Machine Learning

Neural Networks

Lecture 13

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.

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

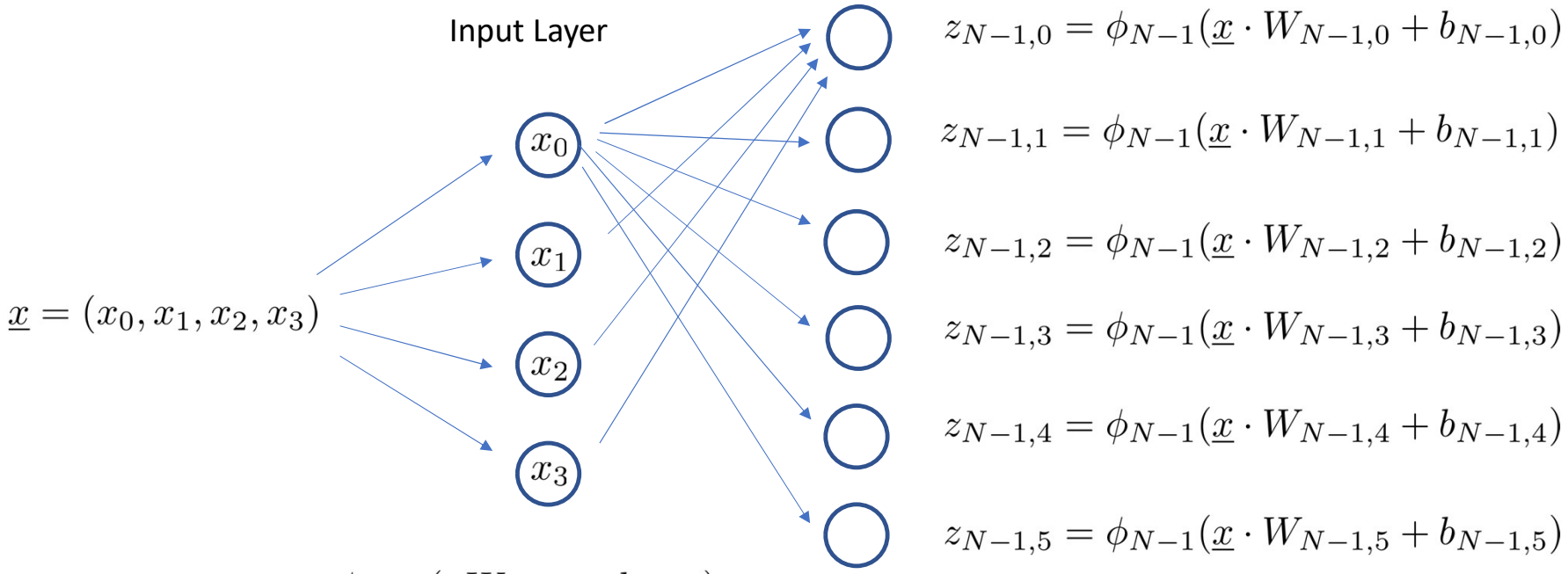
where the matrix W_i, the bias vector $\underline{b}_i$ and the activation function ϕ_i are the data belonging to layer i.

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

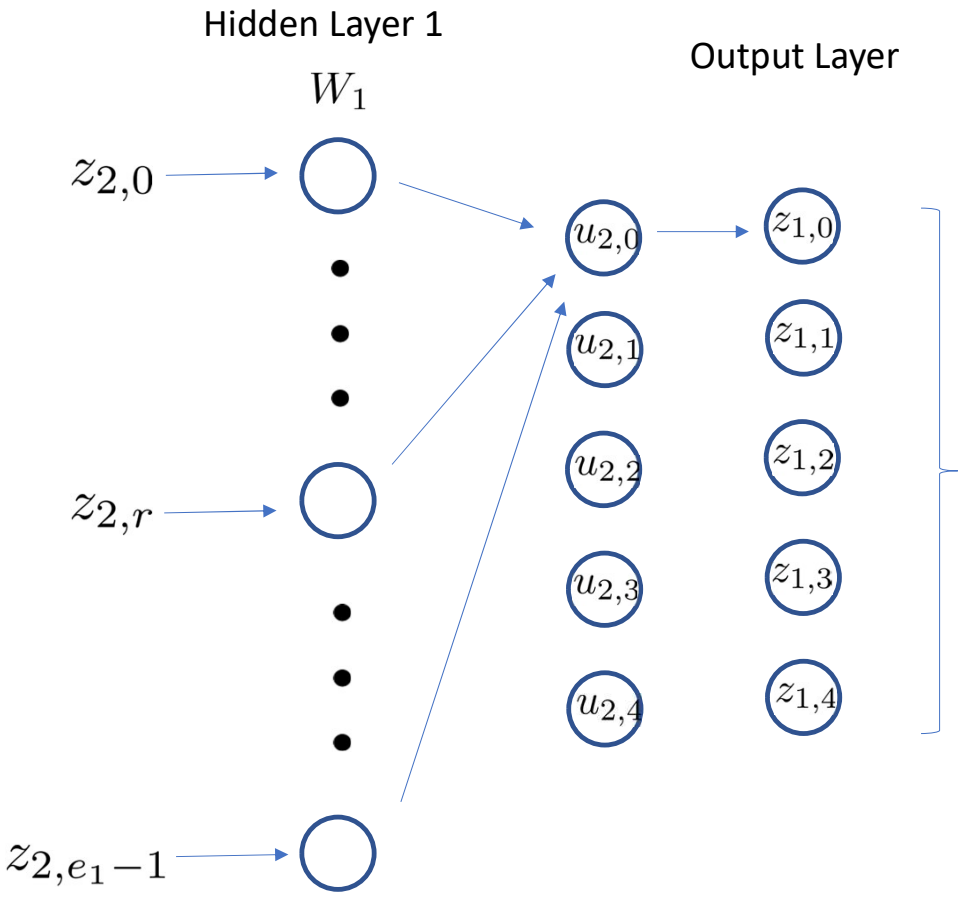
$W_{N-1} = \begin{pmatrix} W_{N-1,0} & W_{N-1,1} & W_{N-1,2} & W_{N-1,3} & W_{N-1,4} & W_{N-1,5} \end{pmatrix}$, of dimension 4×6 so each $W_{N-1,i}$ is a column vector of dimension 4.

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}



$$z_N = \underline{x} \quad \underline{z}_{N-1} = \phi_{N-1}(\underline{x} W_{N-1} + \underline{b}_{N-1}) \quad \underline{u}_N = \underline{x} W_{N-1} + \underline{b}_{N-1}$$

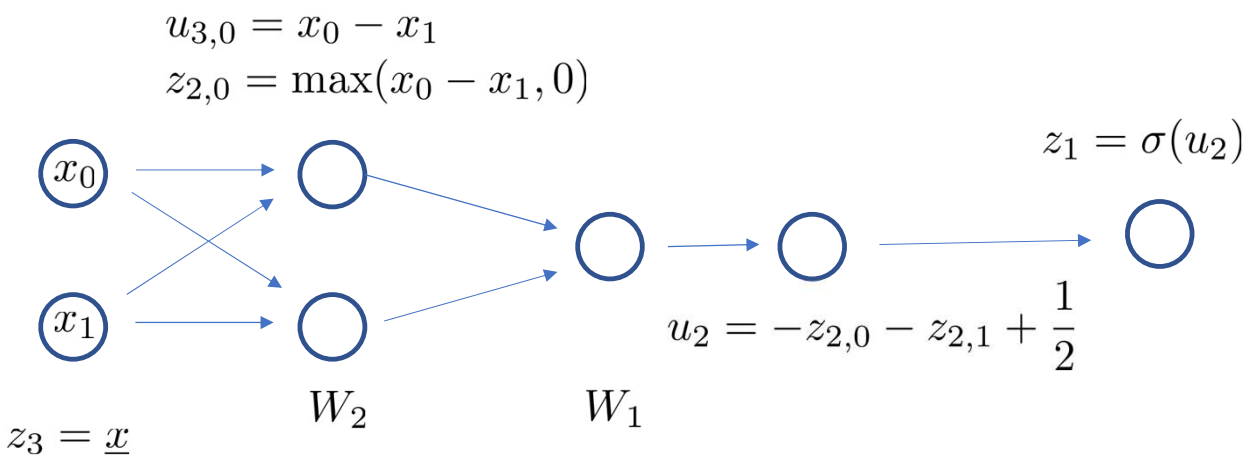
In general $u_{r+1} = z_{r+1} W_r + b_r$ and $z_r = \phi_r(u_{r+1})$



