} \Longrightarrow row\ A\ i\ \$\ j * (map\text{-}vec\ (\lambda T.\ Tensor.lookup\ T\ is)\ Ts)\ \$\ j = lookup\ (A\ \$\$\ (i,\ j) \cdot Ts\ \$\ j)\ is$
using *index-vec* ‹*i<dim-row A*› *row-def* ‹*dim-vec Ts = dim-col A*›
‹$is \lhd ds$› *assms*(*1*) *lookup-smult atLeastLessThan-iff index-map-vec*(*1*) *vec-setI* **by** *metis*

have $lookup\ (mat\text{-}tensorlist\text{-}mult\ A\ Ts\ ds\ \$\ i)\ is = (A *_v (map\text{-}vec\ (\lambda T.\ Tensor.lookup\ T\ is)\ Ts))\ \$\ i$
unfolding *mat-tensorlist-mult-def* **using** *lookup-tensor-from-lookup*[*OF* ‹$is \lhd ds$›] **using** ‹*i<dim-row A*› **by** *auto*
also have ... $= row\ A\ i \bullet map\text{-}vec\ (\lambda T.\ Tensor.lookup\ T\ is)\ Ts$
using ‹*i<dim-row A*› **by** *simp*
also have ... $= (\sum j \in \{0\ ..<\ dim\text{-}vec\ Ts\}.\ row\ A\ i\ \$\ j * (map\text{-}vec\ (\lambda T.\ Tensor.lookup\ T\ is)\ Ts)\ \$\ j)$
unfolding *scalar-prod-def nth-rows*[*OF* ‹*i<dim-row A*›] **by** *simp*
also have ... $= (\sum j \in \{0..<dim\text{-}vec\ Ts\}.\ lookup\ (A\ \$\$\ (i,\ j) \cdot Ts\ \$\ j)\ is)$ **using** *summand-eq* **by** *force*
also have ... $= (\sum A \leftarrow ?Ts'.\ lookup\ A\ is)$ **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* (λ(*n*, *i*). $unit_v$ *n i*) (*zip* (*input-sizes* (*Input M*)) *is*)) = $unit_v$ *M i* **using** ‹*is* = [*i*]› **by** *auto*
then show *?case* **using** *2* ‹*j* < *M*› *base-input-def* **by** (*simp add*: ‹*i* < *M*›)
next
case (*Conv A m j is*)
have *is-valid*:*is* ⊲ *input-sizes m* **using** *Conv.prems* **by** *simp*
have *valid-net*:*valid-net′ m* **using** *Conv.prems*(*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* (λ*T*. *Tensor.lookup T is*) (*tensors-from-net m*) =
evaluate-net m (*base-input m is*)
proof (*rule eq-vecI*)
show *equal-lengths*: *dim-vec* (*map-vec* (λ*T*. *lookup T 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 ⋀*i*. *i* < *dim-vec* (*evaluate-net m* (*base-input m is*)) ⟹
map-vec (λ*T*. *lookup T 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* (λ*T*. *lookup T 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.prems* **by** *auto*
then have (*tensors-from-net* (*Conv A m*)) $ *j* = *tensor-from-lookup* (*input-sizes m*)
(λ*is*. ($A *_v$ (*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* $\lhd$ *input-sizes m1* *is2* $\lhd$ *input-sizes m2*
by (*metis Pool.prems*(*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* $\lhd$ *input-sizes m1*› ‹*is2* $\lhd$ *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.prems*(*1*) *Pool.prems*(*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* ⊗ *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 ⊲ *Tensor.dims* (*tensors-from-net m1* $ *j*)
is2 ⊲ *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.prems*
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.prems* ‹*input-sizes m1* = *map dim-vec* (*base-input m1 is1*)› ‹*valid-net′ m1*›
output-size-correct **by** (*auto*,*metis Pool.prems*(*1*) *Pool.prems*(*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* ⇒ *real mat convnet* ⇒ *nat* ⇒ *real* **where**
extract-weights-Input: *extract-weights shared* (*Input M*) = (λ*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 ⋀*x*. $x <$ *count-weights True* (*remove-weights m*) ⟹ *f x* = *extract-weights True m x*
shows *m* = *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 *extract-matrix f* (*dim-row A*) (*dim-col A*) = *A*
   by (*simp add*: *extract-matrix-cong extract-matrix-flatten-matrix shared-weight-net-Conv.prems*)
  then show *?case*
    using *shared-weight-net-Conv.IH*[*of* (λi. *f* (i + *dim-row A* $*$ *dim-col A*))]
    using *shared-weight-net-Conv.prems* **by** *auto*
next
  case (*shared-weight-net-Pool m1 m2*)
  have *m1* = *insert-weights True* (*remove-weights m1*) *f*
    using *shared-weight-net-Pool.IH*(*1*) *shared-weight-net-Pool.prems* **by** *auto*
  have *m2* = *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.prems* **by** *fastforce*

then show *?case*
using ‹*m1* = *insert-weights True* (*remove-weights m1*) *f*› **by** *auto*
qed

lemma *insert-extract-weights-cong-unshared*:
assumes $\bigwedge x.\ x <$ *count-weights False* (*remove-weights m*) $\Longrightarrow f\ x =$ *extract-weights False m x*
shows *m* = *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 *extract-matrix f* (*dim-row A*) (*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.prems* **by** *auto*
next
case (*Pool m1 m2*)
then show *?case*
using *Pool.IH*(*1*) *Pool.IH*(*2*) *Pool.prems* **by** *auto*
qed

lemma *remove-insert-weights*:
shows *remove-weights* (*insert-weights s 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 *remove-weights* (*insert-weights s m w*) = *m* **using** *Conv.IH* **by** *blast*
then have *remove-weights* (*insert-weights s* (*Conv* (*r1*,*r2*) *m*) *w*) = *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*<*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*) *flatten-matrix-extract-matrix*)
    using *Conv.IH*[*of* - λ*i*. *w* (*i* + *r0* ∗ *r1*)] *Conv.prems*(*1*) *Conv.prems*(*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* ⟹ *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.prems balanced-net.simps convnet.distinct*(*1*) *convnet.distinct*(*5*) *convnet.inject*(*2*) *shared-weight-net-Conv*)
next
  case (*Pool m1 m2*)
  have *balanced-net m1 balanced-net m2*
    using *Pool.prems balanced-net.simps* **by** *blast*+
  have ⋀*x*. *x* < *count-weights True m1* ⟹
      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.prems 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.prems 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* $\{is.\ is \lhd ds\}$
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 $\{is.\ is \lhd d \# ds\} \subseteq (\bigcup i<d.\ \{i \# is\ |is.\ is \lhd ds\})$
  proof (*rule subsetI*)
    fix *is* **assume** $is \in \{is.\ is \lhd d \# ds\}$
    then have $is \lhd d \# ds$ **by** *auto*
    then obtain $i\ is'$ **where** $is = i \# is'$ **by** *blast*
    then have $i<d$ **using** ‹$is \lhd d \# ds$› **by** *blast*
    have $is' \lhd ds$ **using** ‹$is = i \# is'$› ‹$is \lhd d \# ds$› **by** *blast*
    have $is \in \{i \# is\ |is.\ is \lhd ds\}$ **by** (*simp add*: ‹$is = i \# is'$› ‹$is' \lhd ds$›)
    then show $is \in (\bigcup i<d.\ \{i \# is\ |is.\ is \lhd ds\})$ **using** ‹$i < d$› **by** *blast*
  qed
  moreover have $\bigwedge i.$ *finite* $\{i \# is\ |is.\ is \lhd ds\}$ **by** (*simp add*: *Cons.IH*)
  ultimately show *finite* $\{is.\ is \lhd d \# ds\}$ **by** (*simp add*: *finite-subset*)
qed

lemma *setsum-valid-index-split*:
$(\sum is \mid is \lhd ds1\ @\ ds2.\ f\ is) = (\sum is1 \mid is1 \lhd ds1.\ (\sum is2 \mid is2 \lhd ds2.\ f\ (is1\ @\ is2)))$
proof −
  have $1{:}((\lambda(is1, is2).\ is1\ @\ is2)$ ‘ $(\{is1.\ is1 \lhd ds1\} \times \{is2.\ is2 \lhd ds2\})) = \{is.\ is \lhd ds1\ @\ ds2\}$ (**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 $x \lhd ds1\ @\ ds2$ **by** *auto*
    then obtain $x1\ x2$ **where** $x = x1\ @\ x2\ x1 \lhd ds1\ x2 \lhd ds2$ **by** (*metis valid-index-split*)
    then have $(x1, x2) \in (\{is1.\ is1 \lhd ds1\} \times \{is2.\ is2 \lhd ds2\})$ **by** *auto*
    then show $x \in ?A$ **using** *imageI* ‹$x = x1\ @\ x2$› **by** *blast*
  qed
  have $2{:}$*inj-on* $(\lambda(is1, is2).\ is1\ @\ is2)\ (\{is1.\ is1 \lhd ds1\} \times \{is2.\ is2 \lhd ds2\})$
    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 :: *nat* ⇒ *real* **shows** *prod* g {..<n+m} = *prod* g {..<n} ∗ *prod* (λx. 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 < *output-size′ m*
shows *evaluate-net m inputs* \$ j
= (∑ *is*∈{*is*. *is* ⊲ *input-sizes m*}. (∏ k<*length inputs*. *inputs* ! k \$ (*is*!k)) ∗ *Tensor.lookup* (*tensors-from-net m* \$ j) *is*)
using *assms* **proof** (*induction m arbitrary*:*j is inputs*)
case (*Input M*)
then have *length inputs* = *1 input-sizes* (*Input M*) = [*M*] **by** *auto*
{
fix *is* **assume** *is* ⊲ *input-sizes* (*Input M*)
then have *length is* = *1* **by** (*simp add*: *valid-index-length*)
then have *is* = [*hd is*] **by** (*metis One-nat-def length-0-conv length-Suc-conv list.sel*(*1*))
then have *Tensor.lookup* (*tensors-from-net* (*Input M*) \$ j) *is* = (*if hd is*=j *then 1 else 0*)
by (*metis Input.prems*(*3*) ‹*input-sizes* (*Input M*) = [*M*]› ‹*is* ⊲ *input-sizes* (*Input M*)› *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 (∏ k<*length inputs*. *inputs* ! k \$ (*is* ! k)) ∗ *lookup* (*tensors-from-net* (*Input M*) \$ j) *is* =
(*if is*=[j] *then* (∏ k<*length inputs*. *inputs* ! k \$ (*is* ! k)) *else 0*) **using** ‹*is* = [*hd is*]› **by** *auto*
}
then have (∑ *is* | *is* ⊲ *input-sizes* (*Input M*). (∏ k<*length inputs*. *inputs* ! k \$ (*is* ! k)) ∗ *lookup* (*tensors-from-net* (*Input M*) \$ j) *is*)
= (∑ *is* | *is* ⊲ *input-sizes* (*Input M*). (*if is*=[j] *then* (∏ k<*length inputs*. *inputs* ! k \$ (*is* ! k)) *else 0*)) **by** *auto*
also have (∑ *is* | *is* ⊲ *input-sizes* (*Input M*). (*if is*=[j] *then* (∏ k<*length inputs*. *inputs* ! k \$ (*is* ! k)) *else 0*))
= (∏ k<*length inputs*. *inputs* ! k \$ ([j] ! k)) **unfolding** *sum.delta*[*OF finite-valid-index*]
using *Input.prems*(*3*) *valid-index.Cons valid-index.Nil* **by** *auto*
also have ... = *inputs* ! *0* \$ j **using** ‹*length inputs* = *1*› **by** (*simp add*: *prod.lessThan-Suc*)
also have ... = *evaluate-net* (*Input M*) *inputs* \$ j **unfolding** *evaluate-net.simps*

