DeepLearning.scala

Luca Ragazzi - Mat. 0000897452

September 12, 2020

Contents

List of Figures

Listings

Chapter 1

Introduzione

Le Deep Neural Networks (DNN) hanno contribuito in modo significativo allo sviluppo del Machine Learning (ML), facendo avanzare lo stato dell'arte in diversi thread di ricerca tra cui Computer Vision, Natural Language Processing e Transfer Learning.

Il cuore di una DNN è il concetto stesso di rete neurale artificiale. Essa è un modello di calcolo che trasforma l'input in output mediante l'applicazione ripetuta di operazioni matematiche interne alla rete. Una DNN è semplicemente una rete neurale con più strati intermedi, detti hidden layers, tra il layer di input e quello di output. Caratteristica di queste tipologie di reti è l'ampia quantità di trasformazioni matematiche, poiché sono coinvolti più nodi.

Il Deep Learning (DL) è quel campo di ricerca del ML che ha l'obiettivo di trovare i pesi migliori per una rete neurale, per trovare la corretta funzione matematica che trasformi l'input in output, sia che si tratti di una relazione lineare che di una relazione polinomiale. Lo scopo è minimizzare una funzione di errore, la cosiddetta funzione di loss. I pesi giusti vengono trovati utilizzando la discesa del gradiente (stochastic gradient descent), in cui si calcola la derivata della funzione di loss rispetto ai pesi della rete, aggiornandoli di conseguenza. Tale meccanismo è conosciuto come backpropagation, ed è il concetto su cui si basa l'addestramento di una rete neurale.

1.1 Scelta del framework

Diversi sono i framework di ML/DL, ognuno con le proprie caratteristiche. Per capire al meglio le motivazioni alla base della scelta del framework DeepLearning.scala, si elencano i pro dei migliori framework degli ecosis-

temi di Big Data e di Deep Learning.

Le caratteristiche peculiari di alcuni framework di Big Data, come Hadoop, Spark, Akka, sono:

- **Tipizzazione statica**: politica di assegnazione dei tipi alle variabili. Nei linguaggi a tipizzazione statica, il tipo di ogni variabile viene stabilito direttamente nel codice sorgente. Un tentativo di assegnazione di un valore di tipo diverso da quello di una variabile causa un errore terminale. Un esempio di linguaggio a tipizzazione dinamica è Python;
- **Programmazione funzionale**: paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Il principale punto di forza di questo paradigma è la mancanza di side-effect delle funzioni, il che comporta una verifica più semplice della correttezza e della presenza di eventuali bug nel programma.

Le caratteristiche dei migliori framework di Deep Learning, come TensorFlow e PyTorch, sono:

- **DSL (Domain Specific Language) interno**: linguaggio creato per un dominio specifico, consentendo agli utenti di creare funzioni differenziabili da espressioni simili alle ordinarie funzioni non differenziabili. Un DSL esterno, invece, ha la propria sintassi e non è costruito sopra un linguaggio. Tutto ciò che serve per far funzionare un DSL esterno è un parser che interpreti la lingua o che la traduca in un'altra;
- **Reti neurali dinamiche**: l'output delle reti non dipende solo dall'input corrente, come nelle reti feed-forward, ma anche dagli input, output e stati precedenti della rete;
- **Molteplici variabili addestrabili**: dal momento che le reti sono composte da più livelli, ciascuno contenente le proprie variabili addestrabili, è fondamentale poter calcolare le derivati di tutte le variabili addestrabili contemporaneamente per un determinato batch di dati di addestramento;
- **Differenziazione automatica (AD)**: insieme di tecniche per il calcolo automatico delle derivate di una funzione matematica, che permette di rendere completamente automatico, ed efficiente, il meccanismo di backpropagation nell'uso di reti neurali.

DeepLearning.scala è l'unica libreria per Scala con tutte le caratteristiche sopra elencate, e per questo motivo è stato scelto come framework di Deep Learning da analizzare in questo progetto.

Chapter 2

Framework

DeepLearning.scala [1] è una libreria di DL per Scala che, combinando costrutti di OOP (programmazione orientata agli oggetti) e di FP (programmazione funzionale), mira a creare reti neurali dinamiche con tipizzazione statica.

