03-Deep-Learning

March 1, 2021

1 Introduction To Deep Learning

Deep learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised.

Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, machine vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.

Artificial neural networks (ANNs) were inspired by information processing and distributed communication nodes in biological systems. ANNs have various differences from biological brains. Specifically, neural networks tend to be static and symbolic, while the biological brain of most living organisms is dynamic and analog.

```
[1]: from IPython.display import Image
     from IPython.core.display import HTML
     Image(filename= "./img/ann1.jpeg",width=500, height=500)
```

```
[1]:
```

- items
- colors

If you have a text, you wan't to learn

- syntax
- semantic
- grammar
- etc

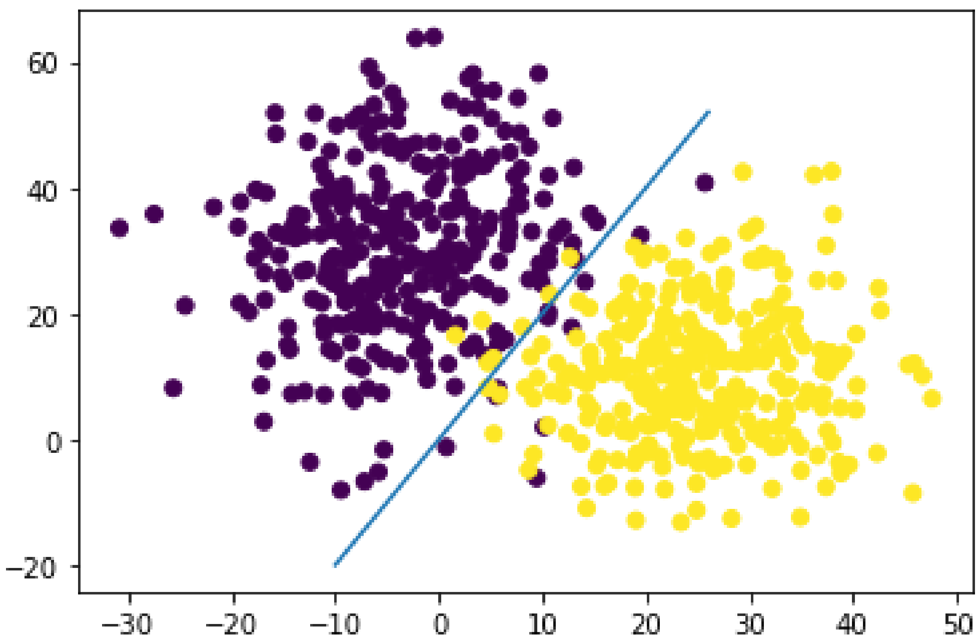
In deep learning, each level learns to transform its input data into a slightly more abstract and composite representation. In an image recognition application, the raw input may be a matrix of pixels; the first representational layer may abstract the pixels and encode edges; the second layer may compose and encode arrangements of edges; the third layer may encode a nose and eyes; and the fourth layer may recognize that the image contains a face. Importantly, a deep learning process can learn which features to optimally place in which level on its own. (Of course, this does not completely eliminate the need for hand-tuning; for example, varying numbers of layers and layer sizes can provide different degrees of abstraction.)

Classifying those kind of features oftenly non-linear. Some of Machine Learning algorithms ;like Kernel Support Vector Machines, Random Forest, KNN, are able to learn non-linear features of data. But in various tasks, Artificial Neural Networks are more effective.

What does 'non-linear' mean? We have seen Logistic Regression classifier. Logistic Regression classifier is a linear classifier as seen:

[11]:

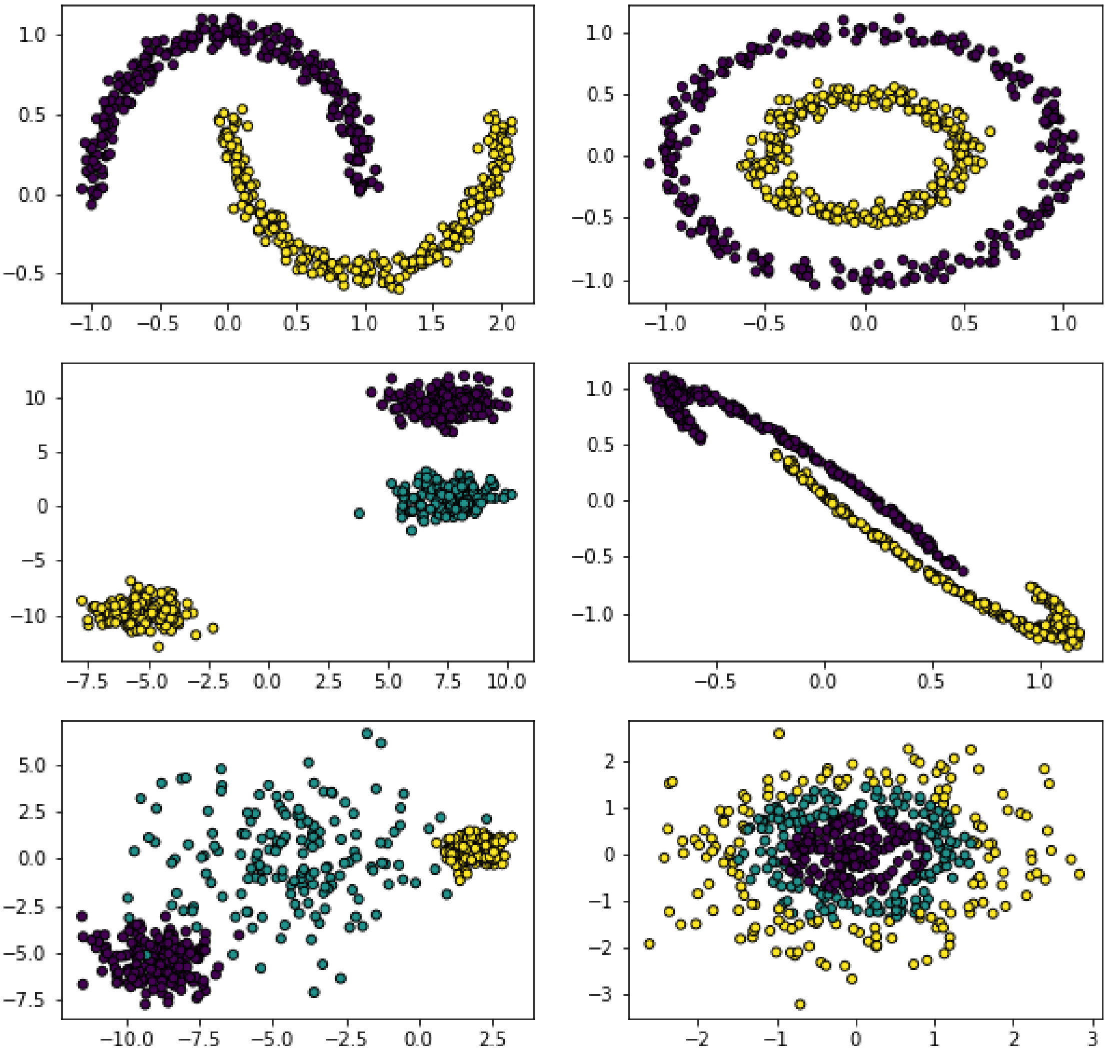
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
x_1 = np.concatenate([np.random.normal(-3, 8, 300),np.random.normal(25, 9,␣
 ↪300)])
x_2 = np.concatenate([np.random.normal(30, 13, 300), np.random.normal(11, 11,␣
 ↪300)])
y = np.concatenate([np.zeros(300),np.ones(300)])
df = pd.DataFrame({'x_1':x_1,'x_2':x_2,'y':y})
plt.scatter(df['x_1'],df['x_2'],c=df['y']);
plt.plot(np.array([point for point in range(-10,27)]),2*np.array([point for␣
 ↪point in range(-10,27)]));
```



Let’s see a non-linear data classifier.

[12]:
```
from sklearn import cluster, datasets
import matplotlib.pyplot as plt

n_samples = 500
X_noisy_circles, Y_noisy_circles = datasets.make_circles(n_samples=n_samples,␣
 ↪factor=.5,
                                      noise=.05)
X_noisy_moons, Y_noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.
 ↪05)
X_blobs, Y_blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)

transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X_noisy_moons, transformation)
Y_aniso = Y_noisy_moons.copy()

X_varied, Y_varied = datasets.make_blobs(n_samples=n_samples,
                             cluster_std=[1.0, 2.5, 0.5],
                             random_state=170)
X_gaussv, Y_gaussv = datasets.make_gaussian_quantiles(n_samples=n_samples)

fig, axs = plt.subplots(3, 2,figsize=(10,10))
axs[0,0].scatter(X_noisy_moons[:, 0], X_noisy_moons[:, 1], marker='o',␣
 ↪c=Y_noisy_moons,
            s=25, edgecolor='k');
```

```
#ax = plt.subplot(2,4,2)
axs[0,1].scatter(X_noisy_circles[:, 0], X_noisy_circles[:, 1], marker='o',␣
 ↪c=Y_noisy_circles,
            s=25, edgecolor='k');
#ax = plt.subplot(2,4,3)
axs[1,0].scatter(X_blobs[:, 0], X_blobs[:, 1], marker='o', c=Y_blobs,
            s=25, edgecolor='k');
#ax = plt.subplot(2,4,4)

axs[1,1].scatter(X_aniso[:, 0], X_aniso[:, 1], marker='o', c=Y_aniso,
            s=25, edgecolor='k');

axs[2,0].scatter(X_varied[:, 0], X_varied[:, 1], marker='o', c=Y_varied,
            s=25, edgecolor='k');

axs[2,1].scatter(X_gaussv[:, 0], X_gaussv[:, 1], marker='o', c=Y_gaussv,
            s=25, edgecolor='k');
```



1.1 Some History

History

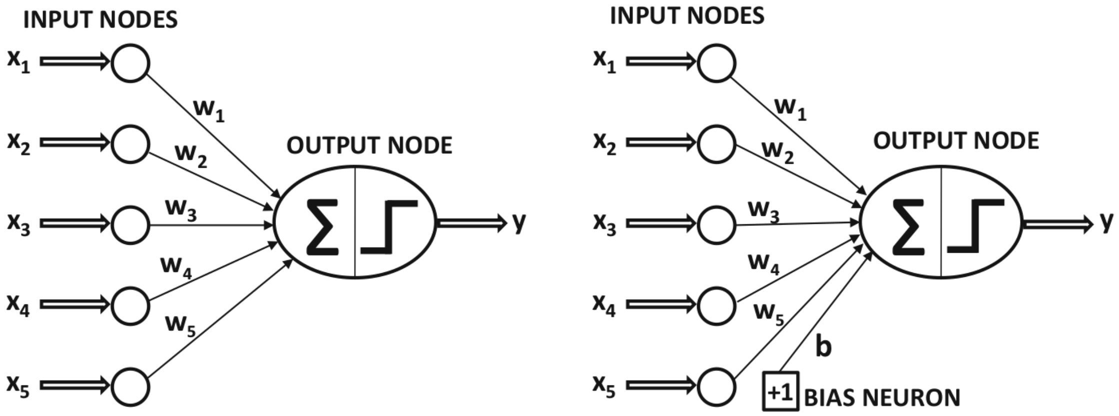
1.2 Single Layer Perceptrons

A single layer perceptron is base architecture behind deep learning model, it can be seen as a shallow neural network.

[1]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/perceptron1.png",width=700, height=700)
```