by (*metis* ‹*length inputs* = *1*› *hd-conv-nth list.size*(*3*) *zero-neq-one*)
finally show *?case* **by** *auto*
next
case (*Conv A m j*)
have *j* < *dim-row A* **using** *Conv.prems*(*3*) **by** *auto*
have *0*:$\bigwedge$*is. is* ◁ *input-sizes* (*Conv A m*) $\Longrightarrow$
($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net* (*Conv A m*) $ *j*) *is* =
($\sum$ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* ∗ (($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net m* $ *i*) *is*))
proof −
fix *is* **assume** *is* ◁ *input-sizes* (*Conv A m*)
then have *is* ◁ *input-sizes m* **by** *simp*
have *0*:*lookup* (*tensors-from-net* (*Conv A m*) $ *j*) *is* =
($\sum$ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* ∗ *lookup* (*tensors-from-net m* $ *i*) *is*)
unfolding *tensors-from-net.simps mat-tensorlist-mult-def index-vec*[*OF* ‹*j* < *dim-row A*›]
lookup-tensor-from-lookup[*OF* ‹*is* ◁ *input-sizes m*›] *index-mult-mat-vec*[*OF* ‹*j* < *dim-row A*›] *scalar-prod-def*
using *index-map-vec* **by** *auto*
show ($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net* (*Conv A m*) $ *j*) *is*
= ($\sum$ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* ∗ (($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*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.prems*(*1*) *convnet.distinct*(*1*) *convnet.distinct*(*5*) *convnet.inject*(*2*) *remove-weights.simps*(*2*) *valid-net.simps*)
have *map dim-vec inputs* = *input-sizes m* **by** (*simp add*: *Conv.prems*(*2*))
have *output-size′ m* = *dim-vec* (*tensors-from-net m*) **by** (*simp add*: ‹*valid-net′ m*› *output-size-correct-tensors*)
have *1*:$\bigwedge$*i. i*<*dim-vec* (*tensors-from-net m*) $\Longrightarrow$ ($\sum$ *is* | *is* ◁ *input-sizes* (*Conv A m*). (($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net m* $ *i*) *is*)) = *evaluate-net m inputs* $ *i* **unfolding** *input-sizes.simps*
using *Conv.IH* ‹*valid-net′ m*› ‹*map dim-vec inputs* = *input-sizes m*› ‹*output-size′ m* = *dim-vec* (*tensors-from-net m*)› **by** *simp*

have ($\sum$ *is* | *is* ◁ *input-sizes* (*Conv A m*). ($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net* (*Conv A m*) $ *j*) *is*)
= ($\sum$ *i* = *0*..<*dim-vec* (*tensors-from-net m*). ($\sum$ *is* | *is* ◁ *input-sizes* (*Conv A m*). *row A j* $ *i* ∗ (($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net m* $ *i*) *is*)))
using *Groups-Big.comm-monoid-add-class.sum.swap 0* **by** *auto*
also have ... = ($\sum$ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* ∗ ($\sum$ *is* | *is* ◁ *input-sizes* (*Conv A m*). (($\prod$*k*<*length inputs. inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net m* $ *i*) *is*)))
by (*simp add*: *sum-distrib-left*)
also have ... = ($\sum$ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* ∗ *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.prems*(*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.prems*(*1*) *Pool.prems*(*3*) *convnet.distinct*(*3*) *convnet.inject*(*3*) *convnet.simps*(*9*)
   output-size.simps(*3*) *remove-weights.simps*(*3*) *valid-net.simps*) **using** *Pool.prems* **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.prems*(*2*) *append-eq-conv-conj input-sizes.simps*(*3*) *inputs1-def take-map*)
    by (*metis Pool.prems*(*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 ($\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ ((is1\ @\ is2)\ !\ k)) = (\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k)) * (\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k))$
unfolding ‹*inputs = inputs1 @ inputs2*› *length-append prod-lessThan-split* **using** *1 2* **by** *metis*
}
note *1 = this*
{
fix *is1 is2* **assume** *is1* ◁ *input-sizes m1 is2* ◁ *input-sizes m2*
then have *is1* ◁ *dims* (*tensors-from-net m1* $ *j*) *is2* ◁ *dims* (*tensors-from-net m2* $ *j*)
using ‹*j* < *dim-vec* (*tensors-from-net m1*)› ‹*j* < *dim-vec* (*tensors-from-net m2*)› *dims-tensors-from-net vec-setI* **by** *force+*
have *lookup* (*tensors-from-net* (*Pool m1 m2*) $ *j*) (*is1* @ *is2*) = *lookup* (*tensors-from-net m1* $ *j*) *is1* ∗ *lookup* (*tensors-from-net m2* $ *j*) *is2*
unfolding *tensors-from-net.simps index-component-mult*[*OF* ‹*j* < *dim-vec* (*tensors-from-net m1*)› ‹*j* < *dim-vec* (*tensors-from-net m2*)›]
lookup-tensor-prod[*OF* ‹*is1* ◁ *dims* (*tensors-from-net m1* $ *j*)› ‹*is2* ◁ *dims* (*tensors-from-net m2* $ *j*)›] **by** *metis*
}
note *2 = this*

have *j-le-eval*:*j* < *dim-vec* (*evaluate-net m1* (*take* (*length* (*input-sizes m1*)) *inputs*))
j < *dim-vec* (*evaluate-net m2* (*drop* (*length* (*input-sizes m1*)) *inputs*))
using ‹*j* < *output-size′ m1*› ‹*map dim-vec inputs1* = *input-sizes m1*› ‹*valid-net′ m1*› *inputs1-def output-size-correct*
using ‹*j* < *output-size′ m2*› ‹*map dim-vec inputs2* = *input-sizes m2*› ‹*valid-net′ m2*› *inputs2-def* **by** *auto*
have ($\sum is \mid is \lhd input\text{-}sizes\ (Pool\ m1\ m2).\ (\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ (Pool\ m1\ m2)\ \$\ j)\ is)$
$= (\sum is1 \mid is1 \lhd input\text{-}sizes\ m1.\ \sum is2 \mid is2 \lhd input\text{-}sizes\ m2.$
$(\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k)) * (\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k)) *$
$lookup\ (tensors\text{-}from\text{-}net\ m1\ \$\ j)\ is1 * lookup\ (tensors\text{-}from\text{-}net\ m2\ \$\ j)\ is2)$
unfolding *input-sizes.simps setsum-valid-index-split* **using** *1 2*
using *mem-Collect-eq sum.cong* **by** (*simp add*: *mult.assoc*)
also have ... = ($\sum is1 \mid is1 \lhd input\text{-}sizes\ m1.\ (\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m1\ \$\ j)\ is1) *$
$(\sum is2 \mid is2 \lhd input\text{-}sizes\ m2.\ (\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m2\ \$\ j)\ is2)$
unfolding *sum-product* **by** (*rule sum.cong*, *metis*, *rule sum.cong*, *metis*, *simp*)
also have ... = *evaluate-net* (*Pool m1 m2*) *inputs* $ *j* **unfolding** *evaluate-net.simps index-component-mult*[*OF j-le-eval*]
using *Pool.IH*(*1*)[*OF* ‹*valid-net′ m1*› - ‹*j* < *output-size′ m1*›] *Pool.IH*(*2*)[*OF* ‹*valid-net′ m2*› - ‹*j* < *output-size′ m2*›]
using ‹*map dim-vec inputs1* = *input-sizes m1*› ‹*map dim-vec inputs2* = *input-sizes m2*› *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 $\bigwedge$*inputs. input-sizes m1 = map dim-vec inputs* $\Longrightarrow$ *evaluate-net m1 inputs = evaluate-net m2 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. base-input m1 is = base-input m2 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. base-input m1 is = base-input m2 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* ⇒ *nat* ⇒ *real mat*
where *id-matrix nr nc = mat nr nc* (λ(*r*, *c*). *if r=c then 1 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 *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 (*id-matrix nr* (*dim-vec v*) $*_v$ *v*) $ *i* = (*if i*<*dim-vec v then v* $ *i else 0*) (**is** *?a* $ *i* = *?b*)
proof −
have *?a* $ *i* = *row* (*id-matrix nr* (*dim-vec v*)) *i* · *v* **using** *index-mult-mat-vec assms id-matrix-dim* **by** *auto*
also have ... = *unit-vec* (*dim-vec v*) *i* · *v* **using** *row-id-matrix assms* **by** *auto*
also have ... = *?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* ⇒ *real vec*
where *all1-vec n* = *vec n* (λ*i*. *1*)

definition *all1-matrix*::*nat* ⇒ *nat* ⇒ *real mat*
where *all1-matrix nr nc* = *mat nr nc* (λ(*r*, *c*). *1*)

lemma *all1-matrix-dim*: *dim-row* (*all1-matrix nr nc*) = *nr dim-col* (*all1-matrix nr nc*) = *nc*
by (*simp-all add*: *all1-matrix-def*)

lemma *row-all1-matrix*:
assumes $i < nr$
shows *row* (*all1-matrix nr nc*) *i* = *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 *all1-vec* (*length xs*) · (*vec-of-list xs*) = *sum-list xs*
proof −
have *all1-vec* (*length xs*) · (*vec-of-list xs*) = (∑ *i* = *0*..<*dim-vec* (*vec-of-list xs*). *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 ... = (∑ *i* = *0*..<*length xs*. *xs* ! *i*) **using** *vec.abs-eq dim-vec vec-of-list.abs-eq* **by** (*metis sum.ivl-cong index-vec*)
also have ... = *sum-list xs* **by** (*simp add*: *sum-list-sum-nth*)
finally show *?thesis* **by** *auto*
qed

lemma *mult-all1-matrix*:
assumes $i < nr$
shows $((all1\text{-}matrix\ nr\ (dim\text{-}vec\ v)) *_v v)\ \$\ i = sum\text{-}list\ (list\text{-}of\text{-}vec\ v)$ (**is** $?a\ \$\ i$ $= sum\text{-}list\ (list\text{-}of\text{-}vec\ v)$)
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 ... $= sum\text{-}list\ (list\text{-}of\text{-}vec\ v)$ **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* ⇒ *nat* ⇒ *real mat*
where *copy-first-matrix nr nc* = *mat nr nc* (λ(*r*, *c*). *if c* = *0 then 1 else 0*)

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

lemma *row-copy-first-matrix*:
assumes $i < nr$
shows *row* (*copy-first-matrix nr nc*) *i* = *unit-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 $(copy\text{-}first\text{-}matrix\ nr\ (dim\text{-}vec\ v) *_v v)\ \$\ i = v\ \$\ 0$ (**is** $?a\ \$\ i = v\ \$\ 0$)
proof −
 have $?a\ \$\ i = row\ (copy\text{-}first\text{-}matrix\ nr\ (dim\text{-}vec\ v))\ i \cdot v$ **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* $\{a \in$ *Collect even*. $a < 2 * n\} = n$
proof (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  have $\{a \in$ *Collect even*. $a < 2 * Suc\ n\} =$ *insert* $(2*n)$ $\{a \in$ *Collect even*. $a < 2 * n\}$
    using *le-eq-less-or-eq less-Suc-eq-le subset-antisym* **by** *force*
  show *?case*
    unfolding ‹$\{a \in$ *Collect even*. $a < 2 * Suc\ n\} =$ *insert* $(2*n)$ $\{a \in$ *Collect even*. $a < 2 * n\}$›
    using *Suc card-insert-disjoint*[*of* $\{a \in$ *Collect even*. $a < 2 * n\}$ $2*n$]
    by (*simp add*: *finite-M-bounded-by-nat less-not-refl2*)
qed

lemma *card-odd*[*simp*]: *card* $\{a \in$ *Collect odd*. $a < 2 * n\} = n$
proof (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  have $\{a \in$ *Collect odd*. $a < 2 * Suc\ n\} =$ *insert* $(2*n+1)$ $\{a \in$ *Collect odd*. $a < 2 * n\}$
    using *le-eq-less-or-eq less-Suc-eq-le subset-antisym* **by** *force*
  show *?case*
    unfolding ‹$\{a \in$ *Collect odd*. $a < 2 * Suc\ n\} =$ *insert* $(2*n+1)$ $\{a \in$ *Collect odd*. $a < 2 * n\}$›
    using *Suc card-insert-disjoint*[*of* $\{a \in$ *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-model′ Y (*r* # *rs*) = *Pool* (*deep-model Y r rs*) (*deep-model Y r rs*) |
deep-model Y r rs = *Conv* (*Y*,*r*) (*deep-model′ r rs*)

