Introduction to Deep Learning

March 15, 2023

INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO (ITAM)

Economic Research Seminar / Applied Research 2[1]

Professor: Horacio Larreguy

TA: Eduardo Zago

[1]We give credits to the Deep Learning course taught at ITAM's Master in Data Science because some of the content in this tutorial is taken from the notes made by PhD. Edgar Francisco Roman-Angel

1 Introduction to Deep Learning

1.1 What is Deep Learning?

Deep learning (DL) is a subset of machine learning that involves the use of artificial neural networks (ANNs) with multiple layers to analyze and model data. The multiple layers in a deep neural network enable it to learn complex patterns and relationships from large amounts of data.

Deep learning has found many applications in various fields such as image and speech recognition, natural language processing, and robotics. Examples of deep learning algorithms include convolutional neural networks (CNNs) for image analysis, recurrent neural networks (RNNs) for natural language processing, and deep belief networks (DBNs) for unsupervised learning.

The success of deep learning is attributed to its ability to automatically learn features from raw data, which reduces the need for manual feature engineering. Deep learning algorithms are also able to generalize well to new data and can perform better than traditional machine learning algorithms in certain tasks.

1.2 Neural Networks

A neural network is a type of machine learning model that is inspired by the structure and function of the human brain. It consists of a series of interconnected processing nodes, called neurons, which work together to perform complex computations on input data.
A typical neural network consists of three types of layers: input, hidden, and output layers.

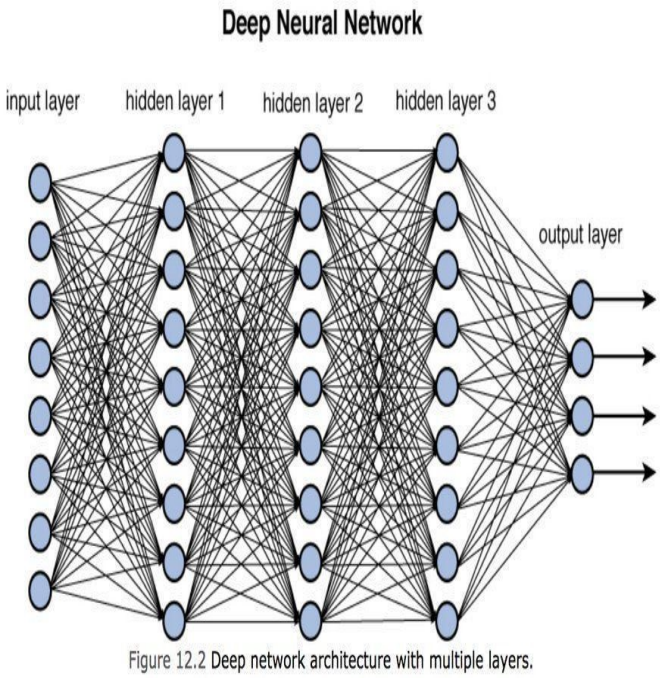
1. The **input layer** receives the raw data, such as images or text, and passes it on

to the hidden layers for processing.

2. The **hidden layers** perform a series of mathematical operations on the input data to extract meaningful features and patterns.

3. Finally, the **output layer produces the model's prediction or classification based on the input data.**

Figure 1: Neural Network



To understand how these three types of layers are connected, we will first understand the simplest of NN, the Perceptron.

1.3 The Easiest NN: Perceptron

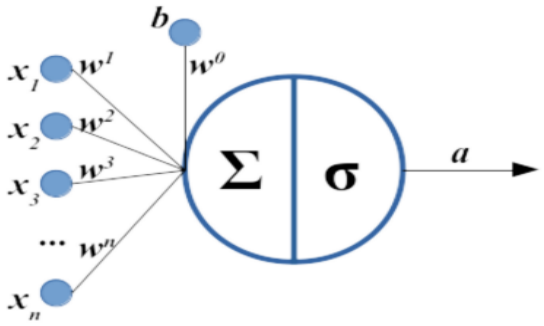
A perceptron is a type of artificial neural network that is commonly used in machine learning and pattern recognition. It consists of a single layer of artificial neurons, which are simple computational units that receive input signals, perform a calculation on those

signals, and produce an output signal.

In general terms (which will be explained further in this Section), each neuron in a perceptron is connected to one or more input nodes, and each input node is associated with a weight. The neuron sums up the weighted inputs and applies an activation function to produce an output signal.

