Deep Learning

Professor: Horacio Larreguy
TA: Eduardo Zago

ITAM Applied Research 2 / Economics Research Seminar,
11/01/2023

General Perspective

Introduction to Deep Learning

Perceptron

Gradient Descent

Back-Propagation

Activation Functions

Regularizers

Neural Net Example using Tensorflow

What is Deep Learning?

- Subset of ML, involves using **Neural Networks** to learn from data.
- **Difference from ML:** more complex models with multiple layers that can learn from raw data.
- Reduces the need for manual feature engineering.
- They have been found to perform better than ML models in complex tasks such as speech recognition, NLP, Image analysis and robotics.

Neural Networks

- Inspired by the structure of the brain.
- Series of interconnected processing nodes which work together to perform complex computations.
- Three Types of Layers:
  1. **Input layer:** receives the raw data
  2. **Hidden layer:** perform a series of mathematical operations on the input data to extract meaningful features and patterns
  3. **Output layer:** makes the prediction or classification.

Neural Network

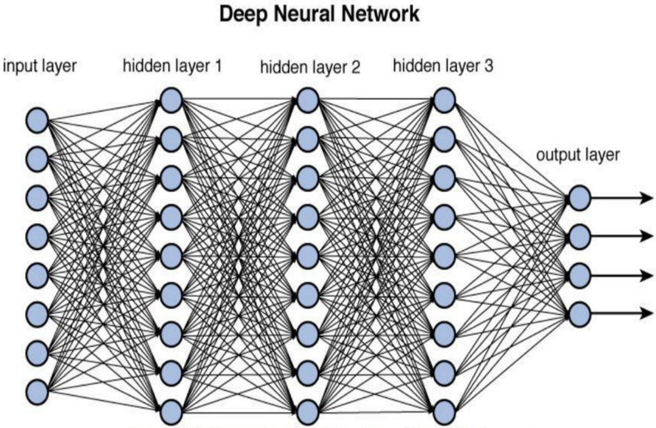
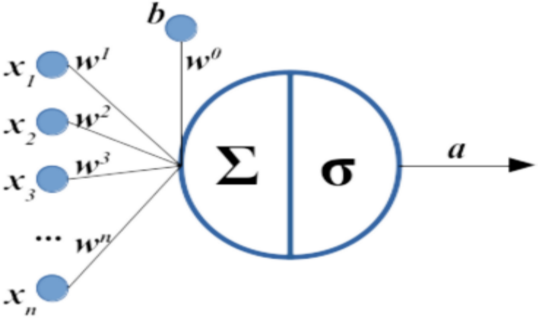


Figure 12.2 Deep network architecture with multiple layers.

Perceptron

- To better understand the structure of a NN let us look at the simplest one: the **perceptron**.
- A **perceptron** is a single neuron (node) connected to one or more input nodes, with a weight associated to each.
- The neuron sums up the weighted inputs and applies an **activation function** to produce an output.
- Let us look at one graphically and mathematically:

Perceptron, Graphically



Perceptron, Mathematically

- Mathematically, if we have a Sigmoid[1] activation function $\sigma(s)$, then:

$$
\begin{aligned}
s &= w^T x, \\
&= \sum_{n=0}^{N} w_n x_n, \\
&= \sum_{n=1}^{N} w_n x_n + w_0 n_0
\end{aligned} \tag{1}
$$

And

$$a = \sigma(s)$$

[1]Similar to what the Logistic function does in a Logit model.

Perceptron, Mathematically

- $a = \sigma(s)$ would be the output signal, or prediction, that takes the weighted sum of inputs s and applies a non-linearity to them.
- The immediate question that follows from this is: *how can we estimate the parameter w_i?*
- We'll use an iterative minimization algorithm called **Gradient Descent** to update our parameters gradually.

Gradient Descent

- Before explaining Gradient Descent we must remember a few things:
- We are using a **data-driven approach**, thus we have a labeled data set.
- We defined the loss as the difference between the **predicted** output and the **actual** output.
- For simplicity:

$$E = (y - \hat{y})^2$$

- Recall that we **minimize** the objective function of a given problem by moving our parameters in the opposite direction of the gradient.