Le sue principali caratteristiche sono:

- **Programmazione differenziabile**: la libreria consente di creare reti neurali da semplici formule matematiche, calcolando direttamente in esse le derivate dei pesi della rete. A tal proposito, supporta diversi tipi di dati, come float, double e array n-dimensionali;
- **Reti neurali dinamiche**: tutte le funzionalità di Scala, incluse funzioni, espressioni e flussi di controllo, sono disponibili all'interno delle reti neurali, che risultano essere programmi a tutti gli effetti;
- **Deep learning monadico**: le reti neurali sono monadi, che possono essere create componendo funzioni higher-order. Per questo motivo l'apprendimento può essere definito monadico;
- **Plugins**: la libreria supporta diversi plugin per fornire alle reti neurali algoritmi, modelli, iperparametri o altre funzionalità.

2.1 Programmazione differenziabile

La libreria offre diversi concetti per una programmazione differenziabile:

- **Variabile addestrabile**: peso scalare o vettoriale all'interno di un modello, da inizializzare prima della sua esecuzione. Il suo tipo deve essere un sottotipo di *Weight*, tra cui:
  - **DoubleWeight**: scalare a singola precisione;

  - **FloatWeight**: scalare a doppia precisione;
  - **INDArrayWeight**: vettore.

- **Espressione differenziabile**: espressione componibile che rappresenta uno strato di una rete, il cui tipo deve essere un sottotipo di *Layer*, tra cui *DoubleLayer*, *FloatLayer* e *INDArrayLayer*. Dopo aver creato un'espressione differenziabile, è possibile eseguire un forward pass per creare grafici computazionali differenziabili;
- **Funzione differenziabile**: funzione Scala che ritorna un'espressione differenziabile. Può rappresentare una funzione di loss, una rete neurale, o un sottoinsieme di una rete neurale, come ad esempio un blocco denso. Questa é una funzione polimorfica che accetta tipi di parametri eterogenei, tra cui un input vettoriale (*INDArray*), una variabile addestrabile (*Weight*) e un'espressione differenziabile (*Layer*).

2.1.1 Meccanismo di backpropagation

Nella backpropagation si calcola la derivata del primo ordine della funzione di loss rispetto ai pesi della rete, e si aggiornano i pesi di conseguenza. Essa, combinata con altri algoritmi di ottimizzazione, modifica i valori dei pesi nelle reti neurali durante l'addestramento, producendo un modello di conoscenza appresa dai dati di input. La differenziazione automatica (AD) sfrutta il fatto che l'implementazione di una funzione, indipendentemente da quanto sia complessa, si riduce all'esecuzione di una serie di operazioni aritmetiche e funzioni elementari. Applicando in modo ripetuto la *chain rule* a tali operazioni, la derivata di una funzione può essere calcolata automaticamente.

In questa sezione si descrive la struttura dati interna utilizzata dalla libreria per eseguire l'AD in modalità inversa, cioè il meccanismo di backpropagation. Per spiegarne il funzionamento si riporta l'esempio tratto dal paper di DeepLearning.scala [2].

L'approccio scelto utilizza una struttura dati simile alla tradizionale AD in modalità forward. La forward AD, infatti, può essere considerata come una computazione su un *DualNumber*, come nel seguente esempio:

```
1 type Data = Double
2 /* Derivata parziale */
3 type PartialDelta = Double
4 case class DualNumber(data: Data, delta: PartialDelta)
```

Listing 2.1: DualNumber per forward AD.

Le operazioni aritmetiche di addizione e moltiplicazione possono essere implementate sui *DualNumber* nel seguente modo:

```
object DualNumber {
    /* Operazione di addizione */
    def plus(left: DualNumber, right: DualNumber): DualNumber = {
        DualNumber(left.data + right.data, left.delta + right.delta)
    }
    /* Operazione di moltiplicazione */
    def multiply(left: DualNumber, right: DualNumber): DualNumber = {
        DualNumber(left.data * right.data, left.data * right.delta + right.data * left.delta)
    }
}
```

Listing 2.2: Operazioni aritmetiche sui DualNumber.

Tuttavia, è difficile utilizzare questo approccio se si desidera supportare più variabili addestrabili, sconosciute prima del runtime. Dal momento che *PartialDelta* rappresenta la derivata parziale delle variabili addestrabili, se si supporta solo una variabile addestrabile, essa deve essere per forza l'input.

