Machine Learning - Assignment 1

Francesco Saverio Zuppichini

October 17, 2017

1 The Perceptron

1.1 Question 1

1.1.1 Vectorized equations

Equation 1 shows the vectorised equation for a single perceptron

$$output = XW + b \tag{1}$$

Where $X = x_1, ... x_n$ is the input set and $W = w_1, ..., w_n$ is the target set. We denote y the output of the perceptron.

1.1.2 Mean Squared Error

Equation 2 denotes the Mean Square Error function for our single perceptron

$$E(w) = \frac{1}{N} \sum_{i=1}^{N} (\underbrace{y(x_i, w_i)}_{\text{predicted}} - \underbrace{t_i}_{\text{actual}})^2) \tag{2}$$

One trick that is usual done is to multiply equation (2) by $\frac{1}{2}$, so when we take the derivative the 2 goes away. This is called One Half Mean Squared Error.

1.1.3 Derivate of the error with respect to the weights

In order to reduce our error function we need to compute the first derivative with respect to the weights. We will use the modified version of equation (2) called One Half Mean Squared Error.

$$\frac{\delta E}{\delta w_i} = y - t \tag{3}$$

1.1.4 Compute the new weight values after one step using a learning rate of 0.02

Since it was not specified that the calculation must be done by hand, we called `train_one_step` in order to compute the new weights, as follows:

```
X,T = get_part1_data()
p = Perceptron()
train_one_step(p,0.02,X,T)
print(p.var['W']) //output weights the values
print(p.var['b']) //output the new bias value
```

Where the new weights and bias are

$$W_{k+1} = \begin{pmatrix} 0.43963636 \\ -0.75109091 \end{pmatrix}, b_{k+1} = \begin{pmatrix} 1.95218181818 \end{pmatrix}$$

1.1.5 Gradient Descend

The gradient descend is an iterative optimisation algorithm that follows the direction of the negative gradient in order to minimised an objective function. It can be effectively used as Learning Algorithm because it reduces the error function, Equation 2, and adjusts the weights properly. Equation 4 shows the generic update rule.

$$w_{k+1} = w_k - \eta \nabla E(w_k) \tag{4}$$

Where η is the step size, also called **learning rate** in Machine Learning. This parameter influences the behaviour of gradient descent, a small number can lead to local minimum, while a bigger learning rate could "over-shoot" and decreasing the converge ration. Later in this project you will how a wrong η can strongly change the output of a Neural Network.

For this reasons, numerous improvements have been proposed to avoid local minima and increase its convergence ration. Some of them are: Conjugate Gradient and Momentum.

1.2 Implement the MSE and dMSE

You can find them in *MSE.py*

1.3 Implement the function forward and backward

The *forward* and *backward* function can be found in *Perceptron.py*. You can find *train_one_step* in *skeletron.py*.

1.4 Implement the *run_part1*

You can find the implementation in the code, *skeletron.py*. Figure 1 shows the final result after 15 steps using a learning rate of 0.02. We can notice that the Perceptron worked as aspected since it correctly classified the two colour sets.
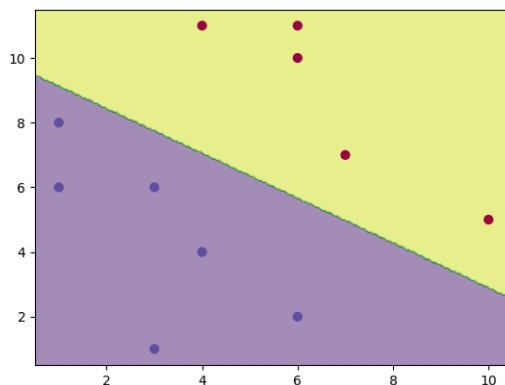

Figure 1: *run_part1* plot

2 A Neural Network

2.1 Implement the activation functions

You can find them inside *activation.py*. I have also implemented the **Relu** activation function for didactic purpose.

