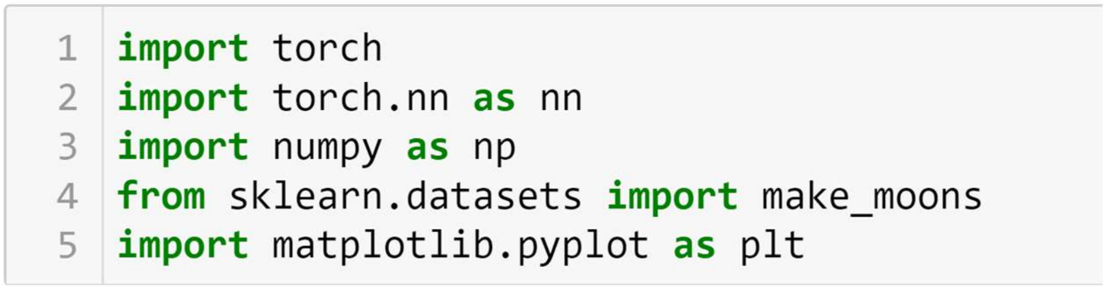
Machine Learning

Building a Pytorch Model

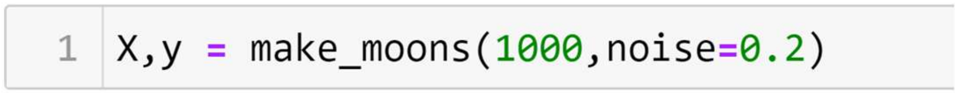
Lecture 14

We begin by importing packages

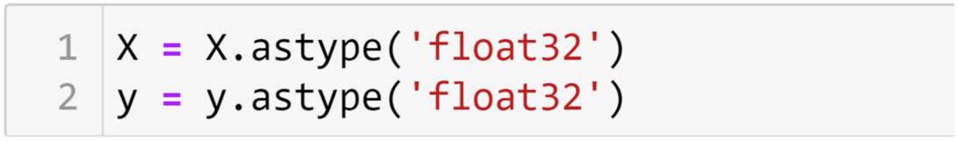


```python
import torch
import torch.nn as nn
import numpy as np
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
```

Making a dataset (and converting the datapoints and labels to float from double)

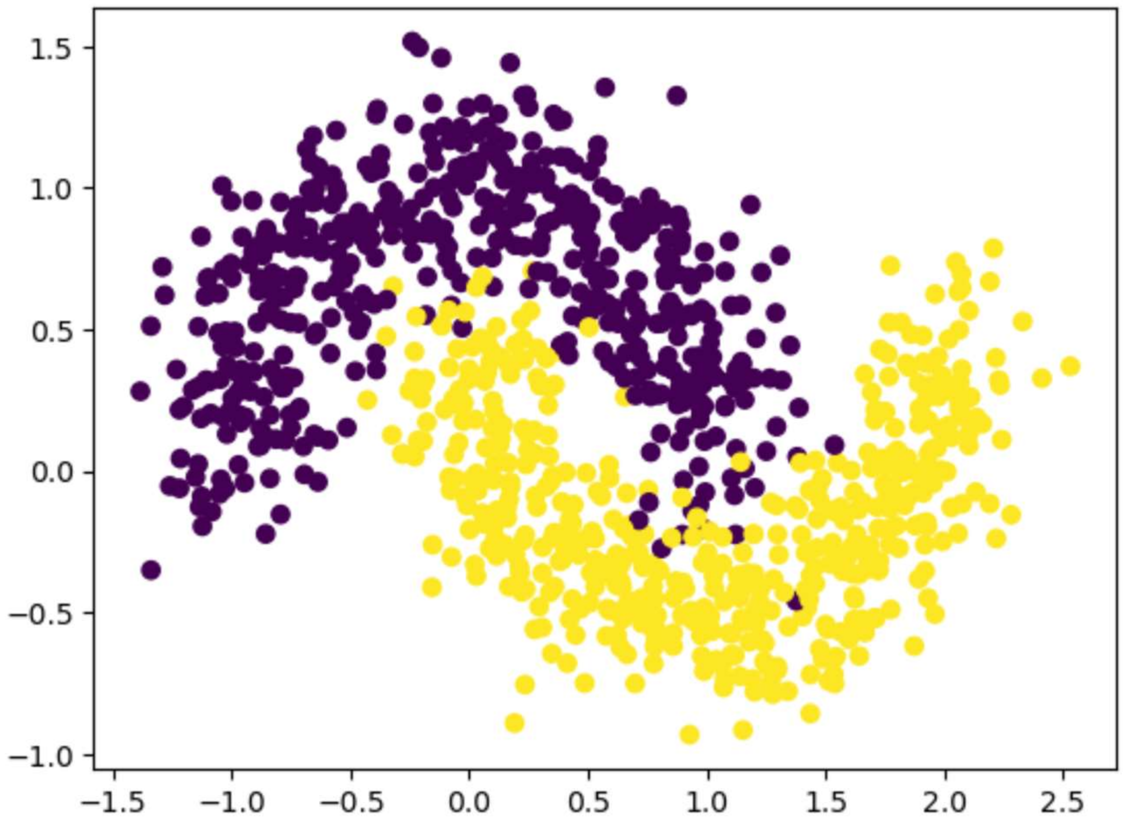


```python
X,y = make_moons(1000,noise=0.2)
```

