Expressiveness of Deep Learning

Alexander Bentkamp

May 14, 2024

Abstract

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

Contents

1 Tensor

theory *Tensor*
imports *Main*
begin

typedef *'a tensor* = {*t*::*nat list* × *'a list*. *length* (*snd t*) = *prod-list* (*fst t*)}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *tensor-from-vec-simp*[*simp*]: *tensor-from-vec* (*dims A*) (*vec A*) = *A*
⟨*proof*⟩

lemma *length-vec*: *length* (*vec A*) = *prod-list* (*dims A*)
⟨*proof*⟩

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

assumes *dims A* = *dims B* **and** *vec A* = *vec B*
shows *A*=*B*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *valid-index-lt*: *is* ◁ *ds* ⟹ *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*
⟨*proof*⟩

lemma *valid-indexI*:
assumes *length is* = *length ds* **and** ⋀*m*. *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*
shows *is* ◁ *ds*
⟨*proof*⟩

lemma *valid-index-append*:
assumes *is1-valid*:*is1* ◁ *ds1* **and** *is2-valid*:*is2* ◁ *ds2*
shows *is1* @ *is2* ◁ *ds1* @ *ds2*
⟨*proof*⟩

lemma *valid-index-list-all2-iff*: *is* ◁ *ds* ⟷ *list-all2* (<) *is ds*
⟨*proof*⟩

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 $a * d \leq length\ v$
shows *concat* (*map* (*fixed-length-sublist v d*) [*0*..<*a*]) = *take* (*a*∗*d*) *v*
⟨*proof*⟩

lemma *concat-parts-eq*:
assumes $a * d = length\ v$
shows *concat* (*map* (*fixed-length-sublist v d*) [*0*..<*a*]) = *v*
⟨*proof*⟩

lemma *tensor-lookup-base*:
assumes *length v* = *prod-list ds*
and $\bigwedge is.\ is \lhd ds \Longrightarrow lookup\text{-}base\ ds\ v\ is = e\ is$
shows *tensor-vec-from-lookup ds e* = *v*
⟨*proof*⟩

lemma *tensor-lookup*:
assumes $\bigwedge is.\ is \lhd dims\ A \Longrightarrow lookup\ A\ is = e\ is$
shows *tensor-from-lookup* (*dims A*) *e* = *A*
⟨*proof*⟩

lemma *concat-equal-length*:
assumes $\bigwedge xs.\ xs{\in}set\ xss \Longrightarrow length\ xs = l$
shows *length* (*concat xss*) = *length xss*∗*l*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *concat-parts*:
assumes $\bigwedge xs.\ xs{\in}set\ xss \Longrightarrow length\ xs = d$ **and** $i{<}length\ xss$
shows *fixed-length-sublist* (*concat xss*) *d i* = *xss* ! *i*
⟨*proof*⟩

lemma *concat-parts'*:
assumes $\bigwedge i.\ i{<}a \Longrightarrow length\ (f\ i) = d$
and $i{<}a$
shows *fixed-length-sublist* (*concat* (*map* (λ*i*. *f i*) [*0*..<*a*])) *d i* = *f i*
⟨*proof*⟩

lemma *length-tensor-vec-from-lookup*:
length (*tensor-vec-from-lookup ds e*) = *prod-list ds*
⟨*proof*⟩

lemma *lookup-tensor-vec*:
assumes $is \lhd ds$
shows *lookup-base ds* (*tensor-vec-from-lookup ds e*) *is* = *e is*
⟨*proof*⟩

lemma *lookup-tensor-from-lookup*:
assumes $is \lhd ds$
shows *lookup* (*tensor-from-lookup ds e*) *is* = *e is*
⟨*proof*⟩

lemma *dims-tensor-from-lookup*: *dims* (*tensor-from-lookup ds e*) = *ds*
⟨*proof*⟩

lemma *tensor-lookup-cong*:
assumes *tensor-from-lookup ds* e_1 = *tensor-from-lookup ds* e_2
and $is \lhd ds$
shows e_1 *is* = e_2 *is* ⟨*proof*⟩

lemma *tensor-from-lookup-eqI*:
assumes $\bigwedge is.\ is \lhd ds \Longrightarrow e_1\ is = e_2\ is$
shows *tensor-from-lookup ds* e_1 = *tensor-from-lookup ds* e_2
⟨*proof*⟩

lemma *tensor-lookup-eqI*:
assumes *dims A* = *dims B* **and** $\bigwedge is.\ is \lhd (dims\ A) \Longrightarrow lookup\ A\ is = lookup\ B\ is$
shows *A* = *B* ⟨*proof*⟩

end

2 Subtensors

theory *Tensor-Subtensor*
imports *Tensor*
begin

definition *subtensor*::*'a tensor* ⇒ *nat* ⇒ *'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* ⇒ *'a tensor list* ⇒ *'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* ≤ *length xs*
shows *length* (*fixed-length-sublist xs l i*) = *l*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *dims-subtensor*[*simp*]:
assumes *dims* $A \neq []$ **and** $i <$ *hd* (*dims* A)
shows *dims* (*subtensor* A i) = *tl* (*dims* A)
⟨*proof*⟩

lemma *subtensor-combine-subtensor*[*simp*]:
assumes *dims* $A \neq []$
shows *subtensor-combine* (*tl* (*dims* A)) (*map* (*subtensor* A) [*0*..<*hd* (*dims* A)]) = A
⟨*proof*⟩

lemma
assumes $\bigwedge A.$ $A \in$ *set As* $\Longrightarrow$ *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*⟩

lemma *subtensor-subtensor-combine*:
assumes $\bigwedge A.$ $A \in$ *set As* $\Longrightarrow$ *dims* A = *ds* **and** $i <$ *length As*
shows *subtensor* (*subtensor-combine ds As*) i = *As* ! i
⟨*proof*⟩

lemma *subtensor-induct*[*case-names order-0 order-step*]:
assumes *order-0*: $\bigwedge A.$ *dims* $A = []$ $\Longrightarrow$ P A
and *order-step*: $\bigwedge A.$ *dims* $A \neq []$ $\Longrightarrow$ ($\bigwedge i.$ $i <$ *hd* (*dims* A) $\Longrightarrow$ P (*subtensor* A i)) $\Longrightarrow$ P A
shows P B
⟨*proof*⟩

