UNIVERSIDAD
DE LA REPUBLICA
URUGUAY


# Verificación de estructura de redes neuronales profundas en tiempo de compilación

## Proyecto TensorSafe

Leonardo Piñeyro

leonardo.pineyro@fing.edu.uy

Tutores

Alberto Pardo y Marcos Viera

{pardo,mviera}@fing.edu.uy

Tesis de Licenciatura en Computación
Facultad de Ingeniería, Instituto de Computación
Universidad de la República

Montevideo – Uruguay
Noviembre de 2019

# RESUMEN

Este documento presenta *TensorSafe*, una biblioteca desarrollada en Haskell que permite la definición y validación estructural de arquitecturas de redes neuronales. En la actualidad, el proceso de desarrollo de modelos de aprendizaje profundo ha sido ampliamente simplificado debido a la disponibilidad de herramientas en la industria. Sin embargo, la mayoría de estas herramientas no provee ningún control de consistencia estructural en tiempo de compilación, haciendo que los desarrolladores tengan que lidiar con errores inesperados en tiempo de ejecución. En particular, la validación estructural de redes neuronales profundas en tiempo de compilación es una tarea compleja, la cual involucra la validación matemática de todas las operaciones que el modelo de aprendizaje profundo va a realizar. Este chequeo estructural requiere un uso avanzado de los sistemas de tipos para la manipulación de tipos abstractos capaces de modelar la construcción de redes neuronales. El uso del paradigma de programación funcional y la programación a nivel de tipos que provee el lenguaje Haskell fueron de particular importancia al momento del desarrollo de TensorSafe. La evaluación experimental realizada muestra que usando TensorSafe es posible la construcción y validación de modelos de aprendizaje profundo bien conocidos, como lo son *AlexNet* o *ResNet50*.

Palabras claves:
programación a nivel de tipos, aprendizaje profundo, computación confiable, programación funcional, haskell.

# Tabla de contenidos

# Capítulo 1

# Introducción

En los últimos años, las aplicaciones de inteligencia artificial se han vuelto omnipresentes en muchos aspectos de nuestras vidas. En particular, los métodos de aprendizaje profundo han mostrado resultados increíblemente buenos para tareas como el reconocimiento de voz, el procesamiento del lenguaje natural y la visión por computadora [15]. En el mundo de aprendizaje profundo, tanto en la comunidad científica como en la industria, la forma en la cual los desarrolladores construyen soluciones es mediante el uso de entornos de desarrollo altamente abstractos, como TensorFlow[1], Keras[2] o PyTorch[3], entre otros [1, 5, 20, 22]. Estos entornos de desarrollo proporcionan Lenguajes de Dominio Específico (DSLs)[4] capaces de describir operaciones entre estructuras algebraicas de datos de varias dimensiones, también denominados *tensores* [13], y herramientas para construir redes neuronales profundas con arquitecturas complejas [22]. Además, proveen componentes abstractos de alto nivel, llamadas *capas*, que funcionan como bloques constructivos con las cuales los programadores pueden crear modelos de aprendizaje profundo con facilidad. Estas capas generalmente representan semánticamente operaciones entre tensores, haciendo mas fácil para los desarrolladores poder crear arquitecturas más grandes de redes neuronales [15].

Sin embargo, la mayoría de estos entornos de desarrollo no proveen ningún tipo de validación estructural en tiempo de compilación. En los escenarios típicos, las bibliotecas para aprendizaje profundo solo proporcionan ciertas

[1] `https://www.tensorflow.org/`
[2] `https://keras.io/`
[3] `https://pytorch.org/`
[4] Domain Specific Languages en inglés

validaciones en tiempo de ejecución, incluso a veces hasta ignorando ciertas validaciones [4]. Crear una arquitectura para un modelo de aprendizaje profundo es en general una tarea ardua. Los desarrolladores tienen que mantener un seguimiento constante de diferentes dimensiones de tensores en sus modelos y siempre estar seguros de que las operaciones que se van a realizar son viables. Por ejemplo, sería inviable la multiplicación entre una matriz y un tensor tridimensional.

La seguridad y confiabilidad en soluciones de aprendizaje profundo es un asunto muy discutido en la actualidad. Las soluciones basadas en inteligencia artificial cada vez son más presentes en aspectos particularmente sensibles de nuestras vidas cotidianas, como su aplicación a la medicina o en la conducción autónoma de vehículos [2, 12]. En este escenario, los lenguajes de programación funcional fuertemente tipados con mecanismos para la programación a nivel de tipos parecen ser una buena elección a la hora de crear soluciones en aprendizaje profundo con mejores niveles de confiabilidad. En este sentido, los avances recientes en las técnicas de programación a nivel de tipos en lenguajes como Haskell proporcionan los medios adecuados para realizar verificaciones estáticas a los modelos de aprendizaje profundo mediante la introducción de restricciones a nivel de los tipos.

En este documento, presentamos *TensorSafe*[1], una biblioteca desarrollada en Haskell para la definición de modelos de aprendizaje profundo que utiliza técnicas de programación a nivel de tipos para la validación de la definición estructural de dichos modelos. *TensorSafe* también ofrece una forma muy sencilla de representar arquitecturas de redes neuronales profundas basadas en capas, utilizando pocas líneas de código Haskell. La característica principal de *TensorSafe* es la definición de arquitecturas de aprendizaje profundo con validación estructural en tiempo de compilación. Además, *TensorSafe* proporciona herramientas que hacen posible la compilación de modelos en entornos externos de desarrollo como Keras para Python o Keras para JavaScript para que puedan ser utilizados y desplegados en diversas aplicaciones.

Este trabajo fue presentado en Setiembre del 2019 en la *Conferencia Brasilera de Software* (CBSoft) que se realizó en la ciudad de Salvador de Bahía, Brasil. Particularmente, se publicló un artículo con el nombre "Structure verification of deep neural networks at compilation time using dependent types" [21] en el *Simposio Brasilero de Lenguajes de Programación* (SBLP), donde

[1] `http://hackage.haskell.org/package/tensor-safe`

nuestro trabajo fue otorgado el premio al mejor artículo.

Este documento está estructurado de la siguiente manera. En el Capítulo 2 se presenta una breve reseña sobre aprendizaje profundo y se listan diferentes enfoques para el desarrollo de bibliotecas de aprendizaje profundo con controles basados en tipos. En el Capítulo 3 introducimos rápidamente el funcionamiento de *TensorSafe* desde un punto de vista pragmático. Posteriormente, en el Capítulo 4 presentamos en detalle aspectos técnicos sobre la implementación de *TensorSafe*. En el Capítulo 5 proporcionamos una descripción detallada sobre las funcionalidades de compilación de *TensorSafe*. En el Capítulo 6 presentamos y discutimos los resultados obtenidos de la evaluación experimental de la biblioteca. Finalmente, en el Capítulo 7 presentamos nuestras conclusiones y reflexiones acerca del proceso de desarrollo y el estado del arte de los entornos de desarrollo para soluciones basadas en aprendizaje profundo.

# Capítulo 2

# Estado del arte y trabajos relacionados

Este capítulo presenta inicialmente una breve reseña sobre el estado del arte de las técnicas de aprendizaje profundo y cómo la comunidad científica utiliza estas técnicas para resolver casos prácticos en la industria. Luego, se presentan hallazgos con respecto al uso de *tipos dependientes*, la programación orientada a tipos y el paradigma de programación funcional en la comunidad científica de aprendizaje profundo. En particular, este capítulo presenta 3 proyectos de investigación, cada uno planteando diferentes enfoques para la creación de un entorno de desarrollo seguro para aprendizaje profundo.

## 2.1. Estado del arte en aprendizaje profundo

Por varios años, las técnicas de aprendizaje automático han sido un foco importante de investigación en la comunidad científica de la computación. Esta rama de la inteligencia artificial tiene como foco de investigación la creación de programas que puedan aprender a realizar una tarea a partir de búsqueda de patrones en datos [18]. Todos los programas en el área de aprendizaje automático comparten siempre las siguientes dos características. En primer lugar, pasan por una etapa de entrenamiento donde se utilizan datos para ajustar parámetros asociados al programa y así mejorar una o más métricas de rendimiento. Y en segundo lugar, estos programas entrenados se utilizan para inferir características a partir de datos nunca antes vistos. Por ejemplo, un programa de aprendizaje automático puede ser entrenado con datos de ven-

tas de propiedades residenciales con el objetivo de poder predecir los precios de ventas. Eventualmente, este programa entrenado con datos de propiedades puede ser utilizado para inferir el precio de una nueva propiedad que no estaba presente en los datos de entrenamiento.

Una de las técnicas de aprendizaje automático es el llamado *aprendizaje profundo* ("deep learning" en inglés). El aprendizaje profundo busca crear estructuras jerárquicas de redes neuronales capaces de reconocer y aprender a partir de estructuras automáticamente generadas, sin necesidad de que un humano indique qué atributos de los datos mirar específicamente. La estructura jerárquica generada hace que estos tipos de programas puedan reconocer patrones muy complicados a partir de conceptos más simples [15]. Esto último habilita a este tipo de algoritmos a poder representar y aprender a resolver tareas intrínsecamente complejas para una computadora, como por ejemplo: detección y clasificación de objetos en imágenes, segmentación de voces en archivos de audio, o generación automática de texto a partir de imágenes [9].