2.2 Question 2

2.2.1 Forward pass

In order to calculate the forward pass of a Neural Network we need to compute the activation of each layer, l, ad use it as input of the next one until we reach the output layer.

Equation 5 shows the activation a of layer l for the j-th neuron on that layer.

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \tag{5}$$

Where w_{jk}^l is the connection from neuron k in the $l-1$ layer to j, a^{l-1} is the activation of the previous layer and b_j^l is the bias of j-th neuron in the l layer. With this in mind, we can rewrite 5 in a efficient vectorised form

$$a^l = \sigma(W^l a^{l-1} + b^l) \tag{6}$$

2.2.2 Delta rules

In a Neural Network the weights are iteratively changed in order to decrease the error function, called E. Equation **??** defines δ_j^l as the output error of neuron j in layer l

$$\delta_j^l = \frac{\delta E}{\delta z_j^l} \tag{7}$$

Strictly speaking, δ_j^l, is how much the error function changes by changing the weighted input on that layer. Applying the chain rule, Equation **??** becomes:

$$\delta_j^l = \frac{\delta E}{\delta a_j^l} \frac{\delta a_j^l}{\delta z_j^l} \tag{8}$$

By knowing that $a_j^l = \sigma(z_j^l)$, Equation 8 can be expressed as

$$\delta_j^l = \frac{\delta E}{\delta a_j^l} \sigma'(z_j^l) \tag{9}$$

2.2.3 Derivatives of the weights

We want to compute $\frac{\delta E}{\delta w_{jk}^l}$. Applying the delta rule:

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta z_j^l} \frac{\delta z_j^l}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_j^l} \frac{\delta a_j^l}{\delta z_j^l} \frac{\delta z_j^l}{\delta w_{jk}^l} \tag{10}$$

$$\frac{\delta E}{\delta w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{11}$$

2.3 Implement the functions forward and backward of the Neural Network class.

You can find them in *NeuralNetwork.py*

2.4 Train Network

2.4.1 Split the data into a train set and a test set

We decided to split the training set and test set with a ratio of 80:20. If we choose a smaller train set we may under-train the network and, on the other hand, if we shrink the test set we may *overfit*. In order to do so, we created a function called `get_train_and_test_data` in *utils.py*.

Figure **??** shows the plot for the train set and the test set. They are created by calling `twospirals` using competition's configuration.
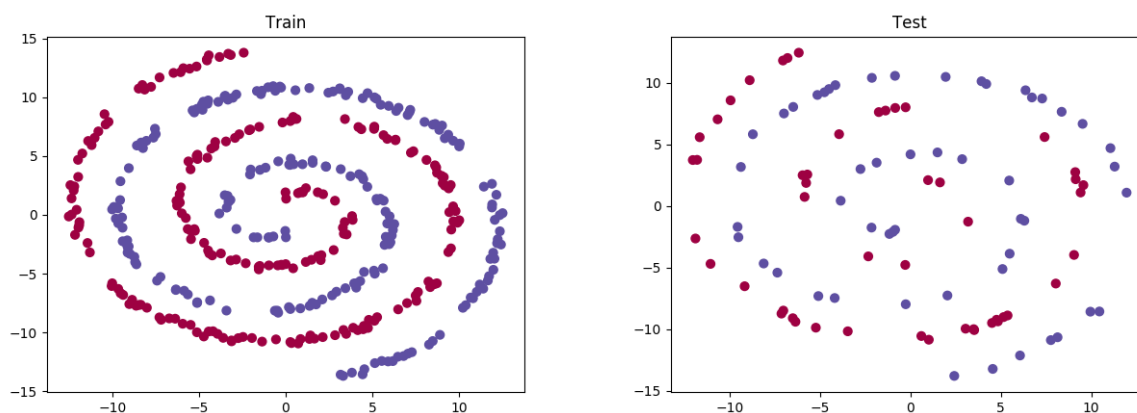