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*⟩

lemma *lookup-subtensor1*[*simp*]:
assumes i # *is* $\lhd$ *dims* A
shows *lookup* (*subtensor* A i) *is* = *lookup* A (i # *is*)
⟨*proof*⟩

lemma *lookup-subtensor*:

assumes *is* $\lhd$ *dims A*
shows *lookup A is* = *hd* (*vec* (*fold* (λ*i A*. *subtensor A i*) *is A*))
⟨*proof*⟩

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*⟩

end

3 Tensor Addition

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

definition *vec-plus a b* = *map* (λ(*x*,*y*). *plus x y*) (*zip a b*)

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

instantiation *tensor*:: (*semigroup-add*) *plus*
begin
  definition *plus-def*: *A* + *B* = (*if* (*dims A* = *dims B*)
                then plus-base A B
                else undefined)
  instance ⟨*proof*⟩
end

lemma *plus-dim1*[*simp*]: *dims A* = *dims B* $\Longrightarrow$ *dims* (*A* + *B*) = *dims A* ⟨*proof*⟩
lemma *plus-dim2*[*simp*]: *dims A* = *dims B* $\Longrightarrow$ *dims* (*A* + *B*) = *dims B* ⟨*proof*⟩
lemma *plus-base*: *dims A* = *dims B* $\Longrightarrow$ *A* + *B* = *plus-base A B* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vec-plus*[*simp*]:
assumes *dims A* = *dims B*
shows *vec* (*A*+*B*) = *vec-plus* (*vec A*) (*vec B*)
  ⟨*proof*⟩

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 $dims\ A \neq []$
shows $subtensor\ (A + B)\ i = subtensor\ A\ i + subtensor\ B\ i$
⟨*proof*⟩

lemma *lookup-plus*[*simp*]:
assumes $dims\ A = dims\ B$
and $is \lhd dims\ A$
shows $lookup\ (A + B)\ is = lookup\ A\ is + lookup\ B\ is$
⟨*proof*⟩

lemma *plus-assoc*:
assumes *dimsA*: $dims\ A = ds$ **and** *dimsB*: $dims\ B = ds$ **and** *dimsC*: $dims\ C = ds$
shows $(A + B) + C = A + (B + C)$
⟨*proof*⟩

lemma *tensor-comm*[*simp*]:
fixes A::$'a$::*ab-semigroup-add tensor*
shows $A + B = B + A$
⟨*proof*⟩

definition $vec0\ n = replicate\ n\ 0$

definition *tensor0*::*nat list* ⇒ $'a$::*zero tensor* **where**
$tensor0\ d = tensor\text{-}from\text{-}vec\ d\ (vec0\ (prod\text{-}list\ d))$

lemma *dims-tensor0*[*simp*]: $dims\ (tensor0\ d) = d$
and *vec-tensor0*[*simp*]: $vec\ (tensor0\ d) = vec0\ (prod\text{-}list\ d)$
⟨*proof*⟩

lemma *lookup-is-in-vec*: $is \lhd (dims\ A) \Longrightarrow lookup\ A\ is \in set\ (vec\ A)$
⟨*proof*⟩

lemma *lookup-tensor0*:
assumes $is \lhd ds$
shows $lookup\ (tensor0\ ds)\ is = 0$
⟨*proof*⟩

lemma
fixes A::$'a$::*monoid-add tensor*
shows *tensor-add-0-right*[*simp*]: $A + tensor0\ (dims\ A) = A$
⟨*proof*⟩

lemma
fixes A::$'a$::*monoid-add tensor*
shows *tensor-add-0-left*[*simp*]: $tensor0\ (dims\ A) + A = A$

$\langle proof \rangle$

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

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

lemma *listsum-Nil*: *listsum ds* [] = *tensor0 ds* $\langle proof \rangle$

lemma *listsum-one*: *listsum* (*dims A*) [*A*] = *A* $\langle proof \rangle$

lemma *listsum-Cons*: *listsum ds* (*A* # *As*) = *A* + *listsum ds As*
$\langle proof \rangle$

lemma *listsum-dims*:
assumes $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
shows *dims* (*listsum ds As*) = *ds*
$\langle proof \rangle$

lemma *subtensor0*:
assumes $ds \neq []$ **and** $i < hd\ ds$
shows *subtensor* (*tensor0 ds*) *i* = *tensor0* (*tl ds*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *listsum0*:
assumes $\bigwedge A.\ A \in set\ As \Longrightarrow A = tensor0\ ds$
shows *listsum ds As* = *tensor0 ds*
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *lookup-listsum*:
assumes $is \lhd ds$
and $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
shows *lookup* (*listsum ds As*) *is* = ($\sum A{\leftarrow}As.\ lookup\ A\ is$)