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

$$\phi_1(\underline{z}_2 W_1 + \underline{b}_2) = \phi_1(\underline{u}_2) = \underline{z}_1$$

Let us consider an extremely simple network, with 1 hidden layer and input dimension 2. $W_2 = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \underline{b}_2 = (0,0)$ and $W_1 = \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \underline{b}_1 = \frac{1}{2}$. The activation functions ϕ_2 is the $ReLU$ and ϕ_1 is the logistic function $\sigma(z) = \frac{1}{1+e^{-z}}$



Consider the four input vectors $\underline{x}_1 = (0,0), \underline{x}_2 = (1,0), \underline{x}_3 = (0,1), \underline{x}_4 = (1,1)$

Then

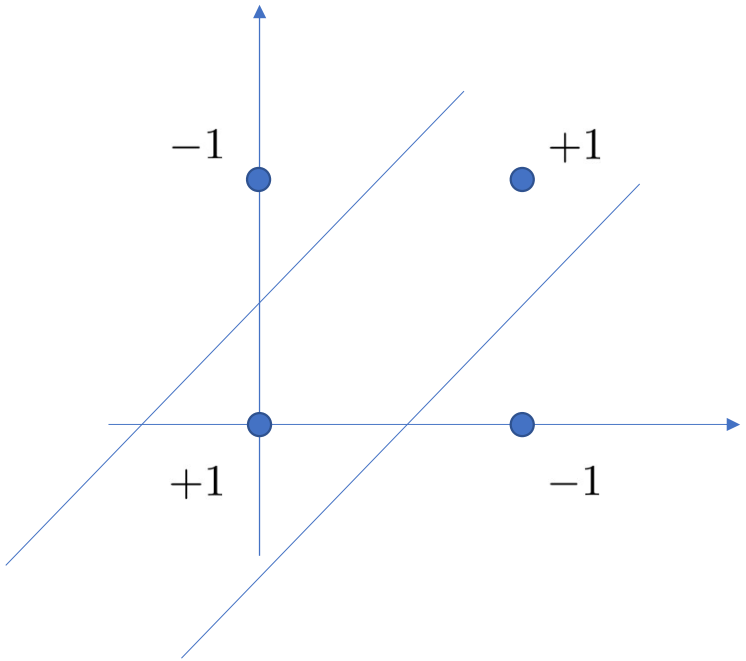
$$\underline{x}_1 \mapsto (0,0) \mapsto \phi_1(0,0) = (0,0) \mapsto \frac{1}{2} \mapsto \sigma(\frac{1}{2})$$

$$\underline{x}_2 \mapsto (1,-1) \mapsto \phi_1(1,-1) = (1,0) \mapsto -1 + \frac{1}{2} \mapsto \sigma(-\frac{1}{2})$$

$$\underline{x}_3 \mapsto (-1,1) \mapsto \phi_1(-1,1) = (0,1) \mapsto -1 + \frac{1}{2} \mapsto \sigma(-\frac{1}{2})$$

$$\underline{x}_4 \mapsto (0,0) \mapsto \phi_1(0,0) = (0,0) \mapsto \frac{1}{2} \mapsto \sigma(\frac{1}{2})$$

Thus this simple network separates the $\neg XOR$ data set



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:

$$f : \underline{x} \mapsto \phi_1(\ldots\left(\phi_{N-1}\left(\underline{x}W_{N-1} + \underline{b}_{N-1}\right)W_{N-2} + \underline{b}_{N-2}\right)\ldots$$

or using the z and u variables

$$\begin{aligned}
x &\mapsto \underline{z}_1 \\
\underline{z}_1 &= \phi_1(\underline{u}_2) \\
\underline{u}_2 &= \underline{z}_2 W_1 + \underline{b}_2 \\
\underline{z}_2 &= \phi_2(\underline{u}_3) \\
\underline{u}_3 &= \underline{z}_3 W_2 + \underline{b}_2 \\
&\vdots
\end{aligned}$$

Let g_i be the function $g_i : \underline{z}_{i+1}, W_i, \underline{b}_i \mapsto \underline{z}_i = \phi_i(\underline{z}_{i+1}W_i + \underline{b}_i)$ and define the *feed-forward functions* $f_i, i = N-1, \ldots, 1$ by

$$f_i(\underline{x}, W_{N-1}, \ldots, W_i, \underline{b}_N, \underline{b}_{N-1}, \ldots, \underline{b}_i) = g_i \circ g_{i+1} \circ \cdots \circ g_{N-1}$$

So

$$\begin{aligned} f_{N-1}(\underline{x}, W_{N-1}, \underline{b}_{N-1}) &= g_{N-1}(\underline{x}, W_{N-1}, \underline{b}_{N-1}) = \underline{z}_{N-1} \\ f_{N-2}(\underline{x}, W_{N-1}, W_{N-2}, \underline{b}_{N-1}, \underline{b}_{N-2}) &= g_{N-2}((g_{N-1}(\underline{z}_{N-1}, W_{N-1}, \underline{b}_{N-1}), W_{N-2}, \underline{b}_{N-2}) = \underline{z}_{N-2} \\ &\vdots \end{aligned}$$

The ANN itself is then the feed-forward function f_1.

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 $W_{N-1}, \ldots, W_1$ and the bias vectors $\underline{b}_{N-1}, \ldots, \underline{b}_1$. 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-1} is the input dimension and d_1 is the output dimension.

For example if $N - 1 = 1$ and $\phi_1 = \sigma$, then the ANN is just the logistic regression $\underline{x} \mapsto \sigma(\underline{x}W + b)$, and there are $d_1 + 1$ parameters where $d_1 = \dim \underline{x}$.

We fix an input vector $\underline{x}$ and the expected output y which is either a label or a real number and consider the loss

$$\mathcal{L}(y, f_1(\underline{x}, W_{N-1}, \ldots, W_1, \underline{b}_{N-1}, \ldots, \underline{b}_1))$$

We want to minimize the loss by doing gradient descent on the parameters.

This means computing the gradient with respect to all the parameters.

Fortunately this is not as difficult as it may seem.

We first compute the derivatives of a function of the form

$$g(\underline{z}, W, \underline{b}) = \phi(\underline{z}W + \underline{b})$$

where $\underline{z}$ is an n-dimensional row vector, W is an $n \times m$ matrix, $\underline{b}$ is m-dimensional and $\phi : \mathbb{R}^m \to \mathbb{R}^m$.

Let $\underline{u} = (u_1, u_2, \ldots, u_m) = \underline{z}W + \underline{b}$ then by the chain rule

$$\frac{\partial g}{\partial z_i} = \sum_k \frac{\partial g}{\partial u_k} \frac{\partial u_k}{\partial z_i}$$

Now

$$\frac{\partial g}{\partial u_k} = \left(\frac{\partial \phi^{(1)}}{\partial u_k}, \frac{\partial \phi^{(2)}}{\partial u_k}, \ldots, \frac{\partial \phi^{(m)}}{\partial u_k} \right)$$

where $\phi = (\phi^{(1)}, \phi^{(2)}, \ldots, \phi^{(m)})$ are the coordinate functions.

$$u_k = z_1 w_{1k} + z_2 w_{2k} + \cdots + z_i w_{ik} + \cdots + z_n w_{nk} + b_k$$

so

$$\frac{\partial u_k}{\partial z_i} = w_{ik}$$

and

$$\frac{\partial g}{\partial z_i} = \left(\sum_k w_{ik} \frac{\partial \phi^{(1)}}{\partial u_k}, \sum_k w_{ik} \frac{\partial \phi^{(2)}}{\partial u_k}, \ldots, \sum_k w_{ik} \frac{\partial \phi^{(m)}}{\partial u_k} \right)$$

Hence $\dfrac{\partial g}{\partial z_i}$ is the i'th row in the matrix product $\nabla\phi \cdot W^T$ where

$$\nabla\phi = \begin{pmatrix} \dfrac{\partial\phi^{(1)}}{\partial u_1} & \dfrac{\partial\phi^{(1)}}{\partial u_2} & \cdots & \dfrac{\partial\phi^{(1)}}{\partial u_m} \\ \dfrac{\partial\phi^{(2)}}{\partial u_1} & \dfrac{\partial\phi^{(2)}}{\partial u_2} & \cdots & \dfrac{\partial\phi^{(2)}}{\partial u_m} \\ \vdots & \vdots & \vdots & \vdots \\ \dfrac{\partial\phi^{(m)}}{\partial u_1} & \dfrac{\partial\phi^{(m)}}{\partial u_2} & \cdots & \dfrac{\partial\phi^{(m)}}{\partial u_m} \end{pmatrix}$$

is the Jacobian matrix of ϕ.