abbreviation *deep-model′-l rs* == *deep-model′* (*rs*!*0*) (*tl rs*)
abbreviation *deep-model-l rs* == *deep-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* ≥ *1*
shows *input-sizes* (*deep-model′-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-sizes* (*deep-model′-l rs*) = [*last rs*]
by (*metis deep-model′.simps*(*1*) *input-sizes.simps*(*1*) *last.simps list.sel*(*3*))
then show *input-sizes* (*deep-model′-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-sizes* (*deep-model′-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.prems 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-model′-l rs* = *Pool* (*deep-model-l rs*) (*deep-model-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-sizes* (*deep-model′-l rs*) = *input-sizes* (*deep-model-l rs*) @ *input-sizes* (*deep-model-l rs*)
using *input-sizes.simps*(*3*) **by** *metis*
also have ... = *input-sizes* (*deep-model′-l* (*tl rs*)) @ *input-sizes* (*deep-model′-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* ^ (*length* (*tl rs*) − *1*)) (*last rs*) @ *replicate* (*2* ^ (*length* (*tl rs*) − *1*)) (*last rs*)
using *IH* **by** *auto*
also have ... = *replicate* (*2* ^ (*length rs* − *1*)) (*last rs*)
using *replicate-add*[*of 2* ^ (*length* (*tl rs*) − *1*) *2* ^ (*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* ≥ *2*
shows *input-sizes* (*deep-model-l rs*) = *replicate* (*2*^(*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*^(*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* ‹*j*<*nr*›, *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 *Convm* **where** *Convm = 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′ Convm* **by** *(simp add: assms id-matrix-dim valid-net.intros(2) Convm-def)*
have *base-input m is = base-input Convm is* **by** *(simp add: Convm-def base-input-def)*
have *i < output-size′ Convm* **unfolding** *Convm-def remove-weights.simps output-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 *Convm-def ‹i < output-size′ Convm› ‹valid-net′ Convm› 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 *Convm-def ‹base-input m is = base-input Convm 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 *Convm* **where** *Convm* = *Conv* (*copy-first-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′ Convm* **by** (*simp add*: *assms copy-first-matrix-dim valid-net.intros*(*2*)
Convm-def)
have *base-input m is* = *base-input Convm is* **by** (*simp add*: *Convm-def base-input-def*)
have *i* < *output-size′ Convm* **unfolding** *Convm-def remove-weights.simps out-
put-size.simps*
copy-first-matrix-dim **using** *assms* **by** *metis*
show *Tensor.lookup* (*?a* \$ *i*) *is* = *Tensor.lookup ?b is*
by (*metis Convm-def* ‹*base-input m is* = *base-input Convm is*› ‹*i* < *output-size′*
Convm›
‹*is* ⊲ *input-sizes m*› ‹*valid-net′ Convm*› *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* ‹*i*<*nr*›, *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* ◁ *Tensor.dims* (*?a* $ *i*)
then have *is* ◁ *input-sizes m*
using ‹*i* < *dim-vec ?a*› *dims-tensors-from-net input-sizes.simps*(*2*) **by** (*metis vec-setI*)
then have *is* ◁ *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*< *output-size′ Convm* **using** *Convm-def* ‹*i* < *dim-vec ?a*› ‹*valid-net′ Convm*›
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* ‹*valid-net′ Convm*› ‹*is* ◁ *input-sizes Convm*› ‹*i*< *output-size′ Convm*›]
by (*metis Convm-def*)
also have ... = *monoid-add-class.sum-list* (*list-of-vec* (*evaluate-net m* (*base-input Convm is*)))
using *evaluate-net-Conv-all1 Convm-def* ‹*is* ◁ *input-sizes Convm*› *assms base-input-length* ‹*i* < *nr*›
by *simp*
also have ... = *monoid-add-class.sum-list* (*list-of-vec* (*map-vec* (λ*A*. *lookup A is*)(*tensors-from-net m*)))
unfolding ‹*base-input Convm is* = *base-input m is*›
using *lookup-tensors-from-net*[*OF* ‹*valid-net′ m*› ‹*is* ◁ *input-sizes m*›]
base-input-length[*OF* ‹*is* ◁ *input-sizes m*›] *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* (λ*A*. *lookup A is*) (*tensors-from-net m*)] *index-map-vec*(*1*) *index-map-vec*(*2*)
by *force*
also have ... = *monoid-add-class.sum-list* (*map* (λ*A*. *lookup A is*) (*list-of-vec* (*tensors-from-net m*)))
using *eq-vecI*[*of vec-of-list* (*list-of-vec* (*map-vec* (λ*A*. *lookup A is*)(*tensors-from-net m*)))
vec-of-list (*map* (λ*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 ... = *Tensor.lookup ?b is* **using** *dims-tensors-from-net set-list-of-vec*
using *lookup-listsum*[*OF* ‹*is* ◁ *input-sizes m*›, *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 ⊗ *unit-vec M j* = *tensor-from-lookup* [*M*,*M*] (λ*is*. *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* ◁ *Tensor.dims* (*unit-vec M j* ⊗ *unit-vec M j*)
  then have *is* ◁ [*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*] ◁ *Tensor.dims* (*unit-vec M j*) [*i2*] ◁ *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 ‹*is* = [*i1*] @ [*i2*]›
    lookup-tensor-prod[*OF* ‹[*i1*] ◁ *Tensor.dims* (*unit-vec M j*)› ‹[*i2*] ◁ *Tensor.dims* (*unit-vec M j*)›]
     lookup-tensor-from-lookup[*OF* ‹*is* ◁ [*M*, *M*]›, *unfolded* ‹*is* = [*i1*] @ [*i2*]›]

```
    lookup-unit-vec[OF ‹i1 < M›] lookup-unit-vec[OF ‹i2 < M›] 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 ⊗ 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-le:j < dim-vec (tensors-from-net (Conv (id-matrix r0 M) (Input M)))
    using output-size-correct-tensors[OF ‹valid-net' (Conv (id-matrix r0 M) (Input
M))›,
    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-le
j-le]  tensors-ht-l0[OF assms]
    by auto
qed

lemma lookup-tensors-ht-l0':
assumes j<r0
and is ⊲ [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 ‹j<M› assms(2)
    by (simp-all add:lookup-tensor-from-lookup)
next
  assume ¬j<M
 then have is ≠ [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  ‹¬j<M› by (simp add: lookup-tensor0[OF assms(2)] ‹is ≠ [j, j]›)
qed

lemma lookup-tensors-ht-l1:
assumes j < r1
and is ⊲ [M,M]
shows Tensor.lookup (tensors-from-net (witness r1 r0 [M]) $ j) is
   = (if is!0 = is!1 ∧ 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 *output-size′* (*witness′ r0* [*M*]) = *r0* **unfolding** *witness-l0′* **using** *witness-l0′-valid*
by (*simp add*: *id-matrix-dim*)
have *dim-vec* (*tensors-from-net* (*witness′ r0* [*M*])) = *r0*
using ‹*output-size′* (*witness′ r0* [*M*]) = *r0*› *witness-l0′-valid output-size-correct-tensors* **by** *fastforce*
have *all0-but1*:$\bigwedge$*i*. *i*≠*is*!*0* ⟹ *i*<*r0* ⟹ *Tensor.lookup* (*tensors-from-net* (*witness′ r0* [*M*]) $ *i*) *is* = *0*
using *lookup-tensors-ht-l0′* ‹*is* ◁ [*M*, *M*]› **by** *auto*

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

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

lemma *length-output-deep-model*:
assumes *remove-weights m = deep-model-l rs*
shows *dim-vec (tensors-from-net m) = rs ! 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 = deep-model′-l rs*
shows *dim-vec (tensors-from-net m) = rs ! 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*:
dim-vec (tensors-from-net (witness-l rs)) = rs ! 0
using *length-output-deep-model witness-is-deep-model* **by** *blast*

lemma *length-output-witness′*:
dim-vec (tensors-from-net (witness′-l rs)) = rs ! 0
using *length-output-deep-model′ witness′-is-deep-model* **by** *blast*

lemma *dims-output-deep-model*:
assumes *length rs ≥ 2*
and *⋀r. r∈set rs ⟹ r > 0*
and *j < rs!0*
and *remove-weights m = deep-model-l rs*
shows *Tensor.dims (tensors-from-net m $ j) = replicate (2^(length rs − 2)) (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 *length rs ≥ 2*
and *⋀r. r∈set rs ⟹ r > 0*
and *j < rs!0*
shows *Tensor.dims (tensors-from-net (witness-l rs) $ j) = replicate (2^(length rs − 2)) (last rs)*
using *dims-output-deep-model witness-is-deep-model assms* **by** *blast*

lemma *dims-output-deep-model′*:
assumes *length rs ≥ 1*
and *⋀r. r∈set rs ⟹ r > 0*
and *j < rs!0*
and *remove-weights m = deep-model′-l rs*
shows *Tensor.dims (tensors-from-net m $ j) = replicate (2^(length rs − 1)) (last*

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 ⋀*r*. *r*∈*set rs* ⟹ *r* > *0*
and *j* < *rs*!*0*
shows *Tensor.dims* (*tensors-from-net* (*witness′-l rs*) $ *j*) = *replicate* (*2*^(*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*:⋀*r*. *r*∈*set rs* ⟹ *0* < *r*
begin

definition *r* = *min* (*last rs*) (*last* (*butlast rs*))
definition *N-half* = *2*^(*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* ⇒ *real*) ⇒ *real tensor*
 where *A ws* = *tensors-from-net* (*insert-weights shared-weights* (*deep-model-l rs*) *ws*) $ *y*
definition *A′* :: (*nat* ⇒ *real*) ⇒ *real mat*
 where *A′ ws* = *ten2mat* (*A ws*)

lemma *dims-tensor-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*
shows *dims* (*tensors-from-net m* $ *y*) = *replicate* (*2* ∗ *N-half*) (*last rs*)
proof −
  have *dims* (*tensors-from-net m* $ *y*) = *replicate* (*2* ^ (*length rs* − *2*)) (*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* = *deep-model-l rs*
shows *order* (*tensors-from-net m* $ *y*) = *2* ∗ *N-half*
  using *dims-tensor-deep-model* **by** (*simp add*: *assms*)

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

lemma *order-A*:
shows *order* (*A ws*) = *2* ∗ *N-half* **using** *dims-A length-replicate* **by** *auto*

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

lemma *dims-A′-pow*:
shows *dim-row* (*A′ ws*) = (*last rs*) ^ *N-half dim-col* (*A′ ws*) = (*last rs*) ^ *N-half*
 unfolding *dims-A′ dims-A nths-replicate set-le-in card-even card-odd prod-list-replicate*
