Neural Networks

An Artificial Neural Network consists of an input layer, a number of hidden layers and finally an output layer.

Assume the data are vectors in $\mathbb{R}^d$ i.e. d-dimensional tuples of real numbers.

We are counting the layers from the top so the output layer is layer 0 and if there are a total of N layers, the input layer is layer N, so the hidden layers are numbered $1, 2, \ldots, N-1$

Each layer consists of 3 pieces of data

- A matrix W
- A bias vector $\underline{b}$
- An activation function $\mathbb{R} \to \mathbb{R}$, which can be for instance the logistic function σ or the hyperbolic tangent $tanh$. Nowadays the most common activation function is the $ReLU$, $x \mapsto \max(x, 0)$

A data vector $\underline{x}$ travels through the network.

$$\underline{x} \mapsto (\underline{u}_N = (\underline{x}W_N + \underline{b}_N)) \mapsto \left(\phi_N(\underline{u}_N) = \underline{z}_{N-1}\right) \mapsto \left(\underline{u}_{N-1} = (\underline{z}_{N-1}W_{N-1} + \underline{b}_{N-1})\right) \mapsto \ldots$$

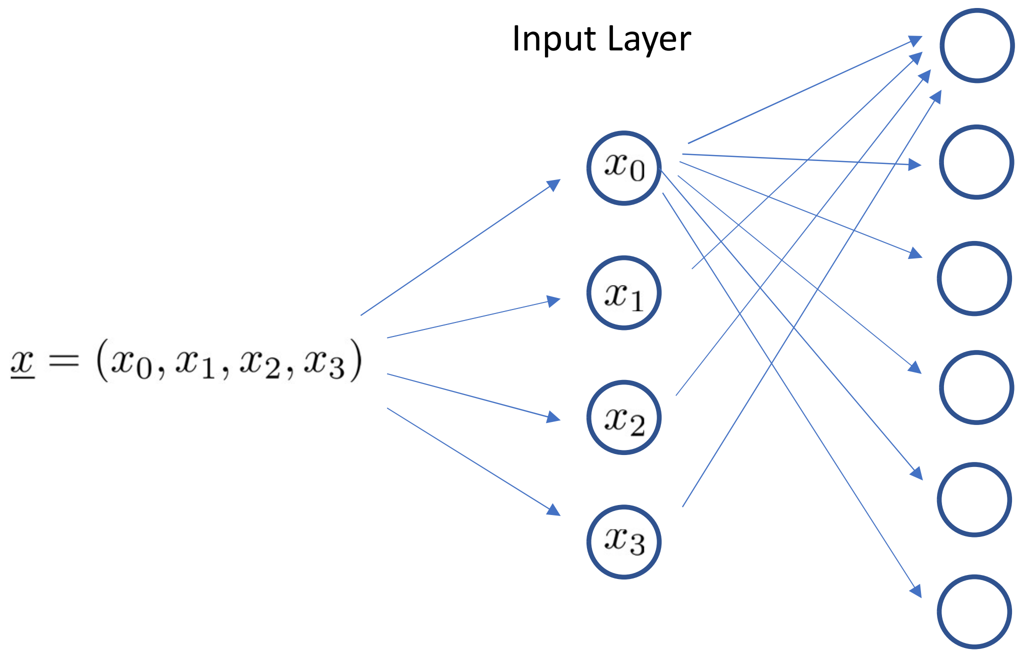
where the matrix W_i, the bias vector $\underline{b}_i$ and the activation function ϕ_i are the data belonging to layer i.

Remark that if W_i is a $d_i \times e_i$ dimensional matrix then $\underline{b}_i$ is an e_i-dimensional vector and W_{i-1} must have e_i rows i.e. W_{i-1} is $d_{i-1} \times e_{i-1}$ with $d_{i-1} = e_i$

Hidden Layer N-1

$W_{N-1} = (W_{N-1,0} \quad W_{N-1,1} \quad W_{N-1,2} \quad W_{N-1,3} \quad W_{N-1,4} \quad W_{N-1,5})$, of dimension 4×6 so each $W_{N-1,i}$ is a column vector of dimension 4.