We write this as

$$\frac{\partial g}{\partial \underline{z}} = \nabla\phi \cdot W^T$$

In our case the activation function is just applying the $ReLU$ function to each coordinate of the vector $\underline{u}$, so the Jacobian matrix is very simple

$$\begin{pmatrix} \frac{\partial ReLU}{\partial u_1} & 0 & 0 & \dots & 0 \\ 0 & \frac{\partial ReLU}{\partial u_2} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \frac{\partial ReLU}{\partial u_m} \end{pmatrix}$$

The derivative of the $ReLU$ is the Heaviside function:

$$h(u) = \begin{cases} 0 \text{ if } u \leq 0 \\ 1 \text{ if } u > 0 \end{cases}$$

So the derivative

$$\frac{\partial g}{\partial \underline{z}} = h(\underline{u}) \cdot W^T$$

where

$$h(u) = \begin{pmatrix} h(u_1) & 0 & 0 & \dots & 0 \\ 0 & h(u_1) & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & h(u_m) \end{pmatrix}$$

Next we have

$$\frac{\partial g}{\partial w_{ij}} = \sum_k \frac{\partial g}{\partial u_k} \frac{\partial u_k}{\partial w_{ij}}$$

and

$$\frac{\partial u_k}{\partial w_{ij}} = \begin{cases} 0 \text{ if } j \neq k \\ z_i \text{ if } j = k \end{cases}$$

so

$$\frac{\partial g}{\partial w_{ij}} = \left(\frac{\partial \phi^{(1)}}{\partial u_j} z_i, \frac{\partial \phi^{(2)}}{\partial u_j} z_i, \dots, \frac{\partial \phi^{(m)}}{\partial u_j} z_i \right) = (0, 0, \dots, h(u_j) z_i, 0, \dots)$$

and we can write

$$\frac{\partial g}{\partial W} = \begin{pmatrix} \underline{z}^T \cdot (h(u_1), 0, \dots, 0) \\ \underline{z}^T \cdot (0, h(u_2), \dots, 0) \\ \vdots \\ \underline{z}^T \cdot (0, 0, \dots, h(u_m)) \end{pmatrix}$$

Remark that this is an array of matrices, each entry is the product of a $m \times 1$ by a $1 \times n$, hence an $m \times n$ matrix. We can view it as a *tensor* of dimension (m, n, m).

Computing

$$\frac{\partial g}{\partial \underline{b}}$$

is very similar (but easier)

$$\frac{\partial g}{\partial b_i} = \sum_k \frac{\partial g}{\partial u_k} \frac{\partial u_k}{\partial b_i} = \sum_k \frac{\partial \phi}{\partial u_k} \frac{\partial (\underline{z} \cdot W_{|k} + b_k)}{\partial b_i} = \frac{\partial \phi}{\partial u_i}$$

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 $i = 1, 2, \ldots, N-1$. Using the chain rule we get

$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial z_1} \cdot \frac{\partial g_1}{\partial z_2} \cdot \frac{\partial g_2}{\partial z_3} \cdot \ldots \cdot \frac{\partial g_{i-1}}{\partial z_i} \cdot \frac{\partial g_i}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial z_1} \cdot \nabla \phi_1 \cdot W_1^T \cdot \nabla \phi_2 \cdot W_2^T \cdot \ldots \cdot \nabla \phi_{i-1} \cdot W_{i-1}^T \cdot \frac{\partial g_i}{\partial W_i}$$

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

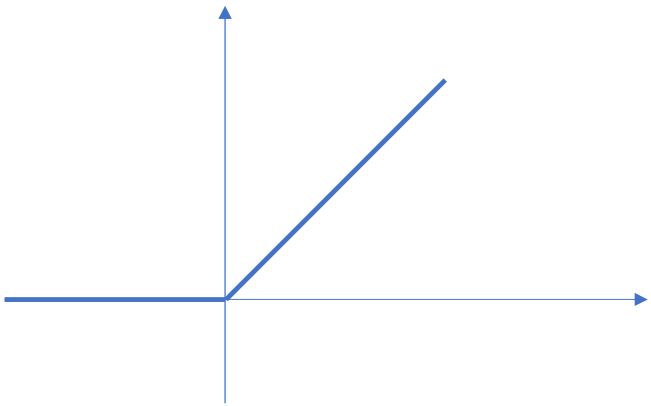
$$\frac{\partial \mathcal{L}}{\partial \underline{b}_i} = \frac{\partial \mathcal{L}}{\partial z_1} \cdot \frac{\partial g_1}{\partial z_1} \cdot \frac{\partial g_2}{\partial z_2} \cdot \ldots \cdot \frac{\partial g_{i-1}}{\partial z_{i-1}} \cdot \frac{\partial g_i}{\partial \underline{b}_i} = \frac{\partial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

Let's illustrate this with an example. We consider a 2-layer binary classification problem. The final activation function is given by sending a data point $\underline{x}$ to $z_1 = \sigma(yf_1(\underline{x}))$. The other activation function ϕ_2 is the $ReLU$ function Thus the NN sends and input vector $\underline{x} = (x_1, x_2, \ldots, x_n)$ to

$$z_1 = g_1(z_2, W_1, b_1) = \sigma(z_2 \cdot W_1 + b_1)$$

and

$$z_2 = g_2(\underline{x}, W_2, \underline{b}_2) = ReLU(\underline{x}W_2 + \underline{b}_2))$$



Here W_1 is an $m \times 1$ matrix, W_2 an $n \times m$ matrix, b_1 a scalar and $\underline{b}_2$ a $1 \times m$ vector.

Let $y = \pm 1$ be the label of $\underline{x}$ then the loss function is

$$\mathcal{L}(z_1) = \begin{cases} -\log(z_1) \text{ if } y = 1 \\ -\log(1 - z_1) \text{ if } y = -1 \end{cases}$$

Hence we get

$$\frac{\partial \mathcal{L}}{\partial W_1} = \begin{cases} \dfrac{\partial \mathcal{L}}{\partial z_1} \cdot \dfrac{\partial g_1}{\partial W_1} = -\dfrac{1}{z_1} \cdot \underline{z_2}^T \cdot \sigma'(u_2) \text{ if } y = 1 \\ \dfrac{1}{1 - z_1} \cdot \underline{z_2}^T \cdot \sigma'(u_2) \text{ if } y = -1 \end{cases}$$

We get

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial z_1} \cdot \frac{\partial g_1}{\partial z_2} \cdot \frac{\partial g_2}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial z_1} \sigma'(u_1) W_1^T \cdot \begin{pmatrix} \underline{x}^T \cdot (h(u_2^{(1)}), 0, 0, \ldots, 0) \\ \underline{x}^T \cdot (0, h(u_2^{(2)}), 0, \ldots, 0) \\ \vdots \\ \underline{x}^T \cdot (0, 0, \ldots, 0, h(u_2^{(m)}) \end{pmatrix}$$

The product $W_1^T \cdot \begin{pmatrix} \underline{x}^T \cdot (h(u_2^{(1)}), 0, 0, \ldots, 0) \\ \underline{x}^T \cdot (0, h(u_2^{(2)}), 0, \ldots, 0) \\ \vdots \\ \underline{x}^T \cdot (0, 0, \ldots, 0, h(u_2^{(m)}) \end{pmatrix}$ is a *tensor contraction*, it is the linear combination of $n \times m$ matrices

$$w_1 \left(\underline{x}^T \cdot (h(u_2^{(1)}), 0, 0, \ldots, 0) \right) + \cdots + w_m \left(\underline{x}^T \cdot (0, 0, \ldots, 0, h(u_2^{(m)}) \right)$$

It is not hard to code a fully connected ANN for binary classification

The constructor declares the dimensions of the layers and initializes the parameters.

