Lecture 10: Fully Connected Neural Networks

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

April 4, 2023

1 Neural Networks

In this chapter[1], we introduce the basics of the neural networks and discuss fully connected neural networks. Nonetheless, we will only be able to scratch the surface, as neural networks cover a wide range of techniques that have applications in many different areas of machine learning. In order to delve deeper into this topic, interested students are encouraged to take a course on deep learning!

1.1 A Brief History of Neural Networks

The concept of neural networks originated from a model of brain function known as "connectionism," which utilized connected circuits to simulate intelligent behavior. In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts portrayed this idea using a simple electrical circuit. Donald Hebb expanded on this idea in his 1949 book, The Organization of Behavior, proposing that neural pathways strengthen over successive use, particularly between neurons that tend to fire at the same time, thus beginning the long journey towards quantifying the complex processes of the brain.

Two significant concepts that precede neural networks are "Threshold Logic," which converts continuous input to discrete output, and "Hebbian Learning," a model of learning based on neural plasticity proposed by Donald Hebb in The Organization of Behavior. It is often summarized by the phrase: "Cells that fire together, wire together." Both of these concepts were proposed in the 1940s. In the 1950s, researchers attempted to translate these networks onto computational systems, and the first Hebbian network was successfully implemented at MIT in 1954.

Around this time, Frank Rosenblatt, a psychologist at Cornell, was studying the comparatively simpler decision systems found in the eyes of flies, which underlie and determine their flee response. In an attempt to understand and quantify this process, he proposed the idea of a Perceptron in 1958, which he called the Mark I Perceptron. It was a system with a straightforward input-output relationship, modeled on a McCulloch-Pitts neuron proposed by Warren S. McCulloch, a neuroscientist, and Walter Pitts, a logician in 1943, to explain the complex decision processes in a brain using a linear threshold gate. A McCulloch-Pitts neuron takes in inputs, takes a weighted sum, and returns "0" if the result is below the threshold and "1" otherwise.

[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

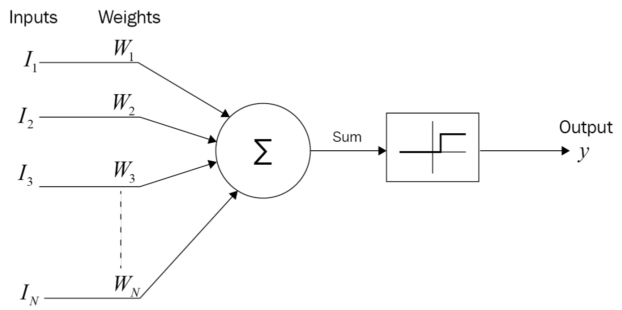


Figure 1: The very first computational model of a neuron, proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943 . For more information, please visit `https://link.springer.com/article/10.1007/bf02478259`.

The Mark I Perceptron was celebrated for its ability to learn through multiple inputs, while minimizing the discrepancy between the desired and actual output. However, it was limited to linearly separable classes, rendering non-linear circuits such as the exclusive-or circuit impossible to learn. Despite this drawback, contemporary artificial neural networks consist of multiple layers of perceptrons. In 1959, Bernard Widrow and Marcian Hoff developed the first neural network, named ADALINE and MADALINE, which addressed noise in phone lines and remains in use to this day. These neural networks differed from perceptrons in terms of their output, which was the weighted input.

The success of these early neural networks sparked considerable excitement and anticipation about the potential of artificial intelligence (AI). [2,3] However, researchers encountered multiple obstacles, including impractically long runtimes and the inability to learn simple boolean exclusive-or circuits. In 1969, Marvin Minsky and Seymour Papert published the book "Perceptrons," which conclusively argued that Rosenblatt's single perception approach to neural networks could not be effectively translated into multi-layered neural networks. Minsky outlined several issues with neural nets, which led the scientific community and funding establishments to the conclusion that further research in this direction would not yield significant results. The funding dried up, and the AI winter began.

The thawing of the AI winter began in 1982, when Jon Hopfield presented his paper on the Hopfield Net.[4] Japan's announcement of its intention to begin its fifth-generation effort on neural networks at the US-Japan conference on Cooperative/Competitive Neural Networks in the same year also helped to reignite interest and funding. The rediscovery of the concept of backpropagation, which attributed reducing significance to each event as one went farther back in the chain of events, was a significant breakthrough. Paul Werbos highlighted their potential in his PhD thesis, but it was not widely recognized until Rumelhart, Hinton, and Williams republished the technique in a clear and detailed framework. The same authors addressed the drawbacks outlined by Minsky in a later text.

Properties of neural networks were then studied by computer scientists, statisticians, and mathematicians extensively. However, after some years the received attentions were diverted to new algorithms at the time (*e.g.* support vector machine, boosting, random forest) some of which outperformed the neural networks designed those days. The reason for this diversion is twofold. First, neural networks needed a lot of fine tuning and tinkering in their early days, whereas other newly invented method were more automatic. The second reason was due to the lack of enough computational power that made neural networks computationally very expensive.

Despite the early disappointment, neural networks resurfaced after 2010 with the new name **deep learning**. Many new architectures and building blocks were invented since then, and these led to a series of remarkable successes for deep learning in problems such as image classification, speech and text recognition, audio

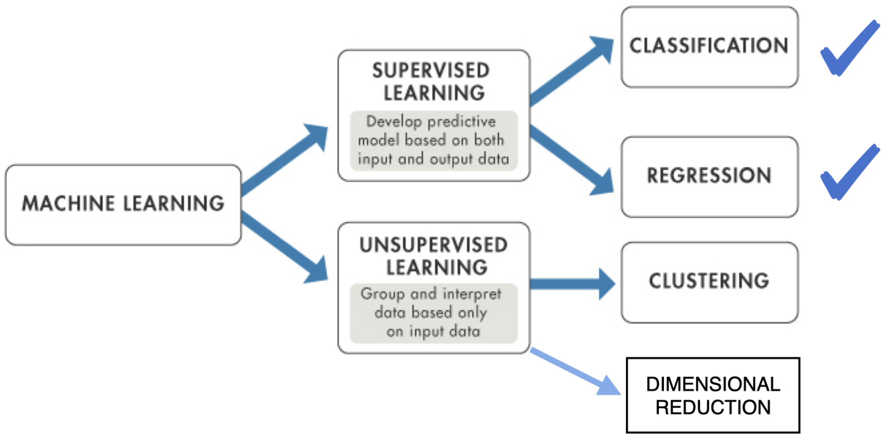
[2] `https://www.nytimes.com/1958/07/08/archives/new-navy-device-learns-by-doing-psychologist-shows-embryo-of.html`