In DeepLearning.scala, invece, deve essere possibile addestrare più variabili, per cui il tipo di delta di uno specifico *DualNumber* deve contenere le derivate per tutte le variabili addestrabili che sono state utilizzate per produrre quel determinato *DualNumber*, non solo la derivata parziale dell'input. Di conseguenza, il tipo di delta varia quando il numero di variabili addestrabili aumenta. A tale scopo, per controllare il delta, gli autori hanno scelto di sostituire *PartialDelta* con *UpdateWeights*, come nell'esempio seguente:

```
type Data = Double
case class ClosureBasedDualNumber(data: Data, backward: UpdateWeights)
```

Listing 2.3: Sostituzione di PartialData per reverse AD.

A differenza di *PartialDelta*, *UpdateWeights* non è un numero che supporta operazioni aritmetiche native, quindi bisogna sostituire le operazioni aritmetiche native su *PartialDelta* con alcune funzioni statiche su *UpdateWeights*, quando si implementano le operazioni aritmetiche per il nuovo *DualNumber*, come mostrato in figura:

```
object ClosureBasedDualNumber {
    /* Operazione di addizione */
    def plus(left: ClosureBasedDualNumber, right: ClosureBasedDualNumber): ClosureBasedDualNumber = {
        ClosureBasedDualNumber(left.data + right.data, UpdateWeights.plus(left.backward(), right.backward()))
    }
```

```
    /* Operazione di moltiplicazione */
    def multiply(left: ClosureBasedDualNumber, right:
    ClosureBasedDualNumber): ClosureBasedDualNumber = {
        ClosureBasedDualNumber(left.data * right.data,
    UpdateWeights.multiply(left.data, right.backward()) +
    UpdateWeights.multiply(right.data, left.backward()))
    }
}
```

Listing 2.4: Sostituzione delle operazioni su PartialDelta con funzioni custom per UpdateWeights.

Dal momento che il tipo *UpdateWeights* in un *DualNumber* deve supportare sia operazioni di addizione che di moltiplicazione scalare, per implementarlo, gli autori lo hanno reso un tipo di funzione che contiene side-effect per aggiornare le variabili addestrabili. A livello matematico, l'addizione può essere definita come:

$$(f0 + f1)(x) = f0(x) + f1(x)$$

Mentre la moltiplicazione:

$$(x0f)(x1) = f(x0x1)$$

A livello di codice, la precedente definizione di operazioni aritmetiche può essere implementata in tipi di dati monadici, come nella figura seguente:

```
type SideEffects = UnitContinuation[Unit]
type UpdateWeights = Do[Double] => SideEffects
object UpdateWeights {
    /* (f0 + f1)(x) = f0(x) + f1(x) */
    def plus(f0: UpdateWeights, f1: UpdateWeights) = { doX: Do[
    Double] =>
        /* |+| rappresenta l'operazione di aggiunta di un
    qualsiasi tipo di dati cumulativo (Semigroup) */
        f0(doX) |+| f1(doX)
    }
    /* (x0f)(x1) = f(x0x1) */
    def multiply(x0: Double, f: UpdateWeights) = { doX1: Do[
    Double] =>
        f(doX1.map(x0 * _))
    }
}
```

Listing 2.5: Operazioni aritmetiche che contengono side-effect.

Bisogna notare che il parametro è un tipo di dati monadico *Do*, che incapsula il calcolo della derivata, ed è un'operazione valutata solo quando necessario.

In DeepLearning.scala, *SideEffects* si basa sul tipo di operazione asincrona *UnitContinuation*. Esso viene utilizzato come tipo di dati monadico per incapsulare i side-effect nella programmazione asincrona.

Una variabile addestrabile può essere rappresentata nel seguente modo:

```
/* Learning rate fisso (statico) per semplificare l'algoritmo
    di ottimizzazione */
def createTrainableVariable(initialValue: Double, learningRate:
     Double): ClosureBasedDualNumber = {
    var data = initialValue
    val backward: UpdateWeights = { doDelta: Do[Double] =>
        val sideEffects: Do[Unit] = doDelta.map { delta =>
            data -= learningRate * delta
    }
    convertDoToContinuation(sideEffects)
  }
  ClosureBasedDualNumber(data, backward)
}
```

Listing 2.6: Creazione di un DualNumber per una variabile addestrabile.