We also define the ReLU function, the Logistic function and the Heaviside function

```python
class ANN():

    def __init__(self,dims):

        self.N = len(dims)
        self.params_W = {'W{}'.format(i):0.1*np.random.randn(dims[i],dims[i-1]) for i in range(1,self.N)}
        self.params_b = {'b{}'.format(i): 0.1*np.zeros((1,dims[i-1])) for i in range(1,self.N)}

    def ReLU(self,u):
        return np.array([t if t > 0 else 0 for t in u[0]]).reshape(u.shape)

    def sigma(self,u):
        return 1./(1 + np.exp(-u))

    def h(self,u):
        return np.array([0 if t <= 0 else 1 for t in u[0]]).reshape(u.shape)
```

The forward pass takes a data point and computes all the u's and the z's through the layers on the NN. The values are stored in dicts

```
def forward_pass(self,x):
    u = {}
    z = {}
    z['z{}'.format(self.N - 1)] = x
    for i in range(self.N-1,0,-1):
        u['u{}'.format(i)] = z['z{}'.format(i)] @ self.params_W['W{}'.format(i)] + self.params_b['b{}'.format(i)]
        z['z{}'.format(i-1)] = self.ReLU(u['u{}'.format(i)])
    z['z0'] = self.sigma(u['u1'])

    return u,z
```

The back_prop function takes a data point and it's label and computes the gradients of the loss function with respect to all the parameters.

We first compute the gradients of the activation functions, they are all Heaviside functions = derivative of ReLU, except the first one which is the derivative of the Logistic function

```
32      def back_prop(self, y, x):
33
34          u, z = self.forward_pass(x)
35
36          grad_phi = {'grad_phi{}'.format(i): np.diag(
37              self.h(u['u{}'.format(i)])[0]) for i in range(2, self.N)}
38          grad_phi['grad_phi1'] = self.sigma(u['u1']) * self.sigma(-u['u1'])
39
```

Next the derivatives of the g functions with respect to the matrix parameters and the biases

```
derivative_g_W = {'gW{}'.format(i): np.array([z['z{}'.format(i)].T @ (np.array(grad_phi['grad_phi{}'.\
                                                                    format(i)][:, j]).reshape(1, -1))
                                              for j in range(grad_phi['grad_phi{}'.format(i)].shape[1])])
                  for i in range(2, self.N)}


derivative_g_W['gW1'] = self.sigma(- u['u1']) * \
    self.sigma(u['u1']) * z['z1'].T
```

```
# Derivatives $\dfrac{\partial g_i}{\partial b_i}$
derivative_g_b = {'gb{}'.format(i): [np.array(grad_phi['grad_phi{}'.format(i)][:, j]).reshape(1, -1)
                                     for j in range(grad_phi['grad_phi{}'.format(i)].shape[1])]
                  for i in range(2, self.N)}
derivative_g_b['gb1'] = self.sigma(u['u1']) * self.sigma(- u['u1'])
```

Remark we use the @ symbol for matrix multiplication in numpy and we use the formula for the derivative of the logistic function

$$\sigma'(u) = \sigma(u)\sigma(-u)$$

Now we can compute the gradients and store them in a dict

```
    grad_W = {}
    products = {}
    products['1'] = grad_phi['grad_phi1'] @ self.params_W['W1'].T
    for i in range(2, self.N-1):
        products['{}'.format(i)] = products['{}'.format(
            i-1)] @ grad_phi['grad_phi{}'.format(i)] @ self.params_W['W{}'.format(i)].T

    if z['z0'][0][0] == 0 and y == 1:
        a = -1e8
    elif z['z0'][0][0] == 1 and y == -1:
        a = 1e8
    else:
        a = -2*y/(2 * y * z['z0'][0][0] - y + 1)

    grad_W['grad_W1'] = (a * z['z1'].T)[0]
    for i in range(2, self.N):
        grad_W['grad_W{}'.format(i)] = a * np.tensordot(
            products['{}'.format(i-1)], derivative_g_W['gW{}'.format(i)], 1)[0]

    grad_b = {}
    grad_b['grad_b1'] = a * self.sigma(-u['u1'])*self.sigma(u['u1'])
    for i in range(2, self.N):
        grad_b['grad_b{}'.format(i)] = a * np.tensordot(
            products['{}'.format(i-1)], derivative_g_b['gb{}'.format(i)], 1)[0]

    return grad_W, grad_b
```

The value *a*

```
a = -2*y/(2 * y * z['z0'][0][0] - y + 1)
```

$$= \begin{cases} \frac{-2}{2z_0} = -\frac{1}{z_0} \text{ if } y = 1 \\ \frac{-2(-1)}{2(-1)z_0 - (-1) + 1} = \frac{1}{1 - z_0} \text{ if } y = -1 \end{cases}$$

is the derivative of

$$-\log(\sigma(yu_1)) = \begin{cases} -\log(z_0) \text{ if } y = 1 \\ -\log(1 - z_0) \text{ if } y = -1 \end{cases}$$

If the denominator = 0 we set the value of *a* to a very large number (or very negative number depending of whether *y* is -1 or +1)

To do gradient descent we have to compute the gradient at every step of the descent sequence for every data point in the training set. The gradient is the average of the gradients over all the data points so if the data set is large each calculation of the gradient will take a long time.

The average can be approximated by taking subsets of the data and computing the average over each subset. If we continually randomly select subsets of the data sets then the averages will converge to the average over the entire data set and if we use the average over the subset as an approximation of the average gradient over the whole training set in every step of the gradient descent sequence it will converge to the same value as by taking the average over the full training set at every step

This is the principle behind *Stochastic Gradient Descent .*

The most coarse approximation is that we just select a single data point at a time as an approximation to the full gradient.

We can also take a *mini-batch* of points and compute the average over the batch. By taking enough batches we will cover the whole training set.

An *epoch* is a pass through the training set

The fit function in our ANN class uses mini-batch SGD

```
def fit(self, X, y, batch_size=16, epochs=10, lr=0.001,shuffle=True):
    losses = []
    X = np.array([x.reshape(1, -1) for x in X])
    n_batches = len(X)//batch_size

    for epoch in range(epochs):

        if shuffle:
            p = np.random.permutation(len(X))
            X = np.array(X)
            y = np.array(y)
            X = X[p]
            y = y[p]

        for j in range(n_batches):
            X_batch = X[j*batch_size:(j+1)*batch_size]
            y_batch = y[j*batch_size:(j+1)*batch_size]

            grad_W_batch = [self.back_prop(y, x)[0]
                            for (y, x) in zip(y_batch, X_batch)]


            grad_b_batch = [self.back_prop(y, x)[1]
                            for (y, x) in zip(y_batch, X_batch)]

            grad_W_batch_avr = {k: np.mean([grad_W_batch[i][k] for i in range(len(X_batch))], axis=0)
                                for k in grad_W_batch[0].keys()}


            grad_b_batch_avr = {k: np.mean([grad_b_batch[i][k] for i in range(len(X_batch))], axis=0)
                                for k in grad_b_batch[0].keys()}
```

If we shuffle (which we will almost always want to) we permute the data points and the labels and then divide the shuffled data into mini-batches of size = the parameter `batch_size`

```
if shuffle:
    p = np.random.permutation(len(X))
    X = np.array(X)
    y = np.array(y)
    X = X[p]
    y = y[p]

for j in range(n_batches):
    X_batch = X[j*batch_size:(j+1)*batch_size]
    y_batch = y[j*batch_size:(j+1)*batch_size]
```

For each mini-batch we compute the average of the gradients over the data in the mini-batch

```
grad_W_batch = [self.back_prop(y, x)[0]
                for (y, x) in zip(y_batch, X_batch)]


grad_b_batch = [self.back_prop(y, x)[1]
                for (y, x) in zip(y_batch, X_batch)]

grad_W_batch_avr = {k: np.mean([grad_W_batch[i][k] for i in range(len(X_batch))], axis=0)
                    for k in grad_W_batch[0].keys()}


grad_b_batch_avr = {k: np.mean([grad_b_batch[i][k] for i in range(len(X_batch))], axis=0)
                    for k in grad_b_batch[0].keys()}
```

We then do the Gradient Descent with these averages (which are approximations to the full gradients of the loss function)

```
for i in range(1, self.N):
    self.params_W['W{}'.format(i)] = self.params_W['W{}'.format(
        i)] - lr * grad_W_batch_avr['grad_W{}'.format(i)]
    self.params_b['b{}'.format(i)] = self.params_b['b{}'.format(
        i)] - lr * grad_b_batch_avr['grad_b{}'.format(i)]
```

For each epoch we compute the loss and aggregate the losses in an array and return this array

```
    loss = np.mean([- np.log(label*self.forward_pass(x)[1]['z0']
                             [0][0] - 0.5*label + 0.5) for label, x in zip(y, X)])
    print(epoch, loss)
    losses.append(loss)
return losses
```

```
1  dimensions = [1,20,40,20,2]
```

executed in 4ms, finished 11:56:50 2019-05-25

```
1  ann = ANN(dimensions)
```

executed in 4ms, finished 11:56:54 2019-05-25

```
1 losses = ann.fit(X,y_,batch_size=10,epochs=50,lr=0.015,shuffle=True)
```