[3] `https://www.youtube.com/watch?v=aygSMgK3BEM`

[4] `http://www.scholarpedia.org/article/Hopfield_network`

processing, time series analysis, and much more. The remarkable accomplishments of deep learning after 2010 were made possible by two major factors, namely the availability of *ever-larger training datasets* due to the use of digitization techniques in industry, and the immense amount of improvement in building more powerful computing units like *graphics and tensor processing units, GPU's and TPU's, respectively*.

Neural networks have applications in all areas of machine learning. In particular, a wide range of deep learning architectures are used in both supervised and unsupervised learnings. In these lectures, we will restrict ourselves to merely some applications of deep learning in the realm of supervised learning.



There are several prominent deep learning architectures used as building blocks to build deep models in supervised learnings. In the course of the next three lectures, we introduce three basic deep learning architectures used in supervised learning. These building blocks are the **fully-connected neural networks**, **convolutional neural networks (CNN)**, and **recurrent neural networks (RNN)**. In today's lecture, we will introduce the fully-connected neural networks, and we will show how they are trained. We do not intend to cover all details and technicalities involved in these architectures. Our treatment should rather be regarded as a soft introduction to the subject!

1.2 Basics of Neural Networks

To illustrate how neural networks work, we start with a simple case. To set the stage, assume that we have d continuous features represented by $\vec{x} \in \mathbb{R}^d$, and a continuous target variable $y \in \mathbb{R}$. A neural network constructs a *nonlinear* model $f(\vec{x})$ to predict the target y. We have already constructed many nonlinear models such as polynomial regressors, tree regressors, random forest regressors, bagged regressors, and boosted regressors. What differentiates a neural network from the previously established models is *the way the network is structured*. Neural networks are structured in a specific way and nonlinearity is introduced in the model in a peculiar way. The following picture depicts a simple *fully connected feed-forward neural network* with only one hidden layer.

Every neural network possesses an input and an output layer. In the input layer (the green layer in Fig. 2), the features of the model are fed into the model. In the output layer (the blue layer in Fig. 2), the model offers its prediction for the target variable. The layers that are placed in between of the input and output layers are called *hidden layers* (the purple layer in Fig. 2). A neural network may have one or many hidden layers (the network depicted in Fig. 2 possesses only one hidden layer). Each hidden layer consists of a number of units (the purple circles). Each unit of a hidden layer is called a **neuron**. For example, in Fig. 2, the network has one hidden layer consisting 5 neurons.

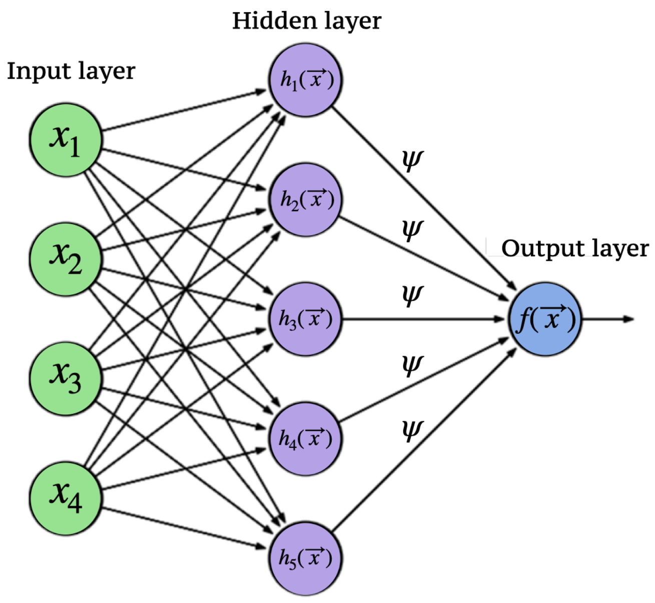


Figure 2: A single layer neural network.

Remark: Note that the term feed-forward network means that all neurons feed their output forward to the next (hidden or output) layer, without any connections feeding to the same or the previous layer.

Remark: The term fully-connected implies that each neuron in a hidden layer has a connection with (*i.e.* receives contribution from) **all** neurons in the previous layer.

Question: How does a neural network introduce nonlinearity into the model?

Answer: Any hidden layer - after receiving the output of its previous layer as a linear combination of the pervious neurons - applies an activation function (*i.e.* a nonlinear function) to the result. Later, we will present a list of most common activation functions used in deep learning.

Now, we need to elaborate on the answer to the previous question, and illustrate how a fully connected feed-forward neural network predicts the target. For simplicity assume that we have only one hidden layer with N neurons. The predicted value of the target variable y is given by the following formula

$$\begin{aligned}\hat{y}^{(i)} = f(\vec{x}^{(i)}) &= \alpha_0 + \sum_{j=1}^{N} \alpha_j h_j(\vec{x}^{(i)}) \\ &= \alpha_0 + \sum_{j=1}^{N} \alpha_j\, \psi\big(\omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_k^{(i)}\big) \,.\end{aligned} \tag{1}$$

As is evident from (1), each neuron h_j is constructed as a linear combination of *all* constituents of the previous layer (in this case a linear combination of all features, as the previous layer is the input layer). Then the activation function ψ is applied to the result of each neuron, and eventually a linear combination of contributions coming from *all* neurons (after the activation ψ is applied) determines the predicted value of the target $\hat{y}^{(i)}$ for the i-th sample. Before we discuss how the above neural network is trained, let us answer two important questions.