Il *DualNumber* che è stato effettivamente implementato in DeepLearning.scala è un *Tape*, una versione generica di *ClosureBasedDualNumber*, avente i parametri *Data* e *Delta*, come mostrato in figura:

```
final case class Tape[+Data, -Delta](
    data: Data,
    backward: Do[Delta] => UnitContinuation[Unit]
)
```

Listing 2.7: DualNumber monadico generico.

Tale *DualNumber* monadico può essere generalizzato a qualsiasi spazio lineare, per cui non solo ai tipi scalari, ma anche agli array n-dimensionali.

In ogni iterazione, una funzione differenziabile che contiene più variabili addestrabili può essere addestrata nel seguente modo:

1. Esecuzione della funzione differenziabile definita dall'utente con un batch di input;
2. Chiamata a *forward* sull'espressione differenziabile per costruire un grafo computazionale, cioè un *Do[Tape[Data,Delta]]*. Inizialmente il contatore di riferimento del grafo è zero;
3. Esecuzione del forward pass dell'espressione differenziabile per costruire un *Tape*, cioè un *Tape[Data, Delta]*, che contiene una coppia del risultato del forward pass e del backward pass. Il contatore di riferimento di ogni nodo in un grafo computazionale viene aumentato durante questa fase;

4. Esecuzione del backward pass del root node del grafo computazionale, con conseguente aggiornamento dell'accumulatore del delta sul root node;
5. Il contatore di riferimento di ogni nodo viene ridotto a zero e si esegue il backward pass di ogni nodo, in modo da aggiornare tutte le variabili addestrabili.

2.1.2 API

Due sono i metodi offerti dalla libreria per addestrare e valutare una rete neurale:

- **train**: restituisce una *Future* che aggiorna i pesi utilizzati internamente dal modello. Con la chiamata a questo metodo si procede con l'addestramento della rete, al fine di aggiornare i pesi interni, come spiegato in precedenza;
- **predict**: restituisce una *Future* che rappresenta la previsione di un modello in risposta ad un determinato input. Utilizzato per valutarne la bontà.

2.2 Plugins

Dalla versione 2.0 di DeepLearning.scala, tutte le funzionalità sono fornite dai plugin.

Un plugin è un *trait* che può essere combinato con altri plugin. Il principale scopo è di fornire nuove funzionalità o modificare i comportamenti delle funzionalità esistenti.

Ci sono 2 tipologie di plugin nella libreria, quelli integrati (built-in) e i non integrati (non built-in). In genere, gli utenti potrebbero contribuire alla libreria creando dei plugin non built-in con nuove funzionalità, come descritto nel seguente link: `https://deeplearning.thoughtworks.school/demo/ContributorGuide.html`.

2.2.1 Plugins built-in

Ogni plugin built-in di DeepLearning.scala 2.0 è una libreria distribuita nel repository centrale di Maven. L'elenco completo di tutti i plugin built-in può essere trovato in Scaladoc nel pacchetto *com.Thoughtworks.deeplearning.plugins*. Quelli che offrono le principali funzionalità sono: *DoubleLayer*, *DoubleWeight*, *FloatLayer*, *FloatWeight*, *INDArrayLayer* e *INDArrayWeight*.

2.2.2 Plugins non built-in

Questi plugin sono distribuiti come semplici file sorgente, come un Gist. Di seguito l'elenco di tutti i plugin non built-in presenti nella libreria:

- **FixedLearningRate**: imposta un learning rate fisso durante l'addestramento degli *INDArrayWeights*;
- **Adagrad**: algoritmo a gradiente adattivo con un learning rate per parametro per gli *INDArrayWeights*;
- **L1Regularization**: regolarizzazione L1;
- **L2Regularization**: regolarizzazione L2;
- **Momentum**: ottimizzatore Momentum per la discesa del gradiente (SGD);
- **RMSprop**: ottimizzatore RMSprop per la SGD;
- **Adam**: ottimizzatore Adam per la SGD;
- **INDArrayDumping**: logger per gli *INDArrayWeights*;
- **CNN**: implementazione di una Convolutional Neural Network.