[1]:



(a) Perceptron without bias

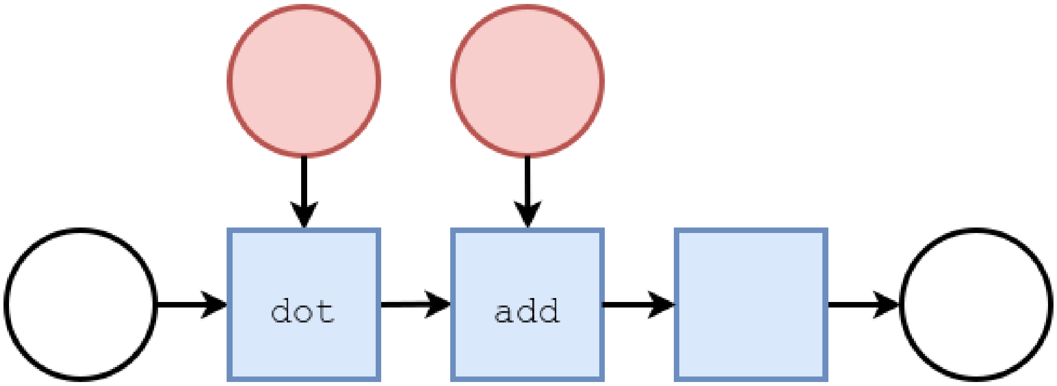
(b) Perceptron with bias

Figure 1.3: The basic architecture of the perceptron

[3]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/mlp1.png",width=700, height=700)
```

[3]:



Let’s see how to compute output from input.

The input matrix is defined as:

$$X = \begin{bmatrix} x_1^1 & x_2^1 & \cdots & x_n^1 \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \vdots & \cdots & \vdots \\ x_1^m & x_2^m & \cdots & x_n^m \end{bmatrix} \quad \in \mathbb{R}^{m \times n}$$

And the weight matrix is defined as:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad \in \mathbb{R}^{n \times 1}$$

and the loss function defined as:

$$L(\hat{y}, y) = \frac{1}{2m} \sum (y - \hat{y})^2$$

to calculate the output, we need to do a matrix multiplication over input matrix and weight matrix, then add the bias term.

$$Xw = \begin{bmatrix} x_1^1 & x_2^1 & \cdots & x_n^1 \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \vdots & \cdots & \vdots \\ x_1^m & x_2^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} + b$$

$$= \begin{bmatrix} b + w_1 x_1^1 + \cdots + w_n x_n^1 \\ b + w_1 x_1^2 + \cdots + w_n x_n^2 \\ \vdots \\ b + w_1 x_1^m + \cdots + w_n x_n^m \end{bmatrix} \quad \in \mathbb{R}^{m \times 1}$$

After that we will get $\hat{y}$ when we input Xw to sigmoid function $\sigma()$

$$\sigma(Xw) = \frac{1}{1+\exp(-Xw)} = \hat{y}$$

$$= \begin{bmatrix} \hat{y}^1 \\ \hat{y}^2 \\ \vdots \\ \hat{y}^m \end{bmatrix}$$

We call this procedure as ‘Forward Propagation’ or ‘Feed Forwarding’.

One can train model’s weigths using gradient descent algorithm. The loss function

$$L(\hat{y}, y) = \frac{1}{2m} \sum (y - \sigma(Xw + b))^2$$

where y is

$$\begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^m \end{bmatrix}$$

The derivation will be

$$\nabla_w L(\hat{y}, y) = -\frac{1}{m} \sum (y - \sigma(Xw))(1 - \sigma(Xw))X$$

Then the update with gradient descent will be

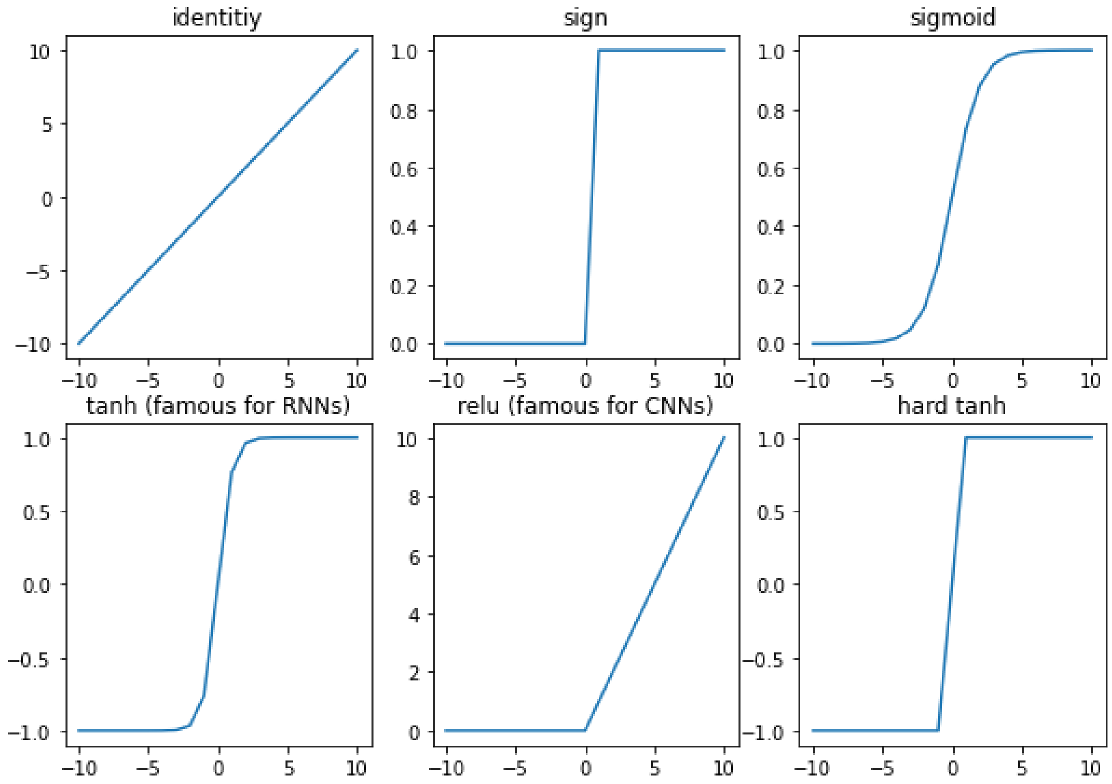
```
$ for i = 0 to epochs $
  $\hat{y} = \sigma(Xw)$
  $w = w - \eta\nabla_w L(\hat{y}, y)$
END
```

1.3 Activation Functions

[44]:

```
import numpy as np
import matplotlib.pyplot as plt
fig, axs = plt.subplots(2,3,figsize = (10,7))
x = range(-10,11)

identity_y = np.array(x)
sign_y = np.array([1 if point>0 else 0 for point in x ])
sigmoid_y = np.array(1/(1+np.exp(-np.array(x))))
```

```
tanh_y = np.array((np.exp(2 * np.array(x)) - 1)/(np.exp(2* np.array(x)) + 1))
relu_y = np.array([0 if point<0 else point for point in x])
hardtanh_y = np.array([maxp if maxp>-1 else -1 for maxp in [point if point<1 else 1 for point in np.array(x)]])


axs[0,0].plot(x,identity_y)
axs[0,0].set_title('identitiy')
axs[0,1].plot(x,sign_y)
axs[0,1].set_title('sign')
axs[0,2].plot(x,sigmoid_y)
axs[0,2].set_title('sigmoid')
axs[1,0].plot(x,tanh_y)
axs[1,0].set_title('tanh (famous for RNNs)')
axs[1,1].plot(x,relu_y)
axs[1,1].set_title('relu (famous for CNNs)')
axs[1,2].plot(x,hardtanh_y)
axs[1,2].set_title('hard tanh');
```



$$sign(z) = \begin{cases} -1, & \text{if } z < 0 \\ 1, & \text{else if } z > 0 \end{cases}$$

$$sigmoid(z) = \frac{1}{1 + \exp(-z)}$$

$$tanh(z) = \frac{\exp(2z) - 1}{\exp(2z) + 1} = 2sigmoid(2 * z) - 1$$

$$ReLU(z) = \max(z, 0)$$

$$hardtanh(z) = \max(\min(z, 1), -1)$$

[ ]:

```
```

1.4 Multilayer Perceptrons

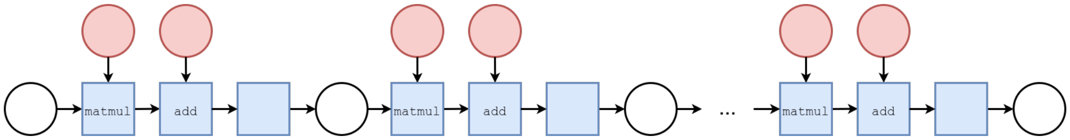
[49]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/perceptron2.png",width=1200, height=1200)
```