Question: What are the parameters of the above model? How many parameters does the above fully connected neural network possess?

Answer: The parameters of the model are the coefficients $\alpha = (\alpha_0, \alpha_1, \cdots, \alpha_N)$ and ω_{jk} that can be cast in the following $N \times (d+1)$ matrix

$$\mathbb{W} = \begin{pmatrix} \omega_{10} & \omega_{11} & \omega_{12} & \cdots & \omega_{1d} \\ \omega_{20} & \omega_{21} & \omega_{22} & \cdots & \omega_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_{N0} & \omega_{N1} & \omega_{N2} & \cdots & \omega_{Nd} \end{pmatrix}. \tag{2}$$

The rows of matrix $\mathbb{W}$ in (2) refers to different neurons, and the columns of ω refer to the d features and the bias terms. In other words, ω_{jk} with $k \geq 1$ is the weight the j-th neuron receives from the k-th feature x_k. In summary, the above fully connected neural network with a single hidden layer possesses

$$\text{Total number of weights } = N(d+1) + N + 1 = N(d+2) + 1 \tag{3}$$

parameters. As is clear from this formula, each new neuron will introduce $d+2$ more weights.

Remark: In the context of deep learning, the parameters of neural network models are referred to as the **weights** of the model.

Question: What activation functions are commonly used to build neural networks?

Answer: In principle, there are no limitations in using different functions as the activations associated with hidden layers. One even has the option to make new activation functions and use them to construct new neural networks. However, some of the most common activation functions used in deep learning are

Activation Function	**Definition**
Sigmoid	$\sigma(x) = \dfrac{1}{1+e^{-x}}$
Rectified Linear Unit	$ReLU(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$
Leaky Rectified Linear Unit	$ReLU_\alpha(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}$
Hyperbolic Tangent	$\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
Sigmoid Linear Unit	$SiLU(x) = x\,\sigma(x) = \dfrac{x}{1+e^{-x}}$
Arc-tangent	$\arctan(x) = \tan^{-1}(x)$

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def sigmoid(x):                  # Define sigmoid function
    return 1/(1 + np.exp(-x))

def relu(x):                     # Define relu function
    if x<0:
        y = 0
    else:
        y = x
    return y

def leakyrelu(x, alpha):            # Define leaky relu function
    if x<0:
        y = alpha*x
    else:
        y = x
    return y

def tanh(x):                        # Define tanh function
    return (np.exp(x)-np.exp(-x))/(np.exp(x) + np.exp(-x))

def silu(x):                        # Define silu function
    return x*sigmoid(x)

def arctan(x):                      # Define arc-tangent function
    return np.arctan(x)

x_vals = np.linspace(-6, 6, 200)
func_range = {'Sigmoid': sigmoid(x_vals), 'ReLU': np.array([relu(a) for a in x_vals]),
              'LeakyReLU': np.array([leakyrelu(a, 0.1) for a in x_vals]), 'tanh':␣
 ↪tanh(x_vals),
              'SiLU': silu(x_vals), 'Arctan': arctan(x_vals)}