2.3 Dipendenze

DeepLearning.scala si basa su altre librerie Java/Scala:

- **ND4J/ND4S**: ND4J [3] è una libreria di computazione scientifica per la JVM. Offre array multidimensionali, noti come *NDArray*, e funzioni di algebra lineare. La sua semantica imita quella delle librerie come Numpy, Matlab e scikit-learn. ND4S [4] è un binding Scala per ND4J;
- **Scalaz**: [5] è una libreria per la programmazione funzionale. Fornisce strutture dati puramente funzionali per completare quelle standard di Scala, definendo un insieme di classi di tipi fondamentali, come *Functor* e *Monad*;
- **feature.scala**: [6] è una raccolta di utility per accedere alle funzionalità del linguaggio Scala a livello di tipo. Queste utility vengono utilizzate in DeepLearning.scala 2.0 per creare il sistema di plugin, permettendo la formazione di mixin dinamici;
- **RAII.scala**: [7] è una raccolta di utility finalizzate alla gestione delle risorse native in Scalaz;
- **ThoughtWorks Each**: [8] è una libreria che converte la sintassi imperativa nativa nell'espressione monadica di Scalaz.

Chapter 3

Caso di studio: previsione di una serie storica

Scopo del progetto è la previsione della serie storica "International Airline Passengers" mediante l'utilizzo di una rete neurale realizzata con la libreria DeepLearning.scala.

3.1 Dataset

Il dataset, reperibile al link `https://www.kaggle.com/andreazzini/international-airline-passengers` e scaricabile al link `https://www.kaggle.com/andreazzini/international-airline-passengers/download`, raccoglie i dati mensili del numero di passeggeri (in migliaia) di compagnie aeree internazionali, dal 1949 al 1960.

Il dataset presenta un unico file in formato CSV composto da 2 colonne:

- **Month**
- **Passengers**

Per l'analisi della serie storica sarà sufficiente gestire i dati relativi al numero di passeggeri.

In fase di pre-processing è stato rimosso l'header contenente le informazioni relative a ciascuna colonna del file CSV.

3.2 Scopo dell'analisi

L'obiettivo è l'addestramento di una rete neurale per prevedere sia i valori del test set, che i valori dell'anno successivo, di cui non si hanno i dati reali.

3.3 Struttura

Il progetto proposto è composto da 7 file, tra cui 2 classi di test, visibili nella seguente immagine:

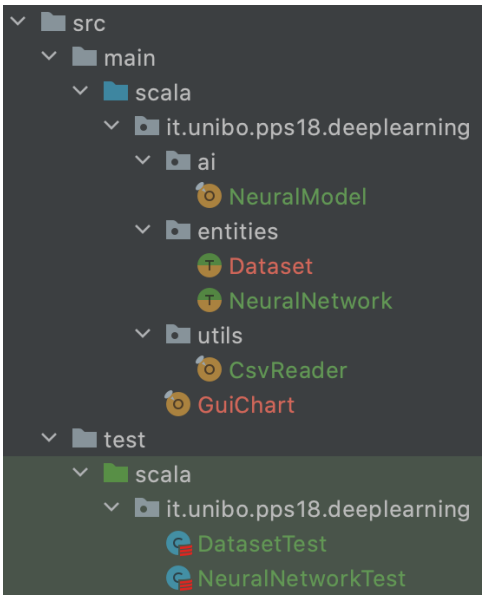

Figure 3.1: Struttura del progetto.

Il package "utils" contiene un oggetto di utility per poter leggere il contenuto di un file CSV. Nel package "entities" sono state modellate le entità del progetto, cioè il dataset e la rete neurale, entrambe testate con *ScalaTest*. Il cuore del progetto risiede all'interno del package "ai", dove è stato creato il modello della rete neurale, con il relativo addestramento e valutazione. Infine, è stata realizzata una GUI con *ScalaFX* per poter visualizzare le previsioni effettuate dal modello neurale.

3.4 Entità

In questa sezione si descrive come sono state modellate le 2 entità del progetto.

3.4.1 Dataset

Il concetto di dataset è stato rappresentato mediante una classe *TimeSeries-Dataset*, con lo scopo di creare un dataset specifico per le analisi di serie storiche.