Gradient Descent

- Therefore, since our parameters are w_i and our objective function (the one we want to minimize) is E, we have that:

$$w^{i+1} = w^i - \nu \frac{\partial E}{\partial w^i}$$

- In our context, since:

$$\begin{aligned} s &= w^T x, \\ \hat{y} &= \sigma(s) \\ E &= \frac{1}{2}(y - \hat{y}) \\ \sigma'(s) &= \sigma(s)(1 - \sigma(s)) \end{aligned} \tag{2}$$

Gradient Descent

- We would have that:

$$\begin{aligned}\frac{\partial E}{\partial w_n^i} &= \frac{\partial (y-\hat{y})^2}{\partial w_n^i}, \\ &= -(y-\hat{y})\sigma(s)(1-\sigma(s))x_n\end{aligned} \tag{3}$$

Therefore,

$$w_n^{i+1} = w_n^i + \nu(y-\hat{y})\sigma(s)(1-\sigma(s))x_n$$

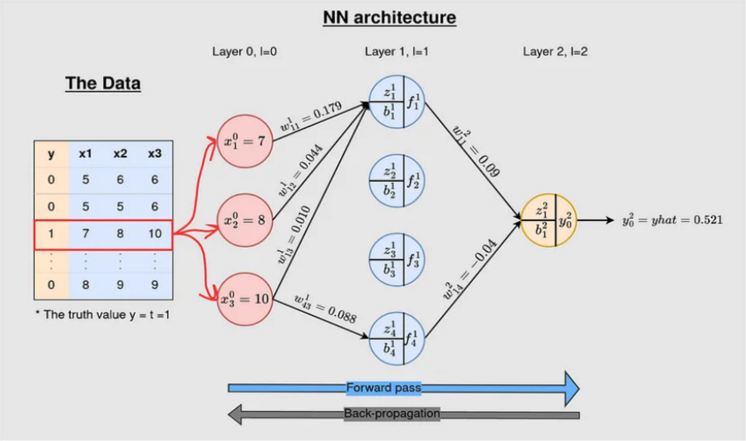
Gradient Descent, Algorithm

- Thus, the training process for batch i at epoch e, is given by:

1. Random initialization of weights $\Longrightarrow w_0$
2. Forward pass $\Longrightarrow \hat{y} = f(x; w)$
3. Error estimation $\Longrightarrow E(y, \hat{y})$
4. Gradient computation $\Longrightarrow \frac{\partial E}{\partial w_n}$
5. Backward pass (weight adjustment) $\Longrightarrow w_{n+1} = w_n - \frac{\partial E}{\partial w_n}$

Information Flow

- Now, how do we do this for more than one neuron and several hidden layers?



Back-Propagation

- **Back-propagation** is a method for supervised learning used by NN to update parameters.
- For Neural Nets with multiple layers, the backward pass can get really complicated, so we must understand how the information flows.
- We must do the backward pass (weight adjustment) of all weights in the model.
- Some of these weights are concatenated inside several activation functions.
- Thus, to obtain the derivative, we must calculate the **Chain Rule**.

Back-Prop, Chain Rule

Recall that the chain rule is used to differentiate any function of the form:

$$h(x) = f(g(x))$$

And the derivative is given by:

$$h'(x) = f'(g(x))g'(x)$$

In a more general context, if we have a function:

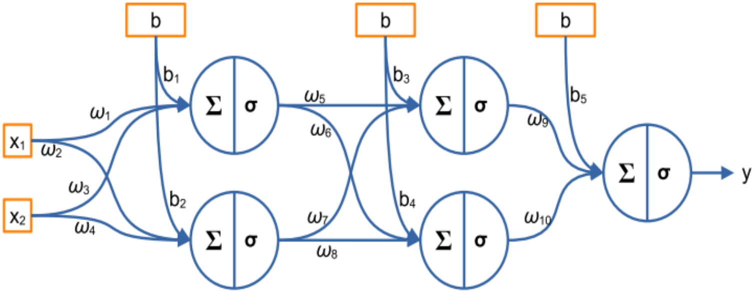
$$f_{1..n} = f_1(f_2(f_3(...f_{n-1}(f_n(x))...)))$$

The derivative would be given by:

$$f'_{1..n} = f'_1(f_{2..n}(x)) \cdot f'_2(f_{3..n}(x)) \cdots f'_{n-1}(f_{n..n}(x)) \cdot f'_n(x)$$