[5]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/mlp2.png",width=1200, height=1200)
```

[5]:



The single layer perceptron is very useful for classifying data sets that are linearly separable. They encounter serious limitations with data sets that do not conform to this pattern as discovered with the non-linear data.

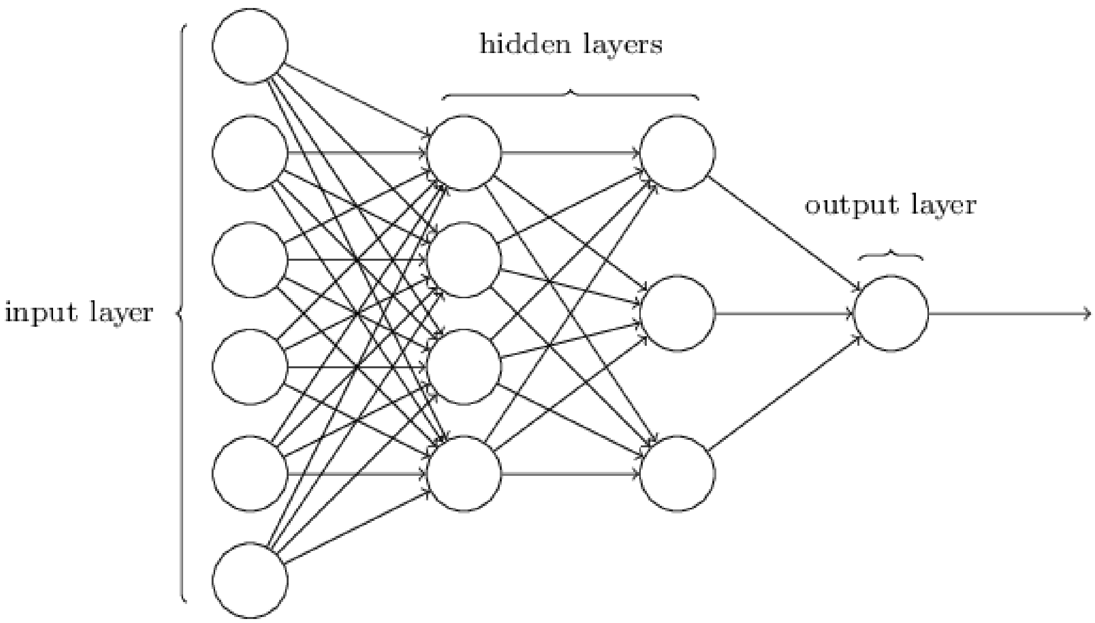
The MultiLayer Perceptron (MLPs) breaks this restriction and classifies datasets which are not linearly separable. They do this by using a more robust and complex architecture to learn regression and classification models for difficult datasets.

Let's compute the forward propagation on this multi layer network:

[51]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/perceptron3.png",width=600, height=600)
```

[51]:



We have 3 different weight matrices that are w_1, w_2, w_3 and bias terms b_1, b_2, b_3. Based on above network, we have 6 features in input data which is $n = 6$ and m samples.

$$X = \begin{bmatrix} x_1^1 & x_2^1 & x_3^1 & x_4^1 & x_5^1 & x_6^1 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 & x_6^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1^m & x_2^m & x_3^m & x_4^m & x_5^m & x_6^m \end{bmatrix} \quad \in \mathbb{R}^{\gtrdot \times \ntriangleleft}$$

$$w_1 = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 & w_4^1 & w_5^1 & w_6^1 \\ w_1^2 & w_2^2 & w_3^2 & w_4^2 & w_5^2 & w_6^2 \\ w_1^3 & w_2^3 & w_3^3 & w_4^3 & w_5^3 & w_6^3 \\ w_1^4 & w_2^4 & w_3^4 & w_4^4 & w_5^4 & w_6^4 \end{bmatrix} \in \mathbb{R}^{\ntrianglerighteq \times \ntriangleleft}$$

$$w_2 = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 & w_4^1 \\ w_1^2 & w_2^2 & w_3^2 & w_4^2 \\ w_1^3 & w_2^3 & w_3^3 & w_4^3 \end{bmatrix} \in \mathbb{R}^{\nVDash \times \ntrianglerighteq}$$

$$w_3 = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 \end{bmatrix} \in \mathbb{R}^{\nVDash \times \nVDash}$$

Let's calculate hidden layers:

- h_1: dim = $(m \times 6) \times (6 \times 4) = (m \times 4)$

$$z_1 = X w_1^t + b_1$$

$$h_1 = \sigma(z_1)$$

- h_2: dim $= (m \times 4) \times (4 \times 3) = (m \times 3)$

$$z_2 = h_1 w_2^T + b_2$$

$$h_2 = \sigma(z_2)$$

- $\hat{y}$: $(m \times 3) \times (3 \times 1) = (m \times 1)$

$$z_3 = h_2 w_3^T + b_3$$

$$\hat{y} = h_3 = \sigma(z_3)$$

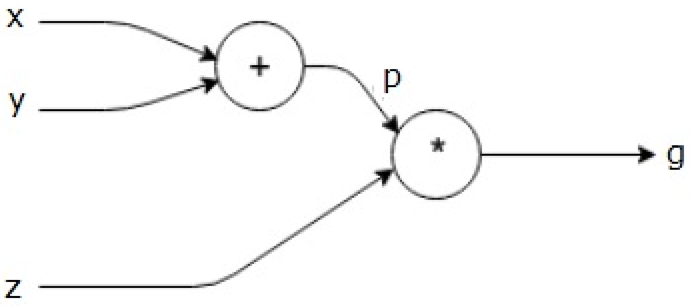
2 Computational Graphs And Backpropagation Algorithm

2.1 Defining Computational Graph

[2]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/graph1.jpg",width=500, height=500)
```

[2]:



A computational graph is defined as a directed graph where the nodes correspond to mathematical operations. Computational graphs are a way of expressing and evaluating a mathematical expression. Above computational graph corresponds:

$$p = x + y$$

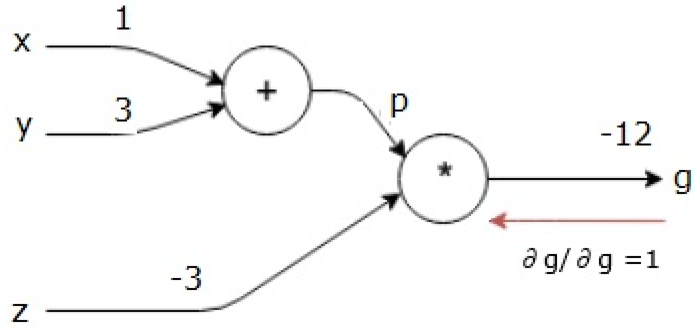
$$g = p \times z$$

Also derivatives can be expressed with computational graphs.

[3]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/graph2.jpg",width=500, height=500)
```

[3]:



$$\frac{\partial g}{\partial g} = 1$$

$$\frac{\partial g}{\partial z} = p = 4$$

$$\frac{\partial g}{\partial p} = z = -3$$

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial x}$$

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial y}$$

$$p = x + y \Rightarrow \frac{\partial p}{\partial x} = 1, \frac{\partial p}{\partial y} = 1$$

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial x} = (-3)\,.1 = -3$$

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial y} = (-3)\,.1 = -3$$

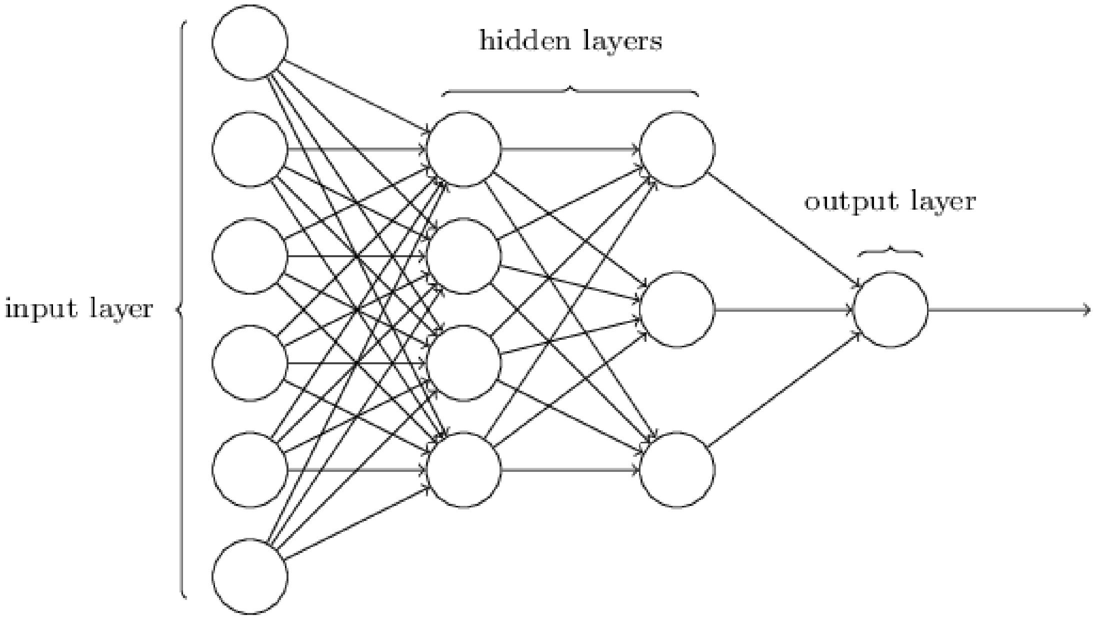
More details for automatic differentation and computational graphs are throughout the lecture. But you can read about it click the link.

