INSTITUTO TECNOLÓGICO DE BUENOS AIRES
72.60 Programación Funcional

Trabajo Práctico Final
Perceptrón Multicapa

Análisis sobre las posibles ventajas al momento de comprender la teoría sobre el perceptrón multicapa como consecuencia de realizar la implementación en Haskell.

AUTORA Fuster, Marina Legajo 57613

DOCENTES Martinez Lopez, Pablo Ernesto (Fidel)
Pennella, Valeria Verónica

BUENOS AIRES
2021

Introducción **2**

Planteo del problema **4**
Breve Marco Teórico 4
Alcance del diseño 5

Sección Técnica **6**
Módulos 8
NeuralNetwork 9
Optimization 10
Evaluation 12
Dynamic 13
Algoritmo de Retropropagación 14
Metodología de comparación 17

Resultados **19**

Mejoras Futuras **19**

Conclusiones **20**

Bibliografía **22**

Apéndice **23**
Medición de tiempo 23
Tabla completa de resultados 23

Introducción

El objetivo de este trabajo es explorar la construcción de redes neuronales, en particular del perceptrón multicapa, desde la perspectiva de la programación funcional, con el propósito de incrementar la comprensión sobre las mismas.

Debido a la presencia de bibliotecas de funciones como PyTorch y Tensorflow, Python es el lenguaje preferido por la mayoría para el desarrollo de áreas que involucran algoritmos de Inteligencia Artificial. Según un artículo de Medium[1], aproximadamente el 57% de desarrolladores eligen Python o C++ para abordar soluciones en el área de IA. Si bien estos lenguajes permiten un rápido desarrollo, creo que estas bibliotecas no son ideales como introducción a los temas teóricos en los cuales se basan las redes neuronales. En mi experiencia personal, provocan “vacíos de conocimiento”. Su utilidad es evidente, pero, como herramienta de educación, pueden tener sus desventajas.

Por otra parte, inclusive sin el uso de las mismas y habiendo programado redes neuronales desde cero en Python, hay ciertos aspectos que no terminan de ser claros para mí, destacándose el abordaje del algoritmo de retropropagación, también llamado algoritmo de backpropagation. Realicé la materia *Sistemas de Inteligencia Artificial* hace un año y este algoritmo era una de las ideas fundacionales en las primeras redes neuronales. Incluso luego de hacer varios trabajos sobre el tema, había algo en la teoría que todavía me costaba.

Dado que fui parte del cuerpo de ayudantes de la materia durante este cuatrimenstre, decidí investigar si era una problemática a la que me había enfrentado yo sola o si otros alumnos también tuvieron dificultades con lo mismo. Se me autorizó a realizar una encuesta sobre el perceptrón multicapa a los estudiantes para conocer sus valoraciones sobre dicho algoritmo. En la misma, participaron 14 personas voluntariamente, todos de la comisión del primer cuatrimestre del año 2021. Por requisitos de la carrera, se debe tener manejo de los conceptos de diferenciación y gradiente de una función multivariable (fundamentales para comprender el algoritmo de retropropagación básico).

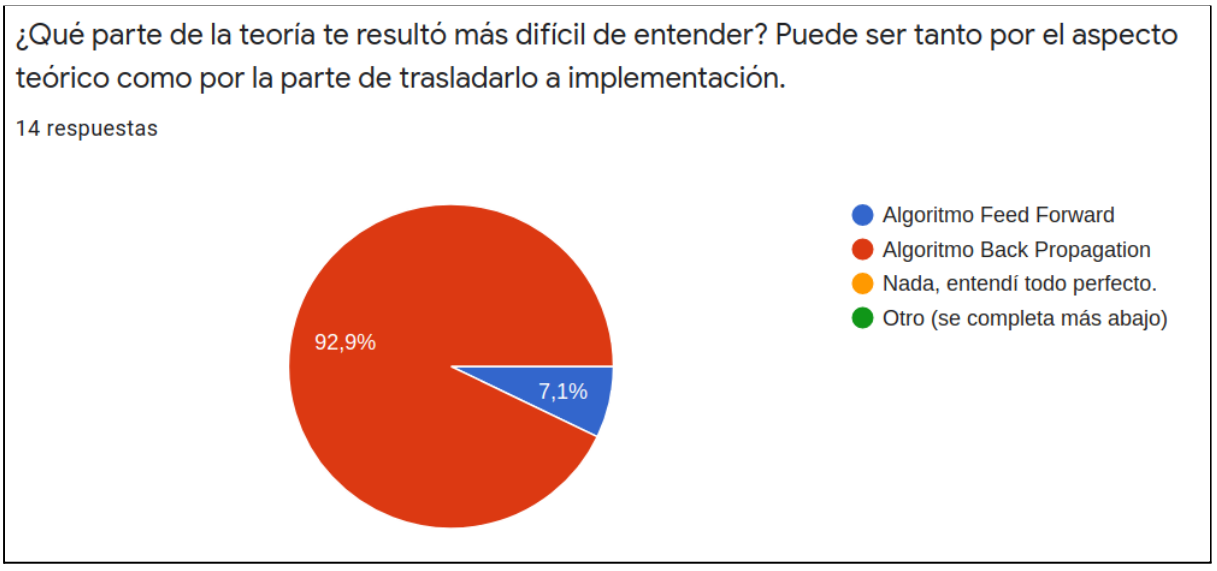


***Figura 1**: resultados sobre el aspecto que generó mayor dificultad con el perceptrón multicapa.*

Como podemos ver en ***Fig. 1***, el algoritmo de backpropagation presentó dificultades a la hora de aprender el perceptrón multicapa (lo cual pude notar que se trasladó a medida que se avanzaba a redes neuronales más complejas). En ***Fig 2***. podemos ver cómo el aspecto teórico fue la principal traba para los alumnos, en particular, el manejo de derivadas y cálculo de errores fue frecuente en los comentarios de los encuestados.

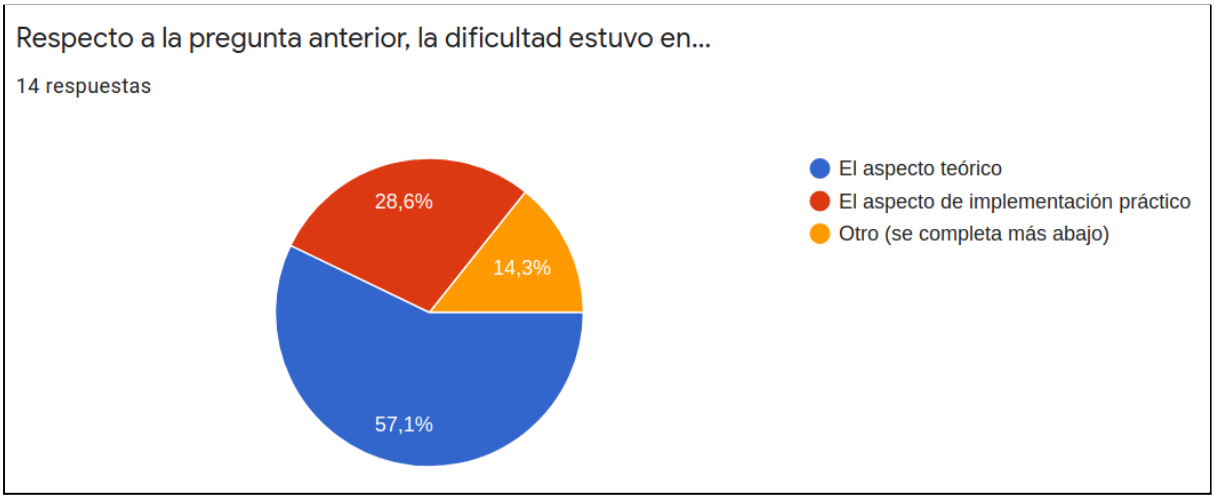


Figura 2*: resultados sobre la dificultad hallada en retropropagación.*

El objetivo que me planteo para este trabajo es analizar si, utilizando un lenguaje de programación funcional como Haskell, se pueden obtener mejoras en la comprensión del algoritmo. Sospecho que esto es posible por la naturalidad que posee la programación funcional para describir operaciones matemáticas (pues se acerca más a la visión denotacional). Al fin y al cabo, una red neuronal puede pensarse como una composición de funciones, lo cual es ideal para el paradigma de programación funcional. Por otra parte, una de las grandes ventajas de un lenguaje como Haskell es su flexibilidad para expresar ideas de muchas formas. Puedo elegir cuál es la que más me conviene para la tarea en cuestión.