Figure 2: Train set and Test set

2.4.2 Initialise the weights randomly and find a good learning rate to train

In order to find the correct learning rate, we decided to fist test the network with different sizes in order have some empirical data. Figure **??** shows the convergence rate with five learning rates. The data was generated using the standard inputs and target from `twospirals` and then normalised the gradient with its average each 100 iterations in order to provide a smoothed line.
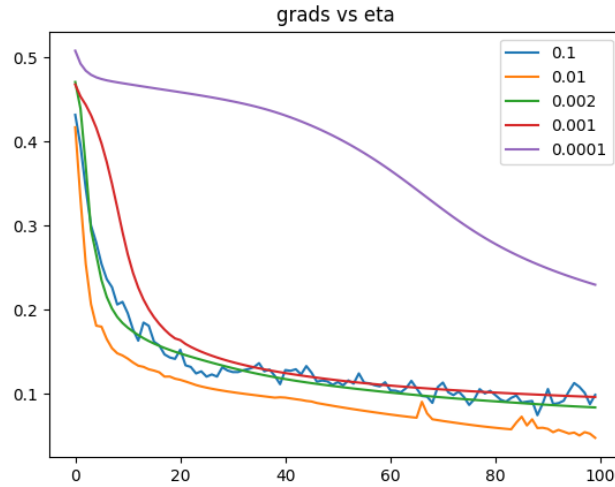
Figure 3: Learning rate convergences

This little benchmark shows that a big step size, like 0.1, do not lead to a very stable converge because it overshoots and it cannot find the correct minimum. On the other hand, a very small eta, 0.0001, converges very slowly and training a model with it will take forever. The best choice is a value between then, 0.01, 0.002 or 0.001.

2.4.3 Plot the learning rate of at least 5 different learning rates on both sets

Since it was not clear what to plot, we decided to plot five boundaries for the five learning rates used in section 2.4.2. We used 20k iterations that are more than enough to make each model converge to a good solution. Figure **??** shows the results Figure **??** and Figure **??** shows differents eta vs gradients and errors respectively