Back-Prop, Example

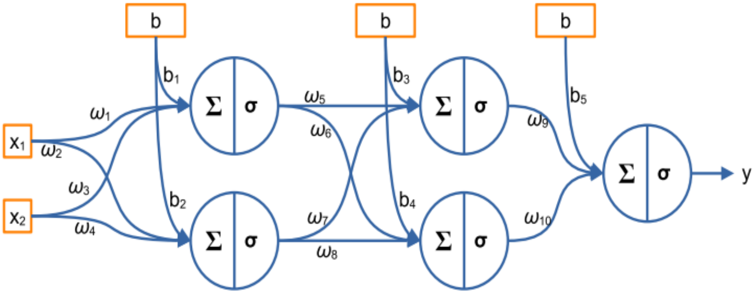
- As an example, let us look at the next NN:



- Also recall that we defined:

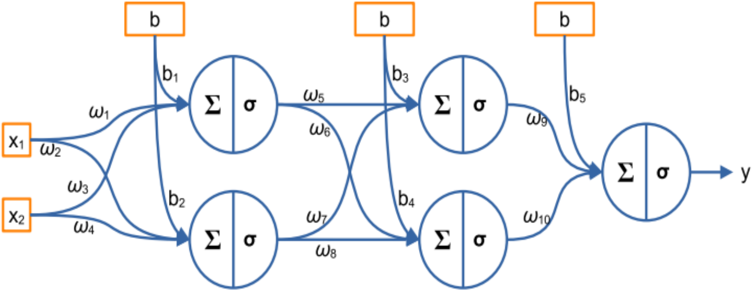
$$s = w^T x, \\ a = \sigma(s) \tag{4}$$

Wide Neural Net



- Notice we have: 5 activation functions (2 for each hidden layer and one in the final layer)
- Also, at each node there is a linear combination of the weights and the input data (for the first layer)
- And a linear combination of the output of the last layer and their respective weights.

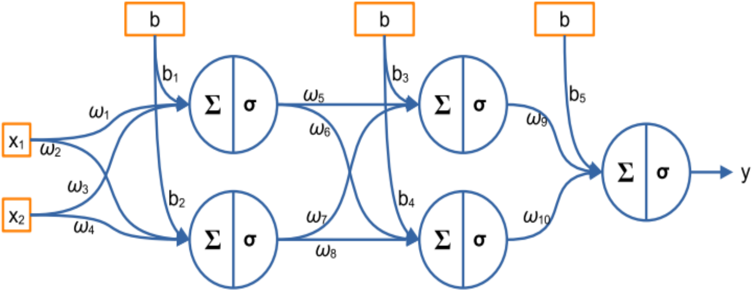
Wide Neural Net



- The linear combination at node 1 and 3 (for example) would be given by:

$$\begin{aligned} s_1 &= b_1 + w_1 \cdot x_1 + w_3 \cdot x_2, \\ s_3 &= b_3 + w_5 \cdot a_1 + w_7 \cdot a_2 \end{aligned} \tag{5}$$

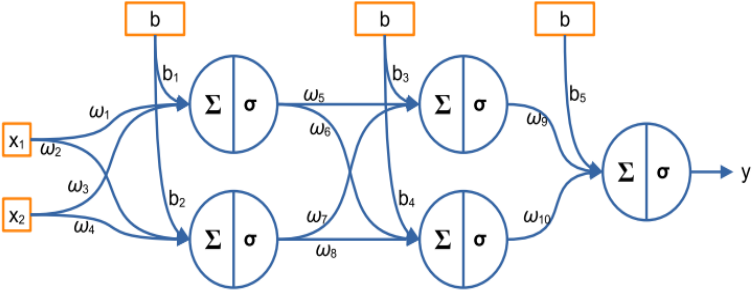
Back-Prop, Taking Derivatives



- Thus, the derivative of E with respect to w_i is:

$$\begin{aligned} \frac{\partial E}{\partial w_1} = & \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_3}\frac{\partial a_3}{\partial s_3}\frac{\partial s_3}{\partial a_1}\frac{\partial a_1}{\partial s_1}\frac{\partial s_1}{\partial w_1}, \\ & + \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_4}\frac{\partial a_4}{\partial s_4}\frac{\partial s_4}{\partial a_1}\frac{\partial a_1}{\partial s_1}\frac{\partial s_1}{\partial w_1} \end{aligned} \tag{6}$$