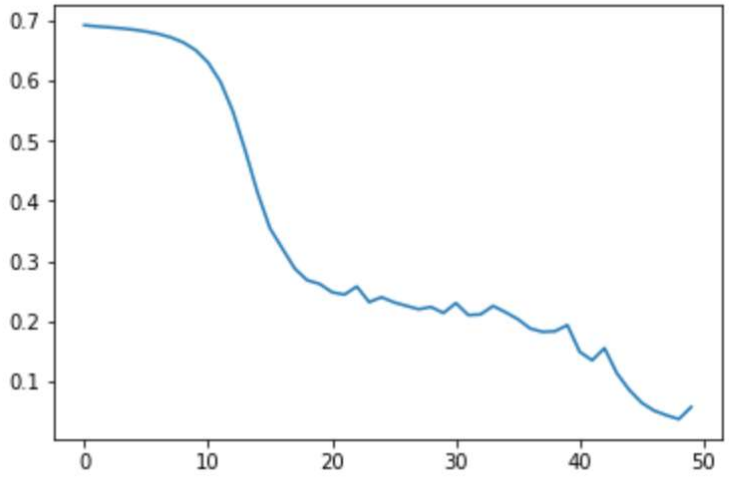
executed in 1m 9.08s, finished 11:15:42 2019-05-25

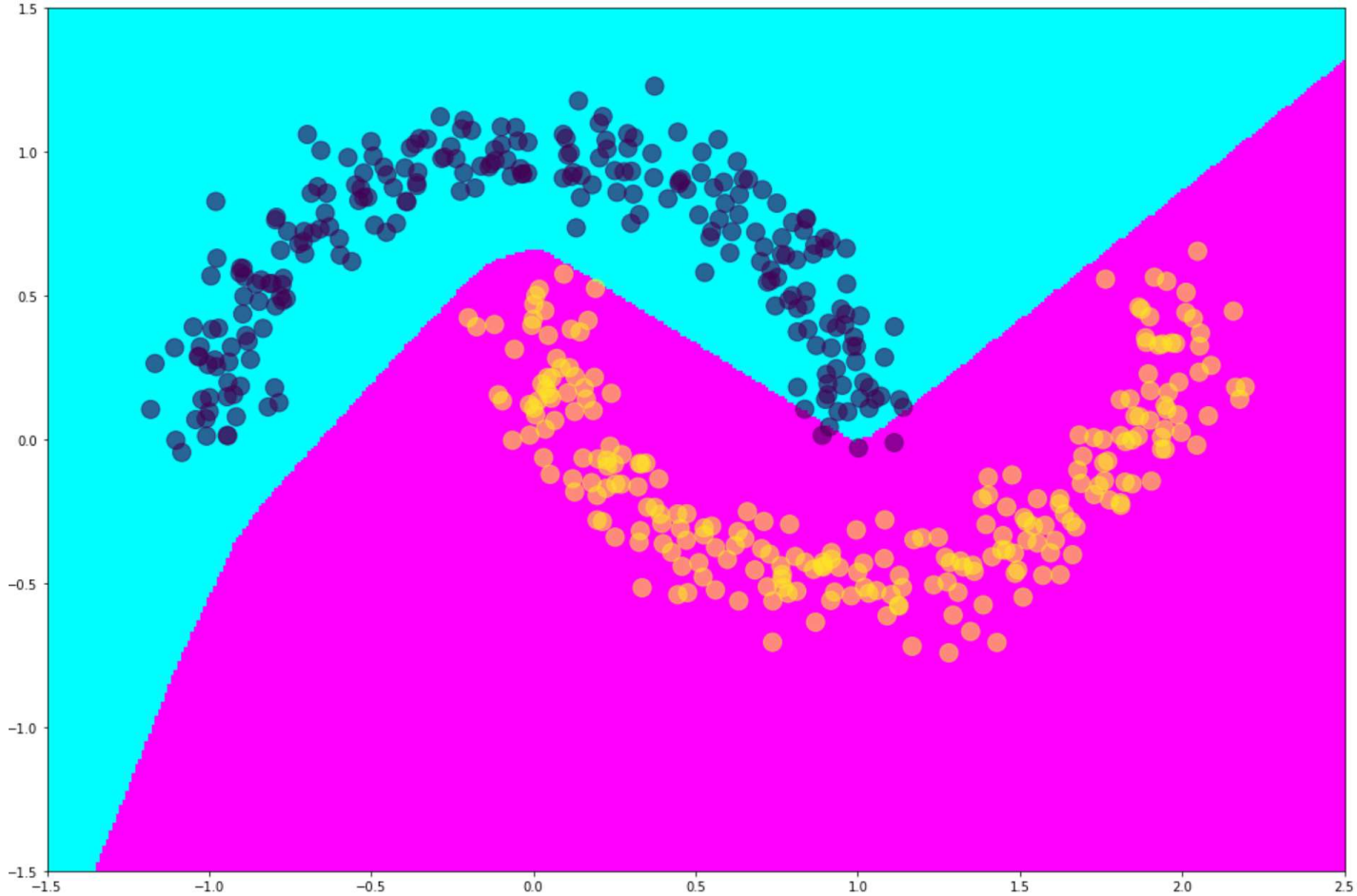
```
0 0.6920914218539412
1 0.6901758439997941
2 0.6886455902547126
3 0.6868494967911604
4 0.6846045436720435
5 0.6816162040300733
6 0.6775454303179168
7 0.6717961636443843
8 0.663353008345407
9 0.6504509800485495
10 0.6299897986183648
11 0.5976712543774952
12 0.5486127516102935
13 0.4831226266805213
14 0.4129028908159626
15 0.3538832266673509
```

```
45 0.06403535214828729
46 0.05102955469464298
47 0.043273385060682075
48 0.03683534461110267
49 0.0570700361862327
```

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

executed in 171ms, finished 11:15:46 2019-05-25




1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

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.

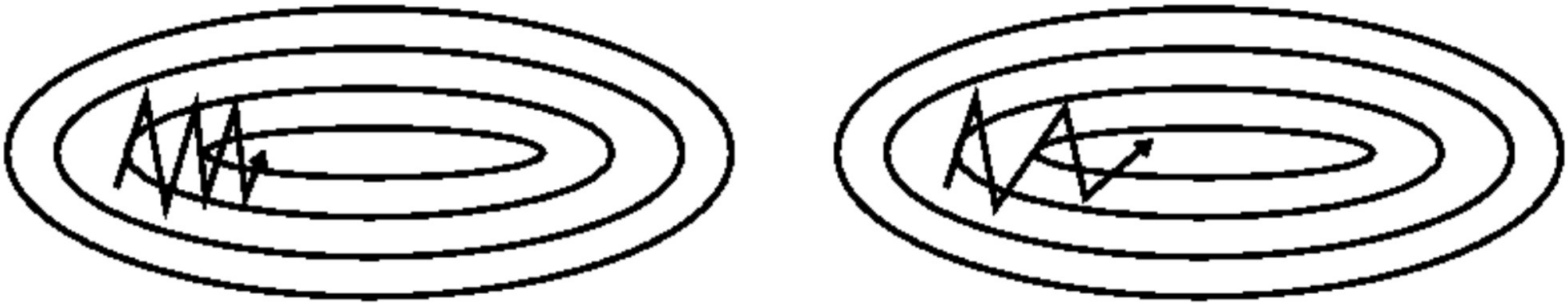
The new push is the convex combination of the previous momentum and the stochastic gradient:

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

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



For the momentum algorithm we need to keep track of the descent steps at all previous epochs. We initialize data structures to hold all these descent steps

```
def momentum(model,X,y,batch_size=10,epochs=25,alpha=0.01,eta=0.4,shuffle=True):


    losses = []

    X = np.array([x.reshape(1, -1) for x in X])

    n_batches = len(X)//batch_size

    deltaW = epochs * [{k:np.zeros((n_batches,model.W_shapes[k][0],model.W_shapes[k][1])) for k in model.params_W.keys()}
    deltab = epochs * [{k:np.zeros((n_batches,model.b_shapes[k][0],model.b_shapes[k][1])) for k in model.params_b.keys()}
```