⟨*proof*⟩

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* $((*)\ \alpha)$ β

lemma *vec-smult0*: *vec-smult* *0* *as* = *vec0* (*length as*)
⟨*proof*⟩

lemma *vec-smult-distr-right*:
shows *vec-smult* $(\alpha + \beta)$ *as* = *vec-plus* (*vec-smult* α *as*) (*vec-smult* β *as*)
⟨*proof*⟩

lemma *vec-smult-Cons*:
shows *vec-smult* α (*a* # *as*) = $(\alpha * a)$ # *vec-smult* α *as* ⟨*proof*⟩

lemma *vec-plus-Cons*:
shows *vec-plus* (*a* # *as*) (*b* # *bs*) = (*a*+*b*) # *vec-plus as bs* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *length-vec-smult*: *length* (*vec-smult* α *v*) = *length v* ⟨*proof*⟩

definition *smult*::$'a$::*ring* $\Rightarrow$ $'a$ *tensor* $\Rightarrow$ $'a$ *tensor* (**infixl** $\cdot$ *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* $\cdot$ *A* = *tensor0* (*dims A*)
⟨*proof*⟩

lemma *dims-smult*[*simp*]:*dims* $(\alpha \cdot A)$ = *dims A*
and *vec-smult*[*simp*]: *vec* $(\alpha \cdot A)$ = *map* $((*)\ \alpha)$ (*vec A*)
⟨*proof*⟩

lemma *tensor-smult-distr-right*: $(\alpha + \beta) \cdot A = \alpha \cdot A + \beta \cdot A$
⟨*proof*⟩

lemma *tensor-smult-distr-left*: *dims A* = *dims B* $\Longrightarrow$ $\alpha \cdot (A + B) = \alpha \cdot A \ + \alpha \cdot B$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *smult-subtensor*:
assumes *dims A* ≠ [] *i* < *hd* (*dims A*)
shows α · *subtensor A i* = *subtensor* (α · *A*) *i*
⟨*proof*⟩

lemma *lookup-smult*:
assumes *is* ◁ *dims A*
shows *lookup* (α · *A*) *is* = α ∗ *lookup A is*
⟨*proof*⟩

lemma *tensor-smult-assoc*:
fixes *A*::′*a*::*ring tensor*
shows α · (β · *A*) = (α ∗ β) · *A*
⟨*proof*⟩

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* (λ*a*. *vec-smult a* (*vec B*)) (*vec A*)))

abbreviation *tensor-prod-otimes* :: ′*a tensor* ⇒ ′*a tensor* ⇒ ′*a tensor* (**infixl** ⊗ *70*)
where *A* ⊗ *B* ≡ *A* ∗ *B*

lemma *vec-tensor-prod*[*simp*]: *vec* (*A* ⊗ *B*) = *concat* (*map* (λ*a*. *vec-smult a* (*vec B*)) (*vec A*)) (**is** *?V*)
and *dims-tensor-prod*[*simp*]: *dims* (*A* ⊗ *B*) = *dims A* @ *dims B* (**is** *?D*)
⟨*proof*⟩

lemma *tensorprod-subtensor-base*:

shows *concat* (*map f* (*concat xss*)) = *concat* (*map* (λ*xs*. *concat* (*map f xs*)) *xss*)
⟨*proof*⟩

lemma *subtensor-combine-tensor-prod*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
shows *subtensor-combine ds As* ⊗ *B* = *subtensor-combine* (*ds* @ *dims B*) (*map* (λ*A*. *A* ⊗ *B*) *As*)
⟨*proof*⟩

lemma *subtensor-tensor-prod*:
assumes *dims A* ≠ [] **and** *i* < *hd* (*dims A*)
shows *subtensor* (*A* ⊗ *B*) *i* = *subtensor A i* ⊗ *B*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *valid-index-split*:
assumes *is* ⊲ *ds1* @ *ds2*
obtains *is1 is2* **where** *is1* @ *is2* = *is is1* ⊲ *ds1 is2* ⊲ *ds2*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *tensor-prod-distr-left*:
assumes *dims A* = *dims B*
shows (*A* + *B*) ⊗ *C* = (*A* ⊗ *C*) + (*B* ⊗ *C*)
⟨*proof*⟩

lemma *tensor-prod-distr-right*:
assumes *dims A* = *dims B*
shows *C* ⊗ (*A* + *B*) = (*C* ⊗ *A*) + (*C* ⊗ *B*)
⟨*proof*⟩

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*)
⟨*proof*⟩

instance ⟨*proof*⟩
end

lemma *order-tensor-one*: *order 1 = 0* ⟨*proof*⟩

lemma *smult-prod-extract1*:
fixes *a*::′*a*::*comm-ring-1*
shows $a \cdot (A \otimes B) = (a \cdot A) \otimes B$
⟨*proof*⟩

lemma *smult-prod-extract2*:
fixes *a*::′*a*::*comm-ring-1*
shows $a \cdot (A \otimes B) = A \otimes (a \cdot B)$
⟨*proof*⟩

lemma *order-0-multiple-of-one*:
assumes *order A = 0*
obtains *a* **where** $A = a \cdot 1$
⟨*proof*⟩

lemma *smult-1*:
fixes *A*::′*a*::*ring-1 tensor*
shows $A = 1 \cdot A$ ⟨*proof*⟩

lemma *tensor0-prod-right*[*simp*]: $A \otimes$ *tensor0 ds = tensor0* (*dims A* @ *ds*)
⟨*proof*⟩

lemma *tensor0-prod-left*[*simp*]: *tensor0 ds* $\otimes A$ = *tensor0* (*ds* @ *dims A*)
⟨*proof*⟩