Back-Prop



- **Why the sum of products?**

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_3}\frac{\partial a_3}{\partial s_3}\frac{\partial s_3}{\partial a_1}\frac{\partial a_1}{\partial s_1}\frac{\partial s_1}{\partial w_1}, \\ &+ \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_4}\frac{\partial a_4}{\partial s_4}\frac{\partial s_4}{\partial a_1}\frac{\partial a_1}{\partial s_1}\frac{\partial s_1}{\partial w_1}\end{aligned} \tag{7}$$

Weight Updating

- Finally, as seen in the last chapter, weight 1 gets updated in the following form:

$$w_1^{i+1} = w_1^i - \frac{\partial E}{\partial w_1^i}$$

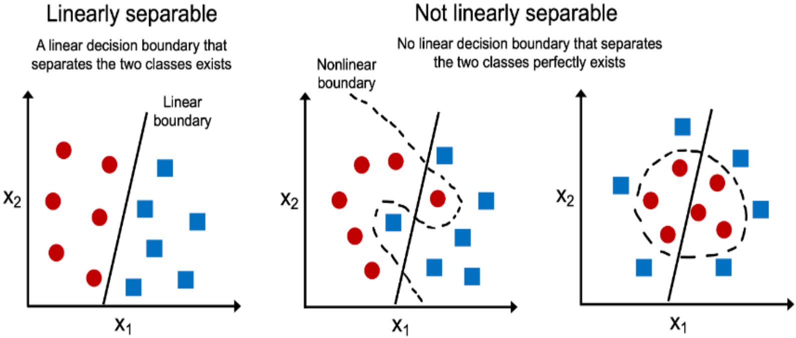
- And this happens to all the weights at each time of training t (batch or epoch).
- Notice that the impact of back-prop is proportional to the depth of the layer.
- Weights in shallow layers are updated more softly with respect to those in deeper layers.

Activation Functions

- An **activation function** defines how the weighted sum of the input is transformed into an output from a node (or nodes) in a layer of a network.
- Introduces a **non-linearity** into the output of the neuron.
- Allows the neural network to learn complex patterns and relationships in the data.
- In the **hidden layers**, the activation function will control how well the model learns.
- In the **output layer**, the activation function will define the type of predictions the model can make.

Linear vs. Non-Linear Problems

- Activation functions allow the Neural Net to perform well in non-linear separable problems[2]:



[2]Image taken from VitalFlux

Linear and ReLU Activations

- **Linear:** used at the output layer for unbounded regression, but also works for binary classification.

$$a = \sum_{n=0}^{N} x_n w_n, \quad \frac{\partial a}{\partial w_i} = x_i$$

- **ReLU:** maps its input to the maximum of 0 and the input value. Very efficient and works well in tons of contexts.

$$a = max(0, x)$$

Sigmoid and TanH Activations

- **Sigmoid:** mostly used at the output layer for binary classification problems and regression with $0 \leq y \leq 1$.

$$a = \sigma(s) = \frac{1}{1+e^{-s}}, \quad \sigma'(s) = \sigma(s)(1-\sigma(s))$$

- **TanH:** Better than sigmoid in hidden layers since it resembles the identity function more closely, $tan(0) = 0$.

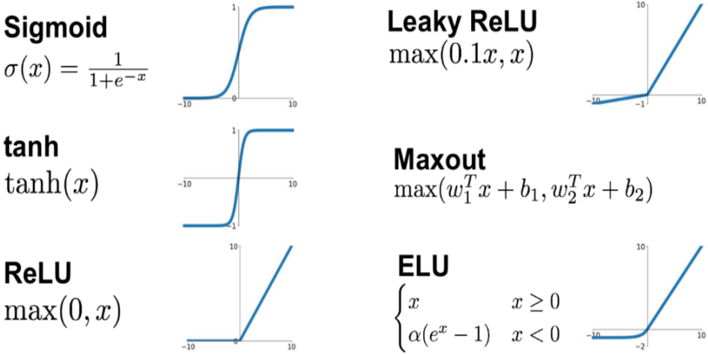
$$a = \frac{e^s - e^{-s}}{e^s + e^{-s}}, \quad a' = 1 - a^2$$

SoftMax

- Used for multi-class classification
- Used to exaggerate the most probable of the elements of the vector.
- It maps its input to a probability distribution over the output classes, allowing the neural network to produce a probability score for each class.

$$a_i = \frac{e^{s_i}}{\sum_j e^{s_j}}, \quad a_i' = a_i(1 - a_i)$$

Activation Functions Graphically



Regularizers

- Recall that regularizers are techniques used to reduce **over-fitting** sometimes at the expense of increased **under-fitting**.
- In a ML context we went over two techniques: L1 and L2 regularization.
- They both could be used in DL models, as we have an objective function (the Error).
- Other techniques used are: **Dataset Augmentation, Early Stopping** and **DropOut**.