La construcción de estos modelos basada en niveles hace que sea intuitiva su representación como un grafo o una serie de capas, también conocidas como "layers" en inglés, que realizan operaciones entre datos estructurados. En la mayoría de los casos, los modelos cuentan con una entrada y salida de datos con formas fijas.

Los datos en los modelos de aprendizaje profundo se suelen representar en diferentes formas algebraicas, como por ejemplo vectores, matrices o arreglos multidimensionales. En el contexto de aprendizaje profundo, se suele utilizar el término *tensor* para referir a cualquier estructura de datos [13]. Con respecto a las capas, éstas especifican cómo es posible en el modelo la búsqueda de patrones y estructuras a partir de los datos utilizados en el proceso de aprendizaje del modelo. Al momento de crear modelos de aprendizaje profundo, los desarrolladores pueden crear relaciones entre capas de manera abstracta con el objetivo de que al momento de entrenamiento las redes en el modelo puedan aprender a resolver una determinada tarea a partir de datos. En general, la cantidad de capas en un modelo influye en su capacidad de aprendizaje, esto es, mientras más capas posea un modelo de aprendizaje profundo, este va a poder representar estructuras o conceptos más complejos.

Existen en la actualidad una amplia variedad de definiciones abstractas de capas que cumplen propósitos específicos dentro de un modelo. A continuación se nombran algunas de las más relevantes y utilizadas en modelos conocidos

por la comunidad. Las capas *totalmente conectadas* son la forma más básica de redes neuronales, las cuales conectan los nodos de una capa con todos los nodos de la siguiente capa en el modelo, donde las activaciones de las neuronas en las capas ayudan al modelo a poder detectar diferentes tipos de patrones en los datos. Por otro lado, las capas de *convolución* en conjunto con las capas de *pooling* logran crear muy buenas representaciones genéricas de patrones en imágenes o en señales. También es muy común utilizar la capa *flatten*, la cual transforma cualquier tipo de estructura de nodos multidimensional en una estructura vectorial de una dimensión. En definitiva, estas capas son los bloques constructores de los modelos de aprendizaje profundo. En los últimos años la comunidad científica se ha enfocado en forma importante a la definición de nuevas capas, así como en la aplicación de las mismas para la creación de modelos más complejos y robustos [5].

En la práctica, los modelos de aprendizaje profundo se suelen implementar utilizando entornos de desarrollo. Al trabajar con estos entornos de desarrollo es habitual usar el lenguaje de programación Python, un lenguaje de programación dinámico y orientado a objetos muy popular en ciencia de datos e inteligencia artificial. Para mencionar algunos de los entornos de desarrollo más populares en la comunidad, está TensorFlow [1], que es una biblioteca muy aceptada con un nivel de abstracción moderado que hace posible la creación de arquitecturas de aprendizaje profundo muy complejas y personalizadas. TensorFlow es especialmente conocido por su foco en el rendimiento a la hora de entrenar los modelos, por ejemplo, haciendo uso de arquitecturas en racimo (clústers) o compilación para GPUs. Otra de las alternativas es Keras [5]. Keras proporciona un nivel de abstracción muy alto donde los desarrolladores pueden usar bloques constructivos, creados por la comunidad científica, con el objetivo de crear modelos más grandes y robustos. Sin embargo, en Python no existen mecanismos de protección ante potenciales fallas en la programación de los modelos de aprendizaje profundo. Si bien Python ha incluido controles estáticos de código en versiones posteriores a `3.6`, como definición de tipos y tipos genéricos, estas funcionalidades aún no son lo suficientemente sofisticadas como para garantizar controles sobre las estructuras de los modelos previo a la ejecución del código.

## 2.2. Aprendizaje profundo y programación funcional

En la última década, la inteligencia artificial ha comenzado a tener múltiples aplicaciones en la industria. Desde algo simple como detectar y etiquetar personas en imágenes, hasta escenarios donde se utiliza inteligencia artificial para controlar vehículos autónomos. Es razonable pensar que mientras más se involucre al software en situaciones de contexto crítico, como vehículos autónomos o medicina, más controles deberían haber sobre estos programas para garantizar seguridad a sus usuarios y a su entorno. En particular, la utilización del paradigma funcional con ayuda de un sistema de tipos robusto es de gran utilidad en estos escenarios, ya que facilita la detección temprana de fallas en el software. En esta sección presentamos 3 trabajos estudiados los cuales involucran el paradigma funcional y el chequeo estático de tipos en aprendizaje profundo.

El primer trabajo que vale la pena mencionar es el propuesto por Chen [4]. En dicho trabajo se propone una solución implementada en *Scala* la cual puede realizar operaciones entre tensores de forma segura con chequeos en tiempo de compilación. Principalmente, el trabajo de Chen propone un sistema de etiquetado semántico de dimensiones estructurales de los datos de forma que sea posible verificar operaciones entre estas estructuras. Por ejemplo, si una imagen tiene tres dimensiones, llamadas *ancho*, *alto* y *canales*, esta forma de etiquetado semántico de las dimensiones hace posible la detección de operaciones ilegales en tiempo de compilación. Esta manera novedosa de otorgar nombres a los ejes se diferencia de otras estrategias adoptadas por los trabajos mencionados a continuación, los cuales toman el álgebra lineal a nivel de tipos como la forma de validar los modelos de aprendizaje profundo.

Los siguientes dos trabajos que vamos a mencionar son soluciones basadas en Haskell que hacen uso de sus funcionalidades de programación a nivel de tipos. Uno de ellos es el trabajo de Bowen J. [3] que provee una interfaz de *TensorFlow* utilizando tipos dependientes[1]. Sin embargo, esta solución no cubre todas las herramientas provistas por *TensorFlow* sino que solo algunas de ellas. Además, el proyecto utiliza una sintaxis bastante complicada de entender debido a los muchos cambios realizados para poder soportar tipos dependientes

[1] `https://github.com/helq/tensorflow-haskell-deptyped`

en Haskell [16].

El otro trabajo a mencionar es *Grenade*[1], otra biblioteca desarrollada con tipos dependientes en el contexto de Haskell y que tuvo mucha influencia en el diseño de *TensorSafe*. En *Grenade* los desarrolladores pueden crear fácilmente modelos de aprendizaje profundo declarando una secuencia de capas y todas las transformaciones de las dimensiones de datos que ocurren en el modelo, lo cual hace posible la verificación estructural en tiempo de compilación. Un aspecto muy importante a destacar de *Grenade* es su utilización de los tipos de datos *singleton* [10] que hacen posible una relación entre los valores que se manipulan a nivel de tipos y los que son manipulados en tiempo de ejecución.

Es imperante notar que todos los proyectos mencionados en esta sección proveen funcionalidades computacionales además de la verificación estructural de los modelos. Eso significa que estos entornos de desarrollo son capaces de entrenar modelos y utilizarlos para predecir valores en tiempo de ejecución. Por el contrario, en *TensorSafe* solo se toma en cuenta la definición estructural de los modelos y se ofrece opciones de compilación de estos modelos a entornos de desarrollo externos, delegándoles la parte del problema computacional.

[1] `https://github.com/HuwCampbell/grenade`

# Capítulo 3

# TensorSafe en la práctica

En este capítulo presentamos a *TensorSafe* de una manera pragmática, tratando de mostrar desde el punto de vista de un desarrollador cómo es posible la creación de modelos de aprendizaje profundo utilizando esta herramienta.

## 3.1. Creación de un modelo

Para empezar con un ejemplo muy simple, en el fragmento de código 3.1 mostramos la implementación utilizando *TensorSafe* de un modelo de aprendizaje profundo que resuelve el problema de reconocimiento de dígitos manuscritos en imágenes con tamaño de $28 \times 28$. Específicamente para este problema, existe la base de datos *MNIST* con un amplio número de muestras de dígitos manuscritos las cuales fueron correctamente etiquetadas para su posterior uso en el entrenamiento de modelos de aprendizaje automático [8]. Es importante tener en cuenta que ésta es solo una de las varias implementaciones posibles de aprendizaje profundo que resuelven el mismo problema. Sin embargo, todas las implementaciones tienen algo en común: obtienen como entrada una imagen de $28 \times 28$ píxeles en escala de grises o en blanco y negro, y su salida también es la misma, que es una predicción del valor del dígito manuscrito que va de 0 a 9. Por otro lado, otras posibles implementaciones difieren en la definición del cuerpo central del modelo de aprendizaje profundo, la cual se encarga principalmente del procesamiento de la entrada del modelo para obtener los valores de predicción más precisos. El rendimiento de los modelos de aprendizaje automático generalmente se obtiene en términos de *exactitud* o de *precisión* y *exhaustividad* [19]. Como consecuencia, para los escenarios

en los cuales se presentan diferentes modelos que resuelven el mismo problema, es posible categorizarlos y ordenarlos por su resultados de evaluación, y potencialmente, en la práctica solo usar el que obtuvo mejores resultados.