Como segundo objetivo, me gustaría analizar si la eficiencia de Haskell es significativamente mejor a la de Python. Esto presenta la primera dificultad que espero encontrar. Dado que no estoy utilizando ninguna biblioteca de funciones de redes neuronales, sino implementaciones que aprovechen operaciones matriciales, será de gran dificultad definir una comparación objetiva. En primer lugar, reutilizaré el perceptrón multicapa programado en Python para la materia que realicé hace un año, con lo cual es probable que el programa no utilice buenas prácticas. Por otro lado, si bien mi intención es “comparar” dos formas de ejecutar, ni siquiera puedo asegurar que ambos programas sean equivalentes, ya que la equivalencia de funciones podemos analizarla únicamente en el mundo denotacional (al menos según las herramientas utilizadas durante la cursada).

Por último, la segunda dificultad que espero encontrar refiere a cómo analizar si la implementación en Haskell permite un mayor entendimiento a la hora de implementar redes neuronales.

A grandes rasgos, la aplicación contará con la funcionalidad necesaria para elegir un conjuntos de datos, armar la red neuronal y optimizarla, obteniendo información sobre el

desempeño de la misma luego de la optimización. Me limité a añadir la funcionalidad completa para problemas de clasificación. La red es lo suficientemente genérica para usarse en problemas de regresión, pero no están los métodos necesarios para proveer métricas sobre desempeño.

Planteo del problema

Breve Marco Teórico

El modelo teórico que se aborda durante el trabajo práctico se basa en el planteo matemático simplificado de una neurona biológica, realizado por Frank Rosenblatt en la década del 50[10]. A este modelo se lo denominó perceptrón simple, el cual podía modelarse según la siguiente ecuación:

$$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n) = f(\sum_{i=1}^{n} w_ix_i) \qquad (1)$$

Donde N es la cantidad de atributos o variables que posee cada dato, w_i son los pesos sinápticos de la neurona y x_i los valores que toman dichos atributos. La función f se denomina función de activación y puede ser tanto lineal como no lineal aunque, a lo largo del desarrollo del trabajo vamos a trabajar con funciones de activación no lineales, ya que nos permiten representar un espacio de funciones más amplio a la hora de modelar nuestro problema.

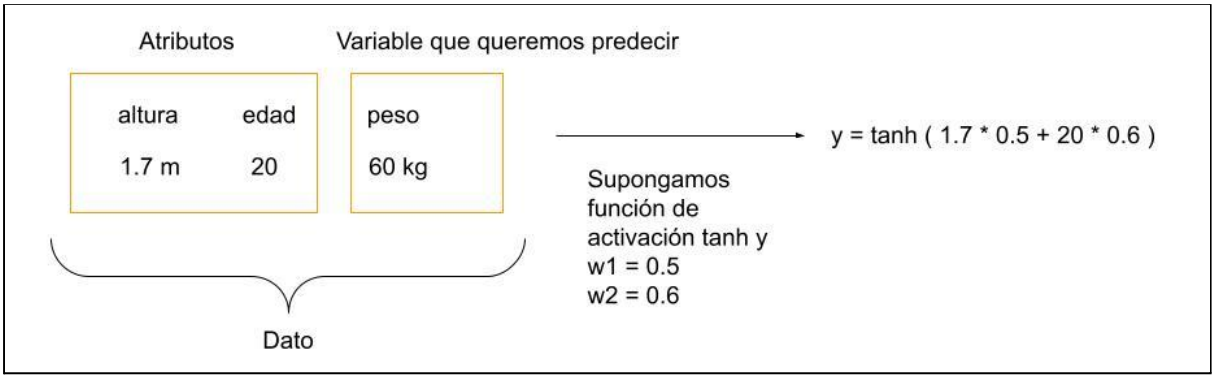


***Figura 3**: ejemplo de aplicación de la ecuación (1).*

Un concepto relevante a la hora de entender redes neuronales es el método de optimización de gradiente descendente. La red neuronal es simplemente una función que busca transformar un dato de entrada en otro dato de salida. A partir de esta transformación se puede definir otra función, llamada función de error o de costo. Esta segunda función es la que queremos minimizar, pues deseamos que la diferencia entre la salida esperada y lo que efectivamente devuelve la red sea mínima.

Esta función de error posee su propio “paisaje” con sus respectivos mínimos locales y, lo que vamos a intentar nosotros, es encontrar el mínimo de esta función usando el gradiente

descendente para movernos a través de la misma. Los gradientes nos van a indicar la dirección en la cual debemos movernos. Podemos formular esta dinámica como:

$$x_{n+1} = x_n - \gamma \cdot \nabla F(x_n) \qquad (2)$$

Donde γ es lo que denominamos ritmo de aprendizaje (comúnmente llamado *learning rate*) y F es la función de costo evaluada en x_n. Nótese que queremos ir en la dirección opuesta al gradiente ya que esto nos permitirá minimizar el valor de la función.

Para aprender relaciones no lineales entre los atributos de entrada y el valor de salida que queremos averiguar, no nos basta con redes neuronales de una sola capa, por más de que tengan muchas neuronas dentro. Para eso debemos pasar a las redes neuronales multicapa, también llamadas perceptrones multicapa. Cada capa de esta red la podemos formalizar como:

$$y = f_N(W_N \cdot (\dots f_2(W_2 \cdot f_1(W_1 \cdot x + b_1) + b_2) \dots) + b_N) \quad (3)$$

Notemos que para cada capa tenemos un vector de pesos Wy de biases b cuyo valor vamos a querer ajustar para que el error sea mínimo.

Como se mencionó previamente, la red neuronal transforma un dato de entrada en un dato de salida. Al algoritmo que se encarga de realizar esta transformación lo llamamos feed forward: dado un conjunto de datos de entrada, obtengo sus transformaciones para una red neuronal definida. Para ir modificando los pesos de todas las capas de la red neuronal durante el entrenamiento utilizamos otro algoritmo, llamado algoritmo de retropropagación o backpropagation.

Notemos que nosotros podemos calcular el error en la última capa. La derivada de esta función de error nos proveerá la dirección y magnitud en la cual tenemos que actualizar los pesos de la capa de salida para lograr el objetivo deseado. ¿Pero cómo actualizamos los pesos de las capas anteriores a la de salida? Se nos presenta el problema de que el gradiente descendente sólo provee información local. Esto es lo que busca resolver el algoritmo de backpropagation: proveer la mecánica para retropropagar la información necesaria para conseguir las actualizaciones de capas centrales a partir de capas más cercanas a la salida.

Existen otros métodos de optimización además de gradiente descendente, como por ejemplo Adam y RMSprop.

Alcance del diseño

Si bien nuestro programa tendrá contenido estático y contenido dinámico, el objetivo principal refiere a un problema de transformación de la información. Una vez definidos ciertos parámetros iniciales que pueden depender de un usuario o de números random, buscaremos que la lógica de la aplicación sea completamente funcional.