2.4.4 Plot the boundary of the model which converged the most on the training data

Figure **??** shows the model boundary after reach a MSE error less than 0.02 using a step size of 0.001. The network used the following seed: 1. You can get the same plot by calling `run_part2`.
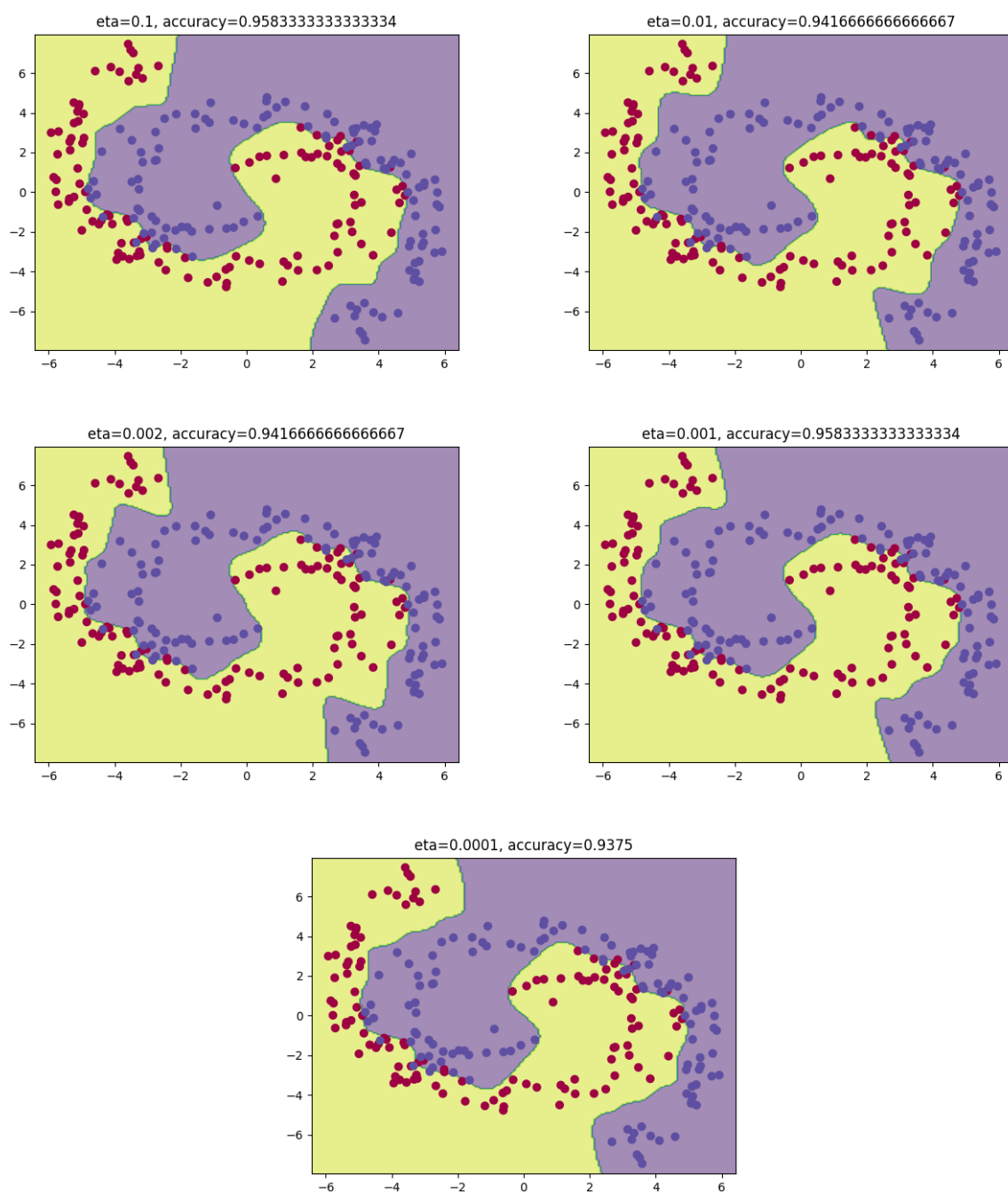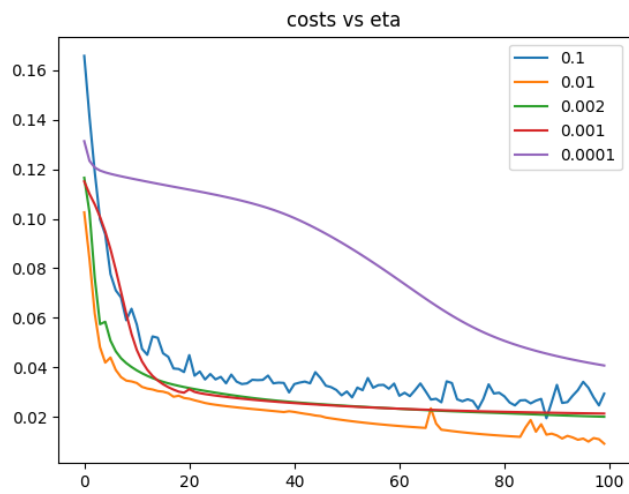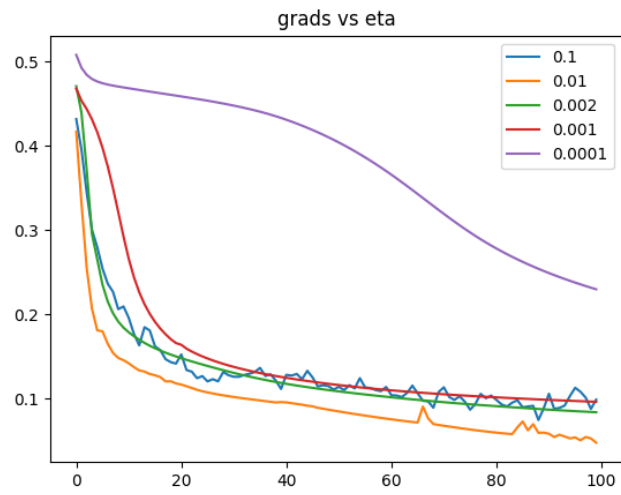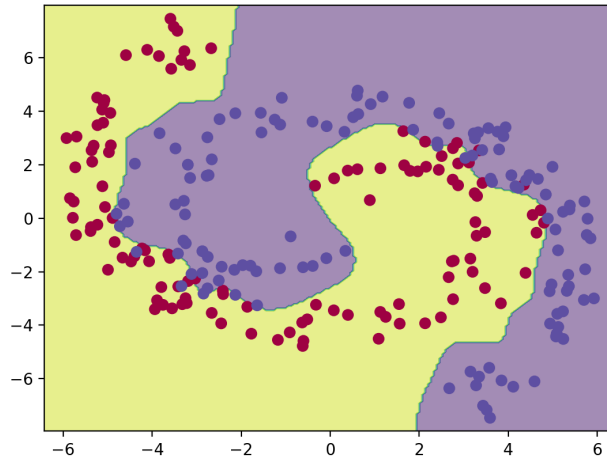
Figure 4: model boundary vs learning rates