  by *simp-all*

definition *Aw* = *tensors-from-net* (*witness-l rs*) $ *y*
definition *Aw′* = *ten2mat Aw*

definition *witness-weights* = *extract-weights shared-weights* (*witness-l rs*)

lemma *witness-weights*:*witness-l rs* = *insert-weights shared-weights* (*deep-model-l rs*) *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*) ^ *N-half dim-col Aw′* = (*last rs*) ^ *N-half*
  unfolding *Aw′-def′ Aw-def′* **using** *dims-A′-pow* **by** *auto*

lemma *witness-tensor*:
assumes *is* ◁ *Tensor.dims Aw*
shows *Tensor.lookup Aw is*
  = (*if nths is* {*n*. *even n*} = *nths is* {*n*. *odd n*} ∧ (∀ *i*∈*set is*. *i* < *last* (*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* ‹*3* ≤ *length rs*›)
  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.prems(*4*) *length-output-witness* ‹*rs* = [*rs* ! *0*, *rs* ! *1*, *rs* ! *2*]› *vec-setI* **by** *metis*
  then have *is* ◁ [*rs*!*2*, *rs*!*2*] **using** *Nil.prems* **by** *metis*
  then have *Tensor.lookup* ((*tensors-from-net* (*witness-l rs*))$*y*) *is*
    = (*if is* ! *0* = *is* ! *1* ∧ *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 $is\ !\ 0 = is\ !\ 1 \wedge is\ !\ 0 < rs\ !\ 1$
$\longleftrightarrow$ *nths is* $\{n.\ even\ n\}$ = *nths is* $\{n.\ odd\ n\} \wedge (\forall i{\in}set\ is.\ i < last\ (butlast\ rs))$
proof −
have *length is* = *2* **by** (*metis One-nat-def Suc-eq-plus1* ‹$is \lhd [rs ! 2, rs ! 2]$› *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* $is ! 0 = is ! 1 \wedge is ! 0 < rs ! 1$ *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*:$\bigwedge r.\ r \in set\ (tl\ rs) \Longrightarrow 0 < r$ **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*: $\bigwedge is'.\ is' \lhd$ *Tensor.dims* (*tensors-from-net* (*witness-l* (*tl rs*)) \$ *0*)
$\Longrightarrow$ *Tensor.lookup* (*tensors-from-net* (*witness-l* (*tl rs*)) \$ *0*) *is′* =

(if $nths$ is' ($Collect$ $even$) = $nths$ is' {n. odd n} ∧ (∀ i∈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* ⊲ *replicate* (2^($length$ rs − 2)) ($last$ rs)
using *Cons.prems(3) dims-output-witness 2* **by** (*metis* (*no-types*, *lifting*) *Cons.prems(1) Cons.prems(3)*
Cons.prems(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* ⊲ *replicate* (2^($length$ (tl rs) − 2)) ($last$ rs) @ *replicate* (2^($length$ (tl rs) − 2)) ($last$ rs)
using *Cons.prems 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* ⊲ *replicate* (2^($length$ (tl rs) − 2)) ($last$ rs) **and**
is2-replicate: *is2* ⊲ *replicate* (2^($length$ (tl rs) − 2)) ($last$ rs) **by** (*metis valid-index-split*)
then have
is1-valid: *is1* ⊲ *Tensor.dims* (*tensors-from-net* (*witness-l* (tl rs)) \$ 0) (**is** *?is1*) **and**
is2-valid: *is2* ⊲ *Tensor.dims* (*tensors-from-net* (*witness-l* (tl rs)) \$ 0) (**is** *?is2*)
proof −
have *last* (tl rs) = *last rs* **by** (*metis 2* ‹*rs* = *r* # *tl rs*› *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* ‹*last* (*tl rs*) = *last rs*› **by** *auto*
qed

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

let *?cond* = λ*is rs*. *nths is* {n. *even* n} = *nths is* {n. *odd* n} ∧ (∀ i∈*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*) ⟷ *?cond is rs*
proof −
have *length is1* = *2* ^ (*length* (*tl rs*) − *2*)
length is2 = *2* ^ (*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* (*nths is2* (*Collect even*)) = *length* (*nths is2* (*Collect odd*))
using *length-nths-even* ‹*even* (*length is2*)› **by** *blast*
have *cond1-iff*: (*nths is1* (*Collect even*) = *nths is1* {*n*. *odd n*} ∧ *nths is2* (*Collect even*) = *nths is2* {*n*. *odd n*})
= (*nths is* (*Collect even*) = *nths is* {*n*. *odd n*})
unfolding ‹*is* = *is1* @ *is2*› *nths-append*
‹{*j*. *j* + *length is1* ∈ {*n*. *odd n*}} = {*n*. *odd n*}› ‹{*j*. *j* + *length is1* ∈ {*n*. *even n*}} = {*n*. *even n*}›
by (*simp add*: ‹*length* (*nths is2* (*Collect even*)) = *length* (*nths is2* (*Collect odd*))›)
have *last* (*butlast* (*tl rs*)) = *last* (*butlast rs*) **using** *Nitpick.size-list-simp*(*2*) ‹*even* (*length is1*)›
‹*length is1* = *2* ^ (*length* (*tl rs*) − *2*)› *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*: (∀ *i*∈*set is1*. *i* < *last* (*butlast* (*tl rs*))) ∧ (∀ *i*∈*set is2*. *i* < *last* (*butlast* (*tl rs*))) ⟷ (∀ *i*∈*set is*. *i* < *last* (*butlast rs*))
unfolding ‹*last* (*butlast* (*tl rs*)) = *last* (*butlast rs*)› ‹*is* = *is1* @ *is2*› *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*) ⊗ (*tensors-from-net* (*witness-l* (*tl rs*))) $ *0*) *is* =
(*if ?cond is rs then 1 else 0*)
using ‹*?cond is1* (*tl rs*) ∧ *?cond is2* (*tl rs*) ⟷ *?cond is rs*›
unfolding ‹*is* = *is1* @ *is2*› *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* ‹*rs* = *r* # *tl rs*› *nth-Cons-Suc*)
have *tl-tl*:(*tl* (*tl rs*)) = ((*rs* ! *2*) # *tl* (*tl* (*tl rs*)))
proof −
have *length* (*tl* (*tl rs*)) ≠ *0*

by (*metis One-nat-def Suc-eq-plus1 diff-diff-left diff-is-0-eq length-tl not-less-eq-eq Cons.prems*(*2*) *numeral-3-eq-3*)
then have *tl* (*tl rs*) $\neq$ []
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.prems*(*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' $$ (i, j)
$= (if\ i{=}j \wedge (\forall\ i0{\in}set\ (digit\text{-}encode\ (nths\ (Tensor.dims\ Aw)\ \{n.\ even\ n\})\ i).\ i0 <$
$last\ (butlast\ rs))\ then\ 1\ else\ 0)$
proof −
define *is* **where** $is = weave\ \{n.\ even\ n\}$
$(digit\text{-}encode\ (nths\ (Tensor.dims\ Aw)\ \{n.\ even\ n\})\ i)$
$(digit\text{-}encode\ (nths\ (Tensor.dims\ Aw)\ \{n.\ odd\ n\})\ j)$
have *lookup-eq*: Aw' $$ $(i, j) = Tensor.lookup\ Aw\ is$
using *Aw′-def matricize-def dims-Aw′(1)[symmetric, unfolded A-def] dims-Aw′(2)[symmetric, unfolded A-def Collect-neg-eq]*
index-mat(1)[OF ‹i < dim-row Aw′› ‹j < dim-col Aw′›] is-def Collect-neg-eq case-prod-conv
by *(metis (no-types) Aw′-def Collect-neg-eq case-prod-conv is-def matricize-def)*
have $is \lhd Tensor.dims\ Aw$
using *is-def valid-index-weave A-def Collect-neg-eq assms digit-encode-valid-index dims-Aw′* **by** *metis*

have *even (order Aw)*
unfolding *Aw-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*: $nths\ (Tensor.dims\ Aw)\ (Collect\ even) = nths\ (Tensor.dims\ Aw)\ \{n.\ odd\ n\}$
proof −
have $0{:}Tensor.dims\ (tensors\text{-}from\text{-}net\ (witness\text{-}l\ rs)\ \$\ y) = replicate\ (2\ \hat{}\ (length\ rs - 2))\ (last\ rs)$
using *dims-output-witness[OF - no-zeros y-valid]* **using** *deep* **by** *linarith*
show *?thesis* **unfolding** *A-def*
using *nths-replicate*
by *(metis (no-types, lifting) 0 Aw-def ‹even (order Aw)› length-replicate length-nths-even)*
qed

have $i = j \longleftrightarrow nths\ is\ (Collect\ even) = nths\ is\ \{n.\ odd\ n\}$
proof
have *eq-lengths*: $length\ (digit\text{-}encode\ (nths\ (Tensor.dims\ Aw)\ (Collect\ even))\ i)$
$= length\ (digit\text{-}encode\ (nths\ (Tensor.dims\ Aw)\ \{n.\ odd\ n\})\ j)$
unfolding *length-digit-encode* **by** *(metis ‹even (order Aw)› length-nths-even)*

then show $i = j \Longrightarrow nths\ is\ (Collect\ even) = nths\ is\ \{n.\ odd\ n\}$ **unfolding** *is-def*
using *nths-weave[of digit-encode (nths (Tensor.dims Aw) (Collect even)) i Collect even digit-encode (nths (Tensor.dims Aw) {n. odd n}) j, unfolded eq-lengths, unfolded Collect-neg-eq[symmetric] card-even mult-2[symmetric] card-odd]*
nths-dimsAw **by** *simp*
show $nths\ is\ (Collect\ even) = nths\ is\ \{n.\ odd\ n\} \Longrightarrow i = j$ **unfolding** *is-def*
using *nths-weave[of digit-encode (nths (Tensor.dims Aw) (Collect even)) i*

Collect even digit-encode (nths (Tensor.dims Aw) {n. odd n}) j, unfolded eq-lengths, unfolded Collect-neg-eq[symmetric] card-even mult-2[symmetric] card-odd]
using ‹*nths (Tensor.dims Aw) (Collect even) = nths (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* ⟹ *set (digit-encode (nths (Tensor.dims Aw) {n. even n}) i) = set is*
unfolding *is-def nths-dimsAw*
using *set-weave[of (digit-encode (nths (Tensor.dims Aw) {n. odd n}) j) Collect even*
(digit-encode (nths (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) = (nths is (Collect even) = nths is {n. odd n})*›*)*
qed

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

lemma *card-low-digits*:
assumes *m>0* ⋀*d. d∈set ds* ⟹ *m ≤ d*
shows *card {i. i<prod-list ds ∧ (∀ i0∈set (digit-encode ds i). i0 < m)} = m ^ (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 ⟷ i < prod-list ds ∧ (∀ i0∈set (digit-encode ds i). i0 < m)* **for** *ds i*
have *card {i. low-digits ds i} = m ^ (length ds)* **unfolding** *low-digits-def*
by *(simp add: Cons.IH Cons.prems(1) Cons.prems(2))*
have *card {i. low-digits (d # ds) i} = card ({..<m} × {i. low-digits ds i})*
proof −
define *f* **where** *f p = fst p + d ∗ snd p* **for** *p*
have *inj-on f ({..<m} × {i. low-digits ds i})*
proof *(rule inj-onI)*
fix *x y* **assume** *x ∈ {..<m} × {i. low-digits ds i} y ∈ {..<m} × {i. low-digits ds i} f x = f y*
then have *fst x<m fst y<m* **by** *auto*
then have *fst x<d fst y<d* **using** *Cons(3)* **by** *(meson list.set-intros(1) not-le order-trans)+*

then have *f x mod d = fst x f y mod d = fst y* **unfolding** *f-def* **by** *simp-all*
have *f x div d = snd x f y div d = snd y* **using** ‹*f x = f y*› ‹*f x mod d = fst x*› ‹*fst y < d*› *f-def* **by** *auto*
show *x = y* **using** ‹*f x = f y*› ‹*f x div d = snd x*› ‹*f x mod d = fst x*› ‹*f y div d = snd y*› ‹*f y mod d = fst y*› *prod-eqI* **by** *fastforce*
qed
have *f* ‘ ({..<*m*} × {*i*. *low-digits ds i*}) = {*i*. *low-digits* (*d* # *ds*) *i*}
proof (*rule subset-antisym*; *rule subsetI*)