Bias vector $\underline{b}_{N-1} = (b_{N-1,0}, b_{N-1,1}, \ldots, b_{N-1,5})$ and activation function ϕ_{N-1}



$$z_{N-1,0} = \phi_N(\underline{x} \cdot W_{N,0} + b_{N,0})$$

$$z_{N-1,1} = \phi_N(\underline{x} \cdot W_{N,1} + b_{N,1})$$

$$z_{N-1,2} = \phi_N(\underline{x} \cdot W_{N,2} + b_{N,2})$$

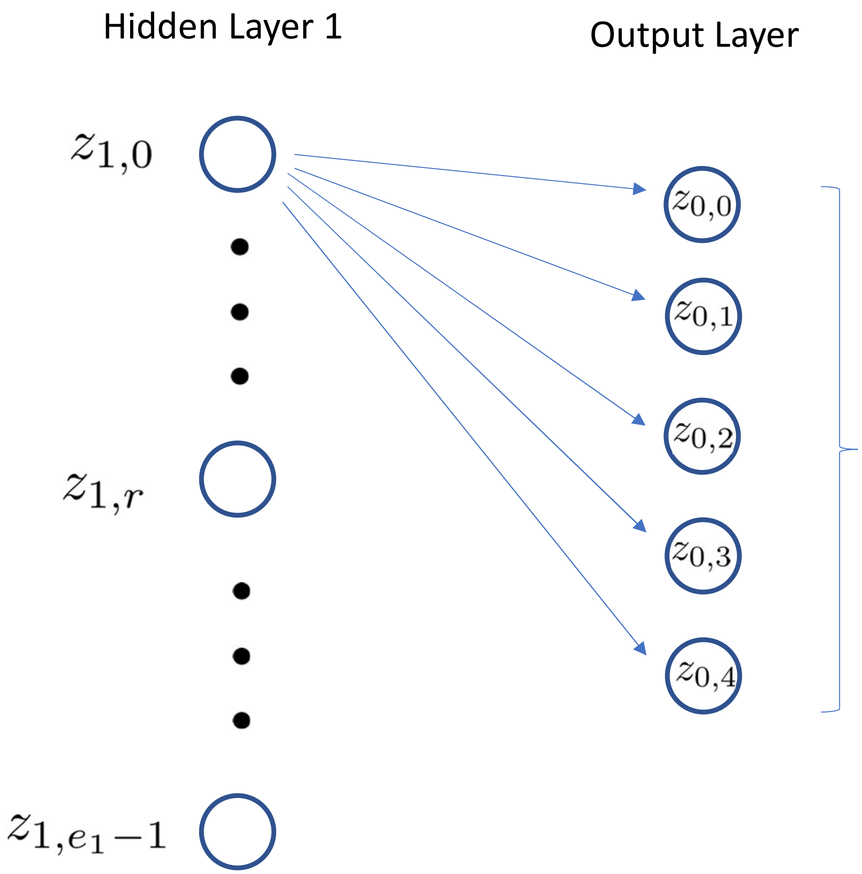
$$z_{N-1,3} = \phi_N(\underline{x} \cdot W_{N,3} + b_{N,3})$$

$$z_{N-1,4} = \phi_N(\underline{x} \cdot W_{N,4} + b_{N,4})$$

$$z_{N-1,5} = \phi_N(\underline{x} \cdot W_{N,5} + b_{N,5})$$

$$\underline{z}_{N-1} = \phi_N(\underline{x}W_N + \underline{b}_N)$$

$$\underline{u}_N = \underline{x}W_N + \underline{b}_N$$



Final activation function , ϕ_1 is typically the softmax or the logistic function (classification) or the identity (regression)

$$\phi_1(\underline{z}_1 W_1 + \underline{b}_1) = \underline{z}_0$$

In a sense the graphic depiction of the ANN is more confusing than illuminating. The reality is that a (in this case *fully connected, feed forward Neural Nework*) is just a composite function:

$$f : \underline{x} \mapsto \phi_1(\dots\left(\phi_{N-1}\left((\phi_N\left(\underline{x}W_N + \underline{b}_N\right))\, W_{N-1} + \underline{b}_{N-1}\right) W_{N-2} + \underline{b}_{N-2}\right)\dots$$

or using the z and u variables

$$\begin{aligned}
x &\mapsto \underline{z}_0 \\
\underline{z}_0 &= \phi_1(\underline{u}_1) \\
\underline{u}_1 &= \underline{z}_1 W_1 + \underline{b}_1 \\
\underline{z}_1 &= \phi_2(\underline{u}_2) \\
\underline{u}_2 &= \underline{z}_2 W_2 + \underline{b}_2 \\
&\vdots
\end{aligned}$$

How do we train an ANN from a dataset i.e. how do we find paramers to fit the data set?

As usual we try to estimate the parameters to minimize a *loss function* $\mathcal{L}$, which can be for instance Cross Entropy in the classification case or Mean Squared Error (MSE) in the regression case.