grads vs eta

costs vs eta

Figure 5: Boundary of the model with MSE less than 0.02

2.5 Explain if this is a good model for the data

The model seems good enough for the data generated by calling `twospirals` with the standard configuration. however, by using the data for part 3 the model loose lots of accuracy due to its gradient descent implementation that does not prevent the algorithm to fall into local minima. A better model is present in detail in the following Section.

3 Further Improvements

In this sections we analyse each extension that we tried, even if they did not improve our solution, in order to create a better model.

3.1 Generic Implementation

Since the NeuralNetwork class has a fixed number of layers, the first improvement that we did was to create a generic implementation.

The `BetterNeuralNetwork` class exposes an API allowing the creation of a generic Neural Network by adding and shaping layers.

After instance the class, is it possible to add a input layer by calling `add_hnput_layer(size_in,size_out)`. Many hidden layers can be added by using `add_hidden_layer(size_out)`, the "size_in" is not necessary since it can be easily calculated by the network. Then method `add_output_layer(size_out)` adds the output lauer. It follows a full example where we create the same Neural Network from Exercise 2.

```
bnn = BNN()
bnn.addInputLayer(2, 20, np.tanh, act.dtanh)
bnn.addHiddenLayer(15, np.tanh, act.dtanh)
bnn.addOutputLayer(1)
```

For each layer is also possible to specified the activation function, as default value *signmoid* is used. We have also implemented `relu` and `dRelu` to try them.

8

`BetterNeuralNetwork` exposes `train` method that performs forward, backpropagation and weight updates directly. It accepts as input the training sets, the targets, max iterations, a dictionary containing the parameters and the gradient descent update method name, the supported ones are: `'gradient_descent'` (default), `'momentum'` and `'adagrad'`

3.2 Initial weight and bias

Weight initialisation is crucial to create a good model. I used the following formula.

$$w = random(n)/\sqrt{n}$$

Where n is the size of the input. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. [1]

Using *numpy*:

```
np.random.randn(in_size,out_size)/np.sqrt(in_size)
```

The bias are initialised with a very small number close to zero, for such reason I scaled down the each random value by a factor of 100.

3.3 Performance

Due to a better and generic implementation, our `BetterNeuralNetwork` is slightly faster than the one from Section 2. Table **??** shows the result of a little benchmark

iterations	time NN	time BNN
10	17.5ms	13.7ms
100	134.5ms	128.2ms
1000	461.6ms	458.84ms
10000	4455.6ms	4157.9ms

Table 1

I have also tried to create a small thread pool, so we can save time in thread allocation, in order to parallelise the weight updates, since they are independent operation but without any real advantages. Probably the network is so small that it take more time to call a free thread and submit a work than to compute the operations sequentially.

3.4 Stochastic Gradient Descent

TODO does it work?

3.5 Momentum

We used Momentum [2] to speed up the convergence of the network, it prevent getting stuck in local minima and pushes the objective more quickly along the shallow ravine. Basically it give a global direction to the gradient descent based on the previous update. Equation 12 shows the new update formula.