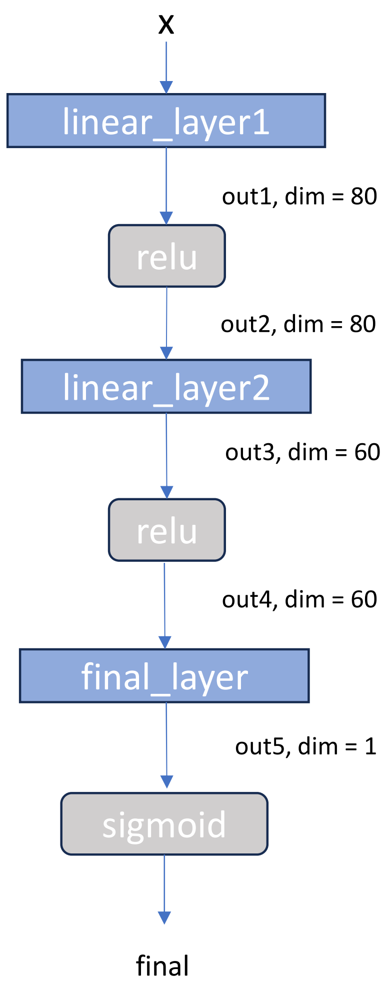


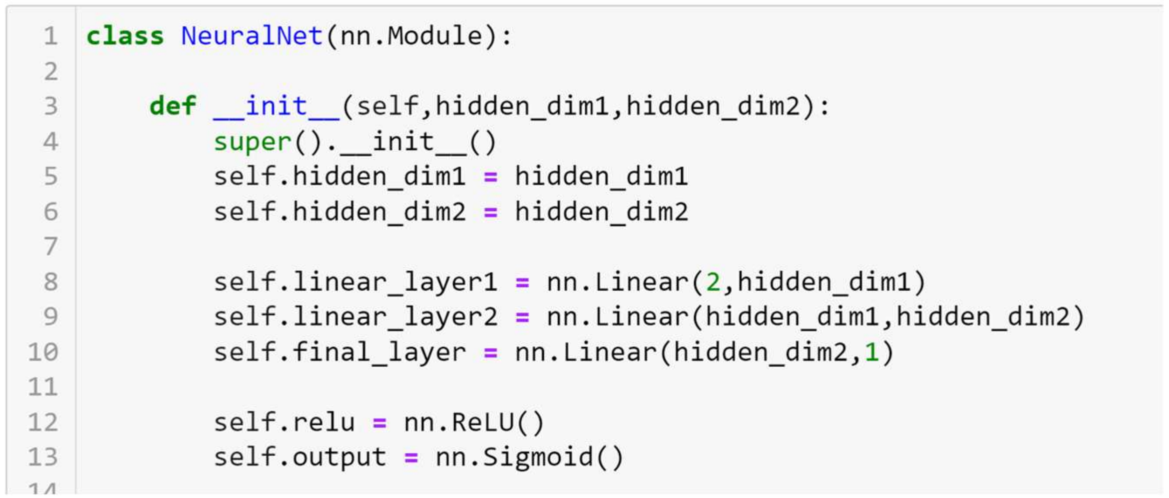
```python
X = X.astype('float32')
y = y.astype('float32')
```