fix *x* **assume** *x* ∈ *f* ‘ ({..<*m*} × {*i*. *low-digits ds i*})
then obtain *i0 i1* **where** *x = i0 + d * i1 i0 < m low-digits ds i1* **using** *f-def* **by** *force*
then have *i0<d* **using** *Cons*(*3*) **by** (*meson list.set-intros*(*1*) *not-le order-trans*)
show *x* ∈ {*i*. *low-digits* (*d* # *ds*) *i*} **unfolding** *low-digits-def*
proof (*rule*; *rule conjI*)
have *i1 < prod-list ds* ∀ *i0*∈*set* (*digit-encode ds i1*). *i0 < m*
using ‹*low-digits ds i1*› *low-digits-def* **by** *auto*
show *x < prod-list* (*d* # *ds*) **unfolding** *prod-list.Cons* ‹*x = i0 + d * i1*› **using** ‹*i0<d*› ‹*i1 < prod-list ds*›
proof −
have *d* ≠ *0*
by (*metis* ‹*i0 < d*› *gr-implies-not0*)
then have (*i0 + d * i1*) *div* (*d * prod-list ds*) = *0*
by (*simp add*: *Divides.div-mult2-eq* ‹*i0 < d*› ‹*i1 < prod-list ds*›)
then show *i0 + d * i1 < d * prod-list ds*
by (*metis* (*no-types*) ‹*i0 < d*› ‹*i1 < prod-list ds*› *div-eq-0-iff gr-implies-not0 no-zero-divisors*)
qed
show ∀ *i0*∈*set* (*digit-encode* (*d* # *ds*) *x*). *i0 < m*
using ‹∀ *i0*∈*set* (*digit-encode ds i1*). *i0 < m*› ‹*i0 < d*› ‹*i0 < m*› ‹*x = i0 + d * i1*› **by** *auto*
qed
next
fix *x* **assume** *x* ∈ {*i*. *low-digits* (*d* # *ds*) *i*}
then have *x < prod-list* (*d* # *ds*) ∀ *i0*∈*set* (*digit-encode* (*d* # *ds*) *x*). *i0 < m* **using** *low-digits-def* **by** *auto*
have *x mod d < m* **using** ‹∀ *i0*∈*set* (*digit-encode* (*d* # *ds*) *x*). *i0 < m*›[*unfolded digit-encode.simps*] **by** *simp*
have *x div d < prod-list ds* **using** ‹*x < prod-list* (*d* # *ds*)›[*unfolded prod-list.Cons*]
by (*metis div-eq-0-iff div-mult2-eq mult-0-right not-less0*)
have ∀ *i0*∈*set* (*digit-encode ds* (*x div d*)). *i0 < m* **by** (*simp add*: ‹∀ *i0*∈*set* (*digit-encode* (*d* # *ds*) *x*). *i0 < m*›)
have *f* ((*x mod d*),(*x div d*)) = *x* **by** (*simp add*: *f-def*)
show *x* ∈ *f* ‘ ({..<*m*} × {*i*. *low-digits ds i*}) **by** (*metis SigmaI* ‹∀ *i0*∈*set* (*digit-encode ds* (*x div d*)). *i0 < m*› ‹*f* (*x mod d*, *x div d*) = *x*› ‹*x div d < prod-list ds*› ‹*x mod d < m*› *image-eqI lessThan-iff low-digits-def mem-Collect-eq*)
qed
then have *bij-betw f* ({..<*m*} × {*i*. *low-digits ds i*}) {*i*. *low-digits* (*d* # *ds*) *i*}
by (*simp add*: ‹*inj-on f* ({..<*m*} × {*i*. *low-digits ds i*})› *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* (*nths* (*Tensor.dims Aw*) (*Collect even*)) ∧
(∀ *i0*∈*set* (*digit-encode* (*nths* (*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* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*). *i0* < *last* (*butlast rs*)
using *rows-with-1-def* **by** *auto*
then have *i* < *prod-list* (*nths* (*dims Aw*) (*Collect even*)) **using** *dims-Aw'* **by** *linarith*
then have *digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ⊲ *nths* (*dims Aw*) (*Collect even*)
using *digit-encode-valid-index* **by** *auto*
have ∀ *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*). *i0* < *r*
proof
fix *i0* **assume** *1*:*i0* ∈ *set* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)
then obtain *k* **where** *k* < *length* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)
digit-encode (*nths* (*dims Aw*) (*Collect even*)) *i* ! *k* = *i0* **by** (*meson in-set-conv-nth*)
have *i0* < *last* (*butlast rs*)
using ‹∀ *i0*∈*set* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*). *i0* < *last* (*butlast rs*)› *1* **by** *blast*
have *set* (*nths* (*dims Aw*) (*Collect even*)) ⊆ {*last rs*} **unfolding** *dims-Aw* **using** *subset-eq* **by** *fastforce*
then have *nths* (*dims Aw*) (*Collect even*) ! *k* = *last rs*
using ‹*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ⊲ *nths* (*dims Aw*) (*Collect even*)›
‹*k* < *length* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)›
nth-mem valid-index-length **by** *auto*
then have *i0* < *last rs*
using *valid-index-lt* ‹*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ! *k* = *i0*›
‹*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ⊲ *nths* (*dims Aw*) (*Collect even*)›
‹*k* < *length* (*digit-encode* (*nths* (*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* (*nths* (*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*:⋀*d*. *d* ∈ *set* (*nths* (*Tensor.dims Aw*) (*Collect even*)) ⟹ *r* ≤ *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 ⋀*i*. *listpr dvd i* ⟹ *i* ∈ *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** ‹⋀*i*. *listpr dvd i* ⟹ *i* ∈ *rows-with-1*›

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*^*N-half*)
proof
show *dim-row* (*submatrix Aw′ rows-with-1 rows-with-1*) = *dim-row* (1_m (*r* ^ *N-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* ^ *N-half*))
by (*metis* ‹*dim-row* (*submatrix Aw′ rows-with-1 rows-with-1*) = *dim-row* (1_m (*r* ^ *N-half*))› *dim-submatrix*(*1*) *dim-submatrix*(*2*) *index-one-mat*(*2*) *index-one-mat*(*3*) *dims-Aw′-pow*)
show ⋀*i j*. *i* < *dim-row* (1_m (*r* ^ *N-half*)) ⟹
j < *dim-col* (1_m (*r* ^ *N-half*)) ⟹ *submatrix Aw′ rows-with-1 rows-with-1* $$ (*i*, *j*) = 1_m (*r* ^ *N-half*) $$ (*i*, *j*)
proof −
fix *i j* **assume** *i* < *dim-row* (1_m (*r* ^ *N-half*)) *j* < *dim-col* (1_m (*r* ^ *N-half*))
then have *i* < *r* ^ *N-half j* < *r* ^ *N-half* **by** *auto*
then have *i* < *card* {*i* ∈ *rows-with-1*. *i* < *dim-row Aw′*} *j* < *card* {*i* ∈ *rows-with-1*. *i* < *dim-col Aw′*}
using *card-rows-with-1 dims-Aw′-pow* **by** *auto*
then have *pick rows-with-1 i* < *dim-row Aw′ pick rows-with-1 j* < *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 ∀ *i0*∈*set* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) (*pick rows-with-1 i*)). *i0* < *last* (*butlast rs*)
using *infinite-rows-with-1 pick-in-set-inf rows-with-1-def* **by** *auto*
then have *Aw′* $$ (*pick rows-with-1 i*, *pick rows-with-1 j*) = (*if pick rows-with-1 i* = *pick rows-with-1 j then 1 else 0*)
using *witness-matricization*[*OF* ‹*pick rows-with-1 i* < *dim-row Aw′*› ‹*pick rows-with-1 j* < *dim-col Aw′*›] **by** *simp*
then have *submatrix Aw′ rows-with-1 rows-with-1* $$ (*i*, *j*) = (*if pick rows-with-1 i* = *pick rows-with-1 j then 1 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* ^ *N-half*))›
‹*dim-row* (*submatrix Aw′ rows-with-1 rows-with-1*) = *dim-row* (1_m (*r* ^ *N-half*))›
‹*i* < *dim-row* (1_m (*r* ^ *N-half*))› ‹*j* < *r* ^ *N-half*› *dim-submatrix*(*1*) *dim-submatrix*(*2*) *index-one-mat*(*3*))
then have *submatrix Aw′ rows-with-1 rows-with-1* $$ (*i*, *j*) = (*if i* = *j then 1 else 0*)
using *pick-eq-iff-inf*[*OF infinite-rows-with-1*] **by** *auto*
then show *submatrix Aw′ rows-with-1 rows-with-1* $$ (*i*, *j*) = 1_m (*r* ^ *N-half*) $$ (*i*, *j*)
by (*simp add*: ‹*i* < *r* ^ *N-half*› ‹*j* < *r* ^ *N-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 $\bigwedge x.\ (A\ x) \in carrier\text{-}mat\ n\ n$
assumes $\bigwedge x\ i\ j.\ i{<}n \Longrightarrow j{<}n \Longrightarrow polyfun\ N\ (\lambda x.\ (A\ x)\ \$\$\ (i,j))$
shows $polyfun\ N\ (\lambda x.\ det\ (A\ x))$
proof −
  {
    fix *p* **assume** $p \in \{p.\ p\ permutes\ \{0..{<}n\}\}$
    then have $p\ permutes\ \{0..{<}n\}$ **by** *auto*
    then have $\bigwedge x.\ x < n \Longrightarrow p\ x < n$ **using** *permutes-in-image* **by** *auto*
    then have $polyfun\ N\ (\lambda x.\ \prod i = 0..{<}n.\ A\ x\ \$\$\ (i,\ p\ i))$
      using *polyfun-Prod*[*of* $\{0..{<}n\}$ *N* $\lambda i\ x.\ A\ x\ \$\$\ (i,\ p\ i)$] *assms* **by** *simp*
    then have $polyfun\ N\ (\lambda x.\ signof\ p * (\prod i = 0..{<}n.\ A\ x\ \$\$\ (i,\ p\ i)))$ **using** *polyfun-const polyfun-mult* **by** *blast*
  }
 moreover have *finite* $\{i.\ i\ permutes\ \{0..{<}n\}\}$ **by** (*simp add*: *finite-permutations*)
  ultimately show *?thesis* **unfolding** *det-def′*[*OF assms*(*1*)]
    using *polyfun-Sum*[*OF* ‹*finite* $\{i.\ i\ permutes\ \{0..{<}n\}\}$›, *of N* $\lambda p\ x.\ signof\ p * (\prod i = 0..{<}n.\ A\ x\ \$\$\ (i,\ p\ i))$]
    by *blast*
qed

lemma *polyfun-extract-matrix*:
assumes $i{<}m\ j{<}n$
shows $polyfun\ \{..{<}a + (m * n + c)\}\ (\lambda f.\ extract\text{-}matrix\ (\lambda i.\ f\ (i + a))\ m\ n\ \$\$\ (i,j))$

unfolding *index-extract-matrix*[*OF assms*] **apply** (*rule polyfun-single*) **using** *two-digit-le*[*OF assms*] **by** *simp*

lemma *polyfun-mult-mat-vec*:
assumes $\bigwedge x.\ v\ x \in$ *carrier-vec n*
assumes $\bigwedge j.\ j<n \Longrightarrow$ *polyfun N* ($\lambda x.\ v\ x$ \$ j)
assumes $\bigwedge x.\ A\ x \in$ *carrier-mat m n*
assumes $\bigwedge i\ j.\ i<m \Longrightarrow j<n \Longrightarrow$ *polyfun N* ($\lambda x.\ A\ x$ \$\$ (i,j))
assumes $j < m$
shows *polyfun N* ($\lambda x.$ (($A\ x$) $*_v$ ($v\ x$)) \$ j)
proof −
  have $\bigwedge x.\ j <$ *dim-row* ($A\ x$) **using** ‹$j < m$› *assms*(*3*) *carrier-matD*(*1*) **by** *force*
  have $\bigwedge x.\ n =$ *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 $i <$ *dim-vec* ($v\ x$) **using** *assms*(*1*) *carrier-vecD* ‹$i<n$› **by** *fastforce*
      have $j <$ *dim-row* ($A\ x$) **using** ‹$j < m$› *assms*(*3*) *carrier-matD*(*1*) **by** *force*
      have *dim-col* ($A\ x$) = *dim-vec* ($v\ x$) **by** (*metis assms*(*1*) *assms*(*3*) *carrier-matD*(*2*) *carrier-vecD*)
      then have *row* ($A\ x$) j \$ $i = A\ x$ \$\$ (j,i) $i<n$ **using** ‹$j <$ *dim-row* ($A\ x$)› ‹$i<n$› **by** (*simp-all add*: ‹$i <$ *dim-vec* ($v\ x$)›)
    }
    then have *polyfun N* ($\lambda x.$ *row* ($A\ x$) j \$ $i * v\ x$ \$ i)
      using *polyfun-mult assms*(*4*)[*OF* ‹$j < m$›] *assms*(*2*) **by** *fastforce*
  }
  then show *?thesis* **unfolding** *index-mult-mat-vec*[*OF* ‹$\bigwedge x.\ j <$ *dim-row* ($A\ x$)›] *scalar-prod-def*