**Fragmento de código 3.1:** Ejemplo de modelo en *TensorSafe*

```
type MNIST = MkINetwork
    '[ Flatten
     , Dense 784 42
     , Relu
     , Dense 42 10
     , Softmax
     ]
    ('D2 28 28)   -- Input
    ('D1 10)      -- Output

mnist :: MNIST
mnist = mkINetwork
```

En *TensorSafe* un modelo de aprendizaje profundo se construye a partir de tres componentes principales. Primeramente, y de mayor importancia, se precisa definir una secuencia de capas, que en la biblioteca son denominadas *Layers*. Cada una de estas capas corresponde a una operación en el modelo de aprendizaje profundo la cual transforma los valores de entrada, posiblemente cambiando la forma de dichos valores. En segundo lugar, se debe proporcionar la especificación de la forma de la estructura de los datos de entrada del algoritmo. Y finalmente, el último componente especifica la forma de los valores de salida, la cuál representa la forma esperada de las predicciones del modelo.

Para ilustrar cómo es posible la definición de un modelo utilizando *TensorSafe*, en el código de ejemplo 3.1 se define un modelo de aprendizaje profundo con cuatro tipos diferentes de capas, estas son: *Flatten*, *Dense*, *Relu* y *Softmax*. En la Figura 3.1 se puede apreciar un diagrama estructural del modelo en cuestión, donde se indica, de izquierda a derecha, la secuencia de transformaciones estructurales de datos que ocurren en el modelo. La primera capa, *Flatten*, transforma una matriz o un arreglo n-dimensional en un vector plano con la secuencia de todos los valores que se encuentran en la estructura de entrada. En segundo lugar, la capa *Dense*, también conocida como una red neuronal totalmente conectada, es una capa que opera con vectores unidimensionales y

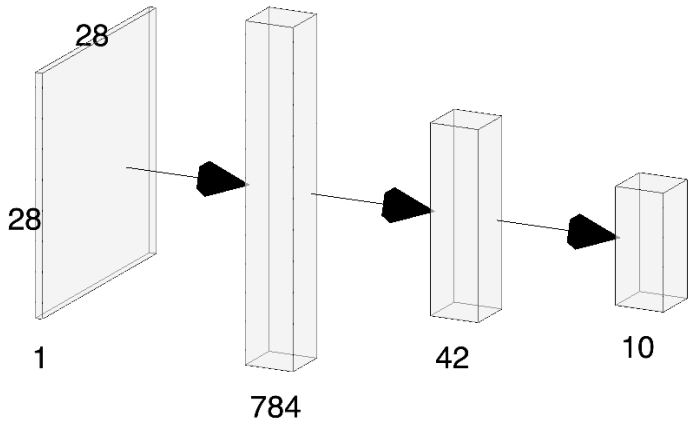



**Figura 3.1:** Diagrama estructural del modelo de aprendizaje profundo presentado en el Fragmento de Código 3.1

requiere dos valores naturales como argumentos: el primer argumento especifica el largo de los valores de entrada, y el segundo el largo de los valores de salida. La capa *Dense* conecta y transforma los valores con cierta forma de entrada en valores con una forma especificada diferente. Finalmente, las capas *Relu* y *Softmax* son capas de activación la cuales habilitan o no cada valor individualmente sin afectar la forma de los valores de entrada [15].

Siguiendo el ejemplo definido en el fragmento de código 3.1, la forma de los valores de entrada del modelo se define como una matriz de tamaño $28 \times 28$ que representa cada uno de los valores de los píxeles de una imagen de un dígito manuscrito. Junto con esto se especifica la forma de los valores de salida que en este caso es un vector de largo 10, donde cada uno de los valores representa la probabilidad de cada dígito sea la respuesta correcta al dígito manuscrito hallado en la imagen de entrada.

En resumen, con *TensorSafe* es posible la definición de un modelo de aprendizaje profundo con tan solo 3 componentes principales: una lista de capas, la forma de los valores de entrada y la forma de los valores de salida. Es especialmente importante mencionar que la definición del modelo se realiza a nivel de tipos en Haskell mediante la introducción de un sinónimo de tipo (en nuestro ejemplo, con la introducción del tipo `MNIST` que es un sinónimo del constructor de tipo MkINetwork aplicado a los tipos que representan a la lista de layers, la forma de la entrada y de la salida). Al especificar el modelo en términos de tipos, es posible ensamblar y verificar automáticamente las estructuras de

datos del modelo. Esta funcionalidad se habilita a través del uso de la función `mkINetwork`, la cual es discutida en el Capítulo 4.

## 3.2. Verificación estructural de modelos en TensorSafe

Al momento de crear modelos de aprendizaje profundo existen varias alternativas en cuanto a herramientas. Más específicamente, estas alternativas pueden ser agrupadas por su nivel de abstracción de código. Por ejemplo, cuanto menos abstracto sea el código, la definición del modelo puede ser muy personalizada y detallada, mientras que si el nivel de abstracción del código es más alto, el nivel de detalle del modelo decrece pero da a lugar a que los desarrolladores puedan crear modelos más extensos y complejos utilizando bloques constructivos abstractos.

Como se mencionó en secciones anteriores, hoy en día existen una cantidad considerable de entornos de desarrollo o bibliotecas para la creación de modelos de aprendizaje profundo, como por ejemplo *TensorFlow*, *Keras* o *PyTorch*. Sin embargo, a pesar de la simplicidad de estas bibliotecas, estas fallan en el momento de dar a los desarrolladores garantías de que la definición estructural del modelo es correcta. Particularmente, la mayoría de las bibliotecas de aprendizaje profundo actuales carecen de controles estructurales estáticos en tiempo de compilación y, en consecuencia, no evitan la aparición de errores de definición en tiempo de ejecución [22].

Para ilustrar estos problemas, en el código 3.2 se muestra la implementación del mismo modelo *MNIST* definido en el fragmento de código 3.1, solo que esta vez está implementado utilizando la biblioteca Keras para el lenguaje Python. Para hacer énfasis en lo que Keras falla en validar, la definición del modelo introduce un error en la forma de la salida esperada del mismo.

**Fragmento de código 3.2:** Ejemplo simple de Keras implementado en Python

```
from keras.layers import Activation
from keras.layers import Dense
from keras.layers import Flatten
from keras.models import Sequential

model = Sequential()
model.add(Flatten(input_shape=(28, 28))
model.add(Dense(42))
model.add(Activation(activation="relu")
model.add(Dense(11))
model.add(Activation(activation="softmax")
```

Como se puede apreciar en el código 3.2, la forma de la salida del modelo es un vector de tamaño 11, lo cual puede afectar drásticamente la ejecución del programa en tiempo de ejecución. Otro error quizás más evidente es la concatenación de capas que no son compatibles entre sí, como por ejemplo la ejecución de una capa de convolución 2D luego de una capa `Flatten`, como se puede apreciar en el siguiente fragmento de código 3.3.

**Fragmento de código 3.3:** Capas incompatibles en modelo implementado en Keras para Python

```
model = Sequential()
model.add(Flatten(input_shape=(28, 28))
model.add(Conv2D(filters=10, kernel_size=3))
model.add(Activation(activation="relu")
model.add(Dense(10))
model.add(Activation(activation="softmax")
```

TensorSafe ataca estos errores estructurales en la definición haciendo uso de sofisticadas funcionalidades del sistema de tipos de Haskell. El uso avanzado de técnicas de programación a nivel de tipos hace posible la aparición de errores de compilación cuando cualquier falla estructural es detectada en el modelo. Por ejemplo, si la forma de la salida del modelo no es igual a la forma especificada en la definición del modelo, como fue mostrado en el fragmento de código 3.1, el compilador crea una excepción en tiempo de compilación con el siguiente mensaje: "`Couldn't match the Shape 'D1 10 with the Shape 'D1 11`", donde la primera forma representa la forma especificada de la salida

del modelo, y la segunda es la forma del resultado del modelo luego de pasar por todas las transformaciones de las capas del mismo. Para ejemplificar este escenario, en el fragmento de código 3.4 se muestra la introducción de un error en el tamaño de salida de la segunda capa `Dense` del modelo, haciendo que el tamaño de salida esperado y el resultante sean diferentes.

**Fragmento de código 3.4:** Introducción de error en la definición de la capa `Dense`

```
type MNIST = MkINetwork
    '[ Flatten
     , Dense 784 42
     , Relu
     , Dense 42 11 -- Introduccion de error
     , Softmax
     ]
    ('D2 28 28)
    ('D1 10)
```

De manera similar, si se deseara realizar un cambio en la implementación como por ejemplo agregar una capa de convolución 2D, como se presentó en el fragmento de código 3.3, y como se ejemplifica en el código de ejemplo 3.5, *TensorSafe* respondería con un error de compilación similar al siguiente: "`Cannot apply the Layer Conv2D 1 10 3 3 1 1 with the input Shape 'D1 42`". Con esta información, los programadores pueden arreglar sus modelos ajustando solamente los parámetros necesarios que están causando problemas estructurales.