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



The NeuralNet class



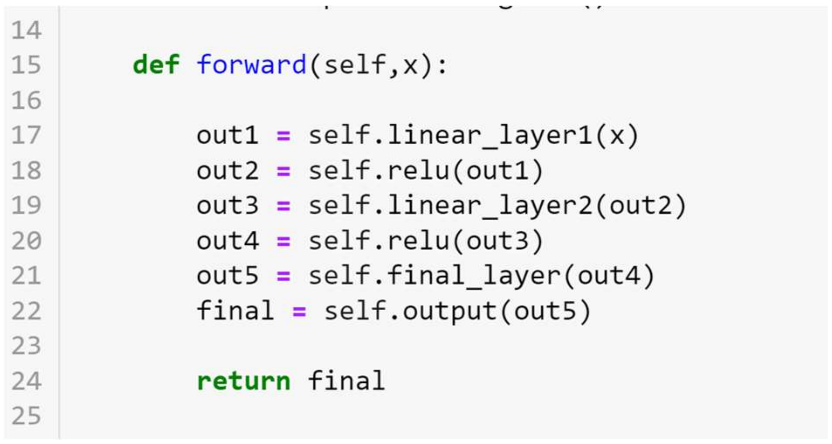

```python
class NeuralNet(nn.Module):

    def __init__(self,hidden_dim1,hidden_dim2):
        super().__init__()
        self.hidden_dim1 = hidden_dim1
        self.hidden_dim2 = hidden_dim2

        self.linear_layer1 = nn.Linear(2,hidden_dim1)
        self.linear_layer2 = nn.Linear(hidden_dim1,hidden_dim2)
        self.final_layer = nn.Linear(hidden_dim2,1)

        self.relu = nn.ReLU()
        self.output = nn.Sigmoid()
```

The *forward* function

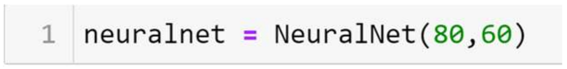


```
    def forward(self,x):

        out1 = self.linear_layer1(x)
        out2 = self.relu(out1)
        out3 = self.linear_layer2(out2)
        out4 = self.relu(out3)
        out5 = self.final_layer(out4)
        final = self.output(out5)

        return final
```

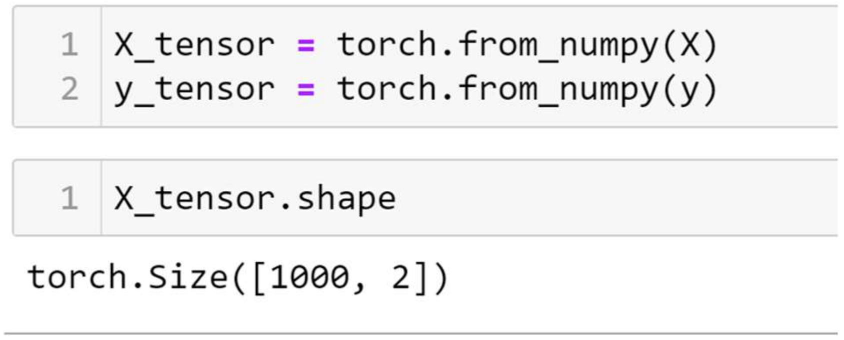
Instantiating the NeuralNet class with parameters hidden_dim1 = 80 And hidden_dim2=60



```python
neuralnet = NeuralNet(80,60)
```

Neural networks operate with tensors i.e. multi-dimensional arrays. So a vector is a 1-dimensional tensor, a matrix is a 2-dimensional tensor, a vector of matrices a 3-dimensional tensor etc.

So we need to turn our data, which are numpy arrays in to tensors. This is very easy in Pytorch

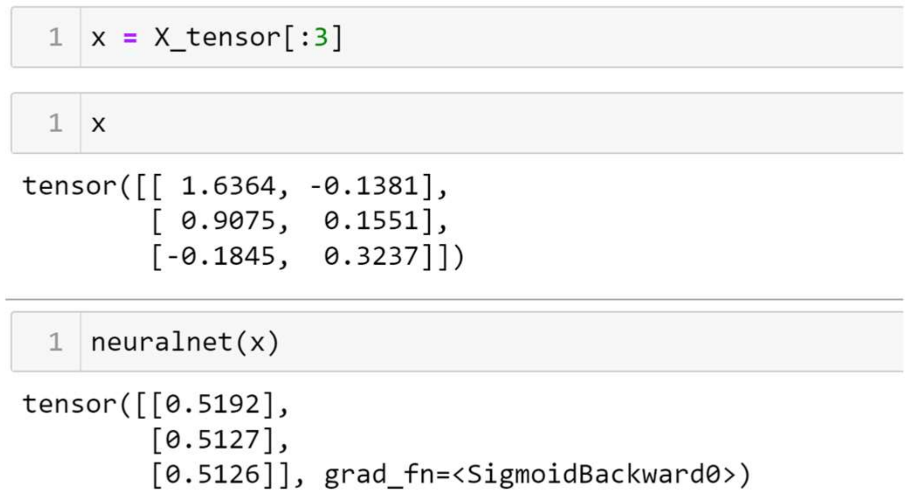


```
X_tensor = torch.from_numpy(X)
y_tensor = torch.from_numpy(y)
```

```
X_tensor.shape
```

```
torch.Size([1000, 2])
```