   using *polyfun-Sum*[*of* $\{0..<n\}$ *N* $\lambda i\ x.$ *row* ($A\ x$) j \$ $i * v\ x$ \$ i] *finite-atLeastLessThan*[*of* $0\ n$] ‹$\bigwedge x.\ n =$ *dim-vec* ($v\ x$)›
    by *simp*
qed

lemma *polyfun-evaluate-net-plus-a*:
assumes *map dim-vec inputs* = *input-sizes m*
assumes *valid-net m*
assumes $j <$ *output-size m*
shows *polyfun* $\{..<a +$ *count-weights s m*$\}$ ($\lambda f.$ *evaluate-net* (*insert-weights s m* ($\lambda i.\ f\ (i + a)$)) *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* (*λi. 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* (*λi. f* (*i* + *x1* ∗ *x2*)))
using *input-sizes-remove-weights remove-insert-weights*
by (*simp add*: *Conv.prems*(*1*))
have *dim-vec* (*evaluate-net* (*insert-weights s m* (*λi. 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* (*λi. 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.prems*(*1*))
have *valid-net m* **using** *Conv.prems*(*2*) *valid-net.cases* **by** *fastforce*
show ⋀*j. j* < *output-size m* ⟹ *polyfun* {..<*a* + *count-weights s* (*Conv* (*x1*, *x2*) *m*)}
(*λf. evaluate-net* (*insert-weights s m* (*λi. f* (*i* + *x1* ∗ *x2* + *a*))) *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.prems*(*2*) ‹*x* = (*x1*, *x2*)› *valid-net.cases* **by** *fastforce*
show ⋀*f. extract-matrix* (*λi. f* (*i* + *a*)) *x1 x2* ∈ *carrier-mat x1* (*output-size m*) **unfolding** ‹*output-size m* = *x2*› **using** *dim-extract-matrix*
using *carrier-matI* **by** (*metis* (*no-types*, *lifting*))

show ⋀*i j. i* < *x1* ⟹ *j* < *output-size m* ⟹ *polyfun* {..<*a* + *count-weights s* (*Conv* (*x1*, *x2*) *m*)} (*λf. extract-matrix* (*λi. f* (*i* + *a*)) *x1 x2* $$ (*i*, *j*))
unfolding ‹*output-size m* = *x2*› *count-weights.simps* **using** *polyfun-extract-matrix*[*of* - *x1* - *x2 a count-weights s m*] **by** *blast*

show *j* < *x1* **using** *Conv.prems*(*3*) ‹*x* = (*x1*, *x2*)› **by** *auto*
qed
next
case (*Pool m1 m2 inputs j a*)
have *A2*:⋀*f. map dim-vec* (*take* (*length* (*input-sizes* (*insert-weights s m1* (*λi. f* (*i* + *a*))))) *inputs*) = *input-sizes m1*

by (*metis Pool.prems*(*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* (λi. *f* (*i* + *a*))))) *inputs*) = *input-sizes m2*
using *Pool.prems*(*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** ‹*valid-net* (*Pool m1 m2*)› *valid-net.simps* **by** *blast*+
have *output-size* (*Pool m1 m2*) = *output-size m2* **unfolding** *output-size.simps*
using ‹*valid-net* (*Pool m1 m2*)› *valid-net.cases* **by** *fastforce*
then have *A4*:*j* < *output-size m1* **and** *B4*:*j* < *output-size m2* **using** ‹*j* < *output-size* (*Pool m1 m2*)› **by** *simp-all*

let *?net1* = λf. *evaluate-net* (*insert-weights s m1* (λi. *f* (*i* + *a*)))
(*take* (*length* (*input-sizes* (*insert-weights s m1* (λi. *f* (*i* + *a*))))) *inputs*)
let *?net2* = λf. *evaluate-net* (*insert-weights s m2* (*if s then* λi. *f* (*i* + *a*) *else* (λi. *f* (*i* + *count-weights s m1* + *a*))))
(*drop* (*length* (*input-sizes* (*insert-weights s m1* (λi. *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* < *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* < *dim-vec* (*?net2 f*) **using** *B4* **by** *metis*
have *cong1*:$\bigwedge xf$. (λf. *evaluate-net* (*insert-weights s m1* (λi. *f* (*i* + *a*)))
(*take* (*length* (*input-sizes* (*insert-weights s m1* (λi. *xf* (*i* + *a*))))) *inputs*) $ *j*)
= (λf. *?net1 f* $ *j*)
using *input-sizes-remove-weights remove-insert-weights* **by** *auto*
have *cong2*:$\bigwedge xf$. (λf. *evaluate-net* (*insert-weights s m2* (λi. *f* (*i* + (*a* + (*if s then 0 else count-weights s m1*)))))
(*drop* (*length* (*input-sizes* (*insert-weights s m1* (λi. *xf* (*i* + *a*))))) *inputs*) $ *j*)
= (λf. *?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* + *count-weights s m1*} {..<*a* + (*if s then max* (*count-weights s m1*) (*count-weights s m2*) *else count-weights s m1* + *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 then 0 else count-weights s m1*) + *count-weights s m2*)} {..<*a* + (*if s then max* (*count-weights s m1*) (*count-weights*

s m2) *else count-weights s m1* + *count-weights s m2*)}])
qed

lemma *polyfun-evaluate-net*:
assumes *map dim-vec inputs* = *input-sizes m*
assumes *valid-net m*
assumes $j <$ *output-size m*
shows *polyfun* {..<*count-weights s m*} (λf. *evaluate-net* (*insert-weights s m f*) *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* $\lhd$ *input-sizes m*
assumes $j <$ *output-size m*
shows *polyfun* {..<*count-weights s m*} (λf. *Tensor.lookup* (*tensors-from-net* (*insert-weights s m f*) \$ *j*) *is*)
proof −
have *1*:$\bigwedge f$. *valid-net'* (*insert-weights s m f*) **by** (*simp add*: *assms*(*1*) *remove-insert-weights*)
have *input-sizes*:$\bigwedge f$. *input-sizes* (*insert-weights s m f*) = *input-sizes m*
unfolding *input-sizes-remove-weights* **by** (*simp add*: *remove-insert-weights*)
have *2*:$\bigwedge f$. *is* $\lhd$ *input-sizes* (*insert-weights s m f*)
unfolding *input-sizes* **using** *assms*(*2*) **by** *blast*
have *3*:$\bigwedge f$. $j <$ *output-size'* (*insert-weights s m f*)
by (*simp add*: *assms*(*3*) *remove-insert-weights*)
have $\bigwedge$*f1 f2*. *base-input* (*insert-weights s m f1*) *is* = *base-input* (*insert-weights s m f2*) *is*
unfolding *base-input-def* **by** (*simp add*: *input-sizes*)
then have $\bigwedge$*xf*. (λf. *evaluate-net* (*insert-weights s m f*) (*base-input* (*insert-weights s m xf*) *is*) \$ *j*)
= (λf. *evaluate-net* (*insert-weights s m f*) (*base-input* (*insert-weights s m f*) *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 $\bigwedge x$. *dims* (*T x*) = *ds*
assumes $\bigwedge$*is*. *is* $\lhd$ *ds* $\Longrightarrow$ *polyfun N* (λx. *Tensor.lookup* (*T x*) *is*)
assumes $\bigwedge x$. *dim-row* (*matricize I* (*T x*)) = *nr*
assumes $\bigwedge x$. *dim-col* (*matricize I* (*T x*)) = *nc*
assumes $i <$ *nr*
assumes $j <$ *nc*
shows *polyfun N* (λx. *matricize I* (*T x*) \$\$ (*i*,*j*))
proof −
let *?weave* = λ *x*. (*weave I*

(*digit-encode* (*nths ds I*) *i*)
(*digit-encode* (*nths ds* (−*I*)) *j*))
have *1*:⋀*x*. *matricize I* (*T x*) $$ (*i*,*j*) = *Tensor.lookup* (*T x*) (*?weave x*) **unfolding** *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 ⋀*x*. *?weave x* ⊲ *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* (λ*x*. *Tensor.lookup* (*T x*) (*?weave x*)) **using** *assms*(*2*) **by** *simp*
then show *?thesis* **unfolding** *1* **using** *assms*(*1*) **by** *blast*
qed

lemma (¬ (*a*::*nat*) < *b*) = (*a* ≥ *b*)
by (*metis not-le*)

lemma *polyfun-submatrix*:
assumes ⋀*x*. (*A x*) ∈ *carrier-mat m n*
assumes ⋀*x i j*. *i*<*m* ⟹ *j*<*n* ⟹ *polyfun N* (λ*x*. (*A x*) $$ (*i*,*j*))
assumes *i* < *card* {*i*. *i* < *m* ∧ *i* ∈ *I*}
assumes *j* < *card* {*j*. *j* < *n* ∧ *j* ∈ *J*}
assumes *infinite I infinite J*
shows *polyfun N* (λ*x*. (*submatrix* (*A x*) *I J*) $$ (*i*,*j*))
proof −
have *1*:⋀*x*. (*submatrix* (*A x*) *I J*) $$ (*i*,*j*) = (*A x*) $$ (*pick I i*, *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* ‹*infinite I*›] *card-le-pick-inf*[*OF* ‹*infinite J*›]
‹*i* < *card* {*i*. *i* < *m* ∧ *i* ∈ *I*}›[*unfolded set-le-in*] ‹*j* < *card* {*j*. *j* < *n* ∧ *j* ∈ *J*}›[*unfolded set-le-in*] *not-less* **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* ⊲ *input-sizes* (*deep-model-l rs*)
shows *polyfun* {..<*weight-space-dim*}
(λ*f*. *Tensor.lookup* (*tensors-from-net* (*insert-weights shared-weights* (*deep-model-l rs*) *f*) $ *y*) *is*)

proof −
have *1*:⋀*f*. *remove-weights* (*insert-weights shared-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*
then have *y* < *output-size* (*deep-model-l rs*) **using** *valid-deep-model y-valid length-output-deep-model* **by** *force*
have *0*:{..<*weight-space-dim*} = *set* [*0*..<*weight-space-dim*] **by** *auto*
then show *?thesis* **unfolding** *weight-space-dim-def* **using** *polyfun-tensors-from-net assms*(*1*) *valid-deep-model*
‹*y* < *output-size* (*deep-model-l rs*)› **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*<(*last rs*) ^ *N-half*
assumes *j*<(*last rs*) ^ *N-half*
shows *polyfun* {..<*weight-space-dim*} (λ*f*. *A′ f* $$ (*i*,*j*))
proof −
have *0*:*y* < *output-size* (*deep-model-l rs*) **using** *valid-deep-model y-valid length-output-deep-model* **by** *force*
have *1*:⋀*f*. *remove-weights* (*insert-weights shared-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*
have *2*:(⋀*f is*. *is* ⊲ *replicate* (*2* ∗ *N-half*) (*last rs*) ⟹
polyfun {..<*weight-space-dim*} (λ*x*. *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* ^ *N-half*
assumes *j* < *r* ^ *N-half*
shows *polyfun* {..<*weight-space-dim*} (λ*f*. *witness-submatrix f* $$ (*i*,*j*))
unfolding *witness-submatrix-def*
proof (*rule polyfun-submatrix*)
have *1*:⋀*f*. *remove-weights* (*insert-weights shared-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*

show $\bigwedge f.\ A'\ f \in$ *carrier-mat* $((last\ rs)\ \widehat{}\ N\text{-}half)\ ((last\ rs)\ \widehat{}\ N\text{-}half)$
using *1 dims-A′-pow* **using** *weight-space-dim-def* **by** *auto*
show $\bigwedge f\ i\ j.\ i < last\ rs\ \widehat{}\ N\text{-}half \Longrightarrow j < last\ rs\ \widehat{}\ N\text{-}half \Longrightarrow$
polyfun $\{..<weight\text{-}space\text{-}dim\}\ (\lambda f.\ A'\ f\ \$\$\ (i,\ j))$
using *polyfun-matrix-deep-model weight-space-dim-def* **by** *force*
show $i <$ *card* $\{i.\ i < last\ rs\ \widehat{}\ N\text{-}half \wedge i \in rows\text{-}with\text{-}1\}$