2.2 Learning Parameters With Backpropagation Algortihm

Lets learn weights of this network

[1]:
```python
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/perceptron3.png",width=600, height=600)
```

[1]:



We defined our network

$$X = \begin{bmatrix} x_1^1 & x_2^1 & x_3^1 & x_4^1 & x_5^1 & x_6^1 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 & x_6^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1^m & x_2^m & x_3^m & x_4^m & x_5^m & x_6^m \end{bmatrix} \quad \in \mathbb{R}^{\gtrdot\times\ntriangleleft}$$

$$w_1 = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 & w_4^1 & w_5^1 & w_6^1 \\ w_1^2 & w_2^2 & w_3^2 & w_4^2 & w_5^2 & w_6^2 \\ w_1^3 & w_2^3 & w_3^3 & w_4^3 & w_5^3 & w_6^3 \\ w_1^4 & w_2^4 & w_3^4 & w_4^4 & w_5^4 & w_6^4 \end{bmatrix} \in \mathbb{R}^{\ntrianglerighteq\times\ntriangleleft}$$

$$w_2 = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 & w_4^1 \\ w_1^2 & w_2^2 & w_3^2 & w_4^2 \\ w_1^3 & w_2^3 & w_3^3 & w_4^3 \end{bmatrix} \in \mathbb{R}^{\nVdash\times\ntrianglerighteq}$$

$$w_3 = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 \end{bmatrix} \in \mathbb{R}^{\nVdash\times\nVDash}$$

We have:

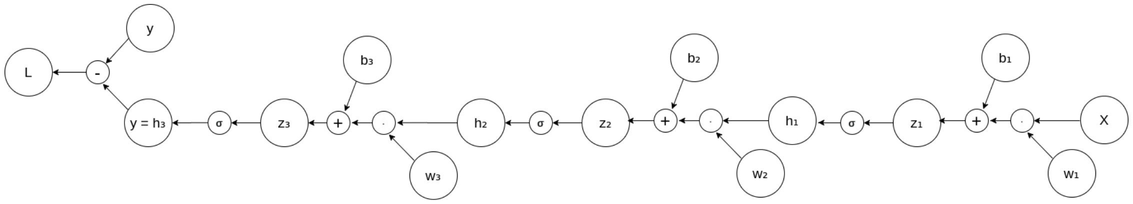
- $$z_1 = Xw_1^t + b_1$$
- $$h_1 = \sigma(z_1)$$
- $$z_2 = h_1 w_2^T + b_2$$
- $$h_2 = \sigma(z_2)$$
- $$z_3 = h_2 w_3^T + b_3$$
- $$\hat{y} = h_3 = \sigma(z_3)$$
- $$L(\hat{y}, y) = \frac{1}{2m} \sum (y - \hat{y})^2$$

Based on this formulation, our computational graph will be

[13]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/derivgraph.png",width=1500, height=1500)
```

[13]:

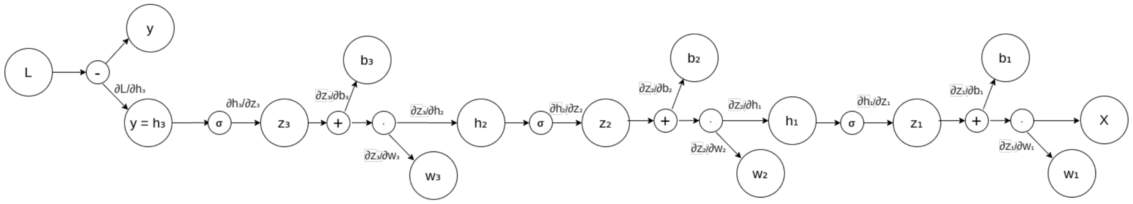


And the partial derivatives based on this computational graph is

[11]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/derivgraph2.png",width=1500, height=1500)
```

[11]:



y
L
-
$\partial L/\partial h_3$
b_3
$\partial z_3/\partial b_3$
b_2
$\partial z_2/\partial b_2$
b_1
$\partial z_1/\partial b_1$
$\partial h_3/\partial z_3$
$\partial z_3/\partial h_2$
$\partial h_2/\partial z_2$
$\partial z_2/\partial h_1$
$\partial h_1/\partial z_1$
$y = h_3$
σ
z_3
+
·
h_2
σ
z_2
+
·
h_1
σ
z_1
+
·
X
$\partial z_3/\partial w_3$
$\partial z_2/\partial w_2$
$\partial z_1/\partial w_1$
w_3
w_2
w_1

So our partial derivatives respect to weights and biases are

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial b_3}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1}$$

From forward propagation formulations, we can easily compute the partials

- $$z_1 = Xw_1^T + b_1$$
- $$h_1 = \sigma(z_1)$$
- $$z_2 = h_1 w_2^T + b_2$$
- $$h_2 = \sigma(z_2)$$
- $$z_3 = h_2 w_3^T + b_3$$
- $$\hat{y} = h_3 = \sigma(z_3)$$
- $$L(\hat{y}, y) = \frac{1}{2m} \sum (y - \hat{y})^2$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3} = \overbrace{\underbrace{-\frac{1}{m} \sum (y - \hat{y})}_{\nabla_{h_3} L} \cdot \underbrace{\sigma(z_3) \cdot (1 - \sigma(z_3))}_{\sigma'(z_3)}}^{\delta_1} \cdot \underbrace{h_2}_{\nabla_{w_3} z_3}$$

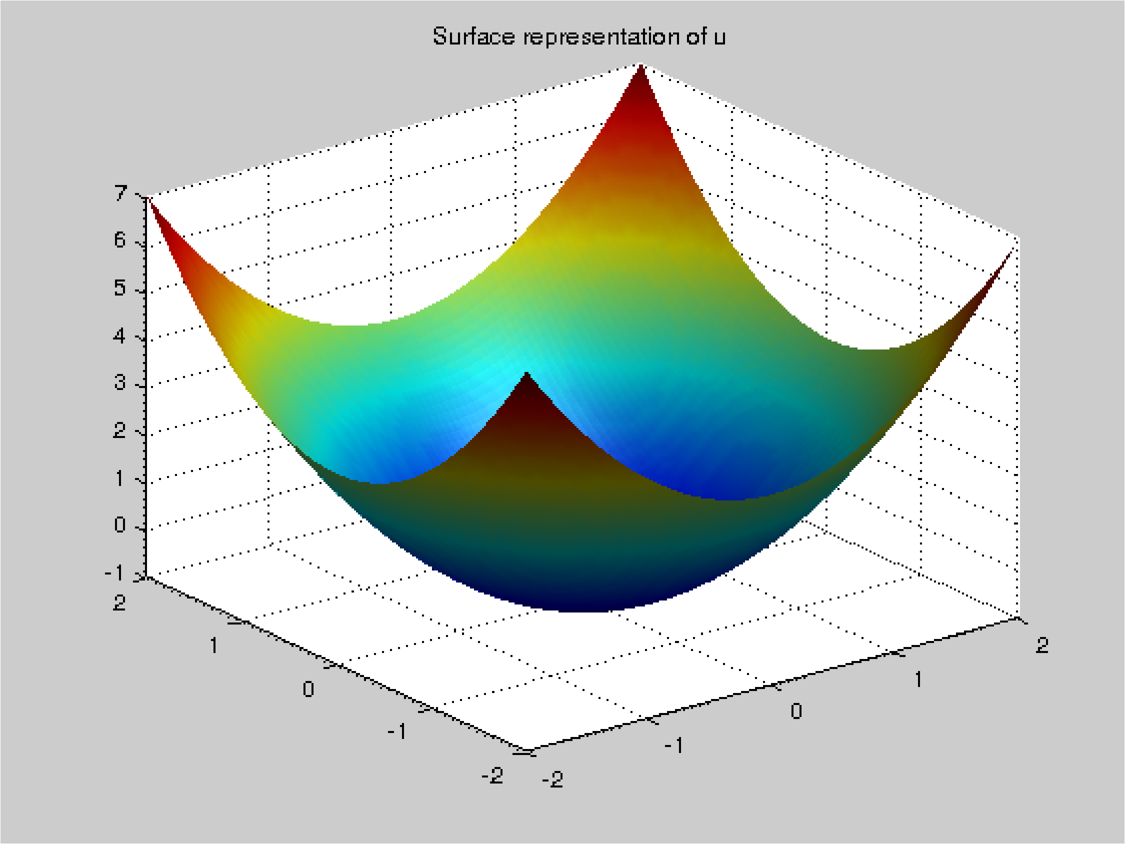
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial h_3}\cdot\frac{\partial h_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial h_2}\cdot\frac{\partial h_2}{\partial z_2}\cdot\frac{\partial z_2}{\partial w_2} = \overbrace{\overbrace{\underbrace{-\frac{1}{m}\sum(y-\hat{y})}_{\nabla_{h_3}L}\cdot\underbrace{\sigma(z_3)\cdot(1-\sigma(z_3))}_{\sigma'(z_3)}}^{\delta_1}\underbrace{w_3^T}_{\nabla_{h_2}z_3}\cdot\underbrace{\sigma(z_2)\cdot(1-\sigma(z_2))}_{\sigma'(z_2)}}^{\delta_2}\cdot\underbrace{h_1}_{\nabla_{w_2}z_2}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h_3}\cdot\frac{\partial h_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial h_2}\cdot\frac{\partial h_2}{\partial z_2}\cdot\frac{\partial z_2}{\partial h_1}\cdot\frac{\partial h_1}{\partial z_1}\cdot\frac{\partial z_1}{\partial w_1} = \overbrace{\underbrace{-\frac{1}{m}\sum(y-\hat{y})}_{\nabla_{h_3}L}\cdot\underbrace{\sigma(z_3)\cdot(1-\sigma(z_3))}_{\sigma'(z_3)}\underbrace{w_3^T}_{\nabla_{h_2}z_3}\cdot\underbrace{\sigma(z_2)\cdot(1-\sigma(z_2))}_{\sigma'(z_2)}}^{\delta_2}\cdot\underbrace{w_2^T}_{\nabla_{h_1}z_2}\cdot$$
$$\underbrace{\sigma(z_1)\cdot(1-\sigma(z_1))}_{\sigma'(z_1)}\cdot\underbrace{X}_{\nabla_{w_1}z_1}$$

One can easily see the repetition over gradients by looking at deltas $\delta_{1,2}$.