We can take out a smaller sub-tensor from X_tensor and send it through the neuralnet



```
x = X_tensor[:3]
```

```
x
```

```
tensor([[ 1.6364, -0.1381],
        [ 0.9075,  0.1551],
        [-0.1845,  0.3237]])
```

```
neuralnet(x)
```

```
tensor([[0.5192],
        [0.5127],
        [0.5126]], grad_fn=<SigmoidBackward0>)
```

Training the neuralnet using SGD (Stochastic Gradient Descent).

We feed sub-tensors into the neural net and compare the outputs to the actual labels using the loss-function

SGD with momentum

The idea of the Momentum algorithm comes from physics. Viewing the parameter vector as a particle moving in space. Each update corresponds to giving the particle a push in a certain direction. But the particle was pushed at the previously and so already is moving in some direction i.e. it has momentum. This will change how the stochastic gradient update affects the parameter.

The idea is that this will speed up the rate of convergence.

Here is the algorithm

At epoch $t-1$ the particle received a push of ΔW_{t-1}. Now at epoch t it receives a push of $-\eta\nabla_W Q_j(W_t)$ where Q_j is the part of the loss-function that comes from the training points in batch j.

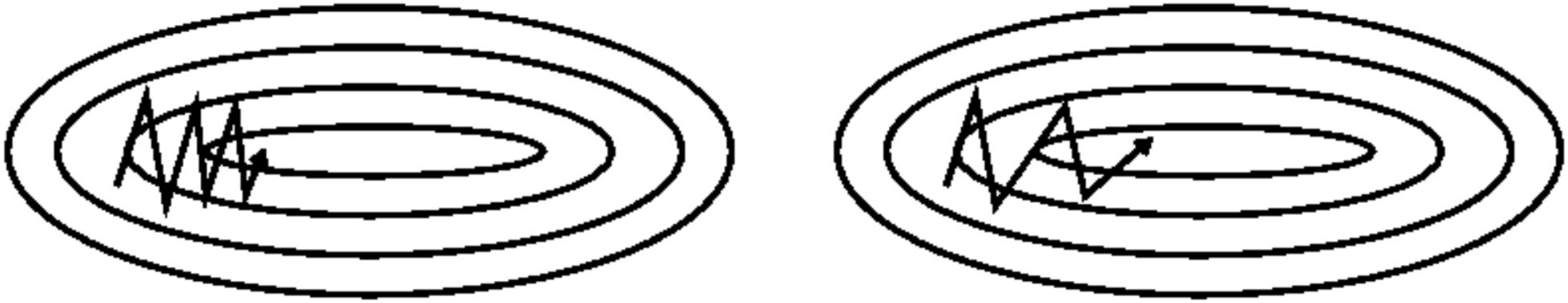
The new push is the convex combination of the previous momentum and the stochastic gradient:

$$\Delta W_t = (1-\eta)\Delta W_{t-1} + \eta\nabla_W(Q_i(W_t))$$

so the update rule is:

$$W_{t+1} = W_t - \alpha\Delta W_t$$

where α is a learning rate parameter. Thus the momentum optimizer requires two parameters α and η



Adagrad

The Adagrad algorithm which means Adaptive Gradient algorithm works by adapting the learning rate at every step . The idea is to smoothen out spikes in the gradient by dividing each component of the gradient at step τ by the square root of the sum of squares of the component at the previous steps

Let $g_{\tau,j} = \sqrt{\sum_t^{\tau} ||\nabla Q_{i_t}(W_t))||^2}$ and let G_τ be the $k \times k$ diagonal matrix with the $g_{\tau,j}$ as the diagonal entries. Then the update rule is given by

$$W_{\tau+1} = W_\tau + \eta G_\tau^{-1} \nabla_W Q_{i_\tau}(W_\tau)$$

The RMSProp (Root Mean Squared Propagation) is similar to Adagrad but a little simpler. Again it works by adjusting the learning rate at each step.

The update rule for the adjustment factor is given by

$$v_{W_t} = \gamma v_{W_{t-1}} + (1-\gamma)||\nabla_W Q_{i_t}(W_t)||^2$$

and the update for the parameters are

$$W_{t+1} = W_t + \frac{\eta}{v_{W_t}} \nabla_W Q_{i_t}(W_t)$$