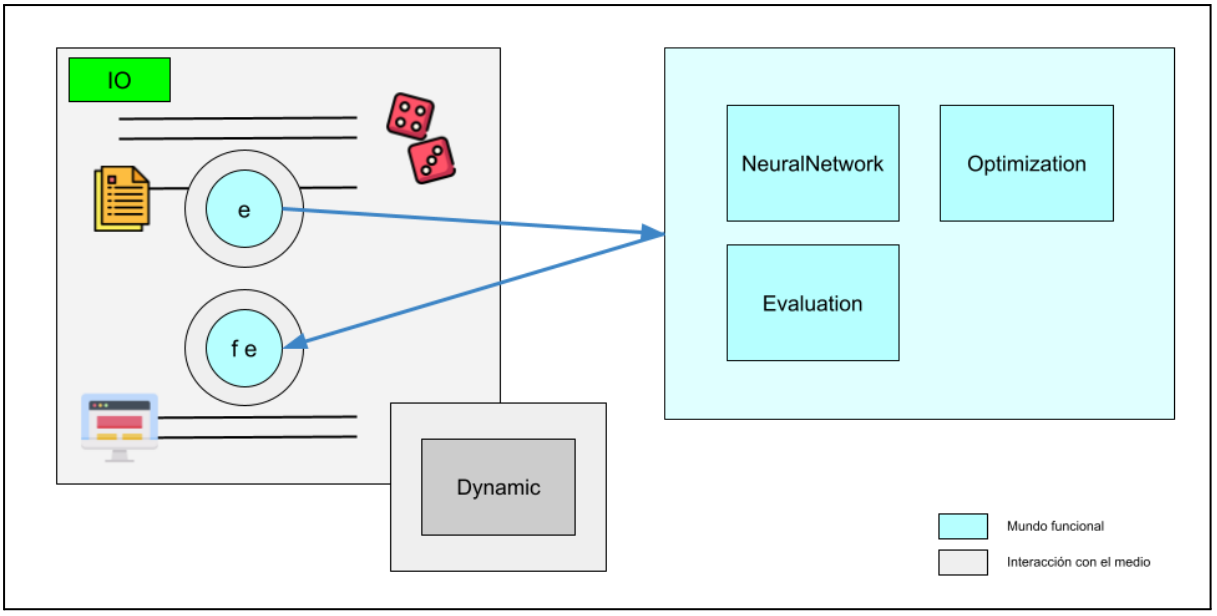


***Figura 4**: esquema de alto nivel de los módulos de la aplicación.*

En ***Fig. 4*** busca mostrarse cómo se organiza la aplicación. Tenemos tres módulos cuyo procesamiento será puramente funcional. `NeuralNetwork` contiene la descripción de todos los tipos de datos referentes a redes neuronales, así como los algoritmos de backpropagation y feed forward. `Optimization` contiene los tipos de datos y métodos necesarios para entrenar la red, como por ejemplo, el método de gradiente descendente. Por último, `Evaluation` contiene métodos para averiguar el desempeño de la red, una vez que la misma está entrenada.

La imagen busca representar cómo nos comunicamos entre el mundo funcional (en este caso, de color celeste) y el mundo físico (que incluye la pantalla de la computadora, los archivos, etc). La mónada con la cual estamos "encerrando" nuestra red neuronal para su análisis y para interaccionar con el medio es la mónada `IO`. En ***Fig. 4***, *"e"* correspondería a nuestra red neuronal y *"f e"* al resultado de las transformaciones que aplicamos.

Dado que el foco del trabajo no se encuentra en exponer esta aplicación a usuarios, decidí no utilizar un archivo de configuración JSON para proveer los detalles de configuración de la red. Se creó un template modelo para que el usuario que desee utilizar el programa sepa cómo configurarlo. La razón principal fue restricción de tiempo, ya que es mucho más conveniente evitar la compilación del programa cuando se quieren correr nuevos experimentos. En lo que respecta a la facilidad para entender cómo armar el programa, creo que Haskell tiene la ventaja de asemejarse a la escritura en papel, especialmente por la ausencia de elementos que generan ruido como los paréntesis innecesarios, tan frecuentes en los lenguajes de programación imperativos.

Sección Técnica

Como se comentó en la sección **Alcance del diseño**, los casos de uso deseados se definen en el archivo `Main`. Para crear una red neuronal y decidir sus características, es posible diseñar experimentos a partir del siguiente template:

```haskell
-- | Experiment example

experimentTemplate :: IO ()
experimentTemplate = do

  -- provision of dataset

  samples <- loadMatrix "datasets/iris_x.dat"  -- PARAM: change datasets/iris_x.dat for another dataset
  targets <- loadMatrix "datasets/iris_y.dat"  -- PARAM: change datasets/iris_y.dat for true values

  -- provision of initial net

  let inputFeatures = 4                        -- PARAM: quantity of columns of datasets/iris_x.dat
      layers = [128, 3]                        -- PARAM: quantity of neurons for each layer of your nn
      activations = [Relu, Sigmoid]            -- PARAM: activation function for each layer of your nn

  net <- buildNetwork inputFeatures layers activations

  -- provision of train parameters

  let learningRate = 0.001                     -- PARAM: learning rate for your optimizer
      optimizer = GradientDescent learningRate -- PARAM: optimizer for your nn
      errFunc = MSE                             -- PARAM: error function to minimize
      epochs = 7000                            -- PARAM: for how many epochs will the nn train

      trainedNet = train net errFunc (samples, targets) epochs optimizer

  print $ takeRows 10 (feedforward trainedNet samples)
```

En el código superior resalté las secciones correspondientes a la "*do-notation*". Esta notación nos permite escribir el código de forma similar a la imperativa. La mónada `IO` posee imlementadas las funciones `return`, `fail` y `bind` (además de muchas otras) y podemos apreciar en este fragmento de código la presencia implícita del bind, al final de cada sentencia que posee un `<-`. Por otro lado, la función `experimentTemplate`, al ser de tipo `IO ()`, indica que no me importa lo que retorna, sino sólo el efecto. Sería el equivalente a un `void` en el lenguaje C.

Se pueden agregar tantos experimentos como uno desee y luego ejecutarlos sucesivamente modificando la última línea del archivo, como puede verse en el fragmento de código debajo. Una vez que se realizaron los cambios deseados, correr el comando `./run.sh` desde la carpeta del proyecto (asegurarse de que el archivo `run.sh` tiene permisos de ejecución).

```haskell
main = experimentTemplate >> experimentTemplate2 >> experimentTemplate3
```

Cuando ejecutamos el programa, podemos obtener algo de la forma:

```
[5 of 5] Compiling Main             ( Main.hs, Main.o )
Linking Main ...
Expected output for first 10 samples
(10><3)
```

```
 [ 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0
 , 1.0, 0.0, 0.0 ]


Network output for first 10 samples
(10><3)
 [ 0.9339870625422344, 0.18098192605032998, 2.9922866023600666e-3
 , 0.8992142706120912, 0.22234009989679346,  3.779440626156692e-3
 , 0.9246578291576253,  0.1940803575608776,  3.428577168338593e-3
 , 0.9023126671847599,  0.2164378714522452,  4.571708449459737e-3
 , 0.9401873017971621, 0.17265523593073517,  3.003192358068613e-3
 , 0.9239630190930103, 0.17371803286621787, 3.5662212461139474e-3
 , 0.9238598676251748, 0.18471869732051857,  3.916643252111004e-3
 , 0.9230148529300087, 0.19463512437228053, 3.5245089549071247e-3
 , 0.8965180951396146, 0.22536580181475066,   4.69972225691125e-3
 , 0.9070599224207982, 0.21957544344798108,   3.90919083399222e-3 ]
```

Aquí podemos ver cómo luego del entrenamiento de la red, la salida de la misma se va pareciendo cada vez más al valor esperado (aumentan los valores de la primera columna y van descendiendo los de la segunda y tercera columna).

Módulos

Para implementar una red neuronal, es necesario tener en cuenta que las operaciones matriciales aparecerán constantemente. Para ello busqué una biblioteca de funciones similar a `numpy` pues, además, uno de los objetivos del trabajo era realizar una comparación con Python. `HMatrix` parece ser una buena opción, según algunos foros de discusión[11]. Luego, a partir de comparaciones de `HMatrix` contra otras bibliotecas que contaban con este tipo de funcionalidades, se pudo observar un buen desempeño en cuando a tiempo de ejecución (aunque requería más memoria que otras alternativas)[12]. Si bien me pareció una buena opción, se necesitaría mayor investigación para concluir si es la mejor. Hay ciertos aspectos que no se tuvieron en cuenta, como por ejemplo, la posibilidad de paralelización (para lo cual no hay evidencia de que `HMatrix` sea la biblioteca indicada para esto).