Most of the rest of the code is the same as for the SGD (the fit function). The only change is the computation of the increments as convex combinations

```
if epoch == 0:

        for k in model.params_W.keys():
            deltaW[epoch][k][j] =  - eta * grad_W_batch_avr['grad_'+k]

        for k in model.params_b.keys():
            deltab[epoch][k][j] =  - eta * grad_b_batch_avr['grad_'+k]


else:


    for k in model.params_W.keys():
        deltaW[epoch][k][j] = (1 - eta) * deltaW[epoch-1][k][j] + eta * grad_W_batch_avr['grad_'+k]

    for k in model.params_b.keys():
        deltab[epoch][k][j] = (1 - eta) * deltab[epoch-1][k][j] + eta * grad_b_batch_avr['grad_'+k]
```

```
1 losses = momentum(ann,X,y_,batch_size=10,epochs=30,alpha=0.04,eta = 0.7,shuffle=True)
```

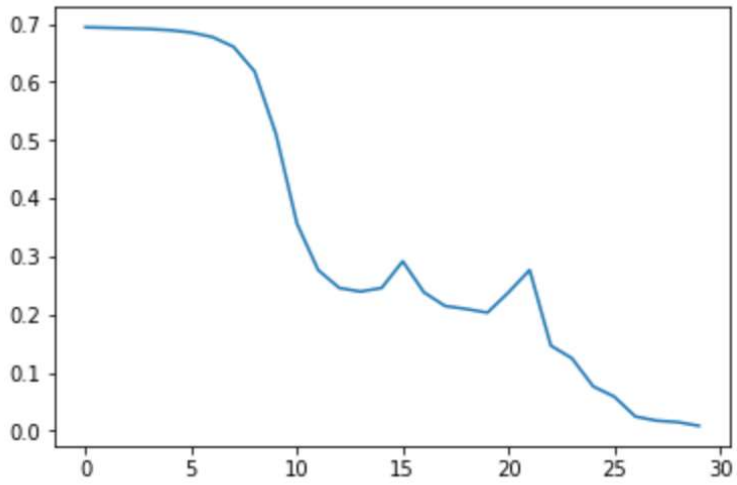
executed in 38.5s, finished 12:14:34 2019-05-25

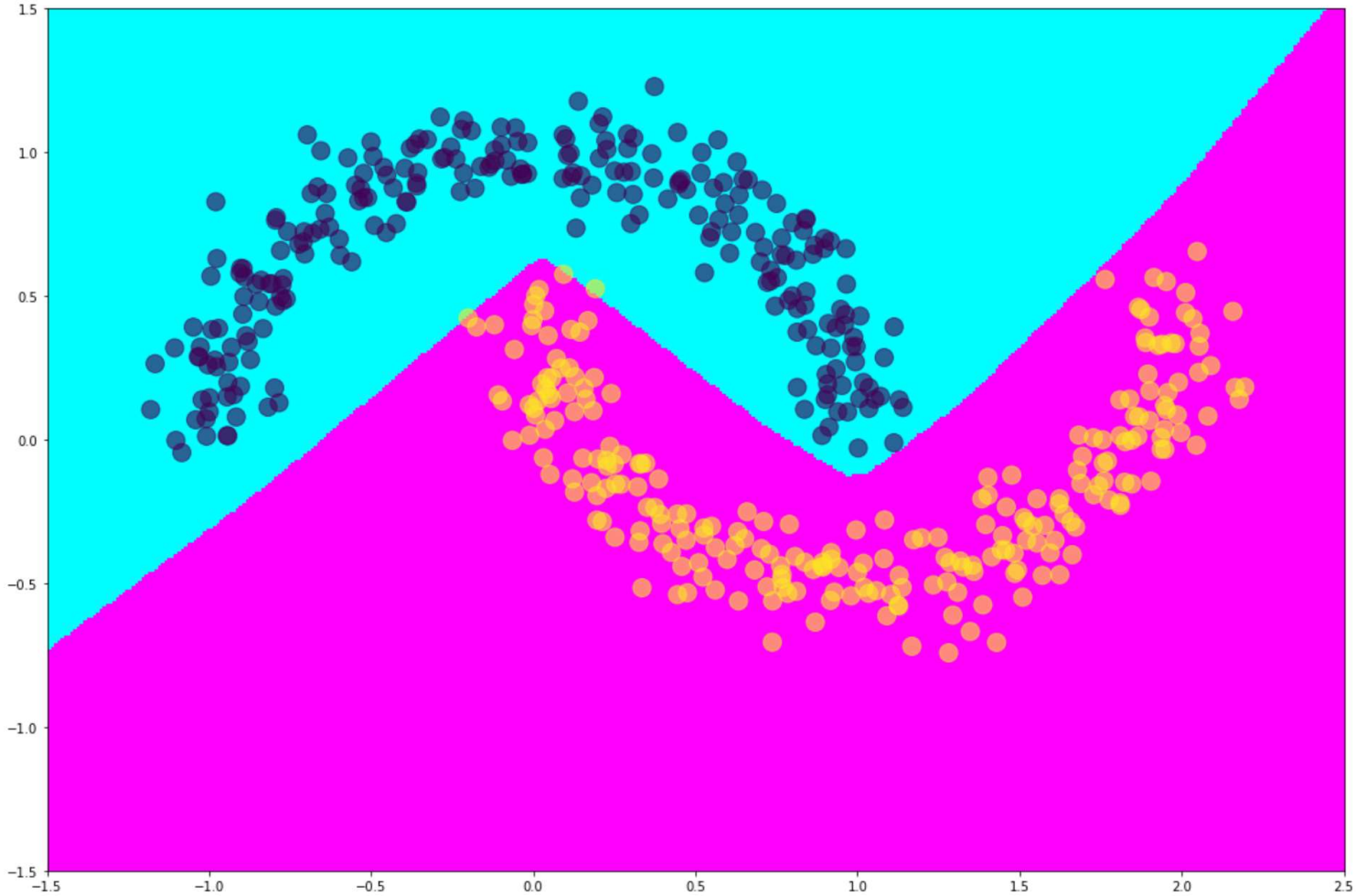
```
0 0.6945607761116326
1 0.6936454576592447
2 0.6926544404584976
3 0.6915573066012943
4 0.6893948069012569
5 0.6854673292227036
6 0.677670207244285
7 0.6608977075768396
8 0.6183204778586071
9 0.5115467295228708
10 0.35632143837554925
11 0.276505060697236
12 0.2456360197853956
13 0.2394659238400982
14 0.24561174932511118
```

```
26 0.02430348418300492
27 0.017279706363026375
28 0.014729325297988484
29 0.008415397459010309
```

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

executed in 128ms, finished 12:14:39 2019-05-25




1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

Adagrad

The Adagrad algorithm which means Adaptive Gradient algorithm works by adapting the learning rate at every step (like the Backtracking algorithm). 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

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

```
def adagrad(model,X,y,batch_size=10,epochs=25,eta=0.04,shuffle=True):

    losses = []

    X = np.array([x.reshape(1, -1) for x in X])

    n_batches = len(X)//batch_size

    GW = epochs * [{k:np.zeros((n_batches,model.W_shapes[k][0],model.W_shapes[k][1])) for k in model.params_W.keys()}]
    Gb = epochs * [{k:np.zeros((n_batches,model.b_shapes[k][0],model.b_shapes[k][1])) for k in model.params_b.keys()}]
```

```
if epoch == 0:

        for k in model.params_W.keys():
            GW[0][k][j] =  grad_W_batch_avr['grad_'+k]**2

        for k in model.params_b.keys():
            Gb[0][k][j] =  grad_b_batch_avr['grad_'+k]**2


else:


    for k in model.params_W.keys():
        GW[epoch][k][j] = GW[epoch-1][k][j] + grad_W_batch_avr['grad_'+k]**2

    for k in model.params_b.keys():
        Gb[epoch][k][j] = Gb[epoch-1][k][j] + grad_b_batch_avr['grad_'+k]**2



for i in range(1, model.N):
        model.params_W['W{}'.format(i)] = model.params_W['W{}'.format(i)] \
        - eta * ((GW[epoch]['W{}'.format(i)][j] + 1e-8)**(-0.5)) * grad_W_batch_avr['grad_W{}'.format(i)

        model.params_b['b{}'.format(i)] = model.params_b['b{}'.format(i)] \
        - eta * ((Gb[epoch]['b{}'.format(i)][j] + 1e-8)**(-0.5)) * grad_b_batch_avr['grad_b{}'.format(i)
```

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

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

```
def rmsprop(model,X,y,batch_size=10,epochs=25,gamma=0.5,eta=0.04,shuffle=True):

    losses = []

    X = np.array([x.reshape(1, -1) for x in X])

    n_batches = len(X)//batch_size

    v_W = epochs * [{k:np.ones(n_batches) for k in model.params_W.keys()}]
    v_b = epochs * [{k:np.ones(n_batches) for k in model.params_b.keys()}]
```

```
if epoch == 0:

        for k in model.params_W.keys():
            v_W[0][k][j] =  1

        for k in model.params_b.keys():
            v_b[0][k][j] = 1

else:


    for k in model.params_W.keys():
        v_W[epoch][k][j] = gamma * v_W[epoch-1][k][j] + (1-gamma) * (np.linalg.norm(grad_W_batch_avr['gr

    for k in model.params_b.keys():
        v_b[epoch][k][j] = gamma * v_b[epoch-1][k][j] + (1-gamma) * (np.linalg.norm(grad_b_batch_avr['gr



for i in range(1, model.N):
        model.params_W['W{}'.format(i)] = model.params_W['W{}'.format(i)] \
        - (eta/np.sqrt(v_W[epoch]['W{}'.format(i)][j]))* grad_W_batch_avr['grad_W{}'.format(i)]

        model.params_b['b{}'.format(i)] = model.params_b['b{}'.format(i)] \
        - (eta /np.sqrt(v_b[epoch]['b{}'.format(i)][j]))  * grad_b_batch_avr['grad_b{}'.format(i)]
```

Now we will use Pytorch to do the same computations we did with our own ANN class.