I dati delle serie temporali possono essere ristrutturati per essere gestiti da un apprendimento supervisionato. Questo è possibile utilizzando i time step precedenti come variabili di input e utilizzare il time step successivo come variabile di output. L'utilizzo di time step precedenti per prevedere la fase temporale successiva è chiamato metodo della finestra scorrevole (sliding window). Il numero di time step precedenti rappresenta la larghezza della finestra (window width). Questo pre-processing è fondamentale per poter trasformare qualsiasi dataset di serie temporali in un problema di apprendimento supervisionato.

```
/** Compute a sliding window matrix from a series of values.
 *
 *  @param data the data to perform windowing
 *  @param nPast the window width
 *  @return a tuple composed of the window matrix to train with the respective value.
 */
private def computeWindows(data: List[Double], nPast: Int): (INDArray, INDArray) = {
  val x: INDArray = data.sliding(nPast).take(data.length - nPast).map(x => x.toArray).toArray.toNDArray
  val y: INDArray = data.takeRight(data.length - nPast).map(x => Array(x)).toArray.toNDArray
  (x, y)
}
```

Figure 3.2: Applicazione della sliding window.

Un'altra operazione effettuata sul dataset è stata una normalizzazione min-max, che è una tecnica di *feature scaling*. In questa normalizzazione, i dati vengono ridimensionati su un intervallo fisso, da 0 a 1, con la seguente formula:

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Figure 3.3: Normalizzazione min-max.

Con i dati normalizzati e l'applicazione della sliding window, il dataset è pronto per essere processato da un modello neurale.

3.4.2 Neural Network

Si è scelto di modellare la rappresentazione di una rete neurale per poterne calcolare il numero di parametri interni. Inoltre, per fini sperimentativi, si è deciso di creare un interpolatore custom per la creazione di una rete neurale.

3.5 Modello neurale

Il modello costruito grazie al supporto della libreria DeepLearning.scala è un modello neurale di regressione. In seguito viene spiegato step-by-step il procedimento adottato per sua la creazione.

3.5.1 Configurazione iperparametri

Gli iperparametri sono la configurazione globale per una rete neurale. Per questo modello, si è impostato il suo learning rate, che determina la velocità con cui il modello cambia i suoi pesi interni durante l'addestramento, così come l'ottimizzatore.

Nella sezione 2.2.2 si sono elencati i plugin non built-in, con i quali è possibile impostare gli iperparametri della rete. Questi plugin non sono altro che piccoli pezzi di codice caricati da un URL, che possono essere importati nel seguente modo:

```
// Fixed learning rate plugin.
import $exec.`https://gist.github.com/Atry/1fb0608c655e3233e68b27ba99515f16/raw/39ba06ee597839d618f2fcfe9526744c60f2f70a/FixedLearningRate.sc`
// Adam optimizer plugin.
import $exec.`https://gist.github.com/Rabenda/0c2fc6ba4cfa536e4788112a94200b50/raw/233cbc83932dad659519c80717d145a3983f57e1/Adam.sc`
```

Figure 3.4: Import di plugins non-built.

Per poter configurare gli iperparametri si crea una *factory* che funziona nel seguente modo: dato un *trait* con membri astratti, come i plugin che si ha importato, quando si crea una *factory* per il *trait*, il metodo *newInstance()* della *factory* accetta una serie di argomenti in base ai membri astratti.

```
// Global configuration for the neural network.
val hyperparameters = Factory[Builtins with FixedLearningRate with Adam].newInstance(learningRate = 0.01)
```

Figure 3.5: Iperparametri.

Il plugin *Builtins* contiene alcuni valori impliciti, che possono essere importati come segue:

```
// Import plugin's implicits
import hyperparameters._
import hyperparameters.implicits._
```

Figure 3.6: Import di valori impliciti degli iperparametri.

3.5.2 Composizione del modello

In DeepLearning.scala una rete neurale è semplicemente una funzione che fa riferimento ad alcuni pesi, che sono variabili mutabili che vengono modificate automaticamente in base ad alcuni obiettivi durante l'allenamento.

In *weights* vengono salvati i pesi della rete, inizialmente casuali.

```
// Weights of the neural network.
val weights = INDArrayWeight(Nd4j.randn(inputNeurons, outputNeurons, seed = 1))
```

Figure 3.7: Pesi del modello.

É possibile notare che *weights* è un peso di un array n-dimensionale, per cui un *INDArrayWeight*.