NeuralNetwork

En el módulo `NeuralNetwork` se encuentran los elementos principales que definen a una red neuronal. Haskell es un lenguaje que permite definir tipos de datos algebraicos rápidamente. Esto nos permite describir los elementos del problema que estamos tratando.

```haskell
-- | Neural Network data types

data Activation = Relu | Tanh | Sigmoid
data Error = MSE

type Weights = Matrix Double     -- layer weights
type Biases = Matrix Double      -- layer biases
type DW = Matrix Double          -- delta for weights
type DB = Matrix Double          -- delta for bias

data Layer = Layer Weights Biases Activation
data Gradients = Gradients DW DB

type NeuralNetwork = [Layer]
```

En el código superior podemos ver la forma de denotar los nuevos conjuntos que nos interesan. Por ejemplo, estamos denotando el conjunto de las funciones de activación para las redes neuronales con el tipo de dato `Activation`. Éste tiene distintos constructores que corresponden a las funciones que queremos incluir. También contamos con constructores con parámetros (los constructores también pueden ser funciones) como es el caso de `Layer` y `Gradients`.

El agregado de nuevos constructores que correspondan a funciones de activación o de error es bastante sencillo: deben incluirse en el tipo de datos y luego se deben dar sus definiciones correspondientes.

```haskell
-- | Functions and derivatives

apply :: Activation -> Matrix Double -> Matrix Double
apply Relu = relu
apply Tanh = tanh_
apply Sigmoid = sigmoid

applyDerivative :: Activation -> Matrix Double -> Matrix Double -> Matrix Double
applyDerivative Sigmoid = \x dY -> sigmoidGradient x * dY
applyDerivative Relu = \x dY -> reluGradient x * dY
applyDerivative Tanh = \x dY -> tanhGradient x * dY

compute :: Error -> Matrix Double -> Matrix Double -> Double
compute MSE = \x y -> sumElements $ 0.5 `scale` (cmap (^ 2) (x - y))

computeDerivative :: Error -> Matrix Double -> Matrix Double -> Matrix Double
computeDerivative MSE = \x y -> x - y
```

Tal como vimos a lo largo de la materia, los constructores no sólo se utilizan para describir ideas sino que también permiten “preguntar” a un elemento si fue construido con dicho constructor. En el código superior vemos resaltado un ejemplo de Pattern Matching, lo cual me permite analizar coincidencia de argumentos con el esquema. Además, permite que la definición de funciones se amolde a los distintos casos, como si tuviéramos un `if` o un `case` implícito pero que, a mi forma de ver, queda mucho más prolijo que definirlo dentro de una misma función, como deberíamos hacer en otros lenguajes.

En el módulo se encuentran, además de lo indicado, todas las funciones necesarias para realizar los algoritmos de backpropagation y feed forward pero que, dado que son centrales para el trabajo, se les dedica la siguiente sección.

Optimization

Este módulo es importante porque los problemas que estamos intentando resolver son problemas de optimización: tenemos una función y queremos hallar la combinación de parámetros para la cual esa función tiene el menor costo o error posible (según la definición de error que estemos tomando). Al momento de decidir qué algoritmo utilizaremos para nuestra red neuronal (y con qué parámetros) no existe una fórmula que nos indique, para la arquitectura en cuestión, la mejor forma de optimizarla.

En este módulo he decidido implementar dos algoritmos de optimización: Gradiente Descendente, pues es un algoritmo simple que fue ampliamente utilizado por mucho tiempo, y Adam (ADApative Moment stimation), ya que junto con RMSprop han demostrado ser útiles al entrenar una amplia gama de arquitecturas de redes neuronales.

```haskell
type LearningRate = Double
type Beta = Double
type Epsilon = Double

data AParameters = AParameters Beta Beta Epsilon Double
data Optimizer = GradientDescent LearningRate | Adam AParameters

train :: NeuralNetwork                -- network to train
   -> Error                            -- error function to use
   -> (Matrix Double, Matrix Double)   -- (samples, targets)
   -> Int                              -- epochs
   -> Optimizer                        -- optimizer to use
   -> NeuralNetwork                    -- resulting neural network

-- | Optimizers' train definitions

train net0 err dataset epochs (GradientDescent lr) = last $ take epochs (iterate step net0)
   where
       step net = zipWith (update lr) net gradients
         where
           (_, gradients) = backprop err net dataset

train net0 err dataset epochs (Adam params) = net
 where
```

```
s0 = initializeAtZero net0
v0 = initializeAtZero net0
(net, _, _) = _adam params err epochs (net0, s0, v0) dataset
```

En el código superior podemos ver nuevamente cómo los tipos de datos algebraicos nos permiten describir fácilmente el conjunto formado por los optimizadores. Para agregar uno nuevo, basta con definir sus parámetros si es necesario y luego agregar un nuevo constructor a `Optimizer`, con la correspondiente función de `train`. Por otra parte, recordemos que en Haskell los datos son inmutables. Cuando vimos árboles, por ejemplo, si teníamos un árbol y queríamos que cambie, construíamos otro árbol con algún valor cambiado. En este caso con la red neuronal estamos haciendo lo mismo en cada paso del entrenamiento.

Otro elemento a destacar es la aparición del patrón especial _. Éste me permite expresar de manera clara que el argumento no me importa y, desde el punto de vista ingenieril no reserva memoria para guardar nada. Notemos que al ser un pattern, nunca puede aparecer a la derecha de una ecuación, sino que debe estar siempre a la izquierda.

Por último, podemos ver la ventaja de las funciones de alto orden para usar la función `train`. La forma en la que se encuetra definida nos da mucha flexibilidad. Por ejemplo, si quisiéramos realizar el entrenamiento con un sólo optimizador, simplemente llamaríamos de la forma:

```
-- provision of dataset

samples <- loadMatrix "datasets/iris_x.dat"
targets <- loadMatrix "datasets/iris_y.dat"

-- provision of initial net

let inputFeatures = 4
    layers = [128, 3]
    activations = [Relu, Sigmoid]

net <- buildNetwork inputFeatures layers activations

-- provision of train parameters

let learningRate = 0.001
    optimizer = GradientDescent learningRate
    errFunc = MSE
    epochs = 7000

    trainedNet = train net errFunc (samples, targets) epochs optimizer

print $ takeRows 10 (feedforward trainedNet samples)
```

Si, por otro lado, quisiéramos experimentar con varios optimizadores y/o parámetros, pero manteniendo todo lo demás, podríamos utilizarla de la siguiente manera:

```
-- provision of dataset

samples <- loadMatrix "datasets/iris_x.dat"
targets <- loadMatrix "datasets/iris_y.dat"

-- provision of initial net

let inputFeatures = 4
    layers = [128, 3]
    activations = [Relu, Sigmoid]

net <- buildNetwork inputFeatures layers activations

-- provision of train parameters

let learningRate = 0.001
    gdOptimizer = GradientDescent learningRate -- gradient descent optimizer
    beta1 = 0.9
    beta2 = 0.999
    epsilon = 1e-8
    lr = 0.001
    adamParams = AParameters beta1 beta2 epsilon lr
    adamOptimizer = Adam adamParams            -- adam optimizer

    epochs = 800
    errFunc = MSE

    trainable = train net errFunc (samples, targets) epochs

    gradientNet = trainable gdOptimizer
    adamNet = trainable adamOptimizer

print $ takeRows 10 (feedforward gradientNet samples)
print $ takeRows 10 (feedforward adamNet samples)
```

Las funciones de alto orden son excelentes para manipular qué tipos de argumentos queremos dejar fijos y cuáles ir variando. Otro caso de uso, si quisiéramos entrenar con distintas épocas, podría ser:

```
trainable = flip (train net errFunc (samples, targets)) optimizer

trainedNet = trainable 800
overfittedNet = trainable 100000
```