using *assms*(*1*) *card-rows-with-1 dims-Aw′-pow set-le-in* **by** *metis*
show $j <$ *card* $\{i.\ i < last\ rs\ \widehat{}\ N\text{-}half \wedge i \in rows\text{-}with\text{-}1\}$
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* $\{..<weight\text{-}space\text{-}dim\}\ (\lambda f.\ det\ (witness\text{-}submatrix\ f))$
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* $\in$ *carrier-mat* $(r\ \widehat{}\ N\text{-}half)\ (r\ \widehat{}\ N\text{-}half)$
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 $\bigwedge i\ j.\ i < r\ \widehat{}\ N\text{-}half \Longrightarrow j < r\ \widehat{}\ N\text{-}half \Longrightarrow$ *polyfun* $\{..<weight\text{-}space\text{-}dim\}$ $(\lambda f.\ witness\text{-}submatrix\ f\ \$\$\ (i,\ j))$
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* ⇒ (*nat* ⇒ *real*) *measure* **where**
lborel-f n = (Π_M *b*∈{..<*n*}. *lborel*)

lemma *product-sigma-finite-interval*: *product-sigma-finite* (λ*b*. *interval-measure* (λ*x*. *x*))
unfolding *product-sigma-finite-def* **using** *sigma-finite-interval-measure* **by** *auto*

lemma *l-borel-f-1*: *distr* (*lborel-f 1*) *lborel* (λ*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*} (λ-. *UNIV*) **unfolding** *lborel-f-def*
unfolding *space-PiM space-lborel space-borel* **by** *metis*

lemma *space-lborel-f-subset*: *space* (*lborel-f n*) ⊆ *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* ∈ *space* (*lborel-f n*)
shows *f*(*n*:=*x*) ∈ *space* (*lborel-f* (*Suc n*))
unfolding *space-lborel-f*
using *assms lessThan-Suc space-lborel-f* **by** *auto*

lemma *integral-lborel-f*:
assumes *f* ∈ *borel-measurable* (*lborel-f* (*Suc n*))
shows $integral^N$ (*lborel-f* (*Suc n*)) *f* = ∫ + *y*. ∫ + *x*. *f* (*x*(*n* := *y*)) ∂*lborel-f n* ∂*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* ∈ *sets* (*lborel-f* (*Suc n*))
assumes ⋀*y*. {*x*∈*space* (*lborel-f n*). *x*(*n* := *y*) ∈ *A*} ∈ *sets* (*lborel-f n*)
shows *emeasure* (*lborel-f* (*Suc n*)) *A* = ∫ + *y*. *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *y*) ∈ *A*} ∂*lborel*
proof −
{
fix *x y* **assume** *x*∈*space* (*lborel-f n*)
then have (*indicator A*) (*x*(*n* := *y*)) = (*indicator* {*x*∈*space* (*lborel-f n*). *x*(*n* := *y*) ∈ *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*: $(\lambda f.\ f(n := x)) \in$ *measurable* (*lborel-f n*) (*lborel-f* (*Suc n*))
proof −
have $x \in$ *space 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 lborel* Pi_M {..<*n*} (λb. *lborel*), *OF* ‹$x \in$ *space lborel*›]
measurable-add-dim[*of n* {..<*n*} λb. *lborel*], *unfolded 0*] *lessThan-Suc* **by** *auto*
qed

lemma *sets-lborel-f-sub-dim*:
assumes $A \in$ *sets* (*lborel-f* (*Suc n*))
shows $\{x.\ x(n := y) \in A\} \cap$ *space* (*lborel-f n*) $\in$ *sets* (*lborel-f n*)
proof −
have $(\lambda f.\ f(n := y))$ −‘ $A \cap$ *space* (*lborel-f n*) $\in$ *sets* (*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 (λf. *restrict f* {..<*m*}) $\in$ *measurable* (*lborel-f n*) (*lborel-f m*)
using *measurable-restrict-subset lborel-f-def assms* **by** *auto*

lemma *lborel-measurable-sub-dim*: (λf. *restrict f* {..<*n*}) $\in$ *measurable* (*lborel-f* (*Suc n*)) (*lborel-f n*)
using *lborel-f-measurable-restrict*[*of n Suc n*] **by** *linarith*

lemma *measurable-lborel-component* [*measurable*]:
assumes *k*<*n*
shows (λx. *x k*) $\in$ *borel-measurable* (*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* ⊆ {..<*n*}
shows (λ*f*. *insertion f p*) ∈ *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* ‹*a* = *0*› *MPoly-Type.monom.abs-eq single-zero*
zero-mpoly.abs-eq[*symmetric*] *insertion-zero* **by** *measurable*
next
case *False*
have *Poly-Mapping.keys m* ⊆ {..<*n*} **using** *monom* **by** (*simp add*: *False vars-monom-keys*)
then show *?thesis* **using** ‹*a*≠*0*›
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* ‹*i* = *0*› **by** *simp*
next
case *False*
then show *?thesis* **unfolding** *insertion-single* **apply** *measurable*
using *vars-monom-single-cases single False insert-subset lessThan-iff* ‹*a*≠*0*› **by** *fastforce*
qed
next
case (*sum m1 m2 x i*)
then have *Poly-Mapping.keys m1* ∩ *Poly-Mapping.keys m2* = {} **by** *simp*
then have *Poly-Mapping.keys m1* ∪ *Poly-Mapping.keys m2* = *Poly-Mapping.keys* (*m1* + *m2*) **using** *keys-add* **by** *metis*
then have *1*:*Poly-Mapping.keys m1* ⊆ {..<*n*} **and** *2*:*Poly-Mapping.keys m2* ⊆ {..<*n*} **using** *sum.prems* **by** *auto*
show *?case* **unfolding** *MPoly-Type.mult-monom*[*of m1 a m2 1*,*simplified*,*symmetric*]
insertion-mult **using** *sum.IH*(*1*)[*OF 1* ‹*a*≠*0*›] *sum.IH*(*2*)[*OF 2*, *of 1*, *simplified*] **by** *measurable*
qed
qed
next
case (*sum p1 p2 m a*)
then have (λ*f*. *insertion f p1*) ∈ *borel-measurable* (*lborel-f n*)
(λ*f*. *insertion f p2*) ∈ *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 $p \neq 0$ *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** ⋀*f*. *insertion f p* = *MPoly-Type.coeff p 0*
    unfolding *insertion-trivial*[*symmetric*] **using** *insertion-irrelevant-vars* **by** *blast*
  have ⋀*m*. *m*≠*0* ⟹ *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** ‹*MPoly-Type.coeff p m* ≠ *0*›
      by (*meson UN-I coeff-keys lookup-not-eq-zero-eq-in-keys*)
    then show *False* **using** ‹*vars p* = {}› **by** *auto*
  qed
  then have *MPoly-Type.coeff p 0* ≠ *0* **using** ‹*p* ≠ *0*›
    by (*metis coeff-all-0*)
  then have {*f*. *insertion f p* = *0*} = {} **using** ‹⋀*f*. *insertion f p* = *MPoly-Type.coeff p 0*› **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* = λ*j*. *MPoly-Type.coeff* (*extract-var p n*) *j*
  obtain *j* **where** *?q j* ≠ *0* **using** ‹*extract-var p n* ≠ *0*›
    by (*metis coeff-all-0*)
  then have *finite* {*x*. *insertion* (λ-. *x*) (*?q j*) = *0*}
    using *univariate-mpoly-roots-finite*[*OF vars-coeff-extract-var*] **by** *metis*
  then have *finite* (⋂*j*. {*x*. *insertion* (λ-. *x*) (*?q j*) = *0*}) **by** *auto*
  moreover have {*x*. ∀*j*. *insertion* (λ-. *x*) (*?q j*) = *0*} = (⋂*j*. {*x*. *insertion* (λ*v*. *x*) (*?q j*) = *0*}) **by** *blast*
  ultimately have *finite* {*x*. ∀*j*. *insertion* (λ-. *x*) (*?q j*) = *0*} **by** *metis*

  define *p-fix1* **where** *p-fix1 x1* = *replace-coeff* (*insertion* (λ-. *x1*)) (*extract-var p n*) **for** *x1*
  define *N* **where** *N* = {*x1*. *p-fix1 x1* = *0*}
  have *N* ⊆ {*x*. ∀*j*. *insertion* (λ-. *x*) (*?q j*) = *0*}

proof
fix *x* **assume** *x*∈*N*
then have *p-fix1 x* = *0* **using** *N-def* **by** *auto*
then have ⋀*m*. *MPoly-Type.coeff* (*p-fix1 x*) *m* = *0* **by** (*metis More-MPoly-Type.coeff-monom monom-zero when-def*)
have ⋀*j*. *insertion* (λ-. *x*) (*?q j*) = *0*
using ‹⋀*m*. *MPoly-Type.coeff* (*p-fix1 x*) *m* = *0*›[*unfolded p-fix1-def coeff-replace-coeff*[*of insertion* (λ-. *x*), *OF insertion-zero*]]
by *metis*
then show *x* ∈ {*x*. ∀ *j*. *insertion* (λ-. *x*) (*MPoly-Type.coeff* (*extract-var p n*) *j*) = *0*} **by** *blast*
qed
then have *finite N* **by** (*simp add*: ‹*finite* {*x*. ∀ *j*. *insertion* (λ-. *x*) (*MPoly-Type.coeff* (*extract-var p n*) *j*) = *0*}› *finite-subset*)

Use the IH:

define *A* **where** *A* = {*f*∈*space* (*lborel-f* (*Suc n*)). *insertion f p* = *0*}

have ⋀*x1*. *vars* (*p-fix1 x1*) ⊆ {..<*n*}
proof −
fix *x1*
have *vars* (*extract-var p n*) ⊆ {..<*n*}
using ‹*vars p* ⊆ {..<*Suc n*}› *lessThan-Suc v-not-in-vars-extract-var vars-extract-var-subset* **by** *fastforce*
then show *vars* (*p-fix1 x1*) ⊆ {..<*n*} **unfolding** *p-fix1-def*
using *vars-replace-coeff*[*of insertion* (λ-. *x1*), *OF insertion-zero*] **by** *blast*
qed
have *set-eq*:⋀*x1*. {*x* ∈ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = {*f* ∈ *space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*}
proof −
fix *x1*
show {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = {*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*}
proof (*rule subset-antisym*;*rule subsetI*)
fix *x* **assume** *x* ∈ {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*}
then have *insertion* (*x*(*n* := *x1*)) *p* = *0 x* ∈ *space* (*lborel-f n*)
using *A-def* **by** *auto*
then have *insertion x* (*p-fix1 x1*) = *0* **unfolding** *p-fix1-def*
unfolding *replace-coeff-extract-var-cong*[*of* λ-. *x1 n x*(*n* := *x1*) *p*, *OF fun-upd-same*[*symmetric*]]
using *insertion-replace-coeff*[*of x*(*n* := *x1*)]
using *insertion-irrelevant-vars*[*of replace-coeff* (*insertion* (*x*(*n* := *x1*))) (*extract-var p n*) *x x*(*n* := *x1*)]
vars-replace-coeff fun-upd-other insertion-zero reduce-nested-mpoly-extract-var subset-eq
v-not-in-vars-extract-var **by** *metis*
then show *x* ∈ {*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} **using** ‹*x* ∈ *space* (*lborel-f n*)› **by** *blast*
next

fix *f* **assume** *f* ∈ {*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*}
then have *f*∈*space* (*lborel-f n*) *insertion f* (*p-fix1 x1*) = *0* **by** *auto*
have *insertion* (*f*(*n* := *x1*)) *p* = *0* **using** ‹*insertion f* (*p-fix1 x1*) = *0*›[*unfolded p-fix1-def*]
insertion-replace-coeff insertion-irrelevant-vars replace-coeff-extract-var-cong
by (*metis* (*no-types*, *lifting*) ‹*insertion f* (*p-fix1 x1*) = *0*› ‹*vars* (*p-fix1 x1*) ⊆ {..<*n*}›
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*) ∈ *A* **unfolding** *A-def* **using** *space-lborel-add-dim*
using ‹*f* ∈ *space* (*lborel-f n*)› *lborel-f-def mem-Collect-eq* **by** *blast*
then show *f* ∈ {*f*∈*space* (*lborel-f n*). *f*(*n* := *x1*) ∈ *A*} **using** ‹*f* ∈ *space* (*lborel-f n*)› **by** *auto*
qed
qed