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*⟩

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*] ⟨*proof*⟩

lemma *lookup-unit-vec*:
assumes $j<n$

shows *lookup* (*unit-vec n i*) [*j*] = (*if i=j then 1 else 0*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *cprank-max1-order-le1*: *order A* ≤ *0* ⟹ *cprank-max1 B* ⟹ *cprank-max1* (*A* ⊗ *B*)
⟨*proof*⟩

lemma *cprank-max1-prod*: *cprank-max1 A* ⟹ *cprank-max1 B* ⟹ *cprank-max1* (*A* ⊗ *B*)
⟨*proof*⟩

lemma *cprank-max1-prod-list*:
assumes ⋀*B. B*∈*set Bs* ⟹ *cprank-max1 B*
shows *cprank-max1* (*prod-list Bs*)
⟨*proof*⟩

lemma *cprank-max1-prod-listE*:
fixes *A*::′*a*::*comm-ring-1 tensor*
assumes *cprank-max1 A*
obtains *Bs a* **where** ⋀*B. B*∈*set Bs* ⟹ *order B* = *1 a* · *prod-list Bs* = *A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cprank-max-plus*: *cprank-max i A* ⟹ *cprank-max j B* ⟹ *dims A* = *dims B* ⟹ *cprank-max* (*i*+*j*) (*A*+*B*)
⟨*proof*⟩

lemma *cprank-max-listsum*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
and ⋀*A*. *A*∈*set As* ⟹ *cprank-max n A*
shows *cprank-max* (*n*∗*length As*) (*listsum ds As*)
⟨*proof*⟩

lemma *cprank-maxE*:
assumes *cprank-max n A*
obtains *BS* **where** (⋀*B*. *B*∈*set BS* ⟹ *cprank-max1 B*) **and** (⋀*B*. *B*∈*set BS* ⟹ *dims A* = *dims B*) **and** *listsum* (*dims A*) *BS* = *A* **and** *length BS* = *n*
⟨*proof*⟩

lemma *cprank-maxI*:
assumes ⋀*B*. *B*∈*set BS* ⟹ *cprank-max1 B*
and ⋀*B*. *B*∈*set BS* ⟹ *dims B* = *ds*
shows *cprank-max* (*length BS*) (*listsum ds BS*)
⟨*proof*⟩

lemma *cprank-max-0E*: *cprank-max 0 A* ⟹ *A* = *tensor0* (*dims A*) ⟨*proof*⟩

lemma *listsum-prod-distr-right*:
assumes (⋀*C*. *C*∈*set CS* ⟹ *dims C* = *ds*)
shows *A* ⊗ *listsum ds CS* = *listsum* (*dims A* @ *ds*) (*map* (λ*C*. *A* ⊗ *C*) *CS*)
⟨*proof*⟩

lemma *cprank-max-prod-order1*:
assumes *order A* = *1*
and *cprank-max n B*
shows *cprank-max n* (*A*⊗*B*)
⟨*proof*⟩

lemma *cprank-max-upper-bound*:
shows *cprank-max* (*prod-list* (*dims A*)) *A*
⟨*proof*⟩

definition *cprank* :: *'a::ring-1 tensor* ⇒ *nat* **where**

cprank A = (*LEAST* n. *cprank-max* n A)

lemma *cprank-upper-bound*: *cprank* $A \leq$ *prod-list* (*dims* A)
⟨*proof*⟩

lemma *cprank-max-cprank*: *cprank-max* (*cprank* A) A
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *digit-decode-encode*[*simp*]:
shows *digit-decode ds* (*digit-encode ds a*) = *a mod* (*prod-list ds*)
⟨*proof*⟩

lemma *digit-decode-encode-lt*[*simp*]:
assumes *a* < *prod-list ds*
shows *digit-decode ds* (*digit-encode ds a*) = *a*
⟨*proof*⟩

lemma *digit-decode-lt*:
assumes *is* ⊲ *ds*
shows *digit-decode ds is* < *prod-list ds*
⟨*proof*⟩

lemma *digit-encode-valid-index*:
assumes *a* < *prod-list ds*
shows *digit-encode ds a* ⊲ *ds*
⟨*proof*⟩

lemma *length-digit-encode*:
shows *length* (*digit-encode ds a*) = *length ds*
⟨*proof*⟩

lemma *digit-encode-0*:
prod-list ds dvd a $\Longrightarrow$ *digit-encode ds a* = *replicate* (*length ds*) *0*
⟨*proof*⟩

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*⟩

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* $\Rightarrow$ *'a mat* $\Rightarrow$ *nat list* $\Rightarrow$ *'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*))
⟨*proof*⟩

lemma *dims-dematricize*: *Tensor.dims* (*dematricize rmodes A ds*) = *ds*
⟨*proof*⟩

lemma *valid-index-nths*:
assumes *is* $\lhd$ *ds*
shows *nths is A* $\lhd$ *nths ds A*
⟨*proof*⟩

lemma *dematricize-matricize*:
shows *dematricize rmodes* (*matricize rmodes T*) (*Tensor.dims T*) = *T*
⟨*proof*⟩