Evaluation

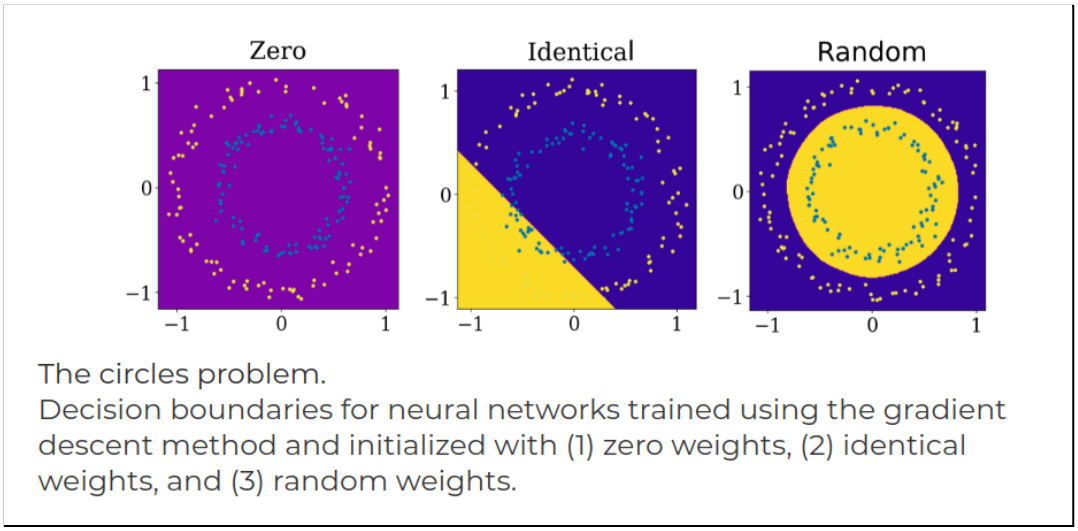
Este módulo busca dar información acerca de la capacidad de la red neuronal para la tarea objetivo. En este caso, decidí acotar el módulo a los problemas de clasificación binaria.

```haskell
-- | Transforms output layer with 1 neuron into binary classification

binaryClassify :: NeuralNetwork -> Matrix Double -> Matrix Double
binaryClassify net samples = cmap (\a -> if a < 0.5 then 0 else 1) (feedforward net
samples)

-- | Returns how many were hits and how many were errors

binaryHits :: NeuralNetwork -> (Matrix Double, Matrix Double) -> (Double, Double)
binaryHits net (samples, targets) = (n - errors, errors)
 where
   n = fromIntegral $ rows targets
   errors = sumElements $ abs (targets - (net `binaryClassify` samples))

-- | Returns accuracy for binary classification problems

binaryAccuracy :: NeuralNetwork -> (Matrix Double, Matrix Double) -> Double
binaryAccuracy net (samples, targets) = 100 * (1 - e / m)
 where
   predictions = net `binaryClassify` samples
   e = sumElements $ abs (targets - predictions)
   m = fromIntegral $ rows targets
```

Las tres funciones nos permitirán clasificar los resultados de la red neuronal, averiguar cuántos se clasificaron bien, cuántos mal y la precisión de la red como porcentaje. Notemos que en el código superior se encuentra resaltada la función `binaryClassify`. Esto es porque, al ser una función binaria, puede usarse como operador infijo. Esta posibilidad es otra prueba de la flexibilidad que provee Haskell. En este caso, decidí usarlo de esta forma porque quería que se asemeje a cómo uno lo diría en voz alta: la red clasifica los ejemplos.

Dynamic

El módulo `Dynamic` contiene funciones que tienen interacción con el medio, orientadas a la inicialización de una red neuronal, en particular a sus pesos y biases. ¿Por qué necesitamos interacción con el medio si, por ejemplo, podríamos inicializar todos los pesos en cero o en un valor idéntico?



The circles problem.
Decision boundaries for neural networks trained using the gradient descent method and initialized with (1) zero weights, (2) identical weights, and (3) random weights.

***Figura 5**: toma de decisión de la red neuronal en función de inicialización de pesos.*

Como podemos observer en ***Fig. 5***, la inicialización de pesos con valores pseudoaleatorios nos proporciona beneficios a la hora de entrenar nuestra red neuronal. Si bien esto puede parecer sencillo de llevar a la práctica, al momento de trasladar este deseo al mundo de la programación funcional, el concepto deja de serlo.

```
buildNetwork :: Int -> [Int] -> [Activation] -> IO NeuralNetwork
buildNetwork features neurons activations = do
   connections <- mapM _generateLayerConnections dimforlayers
   return (zipWith (\(w, b) a -> Layer w b a) connections activations)
   where
       dimforlayers = zip (features:neurons) neurons

_genValues :: (Int, Int) -> IO (Matrix Double)
_genValues (dimensionIn, dimensionOut) = do
   let k = sqrt (1.0 / fromIntegral dimensionIn)
   w <- randn dimensionIn dimensionOut
   return (k `scale` w)

_generateLayerConnections :: (Int, Int) -> IO (Weights, Biases)
_generateLayerConnections (inconnections, neurons) = do
   weights <- _genValues (inconnections, neurons)
   biases <- _genValues (1, neurons)
   return (weights, biases)
```

¿Por qué estas funciones se encuentran dentro de la mónada `IO`? Notemos que, si yo llamo sucesivas veces a la función `_genValues` con los mismos parámetros (y la misma no fuera `IO`), perderíamos la transparencia referencial, a causa de la generación de números pseudoaleatorios. Por ende, tanto `_generateLayerConnections` y `buildNetwork` también lo harían.

Esta situación es similar al ejemplo de `leerChar` que se dió durante la cursada. En aquel caso, el buffer del cual se iban consumiendo los caracteres iba cambiando. La mónada IO se encarga de propagrar de forma apropiada los cambios en el mundo físico. Si en el ejemplo, tomaba la función `getChar` (que leía el caracter y lo consumía) y la transformaba en una función “unsafe” (fuera de la mónada) provocaba la pérdida de las propiedades que me otorga el mundo funcional. En este caso, la función `randn` ocupa el lugar de `getChar`, pues tenemos un estado inicial (la semilla) y los números pseudoaleatorios que nos devuelve van cambiando cada vez que llamo la función.

Algoritmo de Retropropagación

Como ya fue discutido en la introducción, el algoritmo de retropropagación fue problemático al momento de introducirnos en la temática de redes neuronales. Investigando algunos repositorios y revisando la teoría sobre dicho algoritmo me encontré con una opción que me llamó mucho la atención no haber considerado antes: **recursión**.

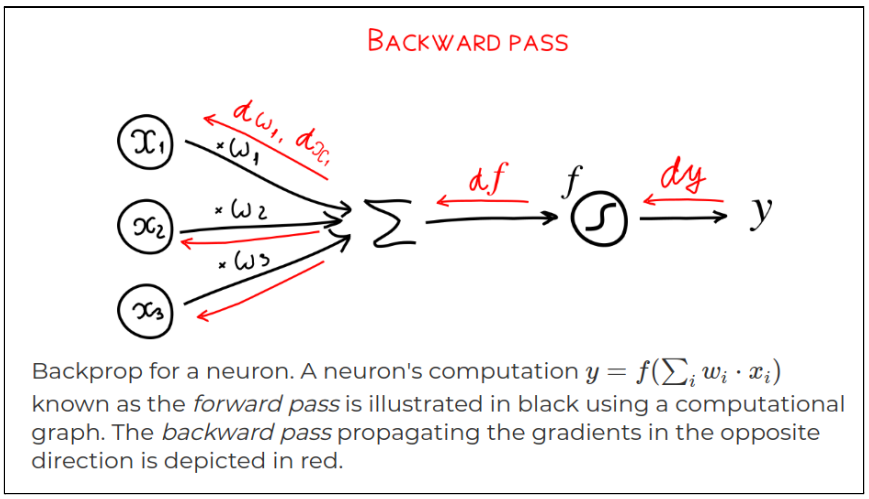


***Figura 6**: esquema del algoritmo de retropropagación para una neurona.*

Como podemos ver en ***Fig. 6***, recursión parece algo "obvio" para la implementación del algoritmo de backpropagaion y, sin embargo, a lo largo de un año de tratar con estos temas, nunca se me había ocurrido verlo de esa forma. Me sorprendió que no era la única que había pasado por alto ese detalle, como puede verse en ***Fig. 7***.