Dataset Augmentation

- This technique consists of increasing the dataset by creating **fake data** and adding it to the training set.
- Usually used in classification tasks to increase the model's robustness.
- Prediction needs to be invariant to a wide variety of transformations.
- A good example of a context where it can be used is in object recognition, since you can rotate, change the color and scale images, to generate new data points.[3]

[3] Image taken from DataCamp.

DATA AUGMENTATION

datacamp

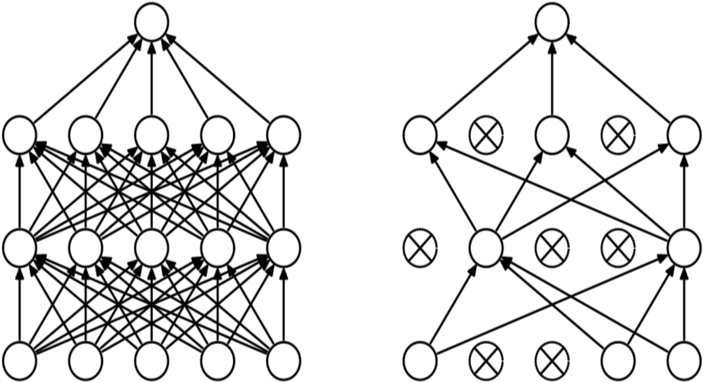
Early Stopping

- Refers to returning to the parameters, epoch, where the validation accuracy last improved, and use them as if they were the last parameters.
- Or, stopping the algorithm when no parameters have improved over the best recorded validation error.
- Very unobtrusive technique.
- Could be seen as a restriction to the parameters to a neighborhood near w^0.

Drop Out

- This technique limits the capacity of the model at random by deactivating neurons (dropout) or weights (drop-connect).
- We must choose the dropout probability, which becomes then another hyper-parameter.
- Very popular technique since it is very computationally cheap (in contrast to L1 or L2)
- Helps the network be robust against variations.

Drop Out



(a) Standard Neural Net (b) After applying dropout.

Neural Net Example using Tensorflow

- `Tensorflow` is an end-to-end open source platform for machine learning.
- `TensorFlow`'s high-level APIs are based on the `Keras` API standard for defining and training neural networks.
- `Keras` enables fast prototyping, state-of-the-art research, and production—all with user-friendly APIs.
- Let's install them:

```
!pip install keras
!pip install tensorflow
```

Load the Packages and Define the Problem

- Let's first load the packages and the data set:

```
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

from sklearn.datasets import load_iris
iris = load_iris()
```

- We will again use the Iris Data Set, but we will increase the difficulty of the problem by a lot.
- We will try to **predict the length and width of sepals using the length and width of the petals.**

Prepare the Data

- As always, let's prepare our data by splitting the sample into training and test:

```
X = iris.data[:, :2]
Y = iris.data[:, 2:]
L = iris.target

# Train Test Split:
from sklearn.model_selection import train_test_split
x_train, x_test, y_train,
    y_test, l_train, l_test = train_test_split(X, Y, L,
    test_size=0.2, random_state = 42)
```

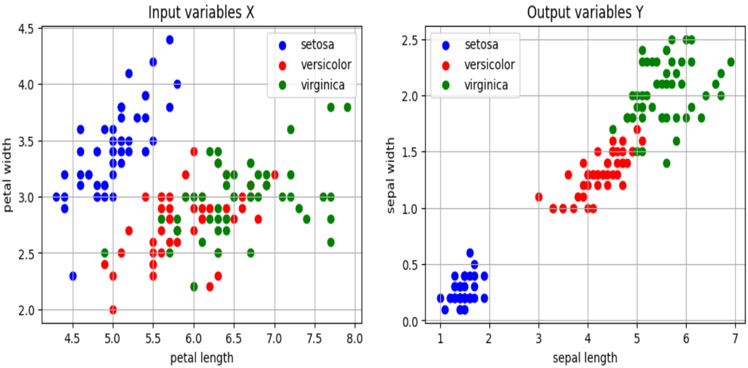
Visualizing the Data

- We can visualize the problem by graphing each set of features and their classes:

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.scatter(X[L==0, 0], X[L==0, 1], c='b', label='setosa')
plt.scatter(X[L==1, 0], X[L==1, 1], c='r', label='versicolor')
plt.scatter(X[L==2, 0], X[L==2, 1], c='g', label='virginica')
plt.legend()
plt.grid(True)
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.title('Input variables X')
plt.subplot(1, 2, 2)
plt.scatter(Y[L==0, 0], Y[L==0, 1], c='b', label='setosa')
plt.scatter(Y[L==1, 0], Y[L==1, 1], c='r', label='versicolor')
plt.scatter(Y[L==2, 0], Y[L==2, 1], c='g', label='virginica')
plt.legend()
plt.grid(True)
plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.title('Output variables Y')
plt.show()
```

Visualizing the Data

- It is indeed a more complex problem, since we are estimating a function that goes from $\mathbb{R}^2$ to $\mathbb{R}^2$.



Defining the Model

- Let's start defining our model:
- We have two inputs, thus the input layer takes a shape of (2,):

```
i = Input(shape=(2,), name='input')
```

- We will define a model with **3 hidden layers**, 32 neurons in the first and the third, and 64 in the 2nd one.
- For activations we will use ReLu for the hidden layer and Linear in the final:

```
h = Dense(units=32, activation='relu', name='hidden1')(i)
h = Dense(units=64, activation='relu', name='hidden2')(h)
h = Dense(units=32, activation='relu', name='hidden3')(h)
o = Dense(units=2, activation='linear', name='output')(h)
```

Defining the Model

- Finally, we aggregate our model using the `Model()` function and obtain an overall description of it:

```
MLP = Model(inputs=i, outputs=o)
MLP.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 2)]               0

 hidden1 (Dense)             (None, 32)                96

 hidden2 (Dense)             (None, 64)                2112

 hidden3 (Dense)             (None, 32)                2080

 output (Dense)              (None, 2)                 66

=================================================================
Total params: 4,354
Trainable params: 4,354
Non-trainable params: 0
_________________________________________________________________
```

Compiling the Model

- To start training the model, we first need to choose another set of hyper-parameters: the optimization procedure and the error function.
- We will use the ones we have seen, **Stochastic Gradient Descent (SGD)** for optimization and **Mean Square Error (MSE)** for Error:

```
# Compile it
MLP.compile(optimizer='sgd', loss='mse')
```

Training the Model

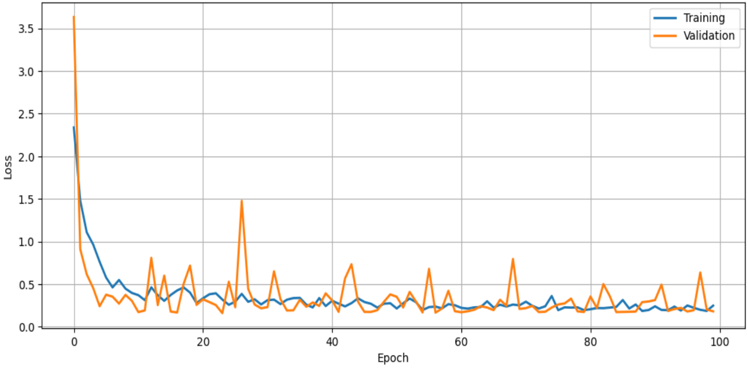
- And we can start training it using our data sets. We choose the last two hyper-parameters: batch size and epochs.

```
# Train it
MLP.fit(x=x_train, y=y_train, batch_size=4,
        epochs=100, verbose=2, validation_split=0.1)
```

- We have 100 values for the loss and the accuracy, both in training in validation.
- To visualize them we will use the Learning Curve we saw in the ML lecture.

Visualizing Metrics

```
plt.figure(figsize=(12, 5))
plt.plot(MLP.history.history['loss'], label='Training', linewidth=2)
plt.plot(MLP.history.history['val_loss'], label='Validation', linewidth=2)
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```



Model's Performance

- And we can conclude that the model is not under-fitted, since the loss is really low in training.
- We also notice that the training loss and the validation loss are really similar at all epochs, therefore the model is not over-fitted.

```
# Evaluate on the test set
MLP.evaluate(x=x_test, y=y_test, verbose=False)

### Output: 0.10164663195610046
```

- We obtained an average loss of 10%, which given that this is a regression problem, is very good.