---

[1]http://cs231n.github.io/neural-networks-2/

$$\Delta w_{t+1} = (\eta \nabla E(w_{t+1}) + \beta \Delta w_t)$$
$$w_{k+1} = w_{k+1} - \Delta w_{t+1}$$

$$(12)$$

$\beta \in [0, 1]$ is a the momentum factor, usually between 0.5 and 0.9. In our benchmarks we used a classic 0.5. In Figure **??**, we can see the converge rate of a classic gradient descent versus momentum for different step sizes. Momentum converges faster in all cases. In my test cases, momentum works better with small step size such as 0.001.
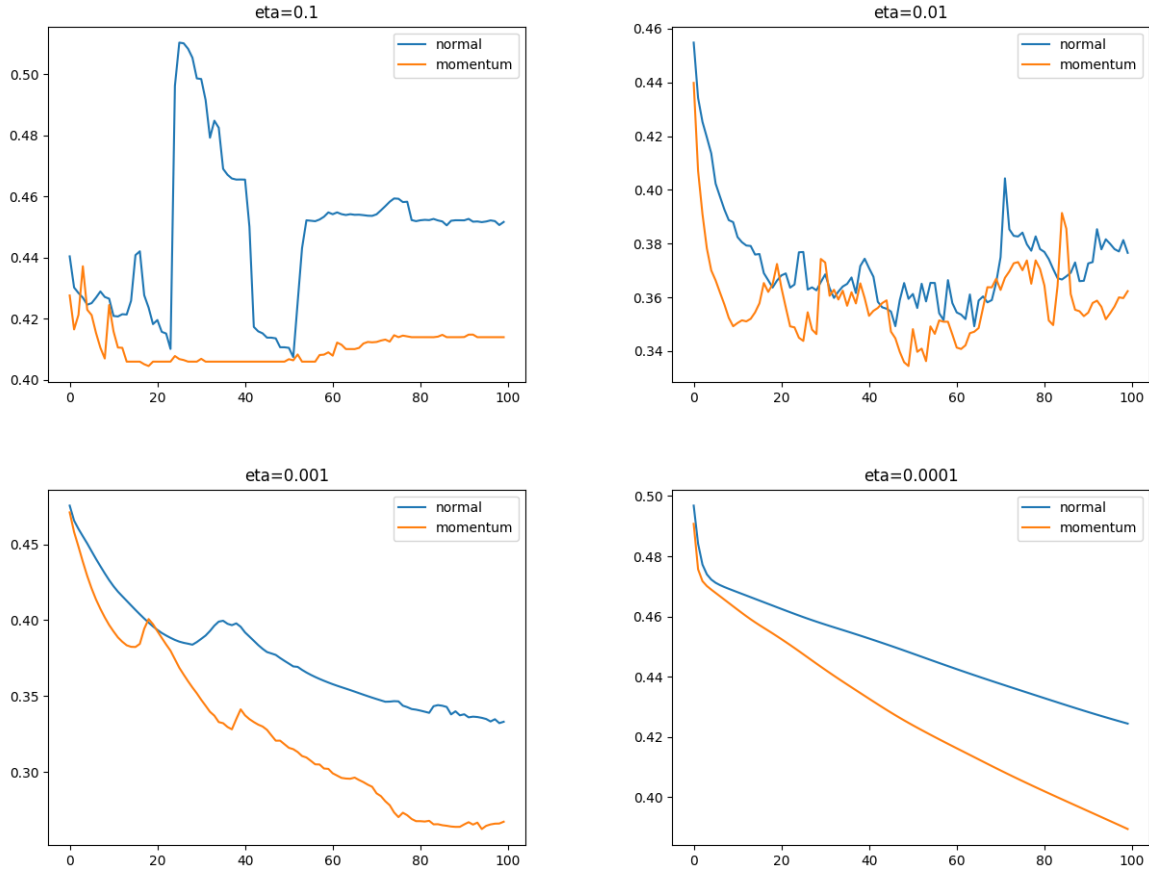