The parameters of the ANN consists of the matrices $W_N, W_{N-1}, \ldots, W_1$ and the bias vectors $\underline{b}_N, \underline{b}_{N-1}, \ldots, \underline{b}_1$. Thus the total number of parameters is

$$\sum_{i=1} d_i e_i + \sum_{i=1} e_i = \sum_{i=1} d_{i-1}(d_i + 1)$$

Here d_N is the input dimension and d_0 is the output dimension.

For example if $N = 1$ and $\phi_1 = \sigma$, then the ANN is just the logistic regression $\underline{x} \mapsto \sigma(\underline{x}W + b)$, and there are $d_N + 1$ parameters where $d_N = \dim \underline{x}$.

Now assume we have a data point $(y, \underline{x})$ and assume we have a some loss function $\mathcal{L}$. We then want to compute $\dfrac{\partial \mathcal{L}}{\partial W_i}$ for $i = 1, 2, \ldots, N$. Using the chain rule we get

$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial z_0} \cdot \frac{\partial g_1}{\partial z_1} \cdot \frac{\partial g_2}{\partial z_2} \cdot \cdots \cdot \frac{\partial g_{i-1}}{\partial z_{i-1}} \cdot \frac{\partial g_i}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial z_0} \cdot \nabla \phi_1 \cdot W_1^T \cdot \nabla \phi_2 \cdot W_2^T \cdot \cdots \cdot \nabla \phi_{i-1} \cdot W_{i-1}^T \cdot \frac{\partial g_i}{\partial W_i}$$

The gradients with respect to the the bias vectors are computed similary:

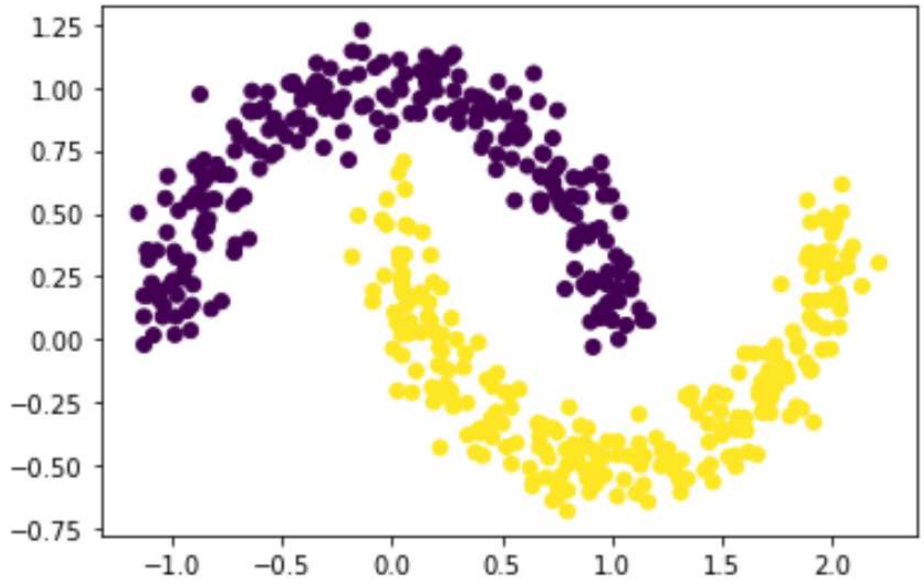
$$\frac{\partial \mathcal{L}}{\partial \underline{b}_i} = \frac{\partial \mathcal{L}}{\partial z_0} \cdot \frac{\partial g_1}{\partial z_1} \cdot \frac{\partial g_2}{\partial z_2} \cdot \cdots \cdot \frac{\partial g_{i-1}}{\partial z_{i-1}} \cdot \frac{\partial g_i}{\partial \underline{b}_i} = \frac{\partial \mathcal{L}}{\partial z_0} \cdot \nabla \phi_1 \cdot W_1^T \cdot \nabla \phi_2 \cdot W_2^T \cdot \cdots \cdot \nabla \phi_{i-1} \cdot W_{i-1}^T \cdot \frac{\partial g_i}{\partial \underline{b}_i}$$

Now we will build a simple Neural Net to perform a classification task.