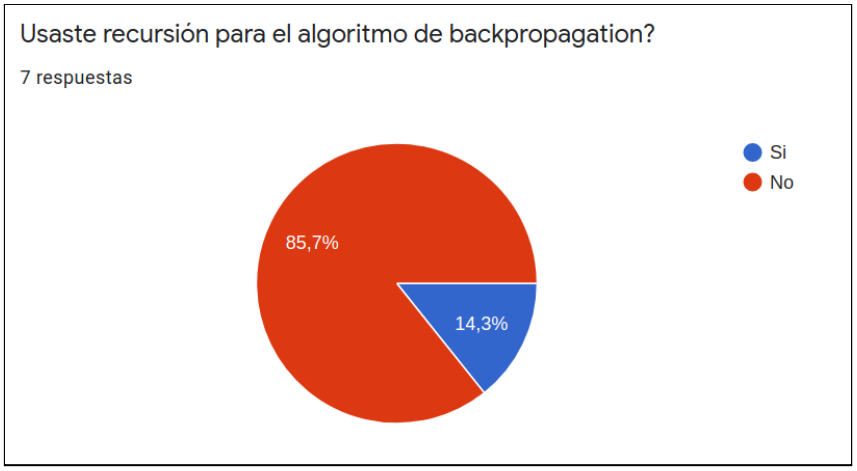


***Figura 7**: resultados sobre la utilización de recursión en el algoritmo de backpropagation.*

Fue durante el desarrollo de este algoritmo, que las funciones de alto orden fueron de gran utilidad, especialmente para definir backpropagation y feed forward a partir de una única función:

```haskell
-- | Backpropagation and Feed Forward algorithms

propagate ::
 Error ->                                  -- error/cost function
 Matrix Double ->                          -- targets
 Matrix Double ->                          -- inputs
 NeuralNetwork ->                          -- layers
 (Matrix Double, Matrix Double, [Gradients]) -- (dX, predictions, list of gradients)

propagate err t inp [] = computeLoss err t inp
propagate err t inp (Layer w b a:xs) = (dX, preds, gradient:gradients)
```

```
  where
    h = getH inp w b
    (dZ, preds, gradients) = propagate err t (apply a h) xs
    (dX, gradient) = computeGradients inp h dZ (Layer w b a)

backprop ::
  Error ->                            -- error function
  NeuralNetwork ->                    -- nn
  (Matrix Double, Matrix Double) ->   -- (samples, targets)
  (Matrix Double, [Gradients])        -- (predictions, list of gradients)

backprop err net (samples, targets) = dropFirst $ propagate err targets samples net

feedforward ::
  NeuralNetwork ->      -- nn
  Matrix Double ->      -- samples
  Matrix Double         -- predictions

-- we use undefined for both targets and error function (we won't need them)
feedforward net samples = fst $ backprop undefined net (samples, undefined)
```

Si bien hay funciones auxiliares que acompañan el método `propagate`, podemos ver claramente en los tres renglones resaltados que el cálculo del gradiente para una capa en particular dependerá de los inputs que llegan a la capa en cuestión (primer línea resaltada), de la derivada de la función de activación de la próxima capa (segunda línea resaltada) y de la excitación y características de la capa en cuestión, como pesos, biases y función de activación (última línea resaltada).

```
computeGradients ::
  Matrix Double ->                  -- inputs
  Matrix Double ->                  -- excitation of layer
  Matrix Double ->                  -- dZ from next layer
  Layer ->                          -- current layer
  (Matrix Double, Gradients)
computeGradients inp h dZ (Layer w b a) = (linearX' w dY, Gradients dW dB)
  where
    dY = applyDerivative a h dZ
    (dW, dB) = (linearW' inp dY, bias' dY)
```

Si analizamos un poco más en detalle el momento en el cual se calculan las actualizaciones dW y dB para la capa en cuestión, podemos ver en ***Fig. 7*** cómo describimos con funciones el flujo de las flechas que vimos en la imagen ***Fig. 6***.

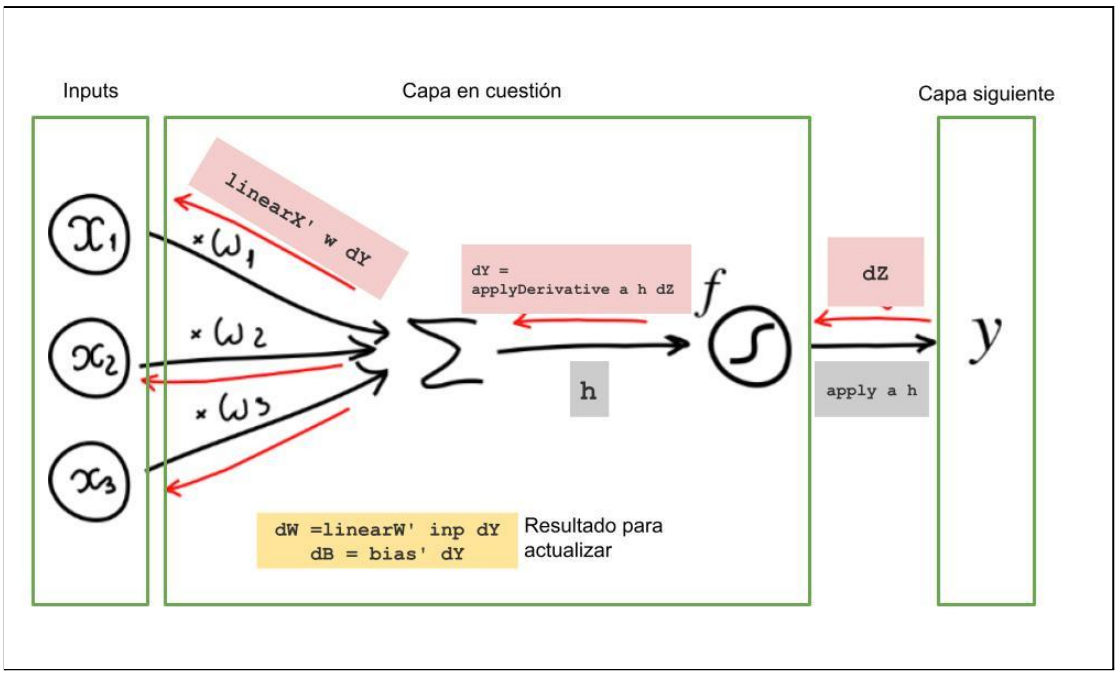


***Figura 7**: flujo de la información dentro de la red neuronal en términos de las funciones del programa en Haskell.*

Por otra parte, notemos que la función `feedForward` se define a partir de backpropagation con la salvedad de que no estamos pasando targets, sino que usamos el valor undefined (bottom).

```
feedforward net samples = fst $ backprop undefined net (samples, undefined)
```

La evaluación LAZY de Haskell nos permite definir la función de esta forma para evitar dos cosas en simultáneo: repetición de código y el cómputo innecesario de gradientes cuando sólo necesitamos las predicciones.

Por último, quería descatar que analicé la posibilidad de usar `foldr` para definir la recursión sobre la red neuronal. Siendo ésta una lista de capas, me pareció una buena posibilidad aprovechar esta función. Algo que buscaba explotar es esta idea de que en otros lenguajes tenemos patrones de diseño que funcionan a modo de recetas para hacer código, mientras que en funcional tenemos “código que hace código”. Luego de plantear algunos esquemas sobre papel, me pareció que la función propagate tenía demasiada información para movilizar y decidí mantener el planteo de la recursión de la forma en que se presenta en el trabajo.