**Fragmento de código 3.5:** Introducción de error al agregar capas con operaciones incompatibles

```
type MNIST = MkINetwork
    '[ Flatten
     , Conv2D 1 10 3 3 1 1,  -- Introduccion de error
     , Dense 784 42
     , Relu
     , Dense 42 10
     , Softmax
     ]
    ('D2 28 28)
    ('D1 10)
```

Una vez que el chequeador de tipos de Haskell valida que un modelo es estructuralmente correcto, los desarrolladores pueden optar por transcribir la estructura del modelo hacia un entorno de desarrollo externo para aprendizaje profundo, y de esa manera, poder entrenar o correr predicciones con la certeza de que no va a ocurrir ningún tipo de problema respecto a la arquitectura del modelo. Respecto a lo anterior, los programadores pueden hacer uso de las funciones `toCNetwork` y `generate` para convertir sus modelos de aprendizaje profundo en instrucciones de código en lenguajes diferentes. Por ejemplo la compilación del modelo `MNIST` (definido en el código 3.1) utilizando *TensorSafe* se muestra en el código de ejemplo 3.6; el resultado de dicha compilación se muestra en el fragmento de código 3.7.

**Fragmento de código 3.6:** Compilación a texto de un modelo creado en TensorSafe

```
mnist :: MNIST
mnist = mkINetwork

mnistCode :: Text
mnistCode = generate Python (toCNetwork mnist)
```

**Fragmento de código 3.7:** Texto generado a partir de compilación de modelo en TensorSafe

```
// Autogenerated code
import tensorflow as tf

model = tf.keras.models.Sequential()
model.add(tf.layers.Flatten(input_shape=[28,28,1], ))
model.add(tf.layers.Dense(units=42, input_shape=784, ))
model.add(tf.layers.ReLu())
model.add(tf.layers.Dense(units=10, input_shape=42, ))
model.add(tf.layers.Activation(activation="softmax", ))
```

# Capítulo 4

# Implementación

En este capítulo se presentan los fundamentos del diseño de *TensorSafe*. Empezamos explicando las estructuras a nivel de tipo con las cuales es posible definir modelos de redes neuronales y las técnicas de programación a nivel de tipos utilizadas para manipular dichas estructuras.

## 4.1. Descripción a nivel de tipos de redes neuronales

La mayoría de las bibliotecas existentes con grandes niveles de abstracción toman al concepto de capas como sus bloques básicos para la creación de arquitecturas de redes neuronales profundas. Como se mencionó en el Capítulo 1, una capa consiste de una o más operaciones a nivel de tensores que involucran valores de entrada y una salida que es el resultado de la operación. Las operaciones sobre tensores actúan funcionalmente sobre las formas de los mismos, significando que dada una forma específica de entrada, sólo hay una única posible forma de salida [15]. La forma de los valores de salida es a menudo definida con una carga semántica por los desarrolladores según el problema que quieran atacar. Para ejemplificar, si el modelo se creó con el objetivo de resolver un problema de clasificación, la forma de los valores de salida del modelo podría ser representada como un vector de largo $n$, donde $n$ es el número de clases que el modelo aprendió a predecir, de forma tal que cada valor del vector representa la probabilidad de que cada clase sea la respuesta correcta a la predicción [15].

En los escenarios más simples que involucran aprendizaje profundo, las

arquitecturas de redes neuronales son definidas como una secuencia de capas. El tipo de arquitectura secuencial de modelos de aprendizaje profundo es de las más elementales y fáciles de comprender: se requiere una forma de los valores de entrada esperados y una lista ordenada de capas que transformarán la forma de estos valores de entrada en otra forma para los valores de salida. Como consecuencia, si se quiere especificar un modelo secuencial como un tipo, esto se puede lograr simplemente creando una estructura a nivel de tipos que represente una secuencia de capas. En el código 4.1 se presenta una definición básica de una estructura de tipos de redes neuronales en Haskell:

**Fragmento de código 4.1:** Definición del tipo de datos `Network`

```
data Network :: [Type] -> Type where
  NNil  :: Network '[]
  (:~~) :: Layer x
        => x
        -> Network xs
        -> Network (x ': xs)
```

La estructura de datos `Network` define un Tipo de Datos Algebraico Generalizado (GADT - Generalized Algebraic Data Type en inglés) que representa una lista heterogénea, donde cada elemento de la lista corresponde a una capa del modelo. La clase `Layer` restringe los posibles tipos que pueden ser utilizados al momento de popular la estructura. Es conveniente observar que el tipo `Network` está indexado por una lista que contiene los tipos de las capas que van a definir el modelo de aprendizaje profundo.

A pesar de que el tipo `Network` logra definir la estructura básica de un modelo secuencial de aprendizaje profundo, éste no logra satisfactoriamente brindar algún tipo de garantía de validez estructural del modelo que se define. De hecho, es posible combinar capas en el modelo que pueden ser no compatibles ente si, por ejemplo: que una capa que opere a nivel de matrices realice una operación sobre un vector. Es más, la forma del resultado de aplicar todas las capas del modelo puede ser diferente a la que el desarrollador esté esperando. Por este motivo, con el objetivo de brindar validaciones estructurales de los modelos es necesario agregar restricciones a nivel de tipo que complementen esta definición y logren capturar este tipo de problemas.

Para representar una red neuronal profunda válida, las estructuras a nivel de tipo necesitan validar dos componentes principales. En primer lugar, es

necesario corroborar la compatibilidad entre capas, y en segundo lugar, el tipo de datos `Network` necesita capturar el concepto de manipulación de las formas de los valores de entrada y salida.

## 4.2. Computando formas a nivel de tipo

Para capturar la idea de manipulación de formas es necesario construir una declaración de un tipo de datos para las formas. *TensorSafe* adopta la definición de un tipo de datos de formas de manera muy similar a como lo afronta la biblioteca *Grenade*[1]. De manera similar a *Grenade*, en *TensorSafe* se define un tipo de datos de formas para tres posibles estructuras de datos: un vector unidimensional, una matriz bidimensional, o un tensor tridimensional.

El tipo `Shape`, definido en el fragmento de código 4.2, es promovido a nivel de tipos utilizando la extensión `DataKinds` [25] de GHC; los tipos promovidos se convierten en tipos de tipos, los que usualmente se denominan como *kinds*.

**Fragmento de código 4.2:** Definición del tipo de datos `Shape`

```
data Shape
  = D1 Nat
  -- Vector. Length.
  | D2 Nat Nat
  -- Matrix. Rows, columns.
  | D3 Nat Nat Nat
  -- Tensor. Rows, columns, channels.
```

Por ejemplo, la forma de los valores para una imagen de $28 \times 28$ píxeles con tres canales de color se puede representar con el tipo `D3 28 28 3`. La extensión `DataKinds` hace posible usar los constructores de datos `D1`, `D2` y `D3` como constructores de tipos. Como resultado, se hace realmente simple manipular formas, o `Shapes`, a nivel de tipos definiendo funciones a nivel de tipos, conocidas en el ecosistema Haskell como `type families` [17].

En nuestro caso, a través de una `type family` podemos describir, para cada capa, la mutación que la misma produce en las formas de los valores. La función a nivel de tipos `Out` toma como valores de entrada una capa (`Layer`) y una forma (`Shape`), y produce como resultado una nueva forma (`Shape`) que

[1] `https://github.com/HuwCampbell/grenade`

representa la mutación de la forma de entrada relacionada con la operación asociada a la capa.

```
type family Out (l :: Type)
                (s :: Shape) :: Shape
```

Para ejemplificar, mostramos parte de la declaración de la función `Out` para la capa `Flatten`, una capa que reestructura cualquier forma en un vector unidimensional.

```
Out Flatten ('D1 x)     = 'D1 x
Out Flatten ('D2 x y)   = 'D1 (x * y)
Out Flatten ('D3 x y z) = 'D1 (x * y * z)
```

En resumen, es posible operar con facilidad a nivel de tipos con formas de valores. Para ello, hacemos uso de una gran diversidad de utilidades que Haskell provee al momento de trabajar con números naturales y estructuras a nivel de tipos.