fig, axs = plt.subplots(2, len(func_range)//2, figsize=(16, 8), tight_layout=True)
for j in range(len(func_range)):
    x_arr = x_vals
    y_arr = func_range[list(func_range.keys())[j]]
    colors = ['royalblue', 'red', 'green', 'crimson', 'salmon', 'orange']
    sns.lineplot(ax=axs[j//3, j%3], x=x_arr, y=y_arr, color=colors[j], label='%s(x)'␣
 ↪%(list(func_range.keys())[j]))
    axs[j//3, j%3].set_title('Graph of %s Function' %(list(func_range.keys())[j]))
plt.show()
```

It should be noted that the role of activation functions in a neural network is essential. If one introduces multiple hidden layers without applying any activations, the whole model would be *equivalent to a linear model with one single layer*. This fact relies on a famous theorem from linear algebra as follows.

Fact from Linear Algebra: The composition $\mathcal{S} \circ \mathcal{T} : A \to C$ of two linear transformations $\mathcal{T} : A \to B$ and $\mathcal{S} : B \to C$, where A, B, and C are vector spaces, is a linear transformation.

Another important role of activation functions is producing complex interactions between the features. To illustrate how this phenomenon occurs, consider the following very elementary example. Suppose we have only two features $\vec{x} = (x_1, x_2)$, and we design a neural network with only one hidden layer with two neurons. We use the activation $\psi(x) = \arctan(x)$. Consider the following values for the weights of this simple model

$$\alpha = (\alpha_0, \alpha_1, \alpha_2) = (0, \frac{1}{2}, \frac{1}{2}), \qquad \mathbb{W} = \begin{pmatrix} \omega_{10} & \omega_{11} & \omega_{12} \\ \omega_{20} & \omega_{21} & \omega_{22} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & -1 \end{pmatrix}. \tag{4}$$

Then, the corresponding neural network predicts the target variable through the following formula

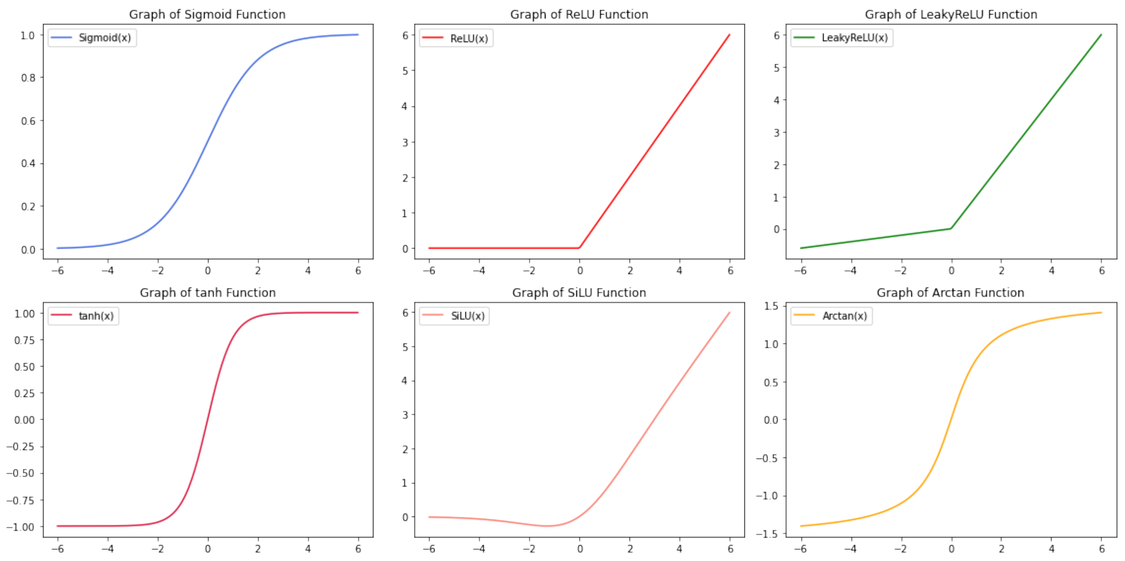


Figure 3: Visualizations of 6 commonly used activation functions.

$$\begin{aligned}\hat{y} = f(\vec{x}) &= \alpha_0 + \sum_{j=1}^{2} \alpha_j \, \psi\big(\omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_j^{(i)}\big) \\ &= \frac{1}{2}\,\psi(x_1 + x_2) + \frac{1}{2}\,\psi(x_1 - x_2) = \frac{1}{2}\big(\arctan(x_1 + x_2) + \arctan(x_1 - x_2)\big) \\ &= x_1 - x_1 x_2^2 - \frac{x_1^3}{3} + 2\, x_1^3 x_2^2 + x_1 x_2^4 + \frac{x_1^5}{5} + \cdots \end{aligned} \tag{5}$$

In getting the last line of (5), we used the Taylor expansion of the arc-tangent function: $\arctan(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1}$. As is clear from the last line of (5), the application of the activation function generates interesting interaction terms between features. Without applying a nonlinear activation function, no interaction terms between the features exist.

We can readily generalize the above structure of neural networks to designs models wherein we have **multiple hidden layers**. Suppose we have d continuous features, $\vec{x} \in \mathbb{R}^d$, and one continuous target variable $y \in \mathbb{R}^n$. We design a fully connected feed-forward neural network with ℓ hidden layers. Each hidden layer i consists of N_i neurons. Figure 4 displays the structure of a fully-connected neural network with many hidden layers schematically.

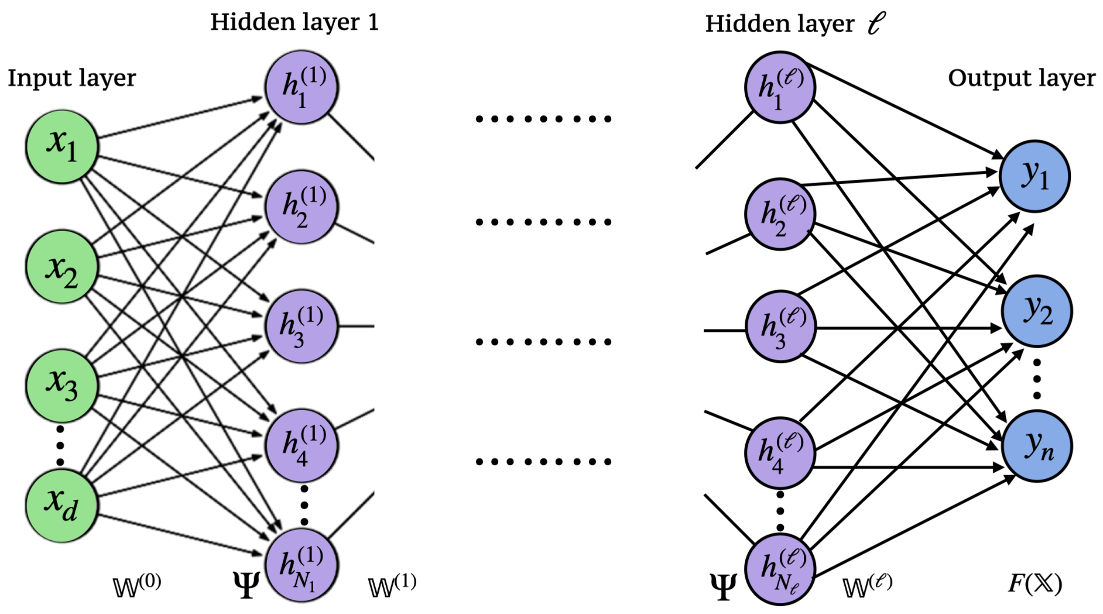


Figure 4: A fully connected neural network with ℓ hidden layers.

Then, the prediction of the this neural network for the target variable y is given by

$$\hat{\mathbb{Y}} = F(\mathbb{X}) = \underbrace{\mathbb{W}^{(\ell)}\Psi\Big(\mathbb{W}^{(\ell-1)}\Psi\cdots\Psi\big(\mathbb{W}^{(1)}\Psi(}_{\ell \text{ times}}\mathbb{W}^{(0)}\mathbb{X}^{\mathsf{T}})\big)\cdots\Big) = \Big(\prod_{j=1}^{\ell}\mathbb{W}^{(j)}\circ\Psi\Big)\big(\mathbb{W}^{(0)}\mathbb{X}^{\mathsf{T}}\big)\,, \tag{6}$$

where the weight matrix $\mathbb{W}^{(i)}$, for $0 \le i \le \ell$, is an $N_{i+1} \times (N_i + 1)$ matrix that encodes the weights of the i-th hidden layer

$$\mathbb{W}^{(i)} = \begin{pmatrix} \omega_{10}^{(i)} & \omega_{11}^{(i)} & \omega_{12}^{(i)} & \cdots & \omega_{1N_i}^{(i)} \\ \omega_{20}^{(i)} & \omega_{21}^{(i)} & \omega_{22}^{(i)} & \cdots & \omega_{2N_i}^{(i)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_{N_{i+1}0}^{(i)} & \omega_{N_{i+1}1}^{(i)} & \omega_{N_{i+1}2}^{(i)} & \cdots & \omega_{N_{i+1}N_i}^{(i)} \end{pmatrix}\,, \tag{7}$$

where $N_0 = d$ and $N_{\ell+1} = n$. In Eq. (6), Ψ acts on a given matrix component-wise (*i.e.* the action of Ψ on a matrix is the same as acting the activation function ψ on all elements of the matrix). In principle, one can even apply different activation functions for different hidden layers. This can be easily accommodated in Eq. (6) by considering a subscript for each activation function (*i.e.* replace Ψ in the right hand side of (6) by $\Psi^{(j)}$). A short analysis reveals that the total number of weights of the model is given by

$$\text{Total number of weights } = \sum_{i=0}^{\ell} N_{i+1}(N_i + 1)\,, \tag{8}$$

In order to obtain a sense about the typical number of parameters involved a neural network, let us set $d = 10$, $n = 1$, and assume we consider 3 hidden layers each with 256 neurons (a very common size for a hidden layer in deep learning). Then, according to (8), the total number of parameters of such neural network is given by

$$\begin{aligned}\text{Total number of weights } &= \sum_{i=0}^{3} N_{i+1}(N_i+1) \\ &= 256\times(10+1)+256\times(256+1)+256\times(256+1)+1\times(256+1) \\ &= 134657\,.\end{aligned} \tag{9}$$

As observed above, even the simplest neural networks involve a large number of parameters. This implies that neural networks possess a *large number of degrees of freedom*, and hence, they can potentially *suffer from an overfitting problem*.