The dataset is generated by one of the dateset generators in *sklearn*

```
X,y = make_moons(500,noise=0.1)
```

```
plt.scatter(X[:,0],X[:,1],c=y);
```



We shall use Pytorch to code our deep learning networks (another option is to use TensorFlow but personally I prefer Pytorch)

You need to install pytorch. Go to the Pytorch website to get instructions on how to do this

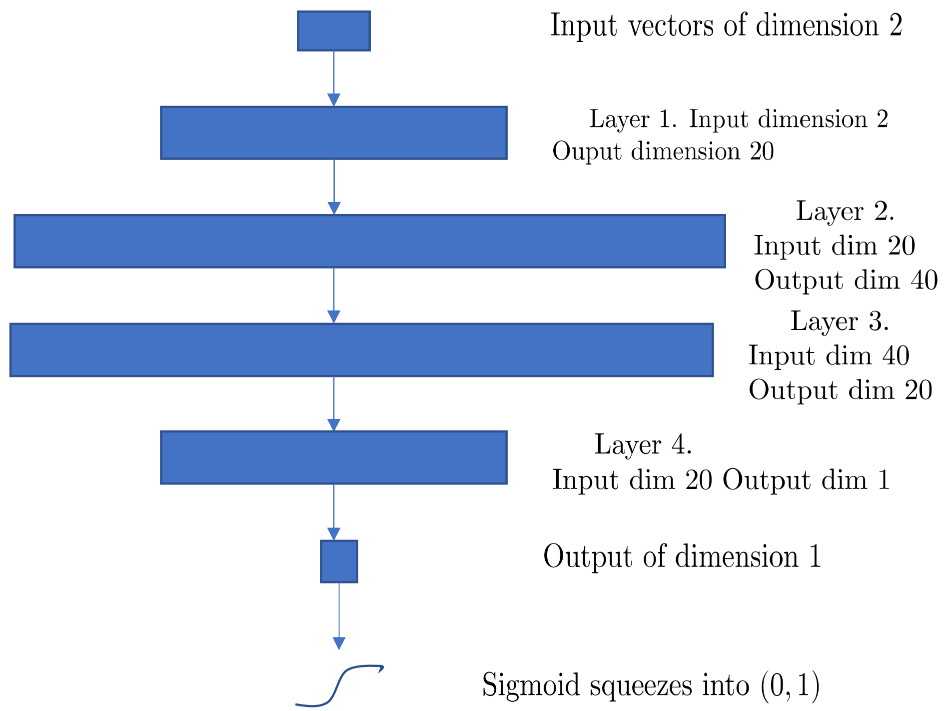
```
import numpy as np
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
%matplotlib inline
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam
```

Pytorch imports, remark that we are importing *torch*

We are going to build a network that has an input and and output layer and 4 hidden layers. At the end we send the output through a sigmoid function

$$x \to \frac{1}{1+\exp(-x)}$$



```python
class Net(nn.Module):

    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2,20)
        self.fc2 = nn.Linear(20,40)
        self.fc3 = nn.Linear(40,20)
        self.fc4 = nn.Linear(20,1)
        self.relu = nn.ReLU()



    def forward(self,x):
        u = self.fc1(x)
        z = self.relu(u)
        u = self.fc2(z)
        z = self.relu(u)
        u = self.fc3(z)
        z = self.relu(u)
        u = self.fc4(z)
        z = torch.sigmoid(u)

        return z

    def predict(self,A):
        with torch.no_grad():
            output = self.forward(A)
            output.cpu()
            output_ = output.cpu().numpy()
            return np.array([0 if x<1/2 else 1 for x in output_])
```

In pytorch a network is a class derived from the *nn.Module* class

Constructor begins by initializing the base class. Then we construct 4 linear layers, remark that the output dimension from a layer has to match the input dimension of the following layer.

The *forward* method overrides the *forward* method in the base class. It sends an input through the entire network. Remark that we have *relu* activations between the linear layers.

2×20

$u = x \cdot W_1 + b_1$

The activation for the last layer is the sigmoid function

Pytorch (and TensorFlow) deals with *tensors.* Tensors can be viewed simply as multi-dimensional arrays, so for instance a 1-tensor is a vector, a 2-tensor is a matrix (=an array of vectors), a 3-tensor, an array of matrices etc. Pytorch has a special Tensor class and methods that transform arrays into Pytorch tensors.

```
X_t = torch.FloatTensor(X)
y_t = torch.FloatTensor(y).unsqueeze(1)
```

```
X_t.size()
```

```
torch.Size([500, 2])
```

```
y_t.size()
```

```
torch.Size([500, 1])
```

```
y_t
```

```
        [0.],
        [0.],
        [1.],
        [0.],
        [0.],
        [1.],
        [0.],
        [1.],
```

The *torch.FloatTensor* turns the arrays into tensors with 32-bit float entries. Pytorch is very picky about tensors having the correct dimensions.
Here we need the y_t tensor to be a column vector i.e. have column dimension = 1.
We use the *unsqueeze* command to add a dimension.

Next we have to code the *training loop*.