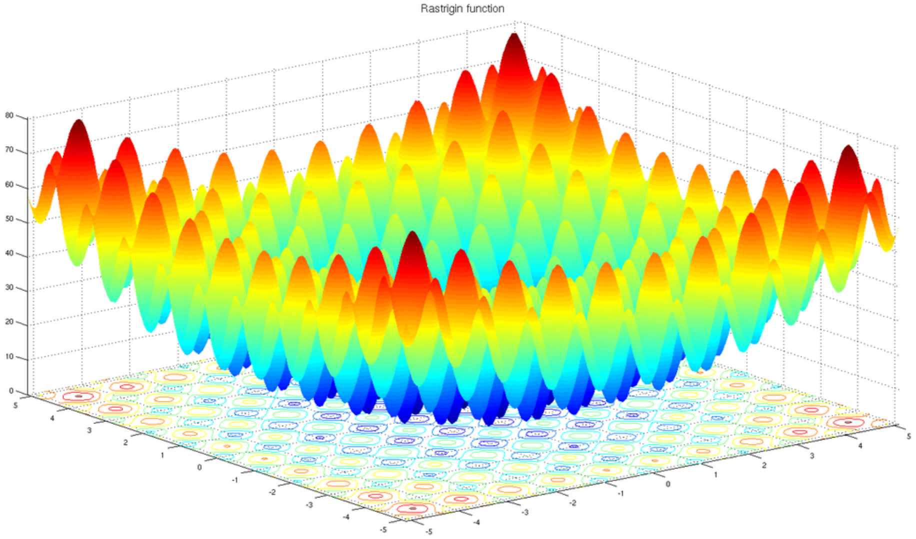
[3]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/convex.png",width=500, height=500)
```

[3]:



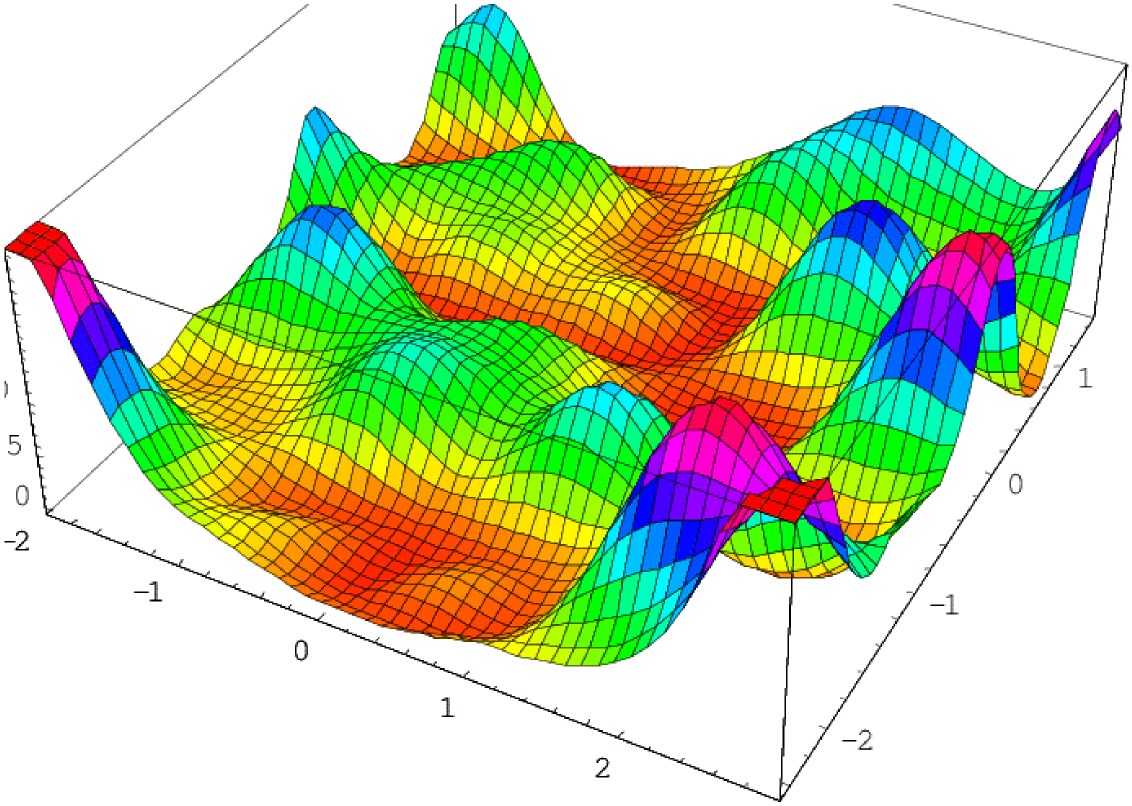
[5]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/nonconvex1.png",width=500, height=500)
```

[5]:



[7]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/nonconvex2.png",width=500, height=500)
```

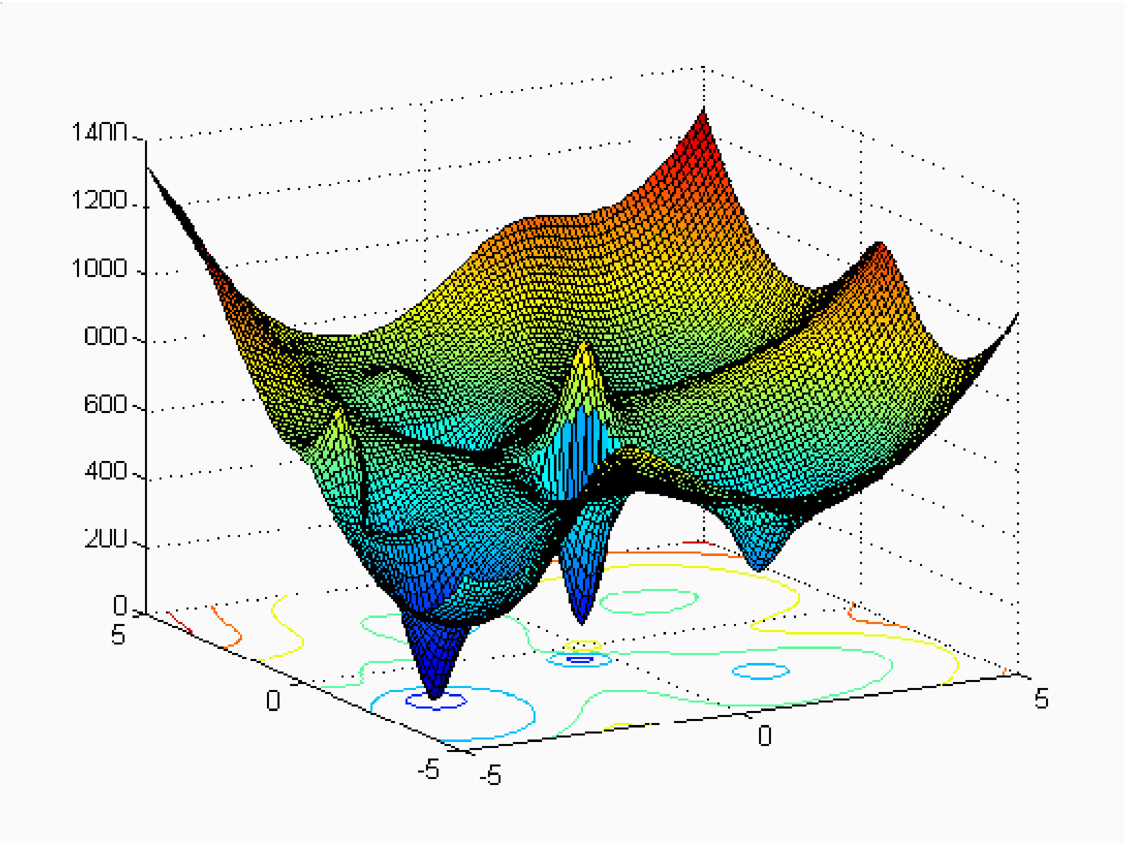
[7]:



[9]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/nonconvex3.png",width=500, height=500)
```

[9]:



2.3 Loss Functions And Optimizers

2.3.1 Batch Gradient

We have seen the Gradient Descent algorithm as a optimizer. Now let's other gradient descent based optimizers.

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters w for the entire training dataset.

The gradient descent is defined as

$$w = w - \eta \nabla_w L(\hat{y}, y)$$

it will look like this when it is implemented

```
for i in range(epochs):
    params_grad = evaluate_gradient ( loss_function , data , params )
    params = params - learning_rate * params_grad
```

2.3.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$w = w - \eta \nabla_w L(\hat{y}^{(i)}, y^{(i)})$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily. While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

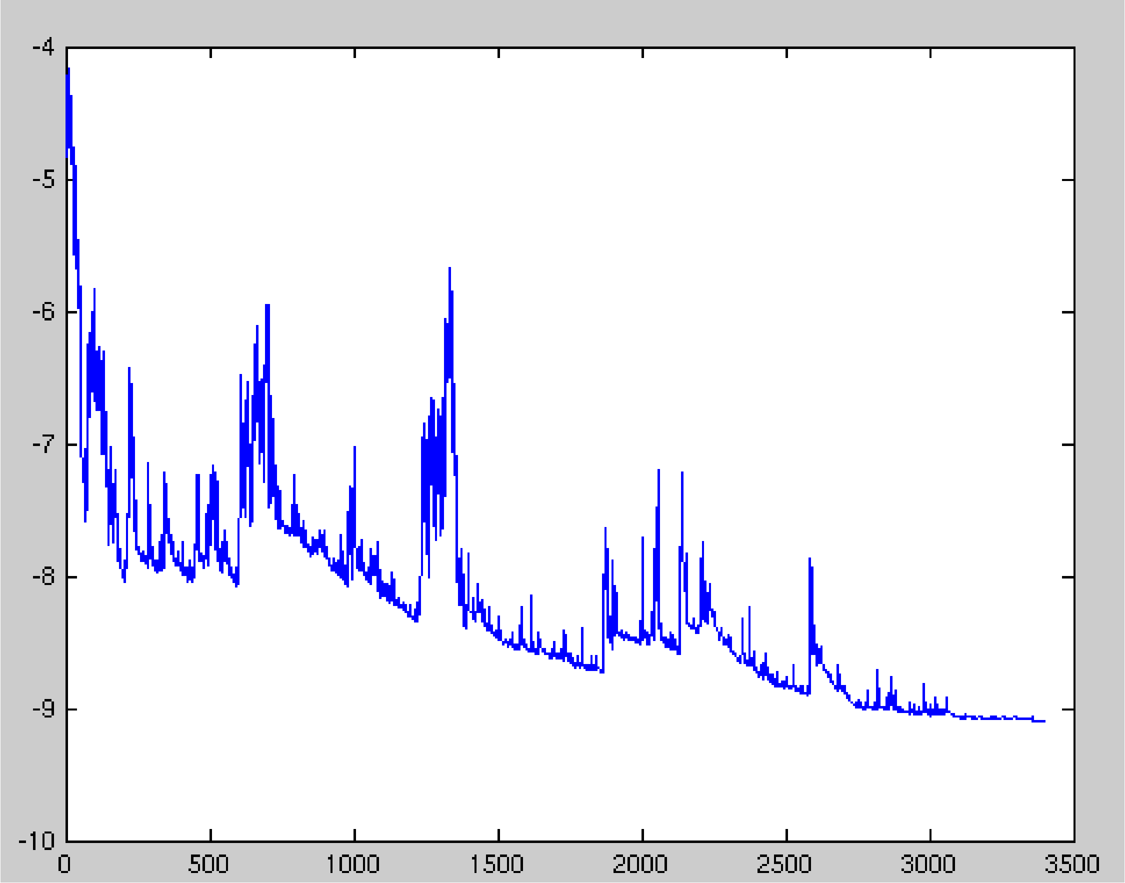
It will look like this when it is implemented:

```
for i in range ( nb_epochs ):
    np . random . shuffle ( data )
    for example in data :
        params_grad = evaluate_gradient ( loss_function , example , params )
        params = params - learning_rate * params_grad
```