To better understand this, let's look at it graphically and **mathematically**. Recall the context of a Logit model, where we had a set of features X (in this case, inputs), set of parameters β (in this case weights w_i) and a prediction or outcome y. For a perceptron we have something similar:

Figure 2: Perceptron



Thus, we have:

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

Where w_0 is b_0 and it is called the bias at layer 0 and $x_0 = 1$. In the context of a

Logit model, the *activation function* to get the prediction $\hat{y}$ would be given by the next equation:

$$\hat{y} = \frac{1}{1 + e^{-s}}$$

For this perceptron we will use an activation function named Sigmoid[2], thus the output is given by:

$$a = \sigma(s)$$

The immediate question that follows from these is how can we estimate the parameter w_i? We'll use an iterative minimization algorithm called Gradient Descent to update our parameters gradually.

1.4 Gradient Descent

Recall from the Machine Learning lecture that we are using a data-driven approach, meaning we have a labeled data-set that contains many examples of each class and we develop learning algorithms that look at these examples and learn about the characteristics and patterns in the data. Also, recall that we defined the **loss** as the difference between the predicted output and the actual output for a given input. Mathematically, it is function that computes the error between the predicted output and the actual output. We'll define it as:

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

Also recall from Calculus, that we minimize the objective function of a given problem by moving our parameters in the opposite direction of the gradient. Therefore, since our parameters are w_i and our objective function (the one we want to minimize) is E, we have that:

[2]We will cover activation functions in detail in the next Section.

$$w_{i+1} = w_i - \nu \frac{\partial E}{\partial w_i}$$

Thus, for a non-linear perceptron where we have:

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

Then,

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

Therefore,

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

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

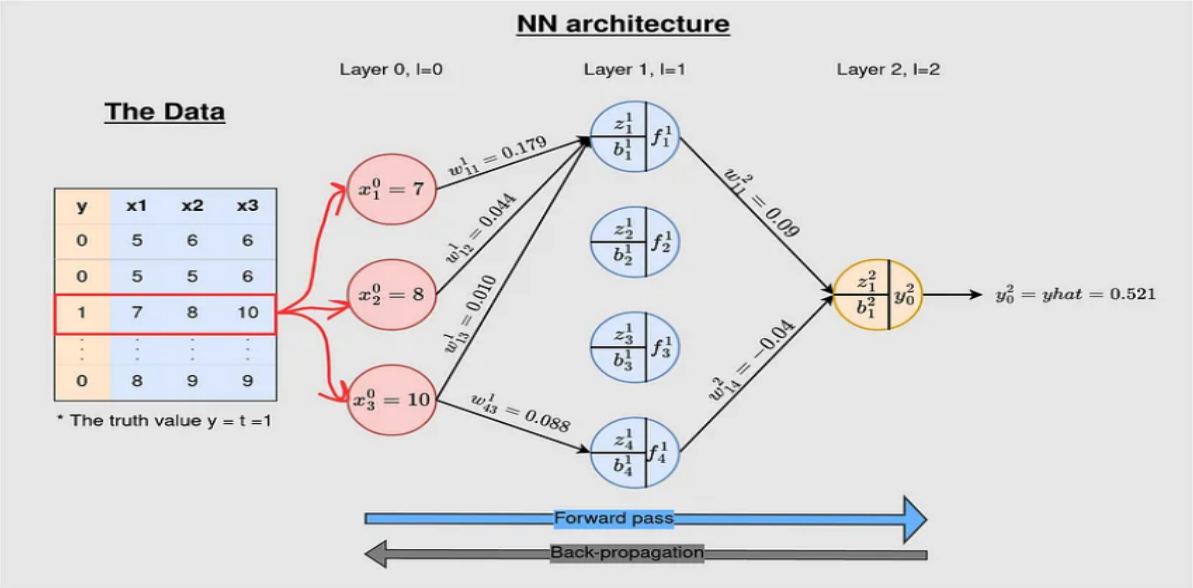
1. Random initialization of weights $\Longrightarrow w_0$
2. Forward pass $\Longrightarrow \hat{y} = f(x; w)$
3. Error estimation $\Longrightarrow E(y, \hat{y})$
4. Gradient computation $\Longrightarrow \frac{\partial E}{\partial w_n}$
5. Backward pass (weight adjustment) $\Longrightarrow w_{n+1} = w_n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

Now the issue is, how would we do it with several layers and more that one perceptron? The answer is **Back-Propagation**

1.5 Back-Propagation

Back-propagation is a method for supervised learning used by NN to update parameters to make the network's predictions more accurate. We have seen how to do this for a single perceptron, however, for multiple layers and nodes things get a bit more complicated. We must do the backward pass (weight adjustment) of all weights in the model, and notice from Figure 3 that these weights might be concatenated inside several activation functions. So to obtain the derivative of each weight with respect to the loss function we must compute the **Chain Rule**.