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*⟩

lemma *matricize-add*:
assumes *dims A = dims B*
shows *matricize I A + matricize I B = matricize I (A+B)*
⟨*proof*⟩

lemma *matricize-0*:
shows *matricize I (tensor0 ds) =* 0_m *(dim-row (matricize I (tensor0 ds))) (dim-col (matricize I (tensor0 ds)))*
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *matricize-cprank-max1*:
fixes *A::'a::field tensor*
assumes *cprank-max1 A*
shows *mrank (matricize I A) ≤ 1*
⟨*proof*⟩

lemma *matrix-rank-le-cprank-max*:
fixes *A :: ('a::field) tensor*
assumes *cprank-max r A*
shows *mrank (matricize I A) ≤ r*
⟨*proof*⟩

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

fixes A :: ($'a$::*field*) *tensor*
shows *mrank* (*matricize* I A) $\leq$ *cprank* A
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *extract-matrix-flatten-matrix*:
extract-matrix (*flatten-matrix* A) (*dim-row* A) (*dim-col* A) = A
⟨*proof*⟩

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
⟨*proof*⟩

lemma *flatten-matrix-extract-matrix*:
flatten-matrix (*extract-matrix* a m n) k = a k **if** $k < m * n$
⟨*proof*⟩

lemma *index-extract-matrix*:
assumes $i{<}m$ $j{<}n$
shows *extract-matrix* a m n $\$\$$ (i,j) = a $(i{*}n + j)$
⟨*proof*⟩

lemma *dim-extract-matrix*:
shows *dim-row* (*extract-matrix* *as* m n) = m
and *dim-col* (*extract-matrix* *as* m n) = n
⟨*proof*⟩

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* $\Rightarrow$ (*nat* $\times$ *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* $\Rightarrow$ *real vec list* $\Rightarrow$ *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* $\Rightarrow$ *real tensor vec* $\Rightarrow$ *nat list* $\Rightarrow$ *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 ($\bigwedge$*i. i*<*count-weights s m* $\Longrightarrow$ *w1 i* = *w2 i*)
shows *insert-weights s m w1* = *insert-weights s m w2*
⟨*proof*⟩

lemma *dims-mat-tensorlist-mult*:
assumes *T*$\in$*set*$_v$ (*mat-tensorlist-mult A Ts ds*)
shows *Tensor.dims T* = *ds*
⟨*proof*⟩

fun *tensors-from-net* :: *real mat convnet* $\Rightarrow$ *real tensor vec* **where**
tensors-from-net (*Input M*) = *Matrix.vec M* (λ*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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *input-sizes-remove-weights*: *input-sizes m* = *input-sizes* (*remove-weights m*)
⟨*proof*⟩

lemma *dims-tensors-from-net*:
assumes $T \in set_v$ (*tensors-from-net m*)
shows *Tensor.dims T* = *input-sizes m*
⟨*proof*⟩

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*⟩

lemma *nth-mat-tensorlist-mult*:
assumes $\bigwedge A.\ A \in set_v\ Ts \Longrightarrow dims\ A = ds$
assumes *i* < *dim-row A*
assumes *dim-vec Ts* = *dim-col A*
shows *mat-tensorlist-mult A Ts ds* $ *i* = *listsum ds* (*map* (λ*j*. (*A* $$ (*i*,*j*)) · *Ts* $ *j*) [*0*..<*dim-vec Ts*])
(**is** - = *listsum ds ?Ts′*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*) =
(λ*x*. *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*) =
(λ*x*. *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* $\times$ *nat*) *convnet* $\Rightarrow$ *bool* **where**
balanced-net-Input: *balanced-net* (*Input M*)
| *balanced-net-Conv*: *balanced-net m* $\Longrightarrow$ *balanced-net* (*Conv A m*)
| *balanced-net-Pool*: *balanced-net m1* $\Longrightarrow$ *balanced-net m2* $\Longrightarrow$
count-weights True m1 = *count-weights True m2* $\Longrightarrow$ *balanced-net* (*Pool m1 m2*)

inductive *shared-weight-net*::*real mat convnet* $\Rightarrow$ *bool* **where**
shared-weight-net-Input: *shared-weight-net* (*Input M*)
| *shared-weight-net-Conv*: *shared-weight-net m* $\Longrightarrow$ *shared-weight-net* (*Conv A m*)
| *shared-weight-net-Pool*: *shared-weight-net m1* $\Longrightarrow$ *shared-weight-net m2* $\Longrightarrow$
count-weights True (*remove-weights m1*) = *count-weights True* (*remove-weights m2*) $\Longrightarrow$
($\bigwedge$*x*. *x* < *count-weights True* (*remove-weights m1*) $\Longrightarrow$ *extract-weights True m1 x* = *extract-weights True m2 x*)
$\Longrightarrow$ *shared-weight-net* (*Pool m1 m2*)

lemma *insert-extract-weights-cong-shared*:
assumes *shared-weight-net m*
assumes $\bigwedge$*x*. *x* < *count-weights True* (*remove-weights m*) $\Longrightarrow$ *f x* = *extract-weights True m x*
shows *m* = *insert-weights True* (*remove-weights m*) *f*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *remove-insert-weights*:
shows *remove-weights* (*insert-weights s m w*) = *m*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *shared-weight-net-insert-weights*: *balanced-net m* $\Longrightarrow$ *shared-weight-net* (*insert-weights True m w*)
⟨*proof*⟩

lemma *finite-valid-index*: *finite* {*is*. *is* $\lhd$ *ds*}

⟨*proof*⟩

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*⟩

lemma *prod-lessThan-split*:
fixes $g :: nat \Rightarrow real$ **shows** $prod\ g\ \{..<n+m\} = prod\ g\ \{..<n\} * prod\ (\lambda x.\ g\ (x+n))\ \{..<m\}$
⟨*proof*⟩