1.3 Training Neural Networks

In the previous sub-section, we explained the basics of fully-connected feed-forward neural networks. We saw how a fully-connected neural network is constructed, and how the model makes predictions for the target variable. In this section, we briefly demonstrate how neural networks are trained. Due to the large of number of parameters (weights) a neural network involves, the adoption of a feasible approach in training neural networks is necessary. This approach takes the advantage of a practical optimization algorithm, known as the **Gradient Descent**. Gradient descent is a first-order iterative optimization algorithm which has many variations in deep learning. In here, we briefly explain the simplest form of the gradient descent algorithm. The goal of the gradient descent algorithm is *to find the local minima of a differentiable (multivariable) function*. The gradient descent approach is based on a simple fact from multivariable calculus:

A Fact from Multivariable Calculus: Let $f(\vec{x})$ be a (first order) differentiable function of n variables $\vec{x} = (x_1, x_2, \cdots, x_n)$, and P be a point in the domain of f in an n-dimensional Euclidean space (*i.e.* $\vec{x}_P \in \mathbb{R}^n$). Then,

- f increases fastest if one goes from P in the direction $\vec{\nabla} f\big|_P$ (*i.e.* gradient of f evaluated at P).
- f decreases fastest if one goes from P in the opposite direction direction of $\vec{\nabla} f\big|_P$ (*i.e.* one moves in the direction $-\vec{\nabla} f\big|_P$).

To minimize a multivariable function f, the gradient descent algorithm starts from a random point P in the domain of the function. In the next step, gradient descent calculates the gradient of f, and it moves by a small step in the opposite direction of the gradient of f (*i.e.* in the direction $-\vec{\nabla} f\big|_P$) to get to point Q. At Q, the algorithm repeats the same steps. It calculate the gradient of f at Q, and moves in the opposite direction of the gradient by the same small step to get to the next point. This process is repeated until the gradient of f is sufficiently small (*i.e.* close to 0). At this point the process stops, and the final point obtained at the end of the process is guaranteed to be sufficiently close to a local minima of f. The following picture depicts how gradient descent operates on a function in two variables.

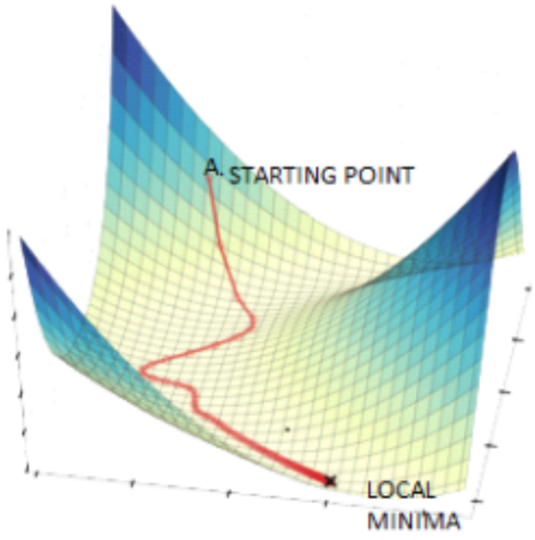


Figure 5: Gradient descent operating on a function with two variables.

Remark: In the context of machine learning, the small step in gradient descent by which one moves in the opposite direction of the gradient is called the **learning rate**.

In order to find the optimal values of the weights of a neural network, an appropriate cost function is optimized. It should be noted that the cost function in the context of deep learning is referred to as a **loss function**. Given a loss function $\mathcal{L}$, the optimal values for weights Ω are found by minimizing the loss function

$$\underset{\Omega}{\text{argmin}} \sum_{i=1}^{N_{train}} \mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = \underset{\Omega}{\text{argmin}} \sum_{i=1}^{N_{train}} \mathcal{L}(y^{(i)}, f_\Omega(\vec{x}^{(i)})) \,, \tag{10}$$

through the training process, where $f_\Omega(\vec{x}^{(i)})$ is the prediction of the neural network for the i-the sample and N_{train} is the number of samples of the training dataset. Throughout the optimization process of (10), the parameters at each step k are updated as follows

$$\Omega \longleftarrow \Omega - \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \vec{\nabla}_\Omega \mathcal{L}(y^{(i)}, f_\Omega(\vec{x}^{(i)})) \,. \tag{11}$$

Note that the minus sign on the right hand side of EQ. (11) implies that *one is moving in the opposite direction of the gradient of the loss function in the parameter space (This means that one is moving toward a local minimum of the loss function)*. If the target variable of the neural network is continuous, a typical choice (there are several choices) for the loss function is the usual *MSE* (mean squared error) function. On the other hand, if the target variable of the neural network is a categorical variable, then an appropriate choice for the loss function would be the *cross-entropy* function.

Remark: The learning rate is a hyperparameter of neural networks that needs to be chosen appropriately. By very small learning rates, one may not be able to get to the vicinity of a local minimum, whereas very large learning rates may pass the local minima! Typical values for the learning rates are $\eta = 0.01$ and $\eta = 0.001$. The following image depicts the problem with too large and too small learning rates schematically.

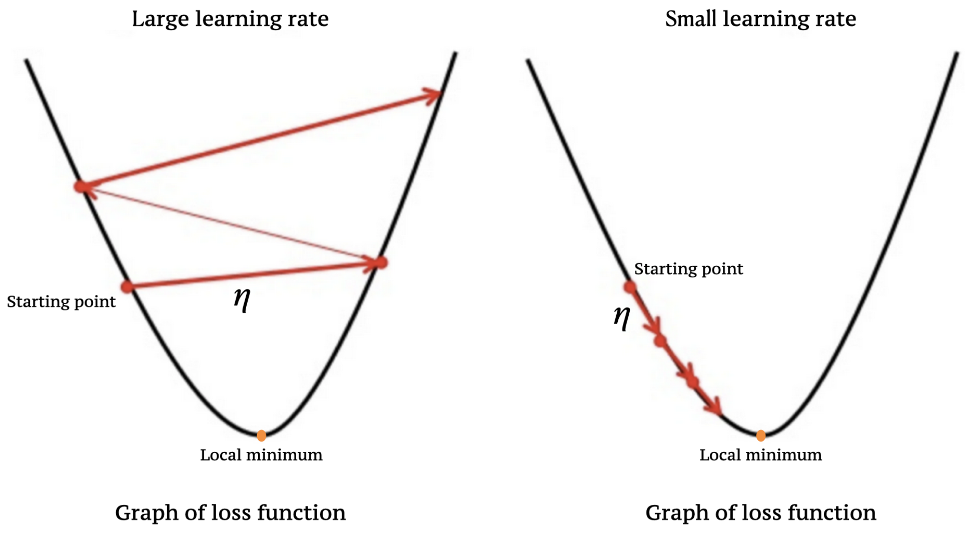


Figure 6: Learning with a large and small learning rate.

Side Remark: Note that in general the loss function is a non-convex function of the weights of the neural network. This implies that the solution to the minimization problem of the loss function may not be unique, and there may exist multiple local minima for the loss function.

Important Remark: Due to the high number of weights a typical neural network involves, neural networks are in danger of suffering from an overfitting problem. To reduce the chance of overfitting, the train data is fit to the model in an iterative process with small steps. Whenever signs of high variance are detected, the training is stopped. This procedure is usually referred to as the **slow learning**. Another method to deal with overfitting in neural networks is to take the advantage of *regularization methods* we introduced earlier (*e.g.* L_1 and L_2 regularizations).