## 4.3. Creación de modelos válidos

Siendo que `Network` sólo define una secuencia de capas, al momento de involucrar formas de valores es importante notar que una simple secuencia de capas puede ser compatible con más de una forma de entrada de valores. Por ejemplo, una misma arquitectura podría ser compatible con imágenes de $28\times28$ o $56\times56$. Por este motivo, introducimos un nuevo tipo de datos, llamado `INetwork`, cuyo nombre hace referencia a "instancia de una Network". Este nuevo tipo representa una secuencia de capas, además de *todas* las mutaciones de formas de valores involucradas en la red, desde la forma de los valores iniciales hasta la última. Es un paso intermedio hacia la existencia de una estructura capaz de detectar errores estructurales. Su definición es la siguiente:

**Fragmento de código 4.3:** Definición de `INetwork`

```
data INetwork :: [Type] -> [Shape] -> Type
where
  INNil :: SingI i
        => INetwork '[] '[i]
  (:~>) :: (SingI i, SingI h, Layer x)
        => x
        -> (INetwork xs (h ': hs))
        -> INetwork (x ': xs) (i ': h ': hs)
```

`INetwork` es una versión ligeramente modificada de la estructura de datos principal utilizada por la biblioteca *Grenade*. El tipo de datos `INetwork` es la piedra fundamental de *TensorSafe*. Representa una secuencia de tipos de capas en su primer parámetro, y una secuencia de formas como segundo parámetro. Las formas que se agregan a la lista estructurada en `INetwork` en cada paso de construcción tiene el objetivo de representar el resultado de una mutación de una forma, correspondiente a la capa que se está insertando, a partir de una forma de valores previa en la lista. Para el caso de la lista vacía, simplemente inicializamos la lista de formas con la forma inicial que esta instancia de red neuronal profunda va a utilizar. A modo de ejemplificación, utilizando la definición del modelo utilizado en el Fragmento de Código 3.1, el tipo `INetwork` resultante de tal definición tiene la estructura como la que se muestra en el siguiente Fragmento de Código 4.4.

**Fragmento de código 4.4:** Estructura del tipo `INetwork` para la estructura del modelo `MNIST` declarado en el fragmento de código 3.1

```
INetwork
    (': Flatten
    (': (Dense 784 42)
    (': Relu
    (': (Dense 42 10)
    (': Sigmoid ('[])
    )))))

    (': Shape ('D3 28 28 1)
    (': Shape ('D1 784)
    (': Shape ('D1 42)
    (': Shape ('D1 42)
    (': Shape ('D1 10)
    (': Shape ('D1 10) ('[])
    ))))))
```

El uso de tipos *Singleton* brinda todas funcionalidades necesarias para la manipulación de las formas a nivel de valores. Los tipos *Singleton* permiten que los programas desarrollados en Haskell puedan simular tipos dependientes, de esta forma es posible operar a nivel de tipos haciendo conexiones con valores de estos [10].

La clase `SingI` es proporcionada por el paquete *singletons* de Haskell [1] y hace posible obtener en tiempo de ejecución los valores de los tipos singleton. Por lo tanto, los valores de las formas pueden ser obtenidos en ejecución, haciendo posible la compilación de los modelos a partir de sinónimos de tipos.

A persar que el tipo de datos `INetwork` soporte el concepto de capas y formas, esto todavía no es suficiente para garantizar la prevención de errores estructurales, considerando que no existen restricciones a nivel de tipos para prevenir a los desarrolladores de ingresar formas o capas equivocadas en la estructura `INetwork`. Como consecuencia, es necesario realizar validaciones estructurales a nivel de tipos para asegurar las funcionalidades deseadas en este trabajo.

Con la intención de crear redes neuronales válidas, es necesaria una restric-

[1] `http://hackage.haskell.org/package/singletons`

ción a nivel de tipos para corroborar que las capas y las formas se corresponden con una estructura de aprendizaje profundo correcta. Para mencionar algunas dificultades con esta declaración de `INetwork`, no hay una restricción explícita con respecto a las longitudes de la lista de capas y formas. Además, no hay ningún tipo de restricción con respecto a la correspondencia entre las transformaciones de las capas y las formas.

Para resolver es tipo de problemas, se creó una clase con el nombre `ValidNetwork`. La misma se definió en Haskell con las siguientes instancias:

**Fragmento de código 4.5:** Declaración de la clase `ValidNetwork`

```
class ValidNetwork
  (xs :: [Type])
  (ss :: [Shape])
where
  mkINetwork :: INetwork xs ss

instance
  (SingI i) => ValidNetwork '[] '[i]
where
  mkINetwork = INNil

instance ( SingI i
         , SingI o
         , Layer x
         , ValidNetwork xs (o ': rs)
         , (Out x i) ~ o
      ) => ValidNetwork
             (x ': xs)
             (i ': o ': rs)
where
  mkINetwork = layer :~> mkINetwork
```

Básicamente, la definición de la clase `ValidNetwork` especifica que una instancia de `INetwork` solamente puede ser válida si esta es vacía o si se agrega a la instancia una capa nueva cuya transformación corresponda con la forma de entrada de la capa siguiente. Es de suma importancia notar la última restricción declarada, `(Out x i)  o`, ya que es esta restricción la que verifica dos aspectos

muy importantes. En primer lugar, que sea posible operar entre la capa $x$ y la forma $i$, y en segundo lugar, que también sea verdad que el resultado de la transformación de la capa $x$ aplicada a la forma $i$ sea *igual* a la forma $o$.

Resumiendo, con una definición de clase suficientemente concreta como se mostró en la definición de `ValidNetwork`, es bastante fácil definir modelos de aprendizaje profundo declarando un tipo `INetwork` y luego instanciando una variable utilizando la función `mkINetwork`, como se muestra en el siguiente ejemplo en el fragmento de código 4.6:

**Fragmento de código 4.6:** Instancia de `INetwork` en *TensorSafe*

```
type MNIST = INetwork
    '[ Flatten
     , Dense 784 42
     , Relu
     , Dense 42 10
     , Sigmoid ]
    '[ 'D2 28 28
     , 'D1 784
     , 'D1 42
     , 'D1 42
     , 'D1 10
     , 'D1 10 ]

mnist :: MNIST
mnist = mkINetwork -- it works!
```

Para concluir, con la estructura de tipos `INetwork` y sumando las restricciones a nivel de clases por parte de `ValidNetowrk`, es posible crear redes neuronales profundas válidas y obtener instantáneamente confirmaciones por parte del compilador si el modelo va a funcionar o no, y en el caso de que no funcione, se pueda confirmar en qué lugar de la red la estructura es incorrecta. Sin embargo, es posible mejorar este enfoque y crear una forma mucho más simple de definir modelos de aprendizaje profundo, como se detallará en la siguiente sección.

## 4.4. Mejoras y simplificaciones a nivel de tipo

Para los programadores, es demasiado complejo pensar en cada mutación de las formas al definir un modelo de red neuronal. Además, cualquier cambio menor en la estructura puede llevar a cambios importantes en las estructuras de formas, haciendo que los desarrolladores reescriban una gran cantidad de código cada vez que realicen actualizaciones menores en sus modelos. Para mitigar estos problemas, *TensorSafe* presenta un nuevo enfoque para la definición de modelos de aprendizaje profundo que consiste en hacer que el desarrollador decida solamente las formas de entrada y salida del modelo, como se mostró en el Capítulo 3.

Para *TensorSafe* es posible saber el resultado de todas las transformaciones tomando como punto de partida una forma de valores inicial, siendo que todo el conocimiento de las transformaciones de las capas está concentrado en la función a nivel de tipos `Out`. Como consecuencia, es fácil crear funciones a nivel de tipos adicionales que contemplen este conocimiento y hagan simple para los desarrolladores poder escribir sus estructuras de modelos de aprendizaje profundo.

Particularmente en *TensorSafe*, una función a nivel de tipos llamada `MkINetwork` (notar la `M` en mayúscula) fue creada con la siguiente firma en Haskell:

**Fragmento de código 4.7:** Declaración de la función de tipos `MkINetwork`

```
type family MkINetwork
    (layers :: [Type])
    (sIn    :: Shape)
    (sOut   :: Shape) :: Type
```

La función a nivel de tipos `MkINetwork` recibe una lista de capas como su primer parámetro, muy similarmente a la estructura de datos `Network` presentada anteriormente en este capítulo. Adicionalmente, esta función de tipos recibe como parámetros una forma de entrada y otra de salida. El tipo resultado de esta función es un tipo `INetwork` en el caso de que la definición sea correcta, lo que significa que el resultado de computar la forma final de los valores a partir de las capas y el parámetro de la forma de salida `sOut` son los mismos. De otra forma, la invocación de esta función crea una excepción a nivel de tipos en tiempo de compilación.

Para poder construir todos los valores intermedios de las formas que representan todas las transformaciones de las capas empezando desde la forma inicial `sIn`, se creó una función `ComposeOut`:

**Fragmento de código 4.8:** Declaración de la función de tipos `ComposeOut`

```
type family ComposeOut'
    (layers :: [Type])
    (s      :: Shape) :: [Shape] where

    ComposeOut' '[] s      = '[]
    ComposeOut' (l : ls) s =
         Out l s ': ComposeOut' ls (Out l s)

type family ComposeOut
    (layers :: [Type])
    (s      :: Shape) :: [Shape] where

    ComposeOut '[] s = '[]
    ComposeOut ls  s = s ': ComposeOut' ls s
```

Como se puede apreciar, la función a nivel de tipos `ComposeOut` construye una lista de formas con el resultado de aplicar la función `Out` a cada una de las capas, utilizando el resultado de las mutaciones de la formas. Como consecuencia, esta función hace mucho más fácil y conveniente para los desarrolladores la tarea de definir todas las formas involucradas en las transformaciones del modelo, de esta forma no es necesario actualizar las formas intermedias ante un eventual cambio en la definición de las capas del modelo.