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*
$= (\sum is \in \{is.\ is \lhd input\text{-}sizes\ m\}.\ (\prod k<length\ inputs.\ inputs\ !\ k\ \$\ (is!k)) * Tensor.lookup\ (tensors\text{-}from\text{-}net\ m\ \$\ j)\ is)$
⟨*proof*⟩

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*⟩

end

12 Concrete Matrices

theory *DL-Concrete-Matrices*
imports *Jordan-Normal-Form.Matrix*
begin

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

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

lemma *id-matrix-dim*: *dim-row (id-matrix nr nc) = nr dim-col (id-matrix nr nc) = nc* ⟨*proof*⟩

lemma *row-id-matrix*:
assumes $i < nr$
shows *row (id-matrix nr nc) i = unit-vec nc i*
⟨*proof*⟩

lemma *unit-eq-0*[*simp*]:
assumes *i*: $i \geq n$
shows *unit-vec n i* $=$ 0_v *n*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *row-all1-matrix*:
assumes $i < nr$
shows *row* (*all1-matrix nr nc*) *i* = *all1-vec nc*
⟨*proof*⟩

lemma *all1-vec-scalar-prod*:
shows *all1-vec* (*length xs*) · (*vec-of-list xs*) = *sum-list xs*
⟨*proof*⟩

lemma *mult-all1-matrix*:
assumes $i < nr$
shows ((*all1-matrix nr* (*dim-vec v*)) $*_v$ *v*) \$ *i* = *sum-list* (*list-of-vec v*) (**is** *?a* \$ *i* = *sum-list* (*list-of-vec v*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *row-copy-first-matrix*:
assumes $i < nr$

shows *row* (*copy-first-matrix nr nc*) *i* = *unit-vec nc 0*
⟨*proof*⟩

lemma *mult-copy-first-matrix*:
assumes *i* < *nr* **and** *dim-vec v* > *0*
shows (*copy-first-matrix nr* (*dim-vec v*) $*_v$ *v*) \$ *i* = *v* \$ *0* (**is** *?a* \$ *i* = *v* \$ *0*)
⟨*proof*⟩

end

13 Missing Lemmas of Finite_Set

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

lemma *card-even*[*simp*]: *card* {*a* ∈ *Collect even*. *a* < *2* ∗ *n*} = *n*
⟨*proof*⟩

lemma *card-odd*[*simp*]: *card* {*a* ∈ *Collect odd*. *a* < *2* ∗ *n*} = *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *valid-deep-model′*: *valid-net* (*deep-model′ r rs*)
⟨*proof*⟩

lemma *input-sizes-deep-model′*:
assumes *length rs* $\geq$ *1*
shows *input-sizes* (*deep-model′-l rs*) = *replicate* (*2*^(*length rs* − *1*)) (*last rs*)
⟨*proof*⟩

lemma *input-sizes-deep-model*:
assumes *length rs* $\geq$ *2*
shows *input-sizes* (*deep-model-l rs*) = *replicate* (*2*^(*length rs* − *2*)) (*last rs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *witness′-is-deep-model*: *remove-weights* (*witness′ Y rs*) = *deep-model′ Y rs*
⟨*proof*⟩

lemma *witness-valid*: *valid-net′* (*witness Y r rs*)
⟨*proof*⟩

lemma *witness′-valid*: *valid-net′* (*witness′ Y rs*)
⟨*proof*⟩

lemma *shared-weight-net-witness*: *shared-weight-net* (*witness Y r rs*)
⟨*proof*⟩

lemma *witness-l0′*: *witness′ Y* [*M*] =
(*Pool*
(*Conv* (*id-matrix Y M*) (*Input M*))
(*Conv* (*id-matrix Y M*) (*Input M*))
)
⟨*proof*⟩

lemma *witness-l1*: *witness Y r0* [*M*] =
Conv (*all1-matrix Y r0*) (*witness′ r0* [*M*])
⟨*proof*⟩

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*])
⟨*proof*⟩

lemma *tensor-prod-unit-vec*:
unit-vec M j ⊗ *unit-vec M j* = *tensor-from-lookup* [*M*,*M*] (*λis. if is*=[*j*,*j*] *then 1 else 0*) (**is** *?A*=*?B*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *length-output-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*
shows *dim-vec* (*tensors-from-net m*) = *rs* ! *0*
⟨*proof*⟩

lemma *length-output-deep-model'*:
assumes *remove-weights m* = *deep-model'-l rs*
shows *dim-vec* (*tensors-from-net m*) = *rs* ! *0*
⟨*proof*⟩

lemma *length-output-witness*:
dim-vec (*tensors-from-net* (*witness-l rs*)) = *rs* ! *0*
⟨*proof*⟩

lemma *length-output-witness'*:
dim-vec (*tensors-from-net* (*witness'-l rs*)) = *rs* ! *0*
⟨*proof*⟩

lemma *dims-output-deep-model*:
assumes *length rs* $\geq$ *2*
and $\bigwedge r.\ r \in set\ rs \Longrightarrow r > 0$
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*)
⟨*proof*⟩

lemma *dims-output-witness*:
assumes *length rs* $\geq$ *2*
and $\bigwedge r.\ r \in set\ rs \Longrightarrow r > 0$
and $j < rs!0$
shows *Tensor.dims* (*tensors-from-net* (*witness-l rs*) \$ *j*) = *replicate* (*2*^(*length rs* − *2*)) (*last rs*)
⟨*proof*⟩

lemma *dims-output-deep-model′*:
assumes *length rs* $\geq$ *1*
and $\bigwedge r.\ r \in set\ rs \Longrightarrow r > 0$
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*⟩

lemma *dims-output-witness′*:
assumes *length rs* $\geq$ *1*
and $\bigwedge r.\ r \in set\ rs \Longrightarrow r > 0$
and $j < rs!0$
shows *Tensor.dims* (*tensors-from-net* (*witness′-l rs*) \$ *j*) = *replicate* (*2*^(*length rs* − *1*)) (*last rs*)
⟨*proof*⟩