Il modello neurale è di tipo feed-forward, il cui output può essere calcolato mediante la moltiplicazione dell'input con i pesi, con la successiva applicazione della funzione di attivazione (in questo caso una ReLU).

```
/** Build the ReLU activation function.
 *
 *  @param input the input to rectify
 *  @return the max between 0 and the input.
 */
def relu(input: INDArrayLayer): INDArrayLayer = max(0.0, input)

/** Compose a neural network.
 *
 *  @param inputs the input data
 *  @return a layer of the neural network, seen as the dot product between inputs and weights.
 */
def neuralNetwork(inputs: INDArray, weights: INDArrayWeight): INDArrayLayer = relu(inputs dot weights)
```

Figure 3.8: Rete neurale feed-forward.

Per la moltiplicazione matriciale si utilizza *dot*, che è una funzione differenziabile offerta dalla libreria. Questo esempio mostra la peculiarità, spiegata in precedenza, delle funzioni differenziabili, cioè che accettano tipi diversi e restituiscono un layer della rete neurale, che può essere composto in un'altra chiamata di funzione differenziabile. Infatti, *inputs* è un tipo vettoriale INDArray, mentre *weights* è un Weight.

Quando si addestra una rete neurale, l'obiettivo dovrebbe sempre essere ridurre al minimo una determinata funzione di errore. A tale scopo, è possibile creare un'altra rete neurale che valuta l'errore tra il risultato previsto dalla rete e i valori reali. Questa rete rappresenta la funzione di loss.

Siccome in questo progetto si deve prevedere una serie storica, si utilizzerà l'errore quadratico medio (MSE) come funzione di loss:

```
/** Loss function to determine the mean squared error (MSE) between predictions and the real values.
 *
 *  @param predictions the predictions of the neural network
 *  @param target the real values
 *  @return the MSE.
 */
def lossFunctionMse(prediction: INDArrayLayer, target: INDArray): DoubleLayer = {
  val error: INDArrayLayer = prediction - target
  (error * error).mean
}
```

Figure 3.9: Funzione di loss.

Questa funzione di loss sarà utilizzata da un modello di regressione durante l'addestramento. Quando il modello viene addestrato, il valore restituito dalla loss tenderà a zero, e il risultato previsto dalla rete dovrebbe avvicinarsi al risultato reale.

```
/** Build a linear regression model.
 *
 *  @param inputs the input data
 *  @param target the target data
 *  @return a layer of the neural network.
 */
def linearRegression(inputs: INDArray, target: INDArray): DoubleLayer = {
  val prediction: INDArrayLayer = neuralNetwork(inputs, weights)
  lossFunctionMse(prediction, target)
}
```

Figure 3.10: Modello di regressione lineare.

3.5.3 Addestramento del modello

Per addestrare una rete neurale si utilizza il metodo **train** per *DoubleLayer*, spiegato in 2.1.2, che è una *ThoughtWorks Future* che esegue un'iterazione del training.

Poiché si vuole addestrare ripetutamente la rete neurale, per un totale di 5000 iterazioni, si necessita di creare un'altra *Future* che esegua le iterazioni del training. Per questo scopo è stata utilizzato *ThoughtWorks Each* per creare tale *Future*.

```scala
val epochs: Int = 5000

/** Training task. You need to use @monadic[T] annotation in order to enable the for comprehension syntax.
 *
 *  @return a scala future with a stream of the loss values.
 */
@monadic[Future]
def trainTask: Future[Stream[Double]] = {
  for (_ <- (0 until epochs).toStream) yield {
    linearRegression(xTrain, yTrain).train.each
  }
}

// Training the neural network.
Await.result(trainTask.toScalaFuture, Duration.Inf)
```

Figure 3.11: Addestramento del modello neurale.