Metodología de comparación

Uno de los objetivos descriptos en la introducción era analizar si Haskell tenía ventajas con respecto a la performance de la red neuronal sobre Python. Para tomar las métricas utilicé mi implementación de Python contra la nueva implementación de Haskell.

Es importante destacar que la comparación tiene sus fallas: en primer lugar la implementación de Python NO utiliza recursión, con lo cual para cada iteración se debe hacer el feed forward y con el resultado aplicar el algoritmo de retropropagación. En

segundo lugar, la implementación hecha en Python no tenía como objetivo ser eficiente. Sí se usa la biblioteca de funciones `numpy` para operaciones matriciales, pero sigue siendo un algoritmo que puede mejorarse muchísimo. Por cuestiones de tiempo y dado que el foco del trabajo se encuentra en la implementación de Haskell, decidí no mejorar el algoritmo de Python.

Para el análisis se mantendrá constante el optimizador (gradiente descendente con learning rate de 0.001), el conjunto datos (300 datos con 2 atributos, cuyo valor a predecir es una clasificación binaria) y la función de error con la cual buscaremos optimizar la red (cuadrados mínimos). En cuanto a las variaciones tenemos:

- Arquitectura con 1 capa oculta de 5 neuronas y salida de 1 neurona. Entrenamiento durante 1000 épocas.
- Arquitectura con 3 capas ocultas de 5, 10 y 15 neuronas respectivamente. Entrenamiento durante 1000 épocas.
- Arquitectura con 7 capas ocultas de 16, 32, 64, 128, 64, 32 y 16 neuronas respectivamente. Entrenamiento durante 1000 épocas.
- Arquitectura con 3 capas ocultas de 5, 10 y 15 neuronas respectivamente. Entrenamiento durante 5000 épocas.

Haremos 3 corridas para cada configuración, tanto en Haskell como en Python. Para el cómputo de tiempo que pasó desde el entrenamiento utilizamos la función `getTime` de `System.Clock`.

```
start <- getTime Monotonic
end <- getTime Monotonic
```

El valor `Monotonic` nos provee una métrica de tiempo que, si bien no es absoluta, asegura que no se actualizará el reloj luego de haber iniciado el programa. Esto puede evitar problemas como medir tiempo durante la actualización del sistema de la computadora, lo cual puede provocar actualizaciones en el tiempo y, por ende, dar medidas inexactas[9].

Una característica muy importante a tener en cuenta de Haskell es que es un lenguaje LAZY, por lo que puede ocurrir que ciertas partes del código no se ejecuten. Esta problemática me la encontré inicialmente para medir cuánto tardaba en entrenar la red. Si no utilizaba la red luego del entrenamiento dentro de la mónada `IO`, el `start` y `end` no se correspondían con lo que quería medir.

```
start <- getTime Monotonic

let trainedNet0 = train net0 errFunction (samples, targets) epochs optimizer
print $ feedforward trainedNet0 (takeRows 1 samples)

end <- getTime Monotonic
```

En el código superior vemos que se entrena la red inicial (definida según los casos detallados más arriba). Por otra parte, el código en Python ejecutado es el siguiente:

```
start_time = time.time()
```

```
mlp.train_weights(training_data, training_data_expected)

end_time = time.time()
```

Como Python no es un lenguaje LAZY, no hay necesidad de posterior utilización de la red entrenada. El código se va a ejecutar. En el apéndice, en la sección Medición de tiempo, se encuentra detallada la definición de variables para el caso de una capa oculta.

Como conclusión, la metodología diseñada tiene sus fallas y no busca ser concluyente sobre la ventaja de las operaciones de un lenguaje por sobre el otro, pero sí aportar evidencia frente a la sospecha de que utilizar Haskell con un esquema recursivo será mejor en términos de performance frente a la implementación realizada en Python. Si esto no ocurriera, sería necesario hacer una revisión completa de los algoritmos (algo raro habría).

Resultados

A partir de lo descripto anteriormente en Metodología de Comparación, se obtuvo la siguiente tabla (los datos completos se encuentran en la sección Tabla completa de resultados, del apéndice):

language	hidden layers	epochs	mean (s)	std (s)
python	1	1000	27,4698	3,2929
haskell	1	1000	0,1656	0,0090
python	3	1000	47,0787	1,5632
haskell	3	1000	0,3827	0,0031
python	7	1000	108,2549	1,8230
haskell	7	1000	14,3367	1,3399
python	3	5000	249,9192	6,3263
haskell	3	5000	2,1567	0,0153

Los resultados aquí indicados nos muestran como, para cada combinación de arquitectura y épocas del algoritmo, el programa implementado en Haskell muestra evidente superioridad en performance respecto del anterior.

Mejoras Futuras

En esta sección me gustaría contar, en orden de prioridad, ciertos aspectos que me gustaría tratar como expansión del trabajo. En primer lugar, el diseño de una metodología más objetiva para comparar funciones en dos lenguajes de programación. Esto requeriría llevar a cabo una pequeña investigación sobre buenos formatos para llevarlo a cabo, acompañado de una implementación en Python que use recursión. Por otro lado, la forma de tomar datos sobre el desempeño de los lenguajes es bastante simple y puede dar espacio a errores. Quizás sería buena idea investigar sobre paquetes en Haskell que estén preparados para hacer *profiling* sobre un programa lo cual me permitiría entender cómo consume recursos.

Una alternativa podría ser investigando el paquete Criterion[7] ya que parece preparado para realizar benchmarking en Haskell. Por otra parte, debería analizarse un paquete equivalente en caso de buscar hacer comparaciones en Python (o desarrollar el código necesario para poder tomar las mismas métricas).

En segundo lugar, me parece importante tener un módulo `Evaluation` que sea genérico a clasificadores y regresiones, especialmente porque la red tiene la capacidad de ajustar problemas de regresión pero, por el momento, no es posible aprovechar esta funcionalidad si uno quiere evaluar cómo se desempeñó la red.

En tercer lugar, creo que sería interesante ampliar el abanico de opciones en lo que a tipos de capas de la red neuronal refiere. Se podrían agregar capas convoluciones, softmax, pooling, etc. Esta variedad permitiría tratar problemas más interesantes, especialmente aquellos que tratan con imágenes. Si bien estas capas también se actualizan con backpropagation son un poco más complejas de implementar, en especial las capas convolucionales.

Por último, agregaría un archivo de configuración o interfaz para que el usuario pueda manejarse más fácilmente con la aplicación y sus funcionalidades (en particular, no tener que volver a compilar el programa). Si bien este ítem podría ser más prioritario, el foco principal del trabajo se encuentra en analizar qué aspectos resultan interesantes y sencillos a la hora de implementar redes neuronales en Haskell, con lo cual se espera que el usuario sea alguien que tiene conocimientos de programación y se sienta cómodo en caso de tener que editar un archivo de código.

Conclusiones

Respecto del objetivo principal acerca de comprender el algoritmo de retropropagación, no tengo más información al respecto que la que puede proveerme una valoración personal. Si bien no fue “mágico” (programé en Haskell y *ualá*, entendí todos los aspectos que me costaban), sí me pasó que tener la implementación en funcional me permitió ver con más claridad cómo se trasladaba el flujo de información y cómo se representaban los dibujos teóricos cuando se los “bajaba a tierra”. Revisé tres implementaciones del algoritmo de backpropagation en Python (mi implementación y las de otros dos trabajos) y la sintaxis de Haskell resulta mucho más limpia. Creo que vale la pena hacerlo de esta forma y, en el futuro, me gustaría seguir armando proyectos en este lenguaje.

A lo largo de la realización del trabajo, me fui encontrando con varios aspectos que me gustaron mucho de la programación funcional, a pesar de que sentí bastante resistencia al principio programando en un lenguaje de este estilo. Me sucedía a veces que quería hacer las cosas de forma imperativa y me costaba pensarlo desde otra perspectiva. No faltó el momento en el que quise agregar un “*for loop*” pero, para mi sorpresa, el programa completo pudo realizarse sin usar ninguno lo cual, en Python, parece ser el default al cual uno acude.