We start with some imports

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4  from torch.optim import SGD, Adagrad, RMSprop, Adam
```

If we want to use GPU computing (which Pytorch makes very simple) we want our data to be in a format that the GPU prefers. GPUs do calculations with 32 bit floats rather than the CPU which uses 64 bit floats. This only a matter of precision in the very last decimals.

We convert our data to these formats

```
1  X = X.astype('float32')
2  y = y.astype('int64')
```

executed in 5ms, finished 20:27:29 2019-05-25

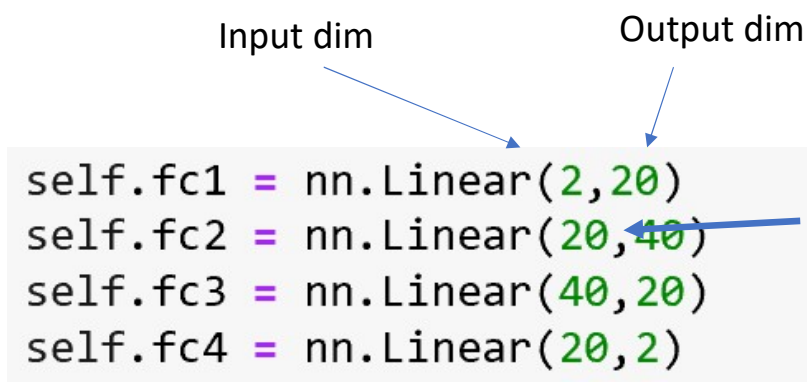
Our neural network is again a class, now it derives from the Pytorch class nn.Module

```
class Net(nn.Module):

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

In the constructor we first initialize the base class

```
def __init__(self):
    super(Net,self).__init__()
```

Here each layer has an input and an output dimension



fc means fully connected, but one can call it anything

Input dimension of the next layer matches the output dimension of the previous layer

These are all Linear layers with a matrix of parameters and bias vector

The class has two methods: the forward method is mandatory

Input data point

Each layer is actually a linear function, u is the output

z is the output of applying the activation function, ReLU to u

```
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
```

The `forward` method is automatically invoked from an instance of the class

```
ann = Net()
```

```
ann(x)
```
=
```
ann.forward(x)
```

We can also write a `predict` method

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

Feeding the data into the gradient descent loop is done by a special object `DataLoader`. To instantiate a DataLoader object we need to input another object of class `TensorDataset`. It is important that Pytorch deals exclusively with Tensor objects. Luckily it is easy to turn np.arrays into Tensors

```
X_p = torch.from_numpy(X)
y_p = torch.from_numpy(y).reshape(-1,1)
```

```
1 X_p
```

executed in 26ms, finished 18:50:34 2019-05-26

```
tensor([[ 1.6621e+00, -4.0047e-01],
        [ 9.4274e-01, -4.4460e-01],
        [ 1.0342e+00, -5.3469e-01],
        [ 2.5350e-01, -1.5420e-01],
        [-3.5460e-02,  9.2338e-01],
        [-2.2555e-01,  8.6350e-01],
        [ 3.1332e-02, -6.3017e-02],
        [ 2.5189e-01, -3.3939e-01],
        [ 2.0539e+00,  3.2519e-01],
        [ 2.9052e-01,  1.0639e+00],
```

```
1 X_p.shape
```

executed in 5ms, finished 18:51:43 2019-05-26

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

The most common error in using Pytorch is to not have tensors with the correct dimensions

```
1  X_p[0:10].size()
```

executed in 5ms, finished 08:59:12 2019-05-27

```
torch.Size([10, 2])
```

```
1  ann(X_p[0:10]).size()
```

executed in 7ms, finished 08:59:52 2019-05-27

```
torch.Size([10, 1])
```

We use the `nn.BCELoss` loss function.

The loss function computes

$$-y \log(\sigma(u)) - (1 - y) \log(1 - \sigma(u))$$

In order for this to make sense the tensors y and $\log(\sigma(u))$ must have the same size.

```
1  X_p[0:10].size()
```

executed in 5ms, finished 08:59:12 2019-05-27

```
torch.Size([10, 2])
```

```
1  torch.log(torch.sigmoid(ann(X_p[0:10]))).size()
```

executed in 10ms, finished 09:11:11 2019-05-27

```
torch.Size([10, 1])
```

If we just take

```
y_p = torch.from_numpy(y)
```

we get

```
1  y_p[0:10].size()
```

executed in 4ms, finished 09:14:07 2019-05-27

```
torch.Size([10])
```

and so does not have the shape we want (each label is a 0-dimensional tensor and we want a 1-dimensional tensor).

We fix this by reshaping

```
y_p = torch.from_numpy(y).reshape(-1,1)
```

The `reshape(-1,1)` command means to keep the first dimension (=length of the data set) and make the second dimension = 1

```
5  y_p = torch.from_numpy(y).reshape(-1,1)
```

executed in 5ms, finished 09:20:10 2019-05-27

```
1  y_p[0:10].size()
```

executed in 5ms, finished 09:20:12 2019-05-27

```
torch.Size([10, 1])
```

We can now instantiate our TensorDataset and DataLoader objects

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

dataset = TensorDataset(X_p,y_p)
training = DataLoader(dataset = dataset,batch_size=10,
                                          shuffle=True)
```

We also have to select a loss function, we choose BCEWithLogitsLoss = Binary Cross Entropy With Logits is our loss function from before

Next we have to select an optimizer.

We first try a simple SGD

```
from torch.optim import SGD
```

We have to write our training loop:

```
losses = []

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

For each epoch we loop through the DataLoader object `training`, which emits a mini-batch of data points and corresponding labels

```
for points,labels in training:
```

We send the data through the forward method to produce the logit u (the input to the sigmoid function)

```
output = ann(points)
```

and we can then compute the loss

```
loss = loss_fn(output,labels)
```

Now the optimizer comes in to minimize the loss:

first we zero all the gradients (if we had run a batch previously all the gradients of the loss function would have been computed and stored in the optimizer object)

```
optimizer1.zero_grad()
```

Then we compute all the gradients of the loss function with the data in the mini-batch. This is done with a single command

```
loss.backward()
```

which computes all the matrix products we had to compute in our hand coded model.