Question: The execute the gradient descent algorithm, we *need a starting point* (*i.e.* point P). How is the starting point in the parameter space picked?

Answer: The gradient descent algorithm that is built in deep learning models automatically picks a *random point* in the parameter space. This means that the coordinates of the starting point (*i.e.* initial values of the parameters of the model) are typically picked from a Gaussian distribution with mean 0.

1.3.1 Backpropagation

As demonstrated above, the core of the gradient descent algorithm is to move in the *opposite direction of the gradient* (of the function being minimized) by small iterative steps. The key question is now how the gradients are calculated? Given the complexity of neural networks, how are the gradients practically calculated? Despite the enormous complexity that a given neural network may involve, the calculation of gradients follow a simple and straightforward rule of calculus, namely the **chain rule**.

To show how the calculation of the gradients of the loss function is implemented, we consider a very simple neural network, namely a neural network with d features $\vec{x} \in \mathbb{R}^d$, a single continuous target variable $y \in \mathbb{R}$, and only one hidden layer h consisting of N neurons with activation ψ. The prediction of the model is given by function f given in equation (1). This neural network has two sets of weights, namely α_j and ω_{jk}. We indicate these $N(d+2)+1$ weights collectively by Ω. Suppose, we choose the loss function $\mathcal{L}$ to be the *MSE* function. Then

$$\begin{aligned}\mathcal{L}(\Omega) = \sum_{i=1}^{N_{train}} \mathcal{L}^{(i)}(\Omega) = \frac{1}{2}\sum_{i=1}^{N_{train}} (y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{N_{train}} \left(y^{(i)} - f_\Omega(\vec{x}^{(i)})\right)^2 \\ = \frac{1}{2}\sum_{i=1}^{N_{train}} \left[y^{(i)} - \left(\alpha_0 + \sum_{j=1}^{N} \alpha_j\, \psi(z_j^{(i)})\right)\right]^2 ,\end{aligned} \tag{12}$$

where $z_j^{(i)} = \omega_{j0} + \sum_{k=1}^{d} \omega_{jk} x_k^{(i)}$.

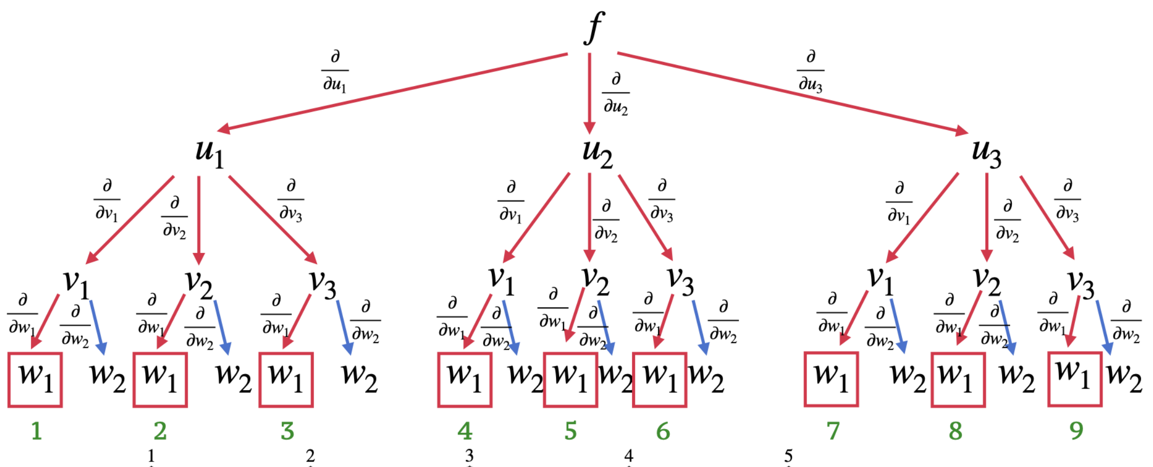
The gradient of the loss function consists of two sets of derivatives:

$$\vec{\nabla}_\Omega \mathcal{L}(\Omega) = \langle \partial_{\alpha_i}\mathcal{L}, \partial_{\omega_{jk}}\mathcal{L}\rangle , \qquad \text{for } i,j = 0,1,\cdots N, \quad \text{and } k = 0,1,\cdots d . \tag{13}$$

In order to calculate the partial derivatives in (13), we need to use the *chain rule* of the multivariable calculus. Using the chain rule, the partial derivatives of the loss function can be easily computed as follows:

$$\begin{aligned}
\partial_{\alpha_0}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \alpha_0} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \alpha_0} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})}\frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \alpha_0} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)}) ,\\
\partial_{\alpha_j}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \alpha_j} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \alpha_j} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})}\frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \alpha_j} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)})\, \psi(z_j^{(i)}) ,\\
\partial_{\omega_{j0}}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \omega_{j0}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \omega_{j0}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})}\frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \psi(z_j^{(i)})}\frac{\partial \psi(z_j^{(i)})}{\partial z_j^{(i)}}\frac{\partial z_j^{(i)}}{\partial \omega_{j0}} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)})\, \alpha_j\, \psi'(z_j^{(i)}) ,\\
\partial_{\omega_{jk}}\mathcal{L}(\Omega) &= \frac{\partial \mathcal{L}}{\partial \omega_{jk}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial \omega_{jk}} = \sum_{i=1}^{N_{train}} \frac{\partial \mathcal{L}^{(i)}}{\partial f_\Omega(\vec{x}^{(i)})}\frac{\partial f_\Omega(\vec{x}^{(i)})}{\partial \psi(z_j^{(i)})}\frac{\partial \psi(z_j^{(i)})}{\partial z_j^{(i)}}\frac{\partial z_j^{(i)}}{\partial \omega_{jk}} = -\sum_{i=1}^{N_{train}} f_\Omega(\vec{x}^{(i)})\, \alpha_j\, \psi'(z_j^{(i)})\, x_k^{(i)} .
\end{aligned} \tag{14}$$

Equation (14) shows how the gradient of the loss function is computed as the sum of individual contributions from the samples. To calculate the contribution of each sample, the chain rule of the multivariable calculus has to be applied. The process of calculating all necessary partial derivatives of the loss function using the chain rule is known as the **backpropagation** in the context of deep learning. Backpropagation is a precise bookkeeping tool to calculate all necessary partial derivatives of the loss function with respect to various weights of the neural network. The core of this bookkeeping device for a simple case has been illustrated in the following picture.



f
u_1
u_2
u_3
v_1
v_2
v_3
w_1
w_2
$\frac{\partial}{\partial u_1}$
$\frac{\partial}{\partial u_2}$
$\frac{\partial}{\partial u_3}$
$\frac{\partial}{\partial v_1}$
$\frac{\partial}{\partial v_2}$
$\frac{\partial}{\partial v_3}$
$\frac{\partial}{\partial w_1}$
$\frac{\partial}{\partial w_2}$
1
2
3
4
5
6
7
8
9

$$\frac{\partial f}{\partial w_1} = \overbrace{\frac{\partial v_1}{\partial w_1}\frac{\partial u_1}{\partial v_1}\frac{\partial f}{\partial u_1}}^{1} + \overbrace{\frac{\partial v_2}{\partial w_1}\frac{\partial u_1}{\partial v_2}\frac{\partial f}{\partial u_1}}^{2} + \overbrace{\frac{\partial v_3}{\partial w_1}\frac{\partial u_1}{\partial v_3}\frac{\partial f}{\partial u_1}}^{3} + \overbrace{\frac{\partial v_1}{\partial w_1}\frac{\partial u_2}{\partial v_1}\frac{\partial f}{\partial u_2}}^{4} + \overbrace{\frac{\partial v_2}{\partial w_1}\frac{\partial u_2}{\partial v_2}\frac{\partial f}{\partial u_2}}^{5}$$