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* $\geq$ *3*
and *no-zeros*:$\bigwedge r.\ r \in set\ rs \Longrightarrow 0 < r$
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*⟩

lemma *order-tensor-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*
shows *order* (*tensors-from-net m* \$ *y*) = *2* ∗ *N-half*
  ⟨*proof*⟩

lemma *dims-A*:
shows *Tensor.dims* (A *ws*) = *replicate* (*2* ∗ *N-half*) (*last rs*)
  ⟨*proof*⟩

lemma *order-A*:
shows *order* (A *ws*) = *2* ∗ *N-half* ⟨*proof*⟩

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*})
  ⟨*proof*⟩

lemma *dims-A′-pow*:
shows *dim-row* (A' *ws*) = (*last rs*) ^ *N-half* *dim-col* (A' *ws*) = (*last rs*) ^ *N-half*
  ⟨*proof*⟩

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*

$\langle proof \rangle$

lemma *Aw-def'*: *Aw* = *A witness-weights* $\langle proof \rangle$

lemma *Aw'-def'*: *Aw'* = *A' witness-weights* $\langle proof \rangle$

lemma *dims-Aw*: *Tensor.dims Aw* = *replicate* (*2* ∗ *N-half*) (*last rs*)
$\langle proof \rangle$

lemma *order-Aw*: *order Aw* = *2* ∗ *N-half*
$\langle proof \rangle$

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*})
$\langle proof \rangle$

lemma *dims-Aw'-pow*: *dim-row Aw'* = (*last rs*) ^ *N-half* *dim-col Aw'* = (*last rs*) ^ *N-half*
$\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *witness-matricization*:
assumes *i* < *dim-row Aw'* **and** *j* < *dim-col Aw'*
shows *Aw'* $$ (*i*, *j*)
= (*if i*=*j* ∧ (∀ *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n. even n*}) *i*). *i0* < *last* (*butlast rs*)) *then 1 else 0*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *card-rows-with-1*: *card* {*i*∈*rows-with-1*. *i*<*dim-row Aw'*} = *r* ^ *N-half*
$\langle proof \rangle$

lemma *infinite-rows-with-1*: *infinite rows-with-1*
⟨*proof*⟩

lemma *witness-submatrix*: *submatrix Aw′ rows-with-1 rows-with-1* = 1_m (*r^N-half*)
⟨*proof*⟩

lemma *witness-det*: *det* (*submatrix Aw′ rows-with-1 rows-with-1*) ≠ *0* ⟨*proof*⟩

end

interpretation *example* : *deep-model-correct-params False* [*10*,*10*,*10*]
⟨*proof*⟩

interpretation *example* : *deep-model-correct-params-y False* [*10*,*10*,*10*] *1*
⟨*proof*⟩

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 ⋀*x*. (*A x*) ∈ *carrier-mat n n*
assumes ⋀*x i j*. *i*<*n* ⟹ *j*<*n* ⟹ *polyfun N* (λ*x*. (*A x*) $$ (*i*,*j*))
shows *polyfun N* (λ*x*. *det* (*A x*))
⟨*proof*⟩

lemma *polyfun-extract-matrix*:
assumes *i*<*m j*<*n*
shows *polyfun* {..<*a* + (*m* ∗ *n* + *c*)} (λ*f*. *extract-matrix* (λ*i*. *f* (*i* + *a*)) *m n* $$ (*i*,*j*))
⟨*proof*⟩

lemma *polyfun-mult-mat-vec*:
assumes ⋀*x*. *v x* ∈ *carrier-vec n*
assumes ⋀*j*. *j*<*n* ⟹ *polyfun N* (λ*x*. *v x* $ *j*)
assumes ⋀*x*. *A x* ∈ *carrier-mat m n*
assumes ⋀*i j*. *i*<*m* ⟹ *j*<*n* ⟹ *polyfun N* (λ*x*. *A x* $$ (*i*,*j*))
assumes *j* < *m*
shows *polyfun N* (λ*x*. ((*A x*) $*_v$ (*v x*)) $ *j*)
⟨*proof*⟩

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*} (λf. *evaluate-net* (*insert-weights s m* (λi. f ($i + a$))) *inputs* \$ j)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *polyfun-tensors-from-net*:
assumes *valid-net m*
assumes *is* ◁ *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*⟩

lemma *polyfun-matricize*:
assumes $\bigwedge x$. *dims* (T x) = *ds*
assumes $\bigwedge$*is*. *is* ◁ *ds* ⟹ *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*⟩

lemma ($\neg$ (a::*nat*) $< b$) = ($a \geq b$)
⟨*proof*⟩

lemma *polyfun-submatrix*:
assumes $\bigwedge x$. (A x) $\in$ *carrier-mat m n*
assumes $\bigwedge x\ i\ j$. $i<m \Longrightarrow j<n \Longrightarrow$ *polyfun N* (λx. (A x) \$\$ (i,j))
assumes $i <$ *card* {i. $i < m \wedge i \in I$}
assumes $j <$ *card* {j. $j < n \wedge j \in J$}
assumes *infinite I infinite J*
shows *polyfun N* (λx. (*submatrix* (A x) I J) \$\$ (i,j))
⟨*proof*⟩