- we feed mini-batches of data points into the network
- we compute the loss between the output from the network and the true labels
- compute the gradient of the loss function on the mini-batch
- the optimizer then does the gradient descent step

Pytorch has a class which will randomly select mini-batches which can be fed into the network, *DataLoader*.

The input to the DataLoader is a a *TensorDataset*

```
from torch.utils.data import TensorDataset, DataLoader

dataset = TensorDataset(X_t,y_t)
dl = DataLoader(dataset = dataset,batch_size=10,
                                        shuffle=True)
```

Construct TensorDataset from the datapoint and the label tensors.
Construct a DataLoader that makes mini-batches of length 10 and shuffles the data so it comes out in random order

Next we have to instantiate the *Net* class to get an actual network.

We then specify the loss function, in this case we use Binary Cross Entropy (*nn.BCELoss*). The loss between an output from the network x (which is a number between 0 and 1) and the label y (which is 0 or 1) given by

$$loss\_fn(x, y) = y\log(x) + (1 - y)\log(1 - x)$$

The optimizer is the Adam optimizer, you should basically always us this optimizer.

```
ann = Net()
```

```
loss_fn = nn.BCELoss()
optimizer = Adam(ann.parameters(),lr = 0.01)
```

The optimizer needs to know which parameters to work with so it takes as input the parameters in the network.
We also specify a learning rate.

Now we can code the training loop

```
losses = []

for epoch in range(100):
    aggr_loss = 0.0
    for points,labels in dl:

        output = ann(points)

        loss = loss_fn(output,labels)

        aggr_loss += loss

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()

    print(aggr_loss.item())
    losses.append(aggr_loss.item())
```

We want to record the losses from the training

An epoch is one pass through the entire dataset so here we run through the entire dataset (as mini-batches of size 10)

The data loader outputs datapoints and corresponding labels

We send the data points through the network, the output is a tensor of dimension (10,1) with entries, numbers between 0 and 1.

The loss_fn compute the loss between the output and the labels using the BCELoss.

We add the loss to the total loss for the epoch

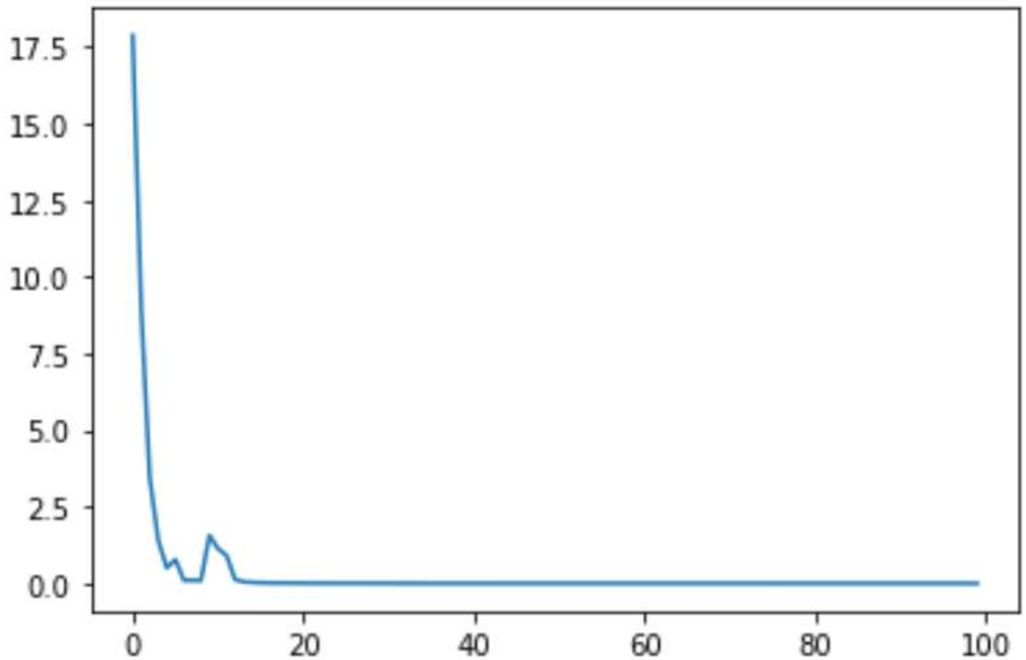
We set all the gradients = 0

The command *.backwards* automaticall computes all the gradients for all the parameters in the network. This is really the main reason for using a framework like Pytorch or TensorFlow, the auto gradient capability

The *optimizer.step* command then does the gradient descent

The losses show that we probably did not have to train for 100 epochs

```
plt.plot(losses);
```



The trained network defines a function $\mathbb{R}^2 \rightarrow 0, 1$