3.6 Risultati

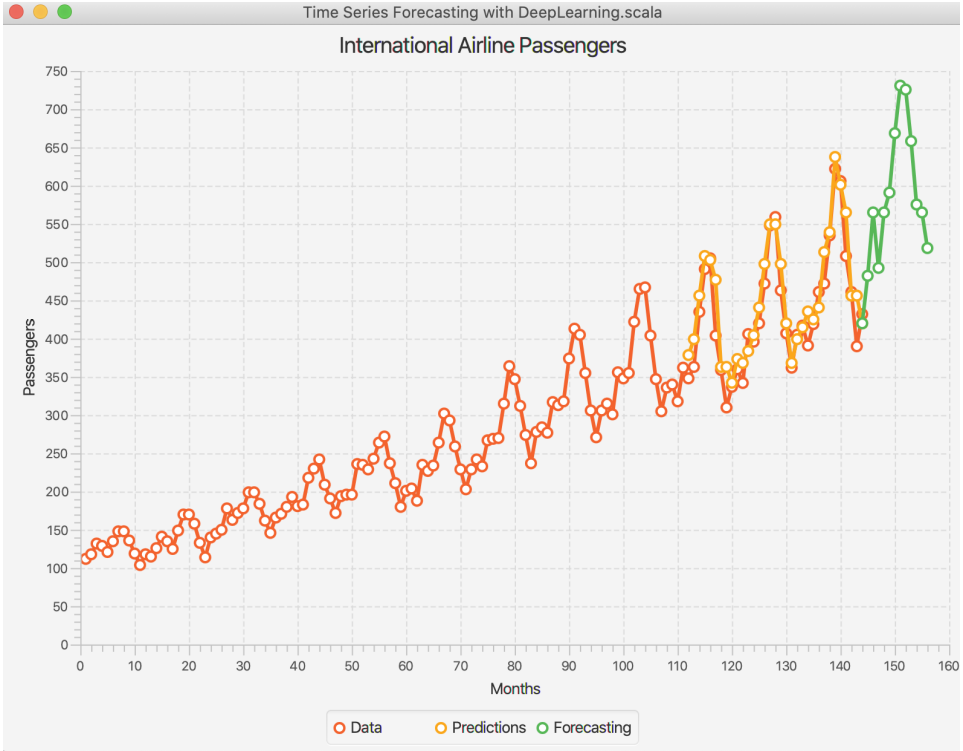

Figure 3.12: Grafico delle previsioni.

Come è possibile notare dalla GUI, la serie rossa rappresenta i dati originali, quella gialla le previsioni fatte dal modello sul test set, mentre in verde le previsioni future, in particolare dell'anno successivo. Si può vedere la fedeltà della previsione nell'approssimare l'andamento del dataset.

Per raggiungere questi risultati si è deciso di utilizzare una finestra temporale di dimensione 11. Ciò significa che la previsione di un valore avviene addestrando il modello sugli 11 valori precedenti. É stato scelto il valore 11 poiché, insieme alla previsione, rappresenta l'andamento annuale.

Riguardo lo split del dataset è stato adottato un tradizionale 70% per il training set e il restante 30% per il test set.

Chapter 4

Conclusione

Analizzando le API di DeepLearning.scala, e mettendole in pratica con un caso di studio, è stato possibile comprendere il punto di forza di questa libreria: creare modelli neurali scrivendo semplice codice Scala. Infatti, ogni layer di una rete neurale non è altro che una funzione componibile con altre. Con queste caratteristiche risulta poi più semplice crearsi dei layer custom come base per esperimenti.

Nello stesso tempo, durante lo sviluppo del progetto si sono riscontrati diversi problemi sull'utilizzo della libreria, in combinazione con l'IDE IntelliJ IDEA. In particolare, il compilatore non è riuscito a compilare correttamente il comando *$exec* per poter importare i plugin della libreria. Questo ha determinato i successivi problemi di una corretta compilazione dei membri astratti dei plugin, insieme ai loro metodi. Questi problemi erano discussi nella documentazione della libreria, assicurando comunque gli utenti del corretto funzionamento.

Bibliography

[1] Deeplearning.scala. https://github.com/ThoughtWorksInc/DeepLearning.scala.

[2] Monadic deep learning. https://deeplearning.thoughtworks.school/assets/paper.pdf.

[3] Nd4j. https://github.com/deeplearning4j/nd4j.

[4] Nd4s. https://github.com/deeplearning4j/nd4s.

[5] Scalaz. https://github.com/scalaz/scalaz.

[6] feature.scala. https://github.com/ThoughtWorksInc/feature.scala.

[7] Raii.scala. https://github.com/ThoughtWorksInc/RAII.scala.

[8] Each. https://github.com/ThoughtWorksInc/each.