Figure 3: Backward and Forward Pass



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

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

In this simple case the derivative of h with respect to x would be given by:

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

In a more general sense, if we have a function of the form:

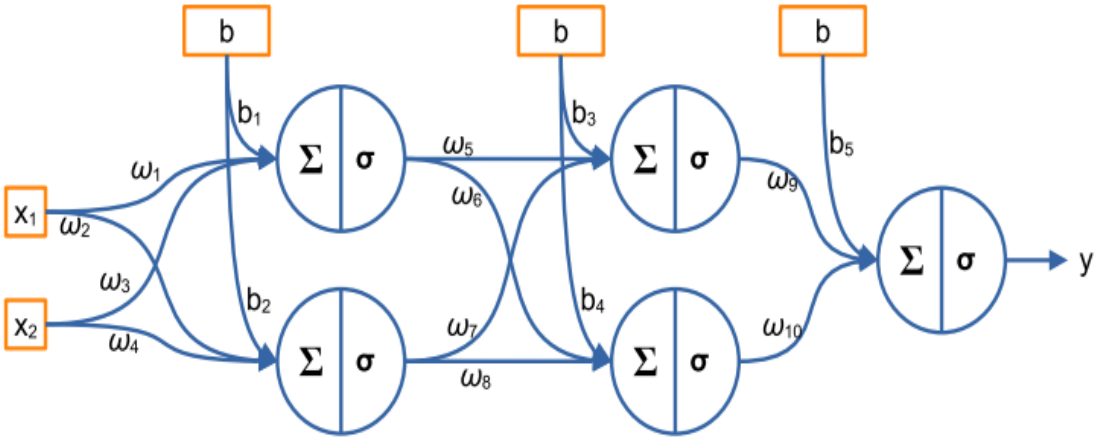
$$f_{1..n} = f_1(f_2(f_3(...f_{n-1}(f_n(x))...)))$$

The derivative would be given by:

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

Now for the context of NN, let's look at Figure 4[3]

Figure 4: Neural Network with Weights



Also, recall that we defined:

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

E is the error function. Notice there are 5 activation functions (2 for each hidden layer and one in the final layer) which we will define as $[a_1, ..., a_5]$. Also, at each node there is a linear combination of the weights and the input data (for the first layer) and then

[3]Image taken from the 3rd set of slides from the Deep Learning course at ITAM's Master in Data Science.

a linear combination of the output of the last layer and their respective weights. Thus the linear combination at node 1 and 3 (for example) would be given by:

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

Given all this, the derivative of the Error with respect to weight 1 (w_i) would be given by (using the Chain Rule):

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

Now you might be asking, why the sum of products? Well, notice that w_1 is so deep in the NN that it actually takes two roads to get to the final prediction. It goes through activation function a_1 and then takes two paths, one goes to s_3 with weight w_5 and the other to s_4 with weight w_6. Thus we must sum each path to obtain the gradient of the weight with respect to the Error. Finally, as seen in the last chapter, weight 1 gets updated in the following form:

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

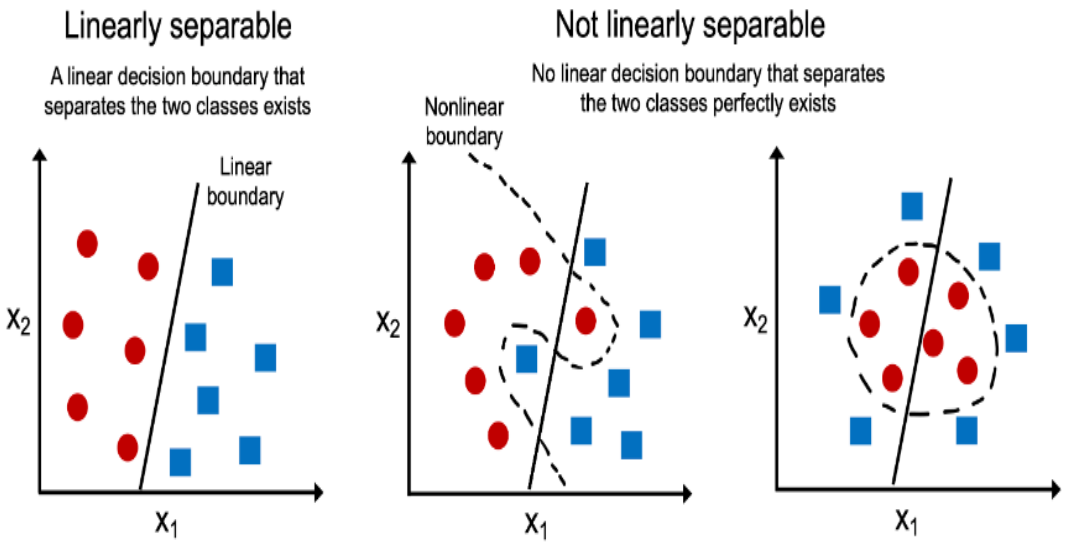
And this happens to all the weights at each time of training t (batch or epoch). A few things to consider about back-prop. The impact of back-prop is proportional to the depth of the layer: weights in shallow layers are updated more softly with respect to those in deeper layers.

1.6 Activation Functions

In a Neural Network framework, 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. The purpose of the activation function is to introduce **non-linearity** into the output of the neuron, which allows the neural network to learn complex patterns and relationships in the data[4].

Figure 5: Linear vs. Non-Linear Problems



The choice of an activation function in the **hidden layer** will control how well the network model learns the training data set[5] The choice of activation function in the **output layer** will define the type of predictions the model can make.

Without an activation function, the output of a neuron would simply be a linear function of its inputs, which would limit the expressive power of the neural network. These are the different types of activation functions used in deep learning:

[4]Image taken from VitalFlux

[5]For more information check Goodfellow, Bengio, and Courville (2016)

Linear Activation Function:

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

Where w_i are the weights and x_i are the data points. This activation function is mostly used at the output layer for unbounded regression, but also works for binary classification.

ReLU (Rectified Linear Unit):

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

This activation function maps its input to the maximum of 0 and the input value. It is computationally efficient and has been shown to work well in many deep learning applications.

Sigmoid:

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

where s is the dot product of w_i and x_i. These activation function is mostly used at the output layer for binary classification problems and regression with $0 \leq y \leq 1$. Their use is discouraged in hidden layers since sigmoidal units saturate across most of their domain, making gradient-based learning very difficult. One can find a more complete explanation of this in this link.

TanH Activation Function:

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

Better than sigmoid in hidden layers since it resembles the identity function more closely, $tan(0) = 0$, which makes training easier and partially solves the vanishing gradient problem[6]

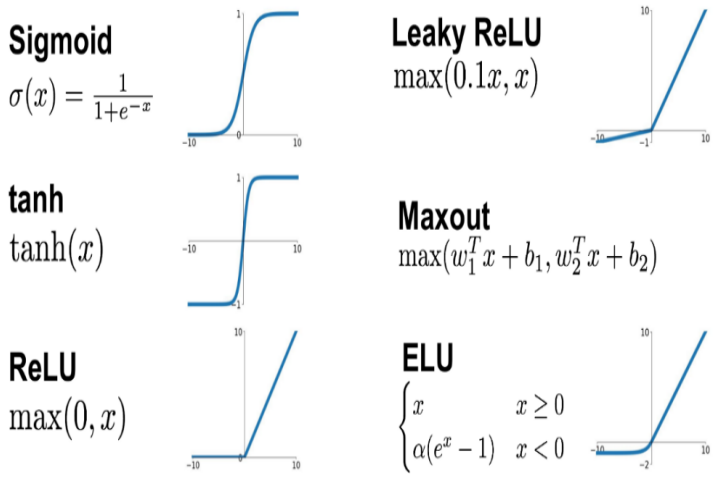
[6]As stated in Goodfellow, Bengio, and Courville (2016), vanishing gradient refers to the fact that in a FNN the back-propagated error signal typically decreases exponentially as a function of the distance

Softmax:

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

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.