Una característica de Haskell que me resultó muy cómoda al momento de programar es su sistema de verificación de expresiones (a través de reglas sintácticas y reglas de asignación

de tipos). Dado que los programas pueden fácilmente tardar horas en correr, en Python puede ocurrir que uno sin darse cuenta haya modificado el archivo, provocando un error de sintaxis, lo que ocasiona que el programa falle. Si uno tiene mala suerte, esto puede ocurrir luego de que haya finalizado todo el entrenamiento de una red neuronal.

Por otra parte, me sorprendió que Haskell es muy rápido. Si bien mencioné ciertos problemas con la objetividad de las comparaciones entre Haskell y Python, he probado arquitecturas grandes con muchísimas épocas en Haskell y me sorprendió lo rápido que funcionaban en relación a lo que mi memoria (no necesariamente la fuente más confiable) recordaba sobre las experimentaciones que llevé a cabo en Python. En la materia de *Sistemas de Inteligencia Aritificial* hay un trabajo práctico donde el principal problema es el factor tiempo. Las arquitecturas deben entrenarse por mucho más que 5000 épocas (como fue el caso de las comparaciones) y comienza a suceder que el programa se vuelve una limitación. Cuando esto se traslada a redes neuronales con millones de parámetros que tardan días en entrenar, optimizaciones en la programación pueden traer enormes beneficios.

Lo que más me gustó sobre el lenguaje fue lo mucho que se asemejaba a escribir en una hoja (lo cual iba haciendo en paralelo para analizar ciertas funciones). En la sección **Alcance del diseño** mencioné lo refrescante que resulta la ausencia de paréntesis, pero me gustaría resaltarlo porque, sin darse cuenta, uno se acostumbra rápidamente a esta característica. Pareciera ser más natural porque es como escribir. Esto, combinado con la enorme flexibilidad de las funciones de alto orden, fue el aspecto que más disfruté del lenguaje. Me dio la sensación de poder programar de manera modular velozmente, incluso teniendo en cuenta que probablemente haya muchos aspectos de mi programa que pueden mejorarse inclusive más.

Debo reconocer que al principio me encontré bastante a ciegas por la ausencia de tener, al alcance de la mano, funciones como `print` y `console log`. No es que no existan en Haskell, pero cuando las funciones son puramente funcionales, querer imprimir información en de la misma no es sencillo (y en muchos casos, no debería hacerse). Me sentí bastante identificada con el chiste de “pedir una canillita” en una habitación. Sólo quería algo chiquito, agregar unos números random, imprimir como venía mi programa… pero si no quería empezar a destrozar mis funciones, necesitaba pensar mejor las cañerías. Todavía no estoy del todo convencida sobre la utilidad del lenguaje funcional en aplicaciones que se utilizan en el día a día. El proyecto que hice fue pequeño, con lo cual no estoy segura de cómo se trasladaría esto a un ambiente comercial/empresarial.

Por último, me pareció especialmente interesante volver a programar en un lenguaje que no conocía o que conocía muy poco, ya que hace bastante tiempo que uso siempre los mismos. Fue beneficioso haberle dedicado mucho tiempo al lenguaje durante la cursada, porque cuando analizaba foros o veía código de otras personas, podía reconocer varios de los elementos del lenguaje. Esto me dio la pauta de que, si bien usaba Python muy seguido, en general no tenía idea de los conceptos teóricos que operaban detrás del lenguaje. Entonces… ¿estaba programando en lenguajes conocidos? ¿o estaba programando por costumbre, un poco a ciegas?

***Figura 8**: mi sensación cada vez que leía respuestas en stack overflow sobre Haskell.*

A modo de chiste (no tan chiste), me quedó la sensación de que suelo dar martillazos todo el día cuando se trata de programar. Cuando buscaba alguna pista para un problema que estaba teniendo, me di cuenta que estaba (estoy) acostumbrada a soluciones rápidas. Creo que esto se incrementó con la virtualidad, no sólo en relación a programar sino a muchas cosas. Cuesta analizar las cosas tan a fondo porque “tenés todo a mano”. Todo lo podés googlear, todo lo podés buscar en el momento. Pero esto provoca que, al final del día, no tengas nada realmente asentado. Dado que es una de las últimas materias de mi carrera universitaria, el trabajo me pareció una buena oportunidad para, no sólo aprender conceptos nuevos, sino reflexionar sobre la forma que uno tiene de ser al momento de dedicarse a aquello para lo cual se estuvo formando durante años.

Bibliografía

[1] Python lenguaje más utilizado:
https://towardsdatascience.com/top-programming-languages-for-ai-engineers-in-2020-33a9f16a80b0
[2] Lenguaje funcional para deep learning:
https://www.welcometothejungle.com/es/articles/btc-deep-learning-clojure-haskell
https://towardsdatascience.com/functional-programming-for-deep-learning-bc7b80e347e9
[3] Gradiente Descendente y Retropropagación:
https://penkovsky.com/neural-networks/day1/
[4] Perceptrones Multicapa: https://penkovsky.com/neural-networks/day2/
[5] Transparencia Referencial: https://es.wikipedia.org/wiki/Transparencia_referencial
[6] Clases y diapositivas de la clases del curso *72.60 Programación Funcional*
[7] Criterion:
https://hackage.haskell.org/package/criterion-measurement-0.1.2.0/docs/Criterion-Measurement.html
[8] Criterion Tutorial: http://www.serpentine.com/criterion/fibber.html
[9] Medición de tiempo: https://chrisdone.com/posts/measuring-duration-in-haskell/
[10] Perceptrón Simple: https://es.wikipedia.org/wiki/Perceptr%C3%B3n
[11] HMatrix:
https://www.reddit.com/r/haskell/comments/hjtttd/what_libraries_do_people_use_for_matrix/
https://www.quora.com/Are-there-Haskell-versions-of-popular-Python-libraries-such-as-NumPy-SymPy-SciPy-et-cetera
[12] HMatrix Benchmark: https://github.com/Magalame/fastest-matrices

Apéndice

Medición de tiempo

Código en Haskell para arquitectura con 1 capa oculta de 5 neuronas y salida de 1 neurona (entrenamiento durante 1000 épocas).

```
net0 <- buildNetwork 2 [5, 1] [Relu, Sigmoid]

let optimizer = GradientDescent 0.001
    errFunction = MSE
    epochs = 1000

start <- getTime Monotonic

let trainedNet0 = train net0 errFunction (samples, targets) epochs optimizer
print $ feedforward trainedNet0 (takeRows 1 samples)

end <- getTime Monotonic

fprint (timeSpecs) start end
```

Código en Python para arquitectura con 1 capa oculta de 5 neuronas y salida de 1 neurona (entrenamiento durante 1000 épocas).

```
mlp = Multilayer_perceptron(
    [5, 1],
    attributes_qty,
    init_lr=0.001,
    lr_decay='None',
    max_epochs=1000
)

start_time = time.time()

mlp.train_weights(training_data, training_data_expected)

end_time = time.time()
```

Tabla completa de resultados

language	hidden layers	epochs	time(secs)
python	1	1000	24,7555828
python	1	1000	31,1329694
python	1	1000	26,5207605
python	3	1000	48,8836303
python	3	1000	46,1588938
python	3	1000	46,1936080

python	7	1000	108,7633395
python	7	1000	106,2316575
python	7	1000	109,7696168
python	3	5000	253,6555007
python	3	5000	253,4873114
python	3	5000	242,6148605
haskell	1	1000	0,1603200
haskell	1	1000	0,1759300
haskell	1	1000	0,1604500
haskell	3	1000	0,3843000
haskell	3	1000	0,3790800
haskell	3	1000	0,3846700
haskell	7	1000	12,8200000
haskell	7	1000	14,8300000
haskell	7	1000	15,3600000
haskell	3	5000	2,1400000
haskell	3	5000	2,1600000
haskell	3	5000	2,1700000