## 4.5. Conclusiones sobre proceso de implementación

Como se pudo apreciar en este capítulo, fue posible demostrar la creación de estructuras sofisticadas de tipos con las cuales los desarrolladores pueden definir modelos de aprendizaje profundo. Encima de esto, también quedó claro que el uso de funciones a nivel de tipos utilizando `type-families`, y el uso de `data kinds` y `singletons` son de gran utilidad y fundamentales al momento

de crear garantías estructurales en tiempo de compilación sobre las definiciones de modelos de redes neuronales. Por último, se presentó una simplificación a la declaración incial propuesta en este capítulo, dando una funcionalidad a los desarrolladores capaz de modelar y verificar en tiempo de compilación modelos de aprendizaje profundo con tan solo una secuencia de capas, una forma de entrada y otra de salida.

# Capítulo 5

# Compilación a entornos de desarrollo externos

En este capítulo se presentan las funcionalidades que *TensorSafe* ofrece para la compilación de modelos de aprendizaje profundos, cuya estructura fue validada, a entornos de desarrollo externos. En primer lugar, se describe cómo es posible la transformación de la estructura del tipo `INetwork` hacia una nueva estructura de datos que simplifica el proceso de compilación. Posteriormente, este capítulo presenta un par de utilidades provistas por *TensorSafe* las cuales hacen posible la verificación y compilación de modelos de aprendizaje profundo.

## 5.1. Construyendo una estructura genérica para redes neuronales

*TensorSafe* provee herramientas que hacen posible la compilación a bibliotecas externas de aprendizaje profundo a partir de estructuras verificadas basadas en el tipo `INetwork`. En particular, se requiere un nuevo tipo de datos genérico para manejar el proceso de compilación a código externo a partir de una estructura verificada `INetwork`. Además de eso, esta nueva estructura de datos necesita ser suficientemente genérica para poder soportar la compilación a entornos de desarrollo externos que potencialmente pueden ser muy diferentes, lo que significa que a partir de una misma estructura original es posible generar instrucciones de código diferentes. Últimamente, este nuevo tipo de datos debe englobar el concepto de secuencia de capas, además de que cada declaración de capa puede requerir sus propios parámetros y metadatos que

hagan posible la transcripción a código externo.

Explícitamente para *TensorSafe*, una estructura de datos llama `CNetwork` se creó con el objetivo de simplificar las tareas compilación y transcripción. La definición de `CNetwork` se puede apreciar en el código de ejemplo 5.1. Adicionalmente, un tipo de datos auxiliar fue necesario para generar la correspondencia entre cada tipo `Layer` y el tipo `DLayer`. Es importante destacar que el tipo de datos `CNetwork` no se creó con la intención de que fuese utilizado directamente por los usuarios de *TensorSafe*, sino que su objetivo básicamente es facilitar la transformación a partir de un tipo de datos `INetwork` a instrucciones de código externas.

**Fragmento de código 5.1:** Definición de `CNetwork`

```
data CNetwork
    = CNSequence CNetwork
    | CNCons CNetwork CNetwork
    | CNLayer DLayer (Map String String)
    | CNReturn
```

A pesar de ser simple, el tipo de datos `CNetwork` define una estructura de datos para arquitecturas de redes neuronales secuenciales. El constructor `CNSequence` se utiliza para empezar la definición secuencial de un modelo de aprendizaje profundo. El constructor `CNCons` añade nuevas definiciones de `CNetwork` de forma recursiva. Con el constructor `CNLayer` es posible agregar capas a la definición de la red, especificando un constructor del tipo `DLayer`. En particular para cada instancia de `CNLayer`, se dispone de un conjunto de parámetros representados por una estructura `Map String String`. Estos parámetros proporcionan información adicional acerca de la invocación de la capa, lo cual resulta potencialmente útil al momento de realizar la compilación de una estructura `CNetwork` a código externo. Finalmente, `CNReturn` es utilizado para declarar el fin de la declaración de una red neuronal.

Al momento de transformar una instancia de `INetwork` a una estructura de tipo `CNetwork`, es necesario crear una función capaz de tomar una instancia de un modelo basado en `INetwork` como entrada y cuya salida corresponda a la representación del mismo modelo en el tipo `CNetwork`. Dicha transformación se puede realizar de forma recursiva, iterando a través de la estructura `INetwork`. Especialmente, para cada declaración de capa en un modelo `INetwork`, se agrega a la estructura `CNetwork` resultante una construcción utilizando la definición

de `CNLayer`. Esto es posible mediante la función `compile`, la cual la disponen todas las instancias de la clase `Layer`. Más en detalle, la función `compile` crea una relación de correspondencia entre los tipos `Layer` y los tipos `CNLayer`. Para sumarizar, una función con las características presentadas puede generar completamente una representación de un modelo en el tipo `CNetwork` a partir de una instancia del tipo `INetwork`.

En *TensorSafe* la función `toCNetwork` convierte a una instancia de tipo `INetwork` en una estructura de tipo `CNetwork`. La función requiere una restricción de tipo `ValidNetwork` en el parámetro de tipo `INetwork` para estar seguros de que la estructura de la instancia `INetwork` es válida. En el siguiente ejemplo de código 5.2 presentamos la firma de la función `toCNetwork`.

**Fragmento de código 5.2:** Firma de la función `toCNetwork`

```
toCNetwork ::
  ( SingI i
  , Layer x
  , ValidNetwork (x ': xs) (i ': ss)
  ) => INetwork (x ': xs) (i ': ss)
    -> CNetwork
```

## 5.2. Compilación a entornos de desarrollo específicos

Una vez que una instancia de un modelo es convertido a una estructura de datos `CNetwork`, es posible transformar la estructura en instrucciones de código para bibliotecas de aprendizaje profundo externas. En la práctica, las bibliotecas de aprendizaje profundo disponible no necesariamente utilizan la misma estructura de código al momento de definir los modelos. Sin embargo, debido al alto nivel de abstracción del tipo `CNetwork`, es posible representar diferentes instrucciones de código a partir de la misma estructura de datos.

Especialmente en *TensorSafe*, se tomó la decisión de realizar la compilación a partir de la estructura interna presentada para dos bibliotecas externas específicas. La primer biblioteca elegida fue *Keras* para *Python*[1], y la segunda biblioteca elegida fue *Keras* para *JavaScript*[2].

[1] `https://keras.io/`
[2] `https://www.tensorflow.org/js/`

La generación de código se puede lograr especificando instancias de generadores por medio de la clase `Generator`, la cual es presentada en el fragmento de código 5.3. Estas instancias de generadores proveen una función `generate` capaz de transformar una estructura `CNetwork` a texto. Los detalles de implementación de esta función involucran la especificación del código generado por cada componente de la estructura de la definición de `CNetwork`. Luego, utilizando comandos provistos por *TensorSafe*, el texto generado por la función `generate` puede ser impreso a la consola, o opcionalmente salvar la definición del modelo en un archivo en disco duro.

**Fragmento de código 5.3:** Definición de clase `Generator`

```
class Generator l where
  generate :: l -> CNetwork -> Text
```

En resumen, *TensorSafe* ofrece funcionalidades de compilación para modelos de aprendizaje profundo correctamente definidos. Además, se presentó la idea de que se necesita de un tipo de datos abstracto para representar estructuras basadas en el tipo `INetwork` y de esta manera hacer posible la generación de código a entornos de desarrollo diferentes. Finalmente, *TensorSafe* ofrece un conjunto de herramientas de línea de comando capaces de transformar modelos de redes neuronales correctamente estructurados en instrucciones de código compatibles con los entornos de desarrollos externos propuestos.

# Capítulo 6

# Resultados y evaluación experimental

En este capítulo presentamos los resultados y análisis sobre experimentos en la práctica utilizando *TensorSafe*. En particular, y como se mencionó al comienzo de este documento, se implementaron utilizando *TensorSafe* tres modelos bien conocidos por la comunidad. La implementación de estos modelos se puede hallar en el módulo `TensorSafe.Examples` de la biblioteca *TensorSafe* presentada en este documento.

En particular, los experimentos fueron ejecutados en una Macbook Pro 2015 (con un procesador Intel Core i7 de 2.2GHz y 16GB de RAM) sobre el sitema operativo MacOS Mojave. La biblioteca *TensorSafe* fue compilada utilizando GHC en su versión `8.6.3` y la herramienta `stack` con versión `1.9.3`.