have ⋀*x1*. *x1*∈*N* ⟹ {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)
and *emeasure-in-N*: ⋀*x1*. *x1*∈*N* ⟹ *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *emeasure* (*lborel-f n*) (*space* (*lborel-f n*))
proof −
fix *x1* **assume** *x1*∈*N*
then have *p-fix1 x1* = *0* **using** *N-def* **by** *auto*
then have ⋀*f*. *insertion f* (*p-fix1 x1*) = *0* **using** *insertion-zero* **by** *auto*
then have {*f* ∈ *space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} = *space* (*lborel-f n*) **by** *simp*
show {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*) **unfolding** *set-eq*
by (*simp add*: ‹{*f* ∈ *space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} = *space* (*lborel-f n*)›)
show *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *emeasure* (*lborel-f n*) (*space* (*lborel-f n*))
unfolding *set-eq*
by (*simp add*: ‹{*f* ∈ *space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} = *space* (*lborel-f n*)›)
qed

have *emeasure-not-in-N*: ⋀*x1*. *x1*∉*N* ⟹ *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *0*
and ⋀*x1*. *x1*∉*N* ⟹ {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)
proof −
fix *x1* **assume** *x1*∉*N*
then have *p-fix1 x1* ≠ *0* **using** *p-fix1-def N-def* **by** *auto*
then have *emeasure* (*lborel-f n*) {*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} = *0*
{*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} ∈ *sets* (*lborel-f n*)
using *Suc.IH*[*OF* ‹*p-fix1 x1* ≠ *0*›] ‹⋀*x1*. *vars* (*p-fix1 x1*) ⊆ {..<*n*}› **by** *auto*
then show *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *0*
{*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)

using ‹{f∈space (lborel-f n). insertion f (p-fix1 x1) = 0} ∈ sets (lborel-f n)›
‹emeasure (lborel-f n) {f∈space (lborel-f n). insertion f (p-fix1 x1) = 0} = 0›
using *set-eq*
by *auto*
qed

have N ∈ null-sets lborel **using** ‹finite N› finite-imp-null-set-lborel **by** *blast*
have *ae-zero*: AE x1 in lborel. emeasure (lborel-f n) {x ∈ space (lborel-f n). x(n := x1) ∈ A} = 0
apply (*rule AE-I'*[*OF* ‹N ∈ null-sets lborel›])
using ‹⋀x1. x1∉N ⟹ emeasure (lborel-f n) {x∈space (lborel-f n). x(n := x1) ∈ A} = 0›
by *force*

have *measurable*: (λx1. emeasure (lborel-f n) {x ∈ space (lborel-f n). x(n := x1) ∈ A}) ∈ borel-measurable lborel
proof (*rule borel-measurableI*)
let ?f = (λx1. emeasure (lborel-f n) {x ∈ space (lborel-f n). x(n := x1) ∈ A})
fix S::ennreal set **assume** *open S*
have 0:0∈S ⟹ − N ⊆ ?f −‘ S
using *emeasure-not-in-N* **by** *auto*
have 1:emeasure (lborel-f n) (space (lborel-f n)) ∈ S ⟹ N ⊆ ?f −‘ S
using *emeasure-in-N* **by** *auto*
have 2:0∉S ⟹ ?f −‘ S ⊆ N **using** *emeasure-not-in-N* **by** *fastforce*
have 3:emeasure (lborel-f n) (space (lborel-f n)) ∉ S ⟹ ?f −‘ S ⊆ −N **using** *emeasure-in-N* **by** *auto*
have ?f −‘ S = {} ∨ ?f −‘ S = N ∨ ?f −‘ S = UNIV ∨ ?f −‘ S = − N
apply (*cases 0∈S*; *cases emeasure* (*lborel-f n*) (*space* (*lborel-f n*)) ∉ *S*)
using *0 1 2 3* **by** *auto*
then show ?f −‘ S ∩ space lborel ∈ sets lborel
using ‹finite N› finite-imp-null-set-lborel borel-comp null-setsD2 sets-lborel **by** *fastforce*
qed

have A ∈ sets (lborel-f (Suc n)) **unfolding** *A-def*
using *pred-eq-const1*[*OF measurable-insertion*[*OF* ‹vars p ⊆ {..<Suc n}›]] *pred-def* **by** *force*
then have *in-sets*: {f ∈ space (lborel-f (Suc n)). insertion f p = 0} ∈ sets (lborel-f (Suc n)) **using** *A-def* **by** *metis*
have ⋀x1. {x∈space (lborel-f n). x(n := x1) ∈ A} ∈ sets (lborel-f n)
using ‹⋀x1. x1∈N ⟹ {x∈space (lborel-f n). x(n := x1) ∈ A} ∈ sets (lborel-f n)›
‹⋀x1. x1∉N ⟹ {x∈space (lborel-f n). x(n := x1) ∈ A} ∈ sets (lborel-f n)› **by** *auto*
have emeasure (lborel-f (Suc n)) A = ∫⁺ y. emeasure (lborel-f n) {x ∈ space (lborel-f n). x(n := y) ∈ A} ∂lborel
using *emeasure-lborel-f-Suc*[*OF* ‹A ∈ sets (lborel-f (Suc n))›]
‹⋀x1. {x∈space (lborel-f n). x(n := x1) ∈ A} ∈ sets (lborel-f n)› **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 output-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.prems*(*1*) *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.prems*(*1*) *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.prems*(*1*) *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 tensors-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 = *insert-weights s* (*shallow-model Y Z M N*) *w*
assumes $y < Y$
shows *cprank* (*tensors-from-net m* \$ *y*) $\leq Z$
proof −
have $s \Longrightarrow$ *shared-weight-net m*
by (*simp add*: *assms*(*1*) *balanced-net-shallow-model shared-weight-net-insert-weights*)
have *cprank-max Z* (*tensors-from-net m* \$ *y*)
proof −
have *dim-extract*: *dim-row* (*extract-matrix w Y Z*) = *Y*
using *dim-extract-matrix*(*1*) **by** *force*
have *dimc-extract-matrix*: *dim-col* (*extract-matrix w Y Z*) = *Z*
using *dim-extract-matrix*(*2*) **by** *force*
have *input-sizes*: (*input-sizes* (*insert-weights s* (*shallow-model′ Z M N*) (λi. *w* ($i + Y * Z$)))) = (*input-sizes* (*shallow-model′ Z M N*))
using *input-sizes-remove-weights remove-insert-weights* **by** *auto*
have *0*:*tensors-from-net m* \$ *y* = *Tensor-Plus.listsum* (*input-sizes* (*shallow-model′ Z M N*))
(*map* (λj. (*extract-matrix w Y Z*) \$\$ (*y*, *j*) · (*tensors-from-net* (*insert-weights s* (*shallow-model′ Z M N*) (λi. *w* ($i + Y * Z$)))) \$ *j*) [*0*..<*Z*])
unfolding ‹*m* = *insert-weights s* (*shallow-model Y Z M N*) *w*› *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* = *map* (λj. *extract-matrix w Y Z* \$\$ (*y*, *j*) · *tensors-from-net* (*insert-weights s* (*shallow-model′ Z M N*) (λi. *w* ($i + Y * Z$))) \$ *j*) [*0*..<*Z*]

have $\bigwedge B$. $B \in$ *set Bs* $\Longrightarrow$ *cprank-max1 B* $\bigwedge B$. $B \in$ *set Bs* $\Longrightarrow$ *dims B* = *input-sizes* (*shallow-model′ Z M N*)
proof −
fix *B* **assume** $B \in$ *set Bs*
then obtain *j* **where** *B* = *Bs* ! *j j* < *length Bs* **by** (*metis in-set-conv-nth*)
then have $j < Z$ **using** *length-map Bs-def* **by** *simp*
have *1*:*cprank-max1* (*tensors-from-net* (*insert-weights s* (*shallow-model′ Z M N*) (λi. *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* $$ (*y*, *j*) · *tensors-from-net* (*insert-weights s* (*shallow-model′ Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*))) $ *j*)
using *smult-prod-extract1 cprank-max1-order0*[*OF 1*, *of extract-matrix w Y Z* $$ (*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-maxI 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* (*submatrix* (*matricize* {*n*. *even n*} (*A w*)) *rows-with-1 rows-with-1*)

lemma *polyfun-polynomial*:
shows *polyfun* {..<*weight-space-dim*} *polynomial-f*
unfolding *polynomial-f-def* **using** *polyfun-det-deep-model* **unfolding** *witness-submatrix-def A′-def* **.**

definition *polynomial-p* = (*SOME p*. *vars p* ⊆ {..<*weight-space-dim*} ∧ (∀ *x*. *insertion x p* = *polynomial-f x*))

lemma
polynomial-p-not-0: *polynomial-p*≠*0* **and**

vars-polynomial-p: *vars polynomial-p* ⊆ {..<*weight-space-dim*} **and**
polynomial-pf: ⋀*w. insertion w polynomial-p* = *polynomial-f w*
proof −
  have *vars polynomial-p* ⊆ {..<*weight-space-dim*} ∧ (∀ *x. insertion x polynomial-p* = *polynomial-f x*) **unfolding** *polynomial-p-def*
    using *someI-ex*[*OF polyfun-polynomial*[*unfolded polyfun-def*]] **.**
  then show *vars polynomial-p* ⊆ {..<*weight-space-dim*} ⋀*w. insertion w polynomial-p* = *polynomial-f w* **by** *auto*
  show *polynomial-p*≠*0* **using** *A′-def Aw′-def′* ‹⋀*w. insertion w polynomial-p* = *polynomial-f w*› *polynomial-f-def witness-det* **by** *auto*
qed

lemma *if-polynomial-0-rank*:
assumes *polynomial-f w* ≠ *0*
shows *r* ^ *N-half* ≤ *cprank* (*A w*)
proof −
  have *r* ^ *N-half* = *dim-row* (*submatrix* (*matricize* {*n. even n*} (*A w*)) *rows-with-1 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 ... ≤ *mrank* (*matricize* {*n. 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 ... ≤ *cprank* (*A w*) **using** *matrix-rank-le-cp-rank* **by** *blast*
  finally show *?thesis* **.**
qed

lemma *if-polynomial-0-evaluate*:
assumes *polynomial-f wd* ≠ *0*
assumes ∀ *inputs. input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶ *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 *Z* ≥ *r* ^ *N-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*)
    = *input-sizes* (*insert-weights shared-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2* ∗ *N-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*)
      = *tensors-from-net* (*insert-weights shared-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*∗*N-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*) ≤ *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* ≠ *0*
shows ¬(∃ *weights-shallow Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
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*))
using *assms if-polynomial-0-evaluate not-le* **by** *blast*

theorem *fundamental-theorem-network-capacity*:
AE x in lborel-f weight-space-dim. *r* ^ *N-half* ≤ *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*.
¬(∃ *ws Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
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*))
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* ≡ *completion* (*lborel-f n*)

lemma *space-lebesgue-f*: *space* (*lebesgue-f n*) = Pi_E {..<*n*} (λ-. *UNIV*)
by (*simp add*: *space-lborel-f*)

theorem *fundamental-theorem-network-capacity-v3*:
assumes
S = {*wd* ∈ *space* (*lebesgue-f weight-space-dim*).
∃ *ws Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
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.