Figure 6: Activation Functions Graphically



Choosing the right activation function can have a significant impact on the performance of a neural network, and different activation functions are better suited for different types of tasks and architectures.

1.7 Regularizers

In the context of DL, when training a node one can also deal with 3 particular situations:

from the final layer. It mostly happens with sigmoid and *tanh* activation functions since they saturate at 1 or 0, thus layers deep in the NN fail to receive useful gradient info and are unable to update correctly the weights.

1. **Under-fitting**: training excluded the true data-generating process and induces bias (high training error).

2. Training matched the true data-generating process.

3. **Over-fitting:** included the generating process but also other possible generating processes. Variance rather than bias dominates the estimation error.

Recall that the role of regularization is going from (3) to (2). Out of these strategies the most relevant ones for DL are: Batch Normalization, Dataset Augmentation, Early Stopping, DropOut and the ones we already saw L1 and L2.[7]

Batch Normalization

BatchNorm refers to the application of normalization to hidden layers to accelerate learning and add noise to make the weights more robust. This also induces independence between layers. However, this technique is not used at the final layer, thus is not applicable on our context.

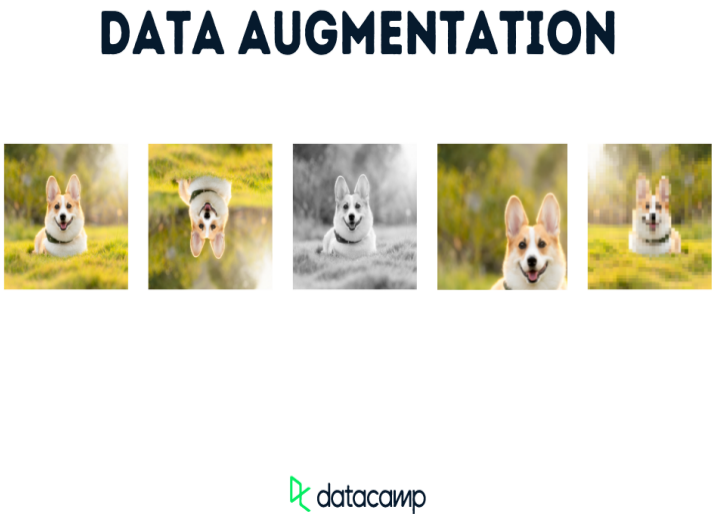
Dataset Augmentation

This technique consists of increasing the dataset by creating fake data and adding it to the training set. It is mostly used for classification tasks since these type of tasks consist of taking a high dimensional X, and summarize within a single category y. Thus needs to be invariant to a wide variety of transformations. This technique does not apply to an NLP context, since it's difficult to think of a simple and adequate way of generating fake texts that are automatically labeled as x or y. A good example of a context where it can be used is in object recognition, since you can rotate, change the color, scale images, to generate new data points.[8]

[7]For a more complete review of these techniques go to Goodfellow, Bengio, and Courville (2016), Zhang et al. (2017) or the following link.

[8]Image taken from DataCamp.

Figure 7



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. Also, stopping the algorithm when no parameters have improved over the best recorded validation error.

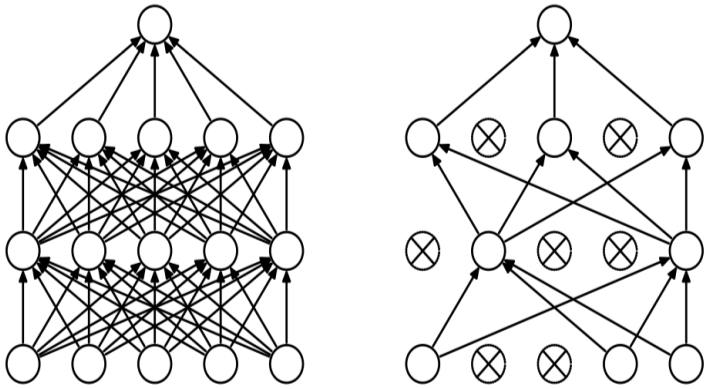
Compared to other regularization techniques, this one is very unobtrusive, since you do not change any of the opt., data pre-processing, etc. However, it can also be seen as a restriction to the optimization procedure to a relatively small volume of parameters' space in the neighborhood of the initial parameter θ_0.