Figure 6: Gradient Discent vs Momentum

In the `BetterNeuralNetwork` class, momentum can be enabled by passing the correct parameters and type to the `train` method. The following snippet shows and example

```
params = {'eta': 0.01,'beta': 0.5} # beta must be included or the default 0.5
                                     value will be used
nn.train(X_train, T_train, 3000, params, 'momentum')   # pass 'momentum' as type
                                     name
```

3.6 Adagrad

Adagrad [1] is an adaptive learning rate algorithm that we implement in order to speed up the convergence. In Figure **??**, we can see the converge rate of a classic gradient descent versus adagrad
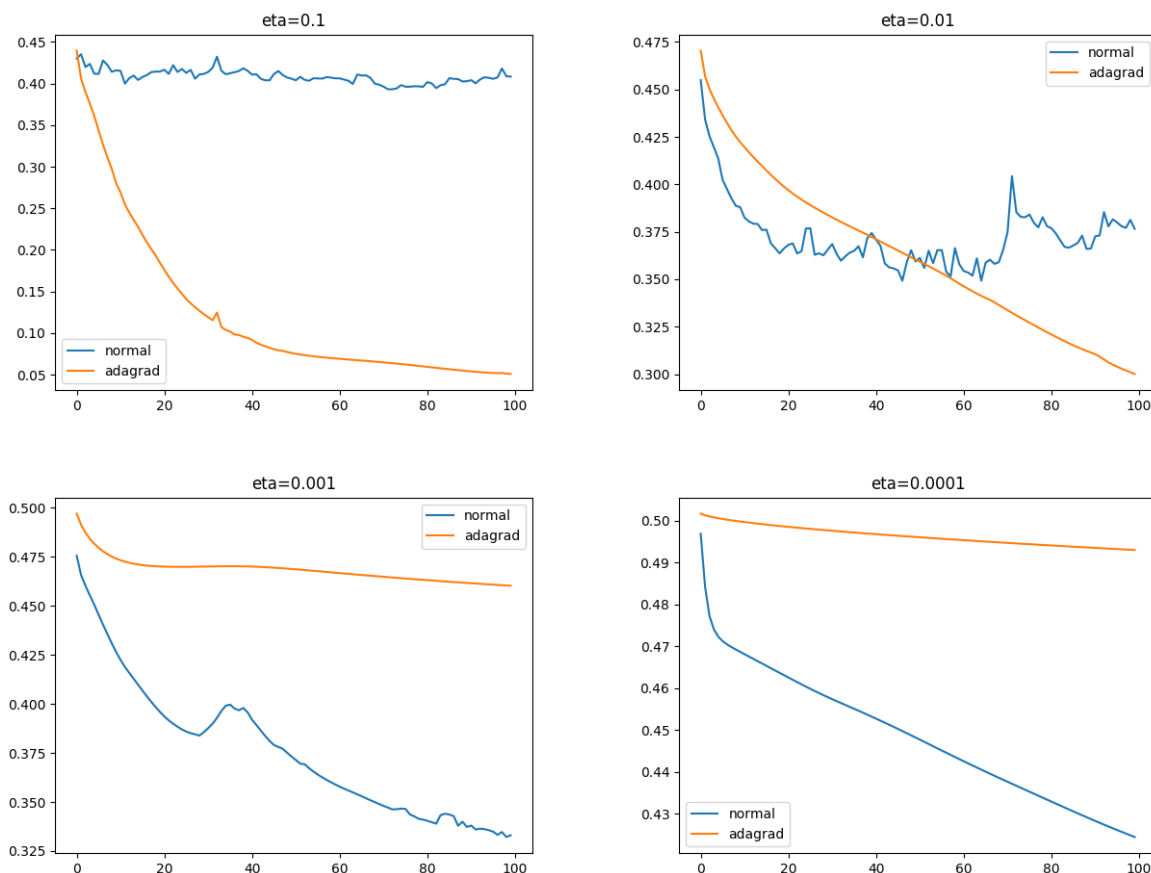
for different step sizes.