[4]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/sgd.png",width=300, height=300)
```

[4]:



2.3.3 Mini-batch Gradient Descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$w = w - \eta \nabla_w L(\hat{y}^{(i:i+n)}, y^{(i:i+n)})$$

This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

It will look like this when it is implemented

```
for i in range ( nb_epochs ):
    np . random . shuffle ( data )
    for batch in get_batches ( data , batch_size =50):
```

```
params_grad = evaluate_gradient ( loss_function , batch , params )
params = params - learning_rate * params_grad
```

The applicability of batch or stochastic gradient descent really depends on the error manifold expected.

Batch gradient descent computes the gradient using the whole dataset. This is great for convex, or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution, either local or global. Additionally, batch gradient descent, given an annealed learning rate, will eventually find the minimum located in it's basin of attraction.

Stochastic gradient descent (SGD) computes the gradient using a single sample. Most applications of SGD actually use a minibatch of several samples, for reasons that will be explained a bit later. SGD works well (Not well, I suppose, but better than batch gradient descent) for error manifolds that have lots of local maxima/minima. In this case, the somewhat noisier gradient calculated using the reduced number of samples tends to jerk the model out of local minima into a region that hopefully is more optimal.

A good balance is struck when the minibatch size is small enough to avoid some of the poor local minima, but large enough that it doesn't avoid the global minima or better-performing local minima

One benefit of SGD is that it's computationally a whole lot faster. Large datasets often can't be held in RAM, which makes vectorization much less efficient. Rather, each sample or batch of samples must be loaded, worked with, the results stored, and so on. Minibatch SGD, on the other hand, is usually intentionally made small enough to be computationally tractable.

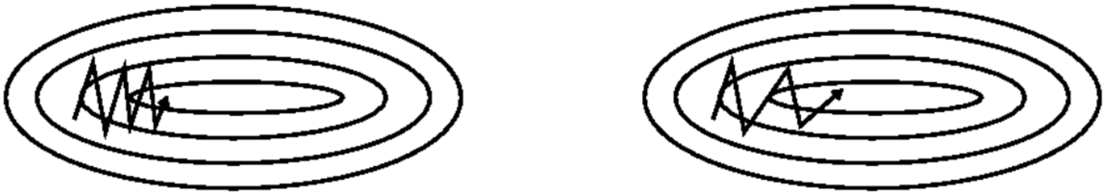
2.3.4 Stochastic Gradient Descent With Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum

[8]:

```
from IPython.display import Image
from IPython.core.display import HTML
Image(filename= "./img/sgd2.png",width=900, height=900)
```

[8]:



(a) SGD without momentum

(b) SGD with momentum

It does this by adding a fraction γ of the update vector of the past time step to the current update vector

$$v_t = \gamma v_{t-1} + \eta \nabla_w L(\hat{y}, y)$$

$$w = w - v_t$$

The momentum term γ is usually set to 0.9 or a similar value. Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way.

Check other optimization methods for better training neural networks:

- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam
- AdaMax
- Nadam

Check out Sebastian Ruder's pre-print paper An overview of gradient descent optimization algorithms.

Check out this blog post Why Momentum Really Works

Book recommendation: Convex Optimization - Stephen Boyd

[3]:

```python
from IPython.display import Image;
Image("./img/sgd3.gif")
```

[3]:

```
<IPython.core.display.Image object>
```

2.4 Cross Entropy Loss For Multiclass Tasks

$$CE = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=0}^{C} y_i^j \log \hat{y}_i^j$$

In case of one-hot vector, for each sample, we only have one correct class. All other classes are zero. This summation over classes C is eliminated.

$$CE = \sum_{i=0}^{m} y^i \log \hat{y}^i$$

Check out other important loss functions like KL-Divergence: Loss Functions

2.5 Softmax

The softmax function, also known as softargmax or normalized exponential function, is a generalization of the logistic function to multiple dimensions. It is used in multinomial logistic regression and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

$$Softmax(h)_i = \frac{\exp(h_i)}{\sum_{j=0}^{K}\exp(h_j)}$$

$$= \hat{y}$$

for more detailed explanation of softmax, see here.

3 Lab.

[1]:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

torch.manual_seed(137);
```

[2]:

```python
class MLP(nn.Module):
    def __init__(self,input_shape,output_shape):
        super(MLP,self).__init__()
        self.input_shape = input_shape
        self.output_shape = output_shape
        self.fc1 = nn.Linear(input_shape,20)
        self.fc2 = nn.Linear(20,20)
        self.out = nn.Linear(20,output_shape)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax()

    def forward(self,x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.out(x)
        #x = self.sigmoid(x)
        #x = self.softmax(x)
        return x
```

[3]:

```
df = pd.read_csv('./data/diabetes.csv')
df.head()
```

[3]:

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            2      138             62             35        0  33.6
1            0       84             82             31      125  38.2
2            0      145              0              0        0  44.2
3            0      135             68             42      250  42.3
4            1      139             62             41      480  40.7

   DiabetesPedigreeFunction  Age  Outcome
0                     0.127   47        1
1                     0.233   23        0
2                     0.630   31        1
3                     0.365   24        1
4                     0.536   21        0
```

[4]:

```
X = df.drop('Outcome', axis = 1).values
y = df['Outcome'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 0)
```

[5]:

```
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)

y_train = torch.LongTensor(y_train)
y_test = torch.LongTensor(y_test)

CUDA = torch.cuda.is_available()

if CUDA:
    X_train = X_train.cuda()
    y_train = y_train.cuda()

    X_test = X_test.cuda()
    y_test = y_test.cuda()
```

[6]:

```
model = MLP(X_train.shape[1],2)

if CUDA:
    model = model.cuda()
```

[7]:
```
model.parameters
```

[7]:
```
<bound method Module.parameters of MLP(
  (fc1): Linear(in_features=8, out_features=20, bias=True)
  (fc2): Linear(in_features=20, out_features=20, bias=True)
  (out): Linear(in_features=20, out_features=2, bias=True)
  (relu): ReLU()
  (sigmoid): Sigmoid()
  (softmax): Softmax()
)>
```

[8]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.01)
```