Finally the optimizer computes the step i.e. subtracting the learning rate multiple of the gradient from the parameters

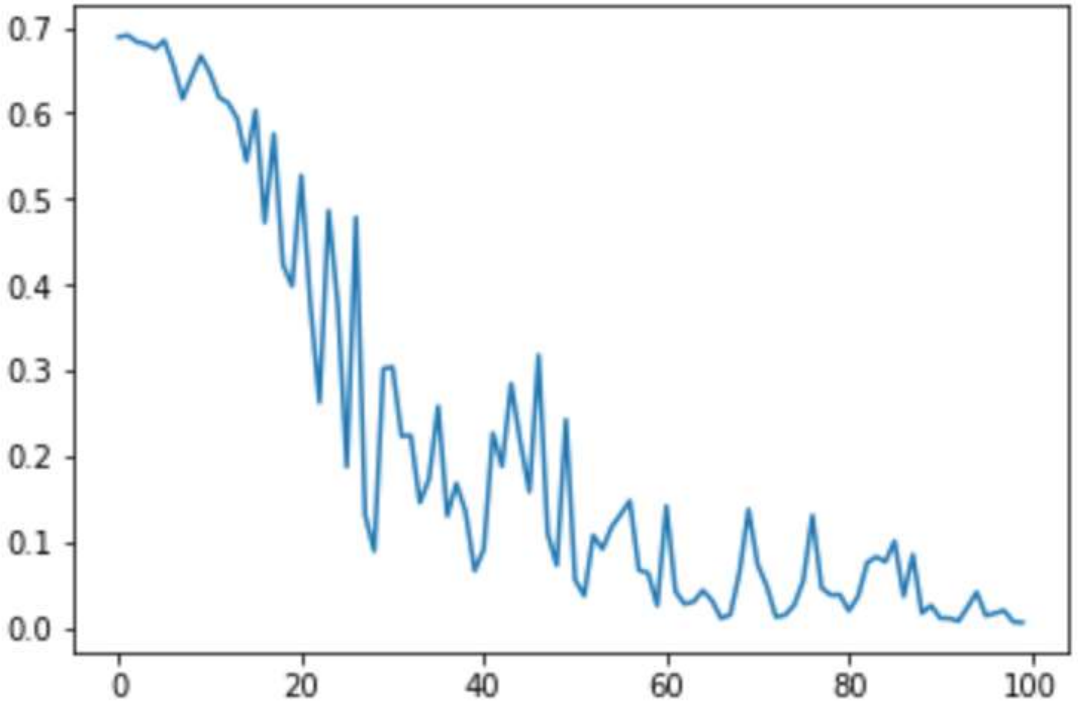
```
optimizer1.step()
```

Here is the complete training loop

```
losses = []

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

        output = ann(points)

        loss = loss_fn(output,labels)

        optimizer1.zero_grad()

        loss.backward()

        optimizer1.step()

    losses.append(loss)
```

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

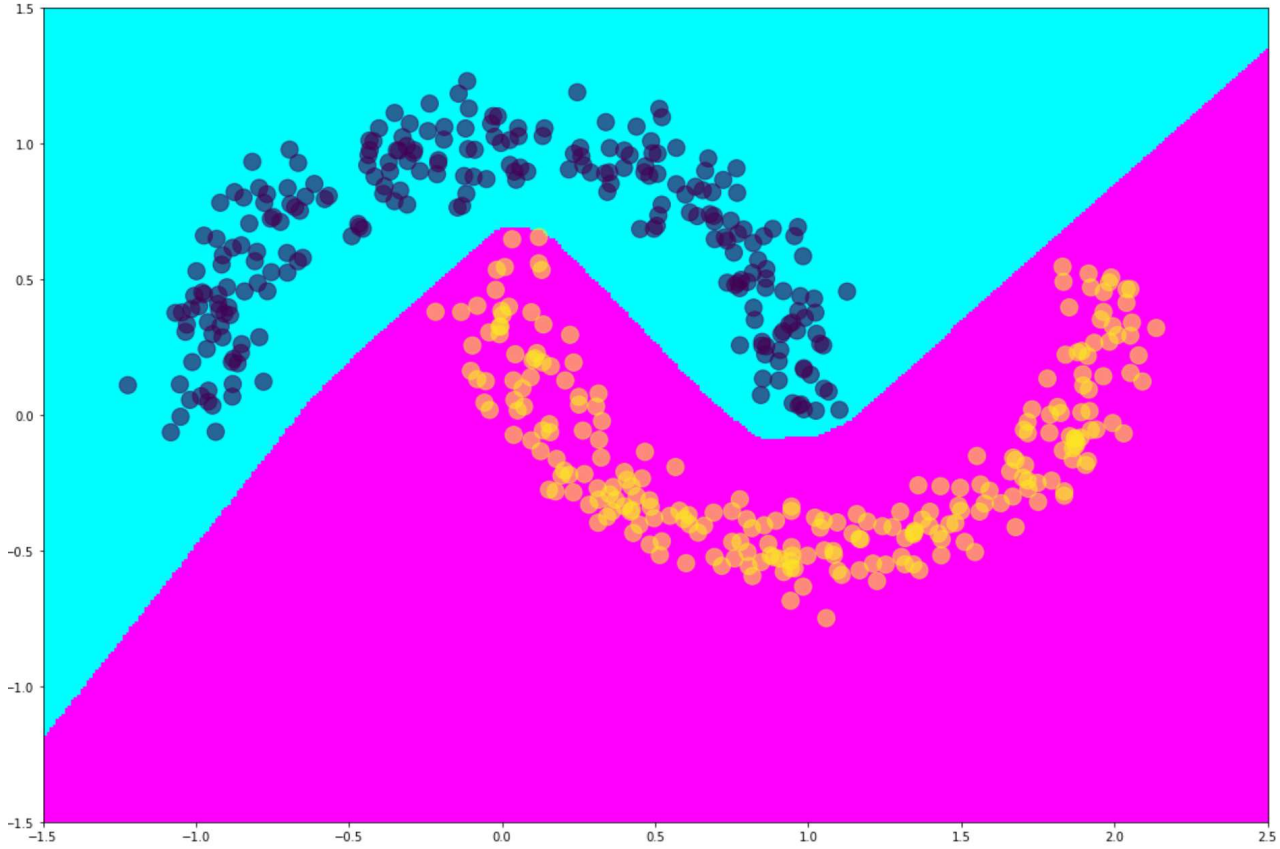
executed in 118ms, finished 09:39:52 2019-05-27



```
1 losses[-1]
```

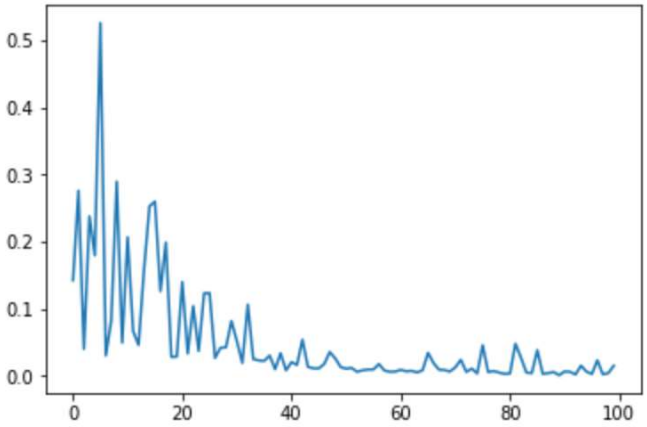
executed in 22ms, finished 09:40:52 2019-05-27

```
tensor(0.0064, grad_fn=<BinaryCrossEntropyWithLogitsBackward>)
```

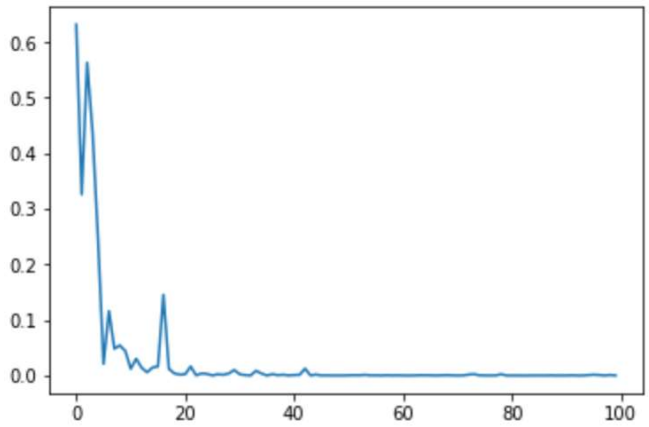


We can try other optimizers

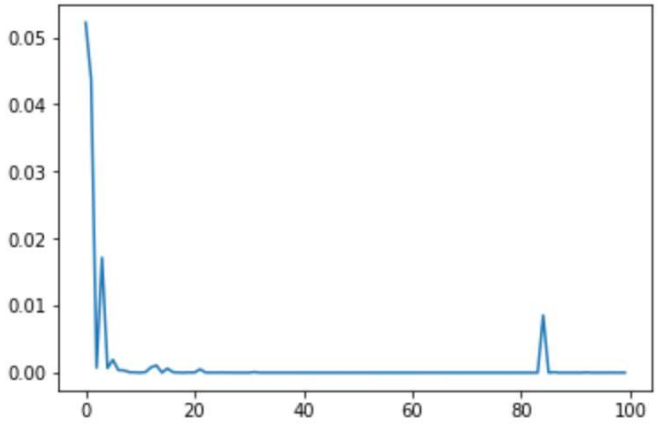
Adagrad



SGD with momentum



RMSProp



Adam