DropOut

Strictly, as stated in Goodfellow, Bengio, and Courville (2016), DropOut trains an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying network. In other words, this technique limits the capacity of the model at random by deactivating neurons (dropout) or weights (drop-connect). One

has to choose the dropout probability, which becomes then another hyper-parameter. This regularizer helps the network be robust against variations, and it is very computationally cheap (Srivastava et al. (2014)), which is needed for complicated models such as BERT. Also, for wider layers, the probability of dropping all paths from inputs to outputs becomes smaller.[9]

Figure 8: DropOut



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

1.8 Example of a NN in Python

In this section we will look at how to build a NN for prediction from scratch using an exercise from the Deep Learning class. We will see how to build a model, define the hyper-parameters, train it and predict with it. For this, we will start using the `keras` and `tensorflow` packages .

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

And we load all the necessary packages and functions.

[9]Image taken from AI Pool.

```
import numpy as np
import matplotlib.pyplot as plt
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 be working with the Iris Data Set, for simplicity. However, we will increase the difficulty of the problem by making it a regression one. We will try to predict the length and width of sepals using the length and width of the petals. As always, we must prepare our data for training.

```
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)
```

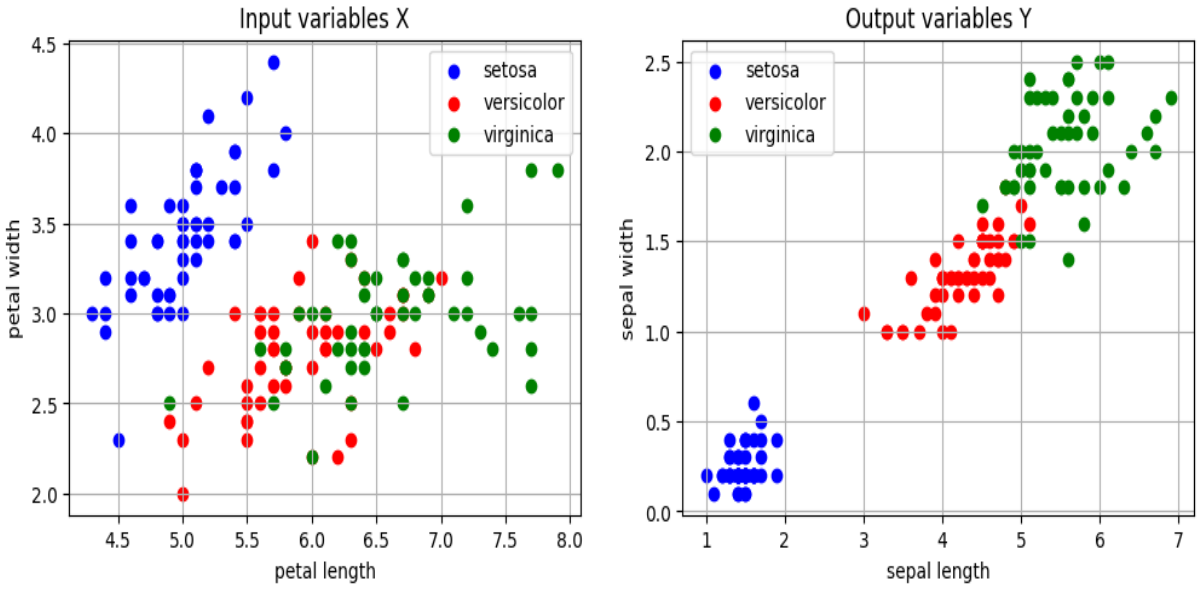
We can graph both our input data and our output data to understand the problem better.

```
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()
```

Notice we are going to find the function that best maps the values of $x = [x_1, x_2]$ to the values of $y = [y_1, y_2]$. Thus the function we are estimating is one that goes from $\mathbb{R}^2$ to $\mathbb{R}^2$.

Figure 9: Distribution of Each Input and Output Variable



Let's start defining our model. We will use `tensorflow`'s functions to start building our own Neural Net. Recall the initial and basic hyper-parameters we must choose: how many hidden layers do we want, how many neurons per layer and what activations do we want at each layer. For this case, we will define a model with 3 hidden layers, 32 neurons in the first and the third layers and 64 in the 2nd one. For activations we will use ReLu for the hidden layer and Linear in the final (recall is the best one for regression problems). We first define the Input layer:

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

Notice we are taking 2 Inputs, since the parameter `shape = (n,)` will always be equal to the number of features you have. We assign it to the object i, since we have to pass it to the next layer as follows:

```
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)
```

Notice we have assigned the number of neurons for each layer using the parameter `units = n` and the activation function for each layer as `activation = 'relu'`[10]. Finally, we aggregate our model using the `Model()` function and obtain an overall description of it:

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

[10]For other activation functions look at the Dense documentation.

Figure 10: Model Description

```
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
_________________________________________________________________
```

Why do we have that many parameters at each layer? Recall that each layer is connected with all of the previous one, and since each layer has a bias, the total number of parameters for each layer is equal to the number of neurons/inputs in the previous layer $\times$ the number of neurons in that layer + the number of neurons in that layer.

Now we can start preparing our model for training, but first, we need to choose another set of hyper-parameters: the **optimization algorithm** 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')
```

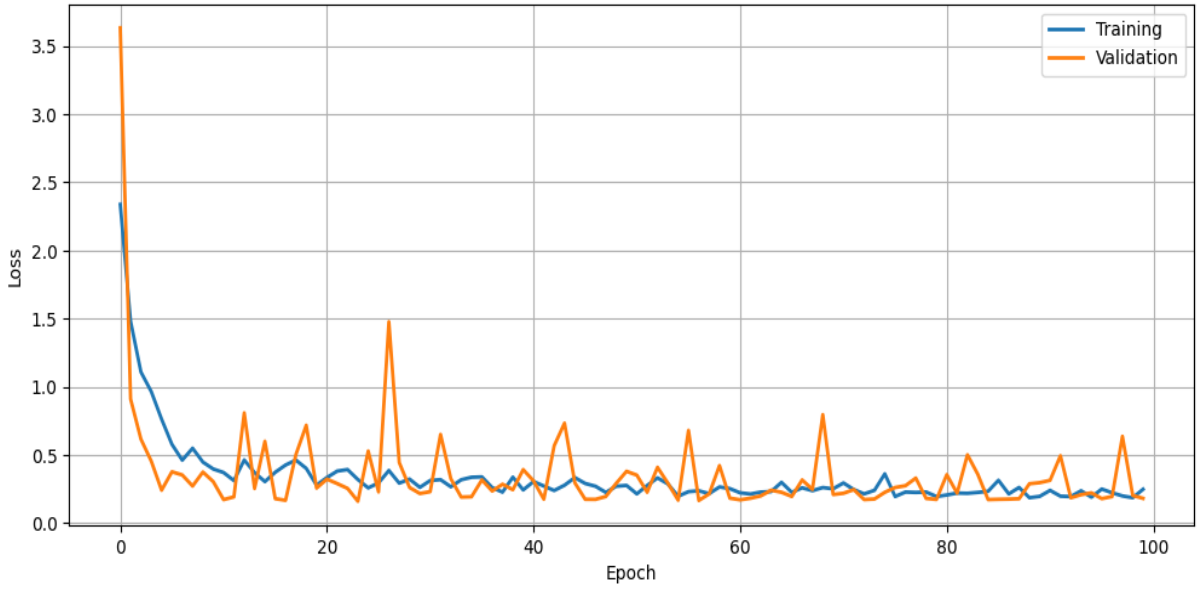
And we can start training it using our data sets. We choose the last two hyper-parameters: batch size and epochs.

```
# 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.

```
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()
```

Figure 11: Metrics Visualization



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. We can conclude that we have a relatively good fit for our task. Now, let's check the final performance of the model:

```
# 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.

References

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf. Springer.

Devlin, Jacob et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv:1810.04805*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2008). *The Elements of Statistical Learning*. https://hastie.su.domains/Papers/ESLII.pdf. Springer Series in Statistics.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv:1502.03167v3*.

Kingma, Diederik P. and Jimmy Ba (2017). "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980v9*.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep Learning". In: *Nature* 521, pp. 436–444.

Reed, Russell and Robert J. Marks (2016). *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press.

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: *arXiv:1609.04747v2*.

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Over-fitting". In: *Journal of Machine Learning Research* 15(1), pp. 1929–1958.

Zhang, Chiyuan et al. (2017). "Understanding Deep Learning Requires Rethinking Generalization". In: *arXiv:1611.03530v2*.