$$+ \underbrace{\frac{\partial v_3}{\partial w_1}\frac{\partial u_2}{\partial v_3}\frac{\partial f}{\partial u_2}}_{6} + \underbrace{\frac{\partial v_1}{\partial w_1}\frac{\partial u_3}{\partial v_1}\frac{\partial f}{\partial u_3}}_{7} + \underbrace{\frac{\partial v_2}{\partial w_1}\frac{\partial u_3}{\partial v_2}\frac{\partial f}{\partial u_3}}_{8} + \underbrace{\frac{\partial v_3}{\partial w_1}\frac{\partial u_3}{\partial v_3}\frac{\partial f}{\partial u_3}}_{9}$$

Figure 7: An illustration of chain rule.

In above, f is a function of u_1, u_2, and u_3 which themselves are functions of v_1, v_2, and v_3. Finally, v_1, v_2, and v_3 are functions of w_1 and w_2. Hence, f is implicitly a function of w_1 and w_2. This structure is depicted by a tree where the original function f is at the top of the tree, and the implicit variables u_1, u_2, and u_3 are branches originating from f. In order to find the partial derivative of f with respect to w_1, one has to consider all paths in the tree that start from f and end at a w_1 leaf. In this example, there are 9 such paths, and the partial derivative of f with respect to w_1 is the sum of all 9 terms. As indicated in the above picture, each arrow represents a partial derivative of the function at the origin of the arrow with respect to the variable appearing at the end of the arrow.

2 Coding for FC Neural Networks

In here, we present a number of examples of fully connected neural networks. As in the theory part, we will not take a systematic approach toward the implementation and execution of neural networks. We rather show how simple neural networks can be built and executed using python libraries.

There are two major python libraries for constructing deep learning models. These two libraries are pytorch and tensorflow. We will use both libraries in here so that you will get a chance to see how each one of them works. We do not aim for a comprehensive treatment of the two libraries, as this goes beyond the scope of this introductory course. Interested readers should consider taking a course on deep learning to learn more about these libraries.

2.1 PyTorch vs. Keras-TensorFlow

PyTorch and Keras-TensorFlow are two of the most widely used deep learning frameworks. They both have their own unique features and strengths, which make them popular among researchers, developers, and data scientists.

PyTorch is an open-source machine learning framework that is widely used for deep learning applications. It was developed by Facebook's AI Research team and is known for its ease of use, flexibility, and dynamic computational graph construction. On the other hand, Keras-TensorFlow is a high-level API that runs on top of TensorFlow. Keras-TensorFlow provides a user-friendly interface to build and train neural networks.

PyTorch is known for its simplicity and ease of use. Its dynamic computational graph construction allows users to easily modify and debug their models. Moreover, PyTorch provides an intuitive interface for defining and manipulating tensors, which are the basic building blocks of deep learning models. On the other hand, Keras-TensorFlow is also easy to use but provides less flexibility than PyTorch. Keras-TensorFlow is built on top of TensorFlow and follows a static graph construction, which makes it less flexible than PyTorch.

PyTorch has a reputation for providing fast performance due to its dynamic graph construction. PyTorch uses a technique called autograd to automatically compute gradients, which results in faster computation times. Moreover, PyTorch allows users to leverage the power of GPUs, which can significantly speed up deep learning models. Keras-TensorFlow also provides fast performance, but it may not be as fast as PyTorch due to the static graph construction of TensorFlow.

In terms of community support, both PyTorch and Keras-TensorFlow have large communities of users who contribute to the development and improvement of the frameworks. PyTorch has a large community of researchers and data scientists who use it for various deep learning applications. Keras-TensorFlow has a large community of developers who use it for building and deploying production-grade deep learning models.

As far as the availability of pre-trained models is concerned, PyTorch has a large collection of pre-trained models, including models for image classification, object detection, and natural language processing. PyTorch also provides a user-friendly interface for fine-tuning pre-trained models on custom datasets. Keras-TensorFlow also provides a wide range of pre-trained models, including models for image classification, object detection, and speech recognition.

The choice between PyTorch and Keras-TensorFlow depends on the user's specific needs and preferences.

2.1.1 PyTorch and Tensors

In PyTorch, tensors are the basic building blocks of deep learning models. Tensors are multi-dimensional arrays that can be used to represent various types of data, including images, text, and numerical data. PyTorch provides a user-friendly interface for defining and manipulating tensors, which makes it easy for researchers and data scientists to build deep learning models.

To create a tensor in PyTorch, you can use the torch.tensor() function. This function takes a list or a NumPy array as input and returns a PyTorch tensor. For example,

[1]:
```
import torch