El primer modelo involucró la implementación de una versión de *MNIST* basada en una red neuronal convolucional. Este modelo fue fácilmente desarrollado utilizando capas de convolución 2D, capas de max-pooling 2D, capas totalmente conectadas (Dense) y finalmente capas de activación [6]. Además, el modelo fue especificado para soportar una entrada con forma $28 \times 28$ y un vector de salida de largo 10. Varios chequeos de compilación fueron probados introduciendo errores en la definición estructural del modelo, como por ejemplo alterando la forma de salida, o configurando un número incorrecto de filtros en las capas convolucionales. La implementación de este modelo se puede encontrar en la biblioteca *TensorSafe* en el módulo `TensorSafe.Examples.Mnist`.

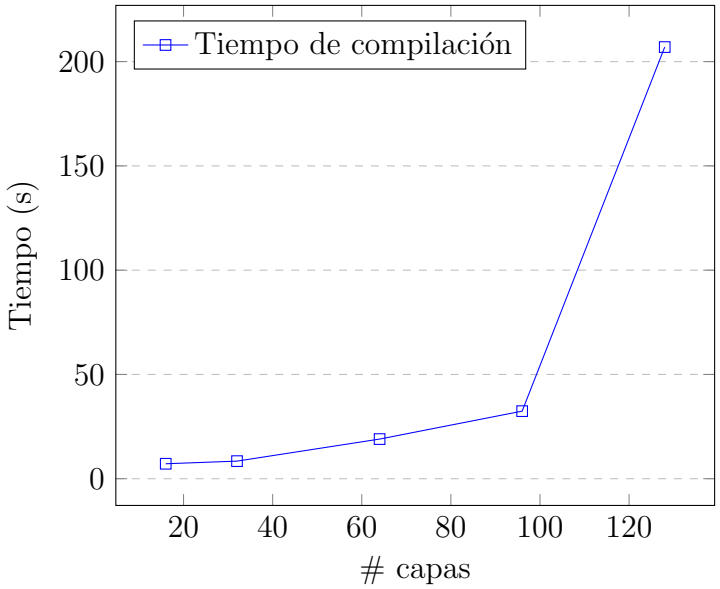
El segundo modelo escogido para implementar fue `AlexNet` [14]. Este modelo es uno de los más reconocidos en la comunidad de aprendizaje profundo,

dado que en el 2012 logró obtener resultados muy superiores en una base de datos de clasificación de imágenes llamada *ImageNet* [7]. Si bien la arquitectura de AlexNet no es muy extensa, la configuración de parámetros es muy sensible a errores, haciendo que ante cualquier cambio pequeño en modelo no de los resultados esperados. Este modelo involucra el uso de 5 capas convolucionales 2D, capas de max-pooling 2D y capas totalmente conectadas (Dense) [14]. La implementación de este modelo también se puede encontrar en el módulo `TensorSafe.Examples.AlexNet` de la biblioteca *TensorSafe*.

Finalmente, el tercer modelo implementado fue *ResNet50*[11], el cuál es mucho más extenso en cantidad de capas que el primer modelo mencionado. Muy similarmente, la definición de este modelo se pudo realizar utilizando las capas provistas por *TensorSafe*. A diferencia del modelo *MNIST* o *AlexNet*, el modelo para `ResNet50` contiene más de 150 capas internas. Si bien *TensorSafe* pudo satisfactoriamente verificar la estructura de este modelo, los tiempos de compilación de programas utilizando GHC se elevaron a ordenes de segundos o incluso minutos debido a los tiempos de enlace en la compilación de programas Haskell.

En la figura 6.1 se puede apreciar una gráfica indicando en el eje $y$ los tiempos de compilación por parte de GHC al momento de compilar programas que definan el modelos con un determinado número de capas. Se puede apreciar un claro patrón exponencial, indicando que cuanto más capas el modelo tenga, más tiempo va a llevar enlazar el programa. Entendemos que este es un fenómeno resultado de problemas de desempeño con la extensión *TypeFamilies* de Haskell.

En resumen, *TensorSafe* mostró buenos resultado en la construcción de modelos bien conocidos por la comunidad. Además, los resultados prueban que con esta biblioteca se pueden compilar modelos de aprendizaje profundo grandes, aunque tome una cierta cantidad de tiempo más grande para modelos con mayor cantidad de capas.




| *# capas* | *Tiempo de compilación* |
|---|---|
| 16 | 7.17 |
| 32 | 8.44 |
| 64 | 19.01 |
| 94 | 32.41 |
| 128 | 206.97 |

**Figura 6.1:** Tiempo de compilación según número de capas en un modelo

# Capítulo 7

# Conclusiones y trabajo a futuro.

En este documento presentamos la definición y detalles de implementación de *TensorSafe*, una biblioteca con la cual se pueden definir modelos de aprendizaje profundo válidos en tiempo de compilación. A lo largo del documento, hubo numerosas referencias a la importancia de las herramientas a nivel de tipos que ofrece Haskell. En particular, el uso de *DataKinds*, *TypeFamilies* y los tipos *singleton* hicieron posible la creación de estructuras de tipos abstractas y complejas capaces de representar arquitecturas de aprendizaje profundo. Posteriormente, presentamos resultados experimentales, donde se demostró la usabilidad de *TensorSafe* en escenarios reales, validando que la herramienta ejecuta verificaciones en tiempo de compilación de la manera que se propuso. Finalmente, mostramos ciertos problemas de rendimiento en tiempos de compilación cuando el modelos posee más de 100 capas.

Las lineas de trabajo futuro están orientadas a extender la cantidad de capas provistas por la biblioteca. Otra posible e interesante linea de investigación podría ser agregar soporte para modelos no secuenciales de aprendizaje profundo, permitiendo a *TensorSafe* definir modelos aún más complejos. Finalmente, sería interesante ver una re-implementación de *TensorSafe* en un lenguaje con soporte nativo de tipos dependientes, como lo son *Idris* o *Agda*, y realizar comparaciones en términos de la expresividad del lenguaje y en el rendimiento en los tiempos de compilación.

# Referencias bibliográficas


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Marco Barreno, Blaine Nelson, Anthony D Joseph, and J Doug Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[3] J. Bowen. Tensorflow haskell deptyped. `https://github.com/helq/tensorflow-haskell-deptyped`, 2019.

[4] Tongfei Chen. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 45–50. ACM, 2017.

[5] François Chollet. keras. `https://github.com/fchollet/keras`, 2015.

[6] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[8] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[9] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[10] Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.

[13] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[16] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. *ACM SIGPLAN Notices*, 48(12):81–92, 2014.

[17] Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

[18] Tom M Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11), 1999.

[19] Thuy TT Nguyen and Grenville J Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[21] Leonardo Piñeyro, Alberto Pardo, and Marcos Viera. Structure verification of deep neural networks at compilation time using dependent types. In *Proceedings of the XXIII Brazilian Symposium on Programming Languages*, pages 46–53. ACM, 2019.

[22] Norman A. Rink. Modeling of languages for tensor manipulation. *CoRR*, abs/1801.08771, 2018.

[23] Wikipedia contributors. ECMAScript — Wikipedia. `https://en.wikipedia.org/w/index.php?title=ECMAScript&oldid=925665822`, 2019. [Online; Recuperado el 21-Noviembre-2019].

[24] Wikipedia contributors. JavaScript, History — Wikipedia. `https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=927237382`, 2019. [Online; Recuperado el 21-Noviembre-2019].

[25] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

[26] Henry Zhu. The State of Babel — Babel Blog. `https://babeljs.io/blog/2016/12/07/the-state-of-babel`, 2016.

# ANEXOS

# Anexo A

# Herramientas de desarrollo de TensorSafe para JavaScript

## A.1. Evolución de JavaScript

El lenguaje de programación JavaScript se ha renovado significativamente en los últimos 10 años. Si bien este lenguaje fue diseñado en sus inicios para su funcionamiento en los primeros navegadores web, como *Netscape* o *Internet Explorer 3*, últimamente ha tomado una relevancia muy grande en otros entornos del software, como en servidores backend, aplicaciones para dispositivos móviles o incluso hasta en software para satélites. Este resurgimiento se debe en gran parte a nuevas especificaciones del lenguaje ECMAScript [1], las cuales dieron forma a JavaScript desde sus comienzos [24].

Las nuevas especificaciones de ECMAScript añadieron un sin fin de funcionalidades nuevas al lenguaje de programación. Desde su versión *ECMAScript 5.0* en el año 2009, también llamado *JavaScript 1.8.5*, se han agregado 3 versiones más, llegando a elaborar la especificación *ECMAScript 8.0* en el año 2018. Si bien estas especificaciones se aprobaron muy rápidamente para los estándares de actualización de un lenguaje de programación, los navegadores web que soportan en la actualidad JavaScript, no soportan todas las nuevas funcionalidades proporcionadas por las últimas versiones de ECMAScript, sino que a medida que avanza el tiempo, los navegadores web agregan a diferente ritmo soporte para nuevas funcionalidades [23].