[11]:
```
final_loss = []
test_loss = []
epochs = 700
for i in range(epochs):
    i = i + 1
    y_pred = model.forward(X_train)
    loss = criterion(y_pred, y_train)
    final_loss.append(loss)



    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    #acc_train = (y_pred == y_train).sum().item() / len(y_pred)

    with torch.no_grad():
        y_pred = model.forward(X_test)
        loss_test = criterion(y_pred, y_test)
        test_loss.append(loss_test)

    if i % 10 == 1:
        print('-'*100)
        print('Epoch number: {} \nTraining Loss: {} \nTest Loss: {}'.format(i,␣
 ↪loss.item(),loss_test.item()))
```

```
--------------------------------------------------------------------------------
--------------------
Epoch number: 1
Training Loss: 5.612581729888916
Test Loss: 4.296512603759766
--------------------------------------------------------------------------------
```

```
--------------------
Epoch number: 11
Training Loss: 1.4223638772964478
Test Loss: 1.4688708782196045
--------------------------------------------------------------------------------
--------------------
Epoch number: 21
Training Loss: 0.6866970062255859
Test Loss: 0.7274264693260193
--------------------------------------------------------------------------------
--------------------
Epoch number: 31
Training Loss: 0.6674190759658813
Test Loss: 0.6509056091308594
--------------------------------------------------------------------------------
--------------------
Epoch number: 41
Training Loss: 0.604386568069458
Test Loss: 0.6314026117324829
--------------------------------------------------------------------------------
--------------------
Epoch number: 51
Training Loss: 0.5876470804214478
Test Loss: 0.6135611534118652
--------------------------------------------------------------------------------
--------------------
Epoch number: 61
Training Loss: 0.5812631845474243
Test Loss: 0.6117417812347412
--------------------------------------------------------------------------------
--------------------
Epoch number: 71
Training Loss: 0.573258101940155
Test Loss: 0.6034999489784241
--------------------------------------------------------------------------------
--------------------
Epoch number: 81
Training Loss: 0.5647507309913635
Test Loss: 0.5940440893173218
--------------------------------------------------------------------------------
--------------------
Epoch number: 91
Training Loss: 0.5581337213516235
Test Loss: 0.5838819146156311
--------------------------------------------------------------------------------
--------------------
Epoch number: 101
Training Loss: 0.5520173907279968
```

```
Test Loss: 0.5810103416442871
--------------------------------------------------------------------------------
--------------------
Epoch number: 111
Training Loss: 0.5466842651367188
Test Loss: 0.5795843601226807
--------------------------------------------------------------------------------
--------------------
Epoch number: 121
Training Loss: 0.5418031811714172
Test Loss: 0.5762171149253845
--------------------------------------------------------------------------------
--------------------
Epoch number: 131
Training Loss: 0.5371609926223755
Test Loss: 0.5732176899909973
--------------------------------------------------------------------------------
--------------------
Epoch number: 141
Training Loss: 0.5326155424118042
Test Loss: 0.5707719326019287
--------------------------------------------------------------------------------
--------------------
Epoch number: 151
Training Loss: 0.5279036164283752
Test Loss: 0.567871630191803
--------------------------------------------------------------------------------
--------------------
Epoch number: 161
Training Loss: 0.5235306024551392
Test Loss: 0.5650922060012817
--------------------------------------------------------------------------------
--------------------
Epoch number: 171
Training Loss: 0.5190896987915039
Test Loss: 0.5614479780197144
--------------------------------------------------------------------------------
--------------------
Epoch number: 181
Training Loss: 0.5147121548652649
Test Loss: 0.5586879253387451
--------------------------------------------------------------------------------
--------------------
Epoch number: 191
Training Loss: 0.510503888130188
Test Loss: 0.5560088753700256
--------------------------------------------------------------------------------
--------------------
```

```
Epoch number: 201
Training Loss: 0.5066581964492798
Test Loss: 0.5543999671936035
--------------------------------------------------------------------------------
--------------------
Epoch number: 211
Training Loss: 0.5031048059463501
Test Loss: 0.5514571070671082
--------------------------------------------------------------------------------
--------------------
Epoch number: 221
Training Loss: 0.4993375539779663
Test Loss: 0.5492119789123535
--------------------------------------------------------------------------------
--------------------
Epoch number: 231
Training Loss: 0.49568870663642883
Test Loss: 0.5450125336647034
--------------------------------------------------------------------------------
--------------------
Epoch number: 241
Training Loss: 0.4921485483646393
Test Loss: 0.5413646101951599
--------------------------------------------------------------------------------
--------------------
Epoch number: 251
Training Loss: 0.48847275972366333
Test Loss: 0.5387511253356934
--------------------------------------------------------------------------------
--------------------
Epoch number: 261
Training Loss: 0.4844721257686615
Test Loss: 0.5351930856704712
--------------------------------------------------------------------------------
--------------------
Epoch number: 271
Training Loss: 0.4799393117427826
Test Loss: 0.5314549803733826
--------------------------------------------------------------------------------
--------------------
Epoch number: 281
Training Loss: 0.4757859408855438
Test Loss: 0.5277162790298462
--------------------------------------------------------------------------------
--------------------
Epoch number: 291
Training Loss: 0.4717782139778137
Test Loss: 0.5236012935638428
```

```
--------------------------------------------------------------------------------
--------------------
Epoch number: 301
Training Loss: 0.4679189920425415
Test Loss: 0.5200663208961487
--------------------------------------------------------------------------------
--------------------
Epoch number: 311
Training Loss: 0.46422314643859863
Test Loss: 0.517448365688324
--------------------------------------------------------------------------------
--------------------
Epoch number: 321
Training Loss: 0.46051064133644104
Test Loss: 0.5152199864387512
--------------------------------------------------------------------------------
--------------------
Epoch number: 331
Training Loss: 0.45703843235969543
Test Loss: 0.5128147602081299
--------------------------------------------------------------------------------
--------------------
Epoch number: 341
Training Loss: 0.45388075709342957
Test Loss: 0.5108883380889893
--------------------------------------------------------------------------------
--------------------
Epoch number: 351
Training Loss: 0.45092952251434326
Test Loss: 0.5079197883605957
--------------------------------------------------------------------------------
--------------------
Epoch number: 361
Training Loss: 0.4480302333831787
Test Loss: 0.5059962868690491
--------------------------------------------------------------------------------
--------------------
Epoch number: 371
Training Loss: 0.44516250491142273
Test Loss: 0.5035369992256165
--------------------------------------------------------------------------------
--------------------
Epoch number: 381
Training Loss: 0.4424757659435272
Test Loss: 0.501268744468689
--------------------------------------------------------------------------------
--------------------
Epoch number: 391
```

```
Training Loss: 0.4398163855075836
Test Loss: 0.49988430738449097
--------------------------------------------------------------------------------
--------------------
Epoch number: 401
Training Loss: 0.43690264225006104
Test Loss: 0.4970690906047821
--------------------------------------------------------------------------------
--------------------
Epoch number: 411
Training Loss: 0.4341602623462677
Test Loss: 0.49272391200065613
--------------------------------------------------------------------------------
--------------------
Epoch number: 421
Training Loss: 0.431776225566864
Test Loss: 0.4922925531864166
--------------------------------------------------------------------------------
--------------------
Epoch number: 431
Training Loss: 0.42962944507598877
Test Loss: 0.49077171087265015
--------------------------------------------------------------------------------
--------------------
Epoch number: 441
Training Loss: 0.42748919129371643
Test Loss: 0.48897162079811096
--------------------------------------------------------------------------------
--------------------
Epoch number: 451
Training Loss: 0.42554476857185364
Test Loss: 0.4886912405490875
--------------------------------------------------------------------------------
--------------------
Epoch number: 461
Training Loss: 0.42423927783966064
Test Loss: 0.48728933930397034
--------------------------------------------------------------------------------
--------------------
Epoch number: 471
Training Loss: 0.4207872152328491
Test Loss: 0.48393648862838745
--------------------------------------------------------------------------------
--------------------
Epoch number: 481
Training Loss: 0.41893526911735535
Test Loss: 0.4818652868270874
--------------------------------------------------------------------------------
```

```
--------------------
Epoch number: 491
Training Loss: 0.41671276092529297
Test Loss: 0.47942158579826355
--------------------------------------------------------------------------------
--------------------
Epoch number: 501
Training Loss: 0.41445064544677734
Test Loss: 0.47868767380714417
--------------------------------------------------------------------------------
--------------------
Epoch number: 511
Training Loss: 0.4136127829551697
Test Loss: 0.4779663383960724
--------------------------------------------------------------------------------
--------------------
Epoch number: 521
Training Loss: 0.4122093617916107
Test Loss: 0.47749441862106323
--------------------------------------------------------------------------------
--------------------
Epoch number: 531
Training Loss: 0.4113883972167969
Test Loss: 0.4761107563972473
--------------------------------------------------------------------------------
--------------------
Epoch number: 541
Training Loss: 0.40898972749710083
Test Loss: 0.47405722737312317
--------------------------------------------------------------------------------
--------------------
Epoch number: 551
Training Loss: 0.40430545806884766
Test Loss: 0.46882903575897217
--------------------------------------------------------------------------------
--------------------
Epoch number: 561
Training Loss: 0.4023095369338989
Test Loss: 0.4675813317298889
--------------------------------------------------------------------------------
--------------------
Epoch number: 571
Training Loss: 0.40045270323753357
Test Loss: 0.4685940444469452
--------------------------------------------------------------------------------
--------------------
Epoch number: 581
Training Loss: 0.3983888328075409
```

```
Test Loss: 0.46418097615242004
--------------------------------------------------------------------------------
--------------------
Epoch number: 591
Training Loss: 0.39779776334762573
Test Loss: 0.46412792801856995
--------------------------------------------------------------------------------
--------------------
Epoch number: 601
Training Loss: 0.39713141322135925
Test Loss: 0.46390148997306824
--------------------------------------------------------------------------------
--------------------
Epoch number: 611
Training Loss: 0.40184536576271057
Test Loss: 0.46535640954971313
--------------------------------------------------------------------------------
--------------------
Epoch number: 621
Training Loss: 0.4058993458747864
Test Loss: 0.47357413172721863
--------------------------------------------------------------------------------
--------------------
Epoch number: 631
Training Loss: 0.392711877822876
Test Loss: 0.4595200717449188
--------------------------------------------------------------------------------
--------------------
Epoch number: 641
Training Loss: 0.3885689079761505
Test Loss: 0.466237336397171
--------------------------------------------------------------------------------
--------------------
Epoch number: 651
Training Loss: 0.38442131876945496
Test Loss: 0.45482197403907776
--------------------------------------------------------------------------------
--------------------
Epoch number: 661
Training Loss: 0.396688312292099
Test Loss: 0.46261051297187805
--------------------------------------------------------------------------------
--------------------
Epoch number: 671
Training Loss: 0.39906299114227295
Test Loss: 0.4652933180332184
--------------------------------------------------------------------------------
--------------------
```

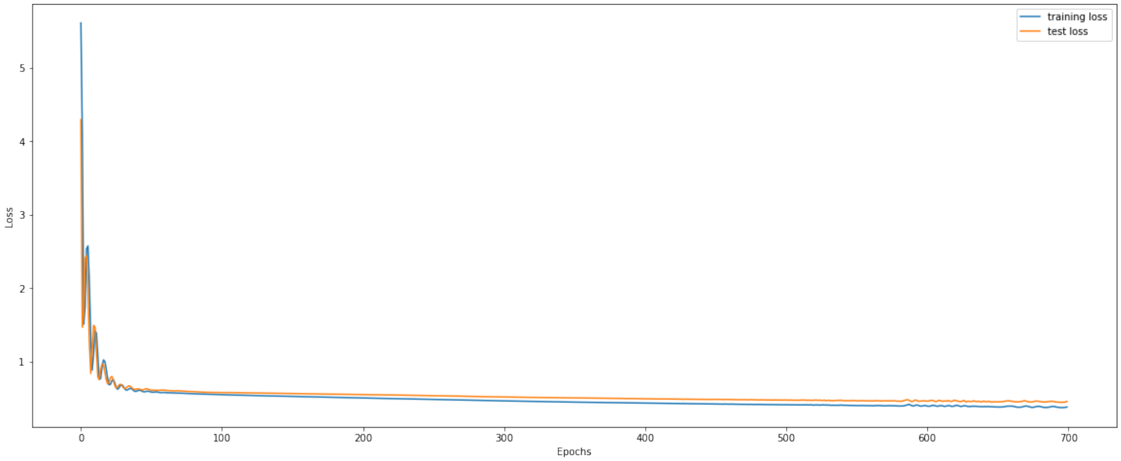
```
Epoch number: 681
Training Loss: 0.3913078308105469
Test Loss: 0.4572509825229645
--------------------------------------------------------------------------------
--------------------
Epoch number: 691
Training Loss: 0.39177799224853516
Test Loss: 0.45765799283981323
```

[12]:

```
fig = plt.figure(figsize=(20,8))
plt.plot(final_loss, label = 'training loss')
plt.plot(test_loss, label = 'test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

[12]:

```
<matplotlib.legend.Legend at 0x7fe3bb9724e0>
```



[179]:

```
pred = []
with torch.no_grad():
    for i, data in enumerate(X_test):
        y_pred = model(data)
        pred.append(y_pred.argmax().item())
```

[180]:

```
y_test = y_test.cpu()
cm = confusion_matrix(y_test, pred)
print(cm)
```

```
[[242  30]
 [ 36  92]]
```

[181]:

```
acc_score = accuracy_score(y_test, pred)
print('Test Accuracy: ',acc_score)
```

```
Test Accuracy:  0.835
```

[84]:

```
torch.save(model, 'diabetes.pt')
model = torch.load('diabetes.pt')
model.eval()
```

```
/home/safak/anaconda3/lib/python3.6/site-packages/torch/serialization.py:250:
UserWarning: Couldn't retrieve source code for container of type MLP. It won't
be checked for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
```

[84]:

```
MLP(
  (fc1): Linear(in_features=8, out_features=20, bias=True)
  (fc2): Linear(in_features=20, out_features=20, bias=True)
  (out): Linear(in_features=20, out_features=2, bias=True)
  (relu): ReLU()
  (sigmoid): Sigmoid()
  (softmax): Softmax()
)
```

[85]:

```
## Let's take the only first column of dataset and change it's values for␣
 ↪prediction
new_data = list(df.iloc[0, :-1])
new_data = torch.tensor(new_data)

if CUDA:
    new_data = new_data.cuda()
```

[86]:

```
with torch.no_grad():
    print(model(new_data))
    print(model(new_data).argmax())
    print(model(new_data).argmax().item())
```

```
tensor([1., 0.], device='cuda:0')
tensor(0, device='cuda:0')
0
```

```
/home/safak/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:20:
UserWarning: Implicit dimension choice for softmax has been deprecated. Change
the call to include dim=X as an argument.
```

[87]:

```
df.head()
```

[87]:

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            2      138             62             35        0  33.6
```

```
1            0       84             82             31      125  38.2
2            0      145              0              0        0  44.2
3            0      135             68             42      250  42.3
4            1      139             62             41      480  40.7

   DiabetesPedigreeFunction  Age  Outcome
0                     0.127   47        1
1                     0.233   23        0
2                     0.630   31        1
3                     0.365   24        1
4                     0.536   21        0
```

[ ]:

```

```

3.1 Lecture Practice

[1]:

```
from sklearn import cluster, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

[2]:

```
torch.manual_seed(137)
```

[2]:

```
<torch._C.Generator at 0x7f8eda7d9450>
```

[3]:

```
X,y = datasets.make_moons(n_samples=10000,noise=.5)
```

[4]:

```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.
 ↪2,random_state=0)
```

[5]:

```
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train)
y_test = torch.LongTensor(y_test)
```

[6]:

```
CUDA = torch.cuda.is_available()
```

[7]:

```
torch.__version__
```

[7]:

```
'1.0.0'
```

[8]:

```
CUDA
```

[8]:
```
True
```

[9]:
```python
if CUDA:
    X_train = X_train.cuda()
    X_test = X_test.cuda()
    y_train = y_train.cuda()
    y_test = y_test.cuda()
```

[10]:
```python
class MLP(nn.Module):
    def __init__(self, x_dim, out_dim):
        super(MLP, self).__init__()

        self.x_dim = x_dim
        self.out_dim = out_dim

        self.fc1 = nn.Linear(self.x_dim,20)
        self.fc2 = nn.Linear(20,20)
        self.out = nn.Linear(20,self.out_dim)

        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.out(x)
        x = self.sigmoid(x)
        return x
```

[11]:
```python
model = MLP(X_train.shape[1],2)
```

[12]:
```python
if CUDA:
    model = model.cuda()
```

[13]:
```python
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.01)
```

[14]:
```python
final_train_loss = []
final_test_loss = []
```

[15]:
```python
epochs = 700
```

[16]:
```python
for i in range(epochs):
    y_pred = model.forward(X_train)
    loss = criterion(y_pred,y_train)
    final_train_loss.append(loss)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    with torch.no_grad():
        y_pred = model.forward(X_test)
        loss_test = criterion(y_pred,y_test)
        final_test_loss.append(loss_test)

    if i % 100 == 0:
        print(100*"-")
        print("Epoch: {0}/{1} \nTraining Loss: {2} \nTest Loss: {3}".
↪format(i,epochs,
                                                                       loss.
↪item(),loss_test.item()))
```

```
--------------------------------------------------------------------------------
--------------------
Epoch: 0/700
Training Loss: 0.7106247544288635
Test Loss: 0.7023131847381592
--------------------------------------------------------------------------------
--------------------
Epoch: 100/700
Training Loss: 0.480049729347229
Test Loss: 0.48768794536590576
--------------------------------------------------------------------------------
--------------------
Epoch: 200/700
Training Loss: 0.47619932889938354
Test Loss: 0.48283660411834717
--------------------------------------------------------------------------------
--------------------
Epoch: 300/700
Training Loss: 0.4756072163581848
Test Loss: 0.48190009593963623
--------------------------------------------------------------------------------
--------------------
Epoch: 400/700
Training Loss: 0.47502031922340393
Test Loss: 0.4815264344215393
--------------------------------------------------------------------------------
--------------------
Epoch: 500/700
```

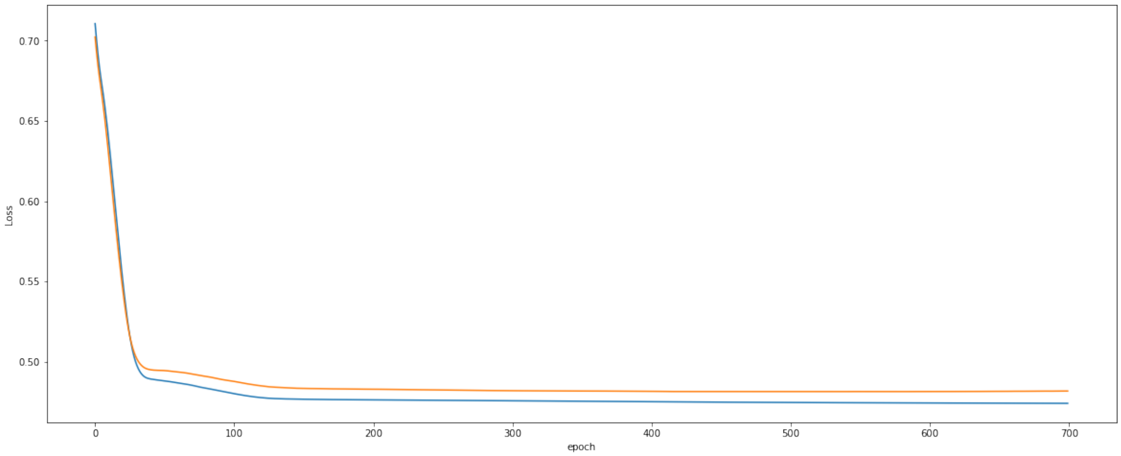
```
Training Loss: 0.4745778441429138
Test Loss: 0.48131540417671204
--------------------------------------------------------------------------------
--------------------
Epoch: 600/700
Training Loss: 0.47425881028175354
Test Loss: 0.4813949167728424
```

[17]:

```
fig = plt.figure(figsize=(20,8))
plt.plot(final_train_loss,label='training loss')
plt.plot(final_test_loss,label='training loss')
plt.xlabel('epoch')
plt.ylabel('Loss')
```

[17]:

```
Text(0, 0.5, 'Loss')
```



[18]:

```
pred = []
with torch.no_grad():
    for i, data in enumerate(X_test):
        y_pred = model.forward(data)
        pred.append(y_pred.argmax().item())
```

[19]:

```
y_test = y_test.cpu()
```

[20]:

```
acc = accuracy_score(y_test,pred)
```

[21]:

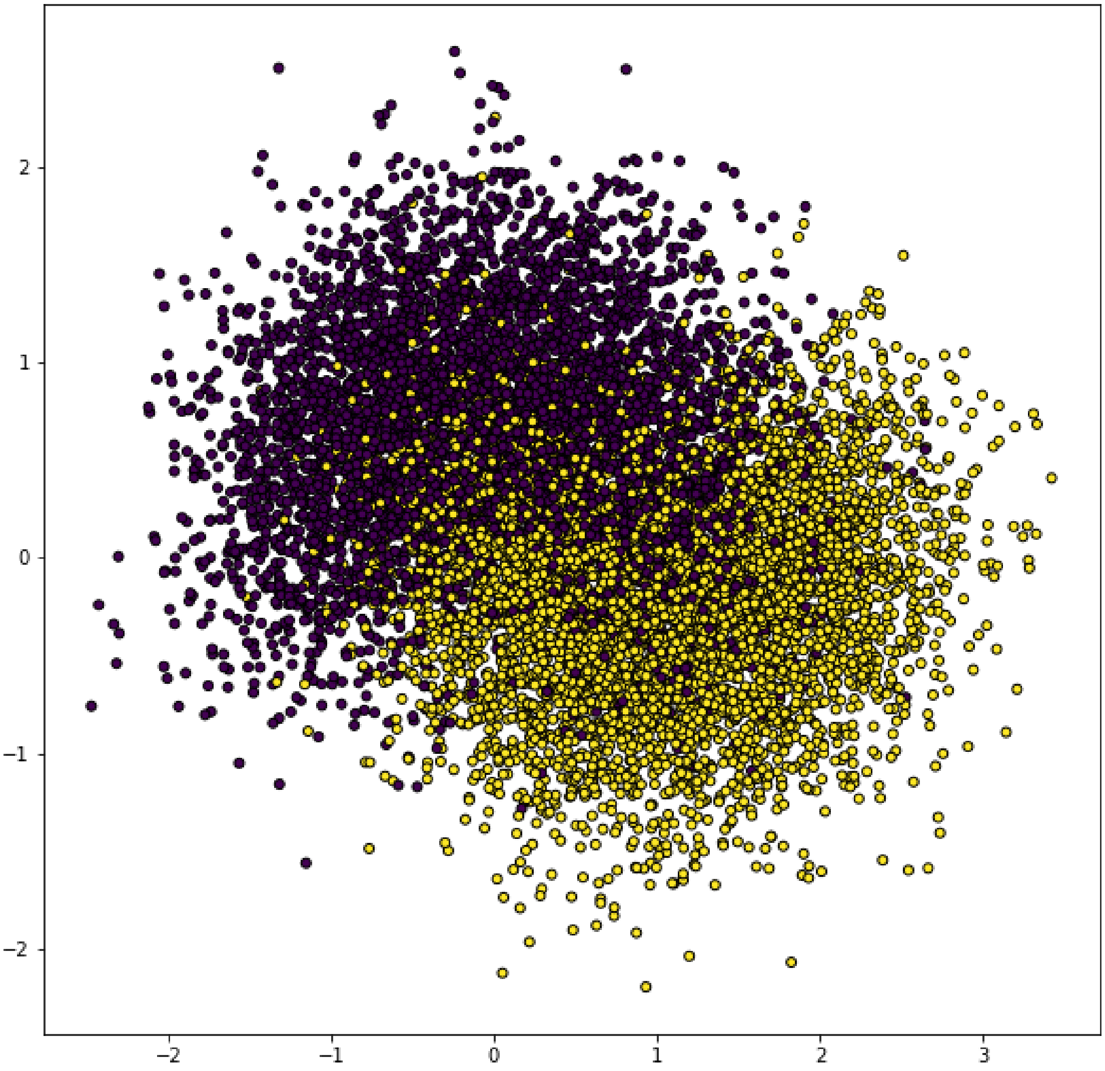
```
acc
```

[21]:

```
0.8175
```

[22]:

```
fig = plt.figure(figsize=(10,10))
plt.scatter(X[:, 0], X[:, 1], marker='o', c=y,
            s=25, edgecolor='k');
```



[ ]:

```
```