# create a tensor from a list
my_list = [1, 2, 3]
my_tensor = torch.tensor(my_list)

# create a tensor from a NumPy array
import numpy as np
my_array = np.array([4, 5, 6])
my_tensor = torch.tensor(my_array)
```

PyTorch tensors can be manipulated using a variety of operations, including arithmetic operations, matrix operations, and comparison operations. PyTorch provides many built-in functions to perform these operations. For example,

[2]:
```
# create two tensors
a = torch.tensor([1, 2, 3])
```

```
b = torch.tensor([4, 5, 6])

# perform arithmetic operations
c = a + b
d = a - b
e = a * b
f = a / b

# perform matrix operations
g = torch.dot(a, b)
h = torch.mm(a.reshape(1, 3), b.reshape(3, 1))

# perform comparison operations
i = (a > b)
j = (a == b)
```

PyTorch tensors can also be used to represent gradients in deep learning models. PyTorch uses a technique called autograd to automatically compute gradients. When a tensor is created with the requires_grad parameter set to True, PyTorch tracks all the operations performed on the tensor and creates a computational graph. This computational graph is then used to compute the gradients of the loss function with respect to the tensor. For example,

[3]:
```
# create a tensor with requires_grad set to True
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# define a function
def y_fn(x):
    y = x**2 + 2*x + 1
    return y

# compute the output and the gradient
y = y_fn(x)
y.backward()

# print the gradient
print(x.grad)
```

In this example, we create a tensor x with requires_grad set to True. We then define a function y_fn that depends on x. We compute the output y and call the backward() function to compute the gradients of y with respect to x. Finally, we print the gradients of x.

2.1.2 Keras-TensorFlow

- Layers are the basic building blocks of neural networks in Keras. A layer consists of a tensor-in tensor-out computation function (the layer's call method) and some state, held in TensorFlow variables (the layer's weights).
- The Sequential model API is a way of creating deep learning models where an instance of the Sequential class is created and model layers are created and added to it.
- There are various types of layers in Keras, e.g., core layers (input, output, dense layer, embedding layer, masking layer), convolution layers, pooling layers, recurrent layers, normalization layers, regularization layers, attention layers, reshaping layers, merging layers, etc. We will talk about some of these layers in the following chapters/lectures.

- In this lecture, we will only use the dense layer. Keras Dense layer means that every neuron in the dense layer takes the input from all the other neurons of the previous layer.