Esto último lleva a que exista una demanda por parte de los desarrolladores

[1] `https://www.ecma-international.org/`

para poder escribir programas en ECMAScript con sus últimas especificaciones, pero además dichos programas puedan ser ejecutados por los navegadores web en la actualidad los cuales no cuentan con un soporte total de ECMAScript. Por ello, surge el proyecto *Babel* [1], un compilador de ECMAScript que transcribe el código a una versión de JavaScript compatible con los navegadores web de la actualidad. Este proyecto es de tal relevancia para la comunidad de desarrolladores JavaScript, que a partir del año 2016 este proyecto cuenta con más de 5 millones de descargas mensuales [26].

*Babel* también abre la posibilidad crear extensiones, o *plugins*, los cuales pueden tomar código incompatible con los estándares de ECMAScript para luego transcribirlo en código compatible con `JavaScript`. Para ejemplificar, en los entornos de desarrollo web modernos, como *Vue* [2] o *React* [3], es posible que el código fuente se conforme de una mezcla de código ECMAScript y `HTML` el cual luego puede funcionar en un navegador, siempre y cuando Babel logre transformar ese código a JavaScript correctamente. Para realizar esto, Babel trabaja sobre un *parser* que genera un árbol de sintaxis abstracta (AST) del código, con el que luego Babel es capaz de decidir qué instrucciones generar a partir de la composición de dicho árbol.

Tener un AST disponible también abre las puertas a otros programas populares entre los desarrolladores de ECMAScript, que son los *linters* o programas que analizan estáticamente definiciones estructurales del código, como por ejemplo: uso de espacios y lineas en blanco, definiciones de variables o funciones sin utilizar, orden de definiciones, entre otros.

En resumen, ECMAScript es un lenguaje muy versátil, y que la comunidad ha adoptado muy satisfactoriamente pudiendo crear ambientes de desarrollo que benefician a los desarrolladores.

## A.2. TensorSafe en JavaScript

Teniendo presente herramientas disponibles para el lenguaje JavaScript, se propuso en este proyecto la creación de un par de utilidades provistas por *TensorSafe* con las cuales pueda ser posible la integración del desarrollo de modelos de aprendizaje profundo utilizando TensorSafe en JavaScript, en un

[1] `https://babeljs.io/`
[2] `https://vuejs.org/`
[3] `https://reactjs.org`

entorno de desarrollo basado en Keras para JavaScript.

El objetivo principal de esta iniciativa es poder reemplazar la definición de un modelo de aprendizaje profundo en código JavaScript por la misma definición del modelo implementado con la sintaxis de código utilizada en TensorSafe. Con esto en mente, se requieren de dos componentes principales. Primero, se precisa un *plugin* de Babel capaz de parsear código de un modelo de deep learning en Haskell adentro de un archivo JavaScript para luego convertirlo en lineas de código JavaScript. Y en segundo lugar un *linter* que pueda ser capaz de notificar al desarrollador que su modelo presenta un error al momento de escribir el código.

De esta manera, los desarrolladores pueden hacer uso de una interfaz web, mientras validan en tiempo de compilación si sus modelos de deep learning poseen algún tipo de problema estructural. Los detalles de implementación sobre el plugin de Babel como además del *linter* se pueden hallar en la carpeta `extra/javascript` del proyecto *TensorSafe* [1].

## A.3. Evaluación experimental

Para evaluar experimentalmente esta funcionalidad, se tomó como ejemplo la implementación del modelo `MNIST` utilizando Keras para JavaScript encontrada en el proyecto *tfjs-examples* [2]. El proyecto cuenta con implementaciones de un par de modelos distintos para resolver `MNIST`: el primero utilizando sólamente capas `Dense`, y el segundo utilizando capas `Conv2D`. Además, al ser un proyecto web, se ofrece una interfaz donde un usuario puede observar el proceso de entrenamiento de los modelos y sus resultados en tiempo de predicción. Esta implementación cuenta con la ventaja de que la definición de los modelos se encuentra separada del resto de la solución, haciendo más fácil la integración con *TensorSafe*.

Para realizar la integración con *TensorSafe*, simplemente se reemplazó el código JavaScript en las funciones donde se definían los modelos de aprendizaje profundo, con la implementación del mismo modelo en *TensorSafe*, como se puede apreciar en el siguiente fragmento de código A.1. Una vez realizada la integración en cuanto a código, el *linter* o analizador de código estático comprueba que el modelo es correctamente definido o indica un error de lo

[1] `https://github.com/leopiney/tensor-safe`
[2] `https://github.com/tensorflow/tfjs-examples`

contrario.

**Fragmento de código A.1:** Ejemplo de modelo en JavaScript utilizando integración con *TensorSafe*

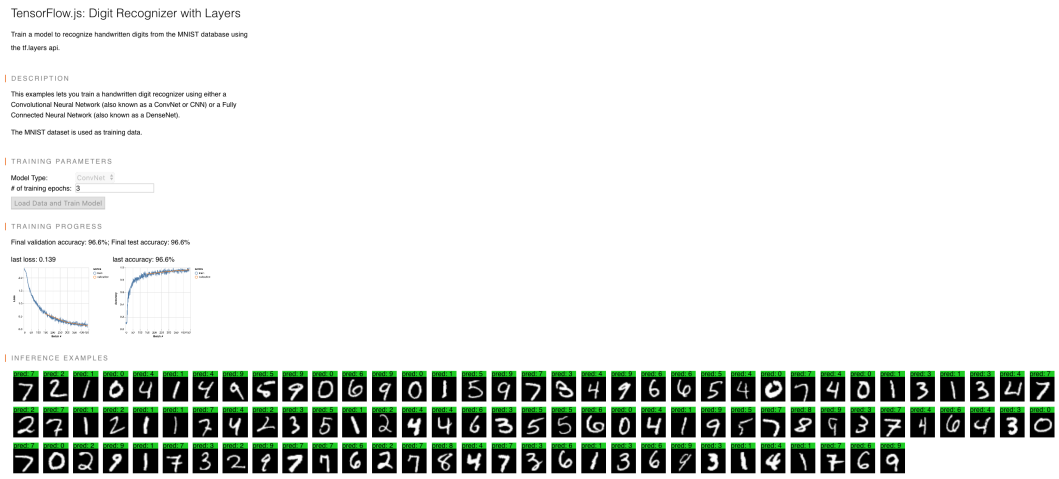
```
/**
 * Creates a convolutional neural network (Convnet)
 * for the MNIST data.
 *
 * @return {tf.Model} An instance of tf.Model.
 */

function createConvModel() {
  const model = safeModel`
    '[
        Conv2D 1 16 3 3 1 1,
        Relu,
        MaxPooling 2 2 2 2,
        Conv2D 16 32 3 3 1 1,
        Relu,
        MaxPooling 2 2 2 2,
        Conv2D 32 32 3 3 1 1,
        Relu,
        Flatten,
        Dense 288 64,
        Sigmoid,
        Dense 64 10,
        Softmax
    ]
    ('D3 28 28 1)
    ('D1 10)
`;

  return model
}
```

Finalmente, se ejecuta el proyecto web en donde se puede apreciar que el modelo es en realidad válido y se puede entrenar para resolver el problema en

**Figura A.1:** Ejecución de proyecto JavaScript integrado con *TensorSafe*




cuestión. En la figura A.1 se puede apreciar la portada de este proyecto web como solución final de la propuesta de integrar *TensorSafe* en JavaScript.

Como conclusión de esta sección, quedó demostrado que es posible realizar una integración entre *TensorSafe* y el desarrollo de modelos de aprendizaje profundo en JavaScript utilizando Keras. Es un problema que se resuelve en varias capas: primero un analizador de código estático verifica que el código Haskell para el modelo implementado en *TensorSafe* es válido, luego *Babel* convierte dicho código en instrucciones JavaScript compatibles con el entorno de desarrollo Keras, resultando en un proyecto totalmente JavaScript el cual puede ejecutarse en un navegador web moderno.

# Anexo B

# Extensibilidad de TensorSafe

La implementación de *TensorSafe* fue planeada de tal forma que sea fácil la agregación de nuevas funcionalidades, ya que desde el comienzo del proyecto estuvo estipulado que *TensorSafe* solo implementaría un conjunto reducido de capas disponibles en entornos de desarrollos modernos como *TensorFlow* o *PyTorch*. Con esto en mente, agregar nuevas capas resulta fácil en TensorSafe.

La agregación de nuevas capas a *TensorSafe* consta de 4 simples pasos. Primero, se debe agregar una nueva entrada de capa auxiliar para el tipo de datos `DLayer` en módulo `TensorSafe.Compile.Expr`. Esto hará posible la compilación de la capa para todas las instancias de `Generator`. Además, agregue a la entrada `LayerGenerator` para la capa recién agregada. Luego, se tiene que agregar la definición de capa a la carpeta `TensorSafe/Layers`, para lo cual puede resultar útil basarse en las definiciones de las capas ya definidas. En tercer lugar, hay que simplemente importar y exponer la definición de capa en el módulo `TensorSafe.Layers`. Y finalmente, se debe declarar cómo la capa transforma una forma específica en la función de tipos de `Out` en el módulo `TensorSafe.Network`.