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*}
(*λf. Tensor.lookup* (*tensors-from-net* (*insert-weights shared-weights* (*deep-model-l rs*) *f*) $ *y*) *is*)
⟨*proof*⟩

lemma *input-sizes-deep-model*: *input-sizes* (*deep-model-l rs*) = *replicate* (*2* ∗ *N-half*) (*last rs*)
⟨*proof*⟩

lemma *polyfun-matrix-deep-model*:
assumes *i*<(*last rs*) ^ *N-half*
assumes *j*<(*last rs*) ^ *N-half*
shows *polyfun* {..<*weight-space-dim*} (*λf. A′ f* $$ (*i*,*j*))
⟨*proof*⟩

lemma *polyfun-submatrix-deep-model*:
assumes *i* < *r* ^ *N-half*
assumes *j* < *r* ^ *N-half*
shows *polyfun* {..<*weight-space-dim*} (*λf. witness-submatrix f* $$ (*i*,*j*))
⟨*proof*⟩

lemma *polyfun-det-deep-model*:
shows *polyfun* {..<*weight-space-dim*} (*λf. det* (*witness-submatrix f*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *l-borel-f-1*: *distr* (*lborel-f 1*) *lborel* (λ*x*. *x 0*) = *lborel*
⟨*proof*⟩

lemma *space-lborel-f*: *space* (*lborel-f n*) = Pi_E {..<*n*} (λ-. *UNIV*) ⟨*proof*⟩

lemma *space-lborel-f-subset*: *space* (*lborel-f n*) ⊆ *space* (*lborel-f* (*Suc n*))
⟨*proof*⟩

lemma *space-lborel-add-dim*:
assumes *f* ∈ *space* (*lborel-f n*)
shows *f*(*n*:=*x*) ∈ *space* (*lborel-f* (*Suc n*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *lborel-f-measurable-add-dim*: (λ*f*. *f*(*n* := *x*)) ∈ *measurable* (*lborel-f n*) (*lborel-f* (*Suc n*))
⟨*proof*⟩

lemma *sets-lborel-f-sub-dim*:
assumes *A* ∈ *sets* (*lborel-f* (*Suc n*))
shows {*x*. *x*(*n* := *y*) ∈ *A*} ∩ *space* (*lborel-f n*) ∈ *sets* (*lborel-f n*)
⟨*proof*⟩

lemma *lborel-f-measurable-restrict*:
assumes *m* ≤ *n*
shows (λ*f*. *restrict f* {..<*m*}) ∈ *measurable* (*lborel-f n*) (*lborel-f m*)
⟨*proof*⟩

lemma *lborel-measurable-sub-dim*: (λ*f*. *restrict f* {..<*n*}) ∈ *measurable* (*lborel-f* (*Suc n*)) (*lborel-f n*)

$\langle proof \rangle$

lemma *measurable-lborel-component* [*measurable*]:
assumes $k<n$
shows $(\lambda x.\ x\ k) \in$ *borel-measurable* (*lborel-f* n)
$\langle proof \rangle$

end

17 Lebesgue Measure of Polynomial Zero Sets

theory *Lebesgue-Zero-Set*
imports
Polynomials.More-MPoly-Type
Lebesgue-Functional
Polynomials.MPoly-Type-Univariate
begin

lemma *measurable-insertion* [*measurable*]:
assumes *vars* $p \subseteq \{..<n\}$
shows $(\lambda f.$ *insertion* $f\ p) \in$ *borel-measurable* (*lborel-f* n)
$\langle proof \rangle$

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* $p \subseteq \{..<n\}$
shows $\{f \in$ *space* (*lborel-f* n). *insertion* $f\ p = 0\} \in$ *null-sets* (*lborel-f* n)
$\langle proof \rangle$

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*)
⟨*proof*⟩

lemma *output-size-shallow-model'*: *output-size* (*shallow-model' Z M N*) = *Z*
⟨*proof*⟩

lemma *valid-shallow-model*: *valid-net* (*shallow-model Y Z M N*)
⟨*proof*⟩

lemma *output-size-shallow-model*: *output-size* (*shallow-model Y Z M N*) = *Y*
⟨*proof*⟩

lemma *input-sizes-shallow-model*: *input-sizes* (*shallow-model Y Z M N*) = *replicate* (*Suc N*) *M*
⟨*proof*⟩

lemma *balanced-net-shallow-model'*: *balanced-net* (*shallow-model' Z M N*)
⟨*proof*⟩

lemma *balanced-net-shallow-model*: *balanced-net* (*shallow-model Y Z M N*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*) ≤ *Z*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *if-polynomial-0-rank*:
assumes *polynomial-f w* ≠ *0*
shows *r* ^ *N-half* ≤ *cprank* (*A w*)
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

theorem *fundamental-theorem-network-capacity*:
AE x in lborel-f weight-space-dim. r ^ *N-half* ≤ *cprank* (*A x*)
⟨*proof*⟩

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*))
⟨*proof*⟩

abbreviation *lebesgue-f* **where** *lebesgue-f n* ≡ *completion* (*lborel-f n*)

lemma *space-lebesgue-f*: *space* (*lebesgue-f n*) = Pi_E {..<*n*} (λ-. *UNIV*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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.