Figure 7: Gradient Discent vs Adagrad

Adagrad seems to work very well with big step size, but too slow with small step. With little step sizes, Adagrad continue to accumulate the squared gradients in the denominator during training leading to a huge reduction of the step sizes that becomes close to zero. Therefore it cannot converge anymore and we get that really horizontal line in the last two plots of Figure ??

The following snippet shows how to enable it

```
nn.train(X_train, T_train, 3000, params, 'adagrad') // pass 'adagrad' as type name
```

3.7   Network deep

We used an empirical approach to find a correct number of hidden layer. We fist tried to add thinner layers with a size between four and eight with no success. After some other benchmarks, our the best configuration is a network with one input layer of size $(2, 20)$, two hidden layer of size $(20, 15)$ and$(15, 15)$ and one output layer of size $(15, 1)$. Basically we just added one hidden layer to the model from Section 2. We trying other techniques with no luck. We also include them in this report.

11

3.8 Stochastic Gradient Descent

We implemented Stochastic Gradient Descent, it can be enabled by passing the `batch_size` value inside the `params` dictionary

```
model.train(train_X, train_T, 4000, { 'eta' : 0.1, 'beta' : 0.5, 'batch_size': 10
                                    }, 'adagrad',testX, testT)
```

3.9 Dropout

Dropout is a recently introduced regularisation technique. Basically, each node has a probability to be enable on disable in each forward pass. We implemented it by changing the `forward` method inside `BetterNeuralNetwork`

```python
def forward(self, inputs):
    self.A = [inputs]
    self.Z = []
    # dropout probability
    p = 0.5

    a = inputs

    for l in self.layers:
        z = a.dot(l.W) + l.b
        # create dropout mask
        u = (np.random.rand(*z.shape) < p) / p
        # apply mask
        z *= u
        a = l.activation(z)
        # store
        self.A.append(a)
        self.Z.append(z)

    return a
```

At each pass, a binary mask `u` is created and applied to the layer weighted output `z`. Unfortunately, this approach did not work well for us and lead to a bad convergence, probably because the network is too small.

3.10 Results

Figure **??** shows the boundary of our new model with one of the best seed: 1508255316.
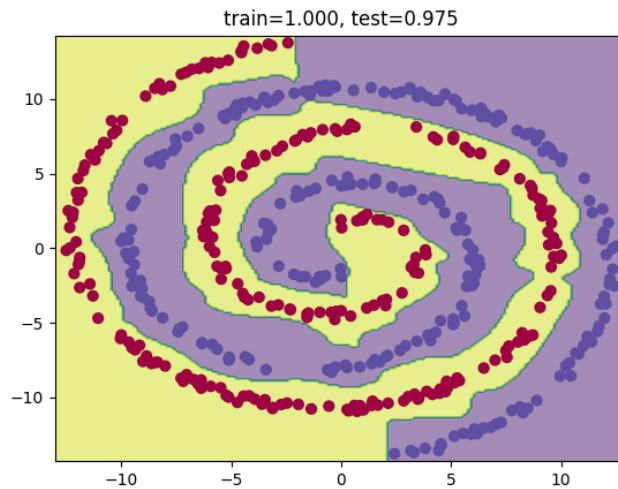
Figure 8: Boundary of the model

The following snippet show our to re-create the model with our API.

```python
# use same seed
np.random.seed(1508186659)
  model = BetterNeuralNetwork()
  # add layers
  model.add_input_layer(2, 20, act.tanh, act.dtanh)
  model.add_hidden_layer(15, act.tanh, act.dtanh)
  model.add_hidden_layer(15, act.tanh, act.dtanh)
  model.add_output_layer(1)

  model.train(train_X, train_T ,8000, { 'eta' : 0.1, 'beta' : 0.5 },'adagrad')
```

Figure **??** shows the errors, gradients, error and accuracy for this model with respect to the one from section 2. Obviously the new model is better in every aspect.
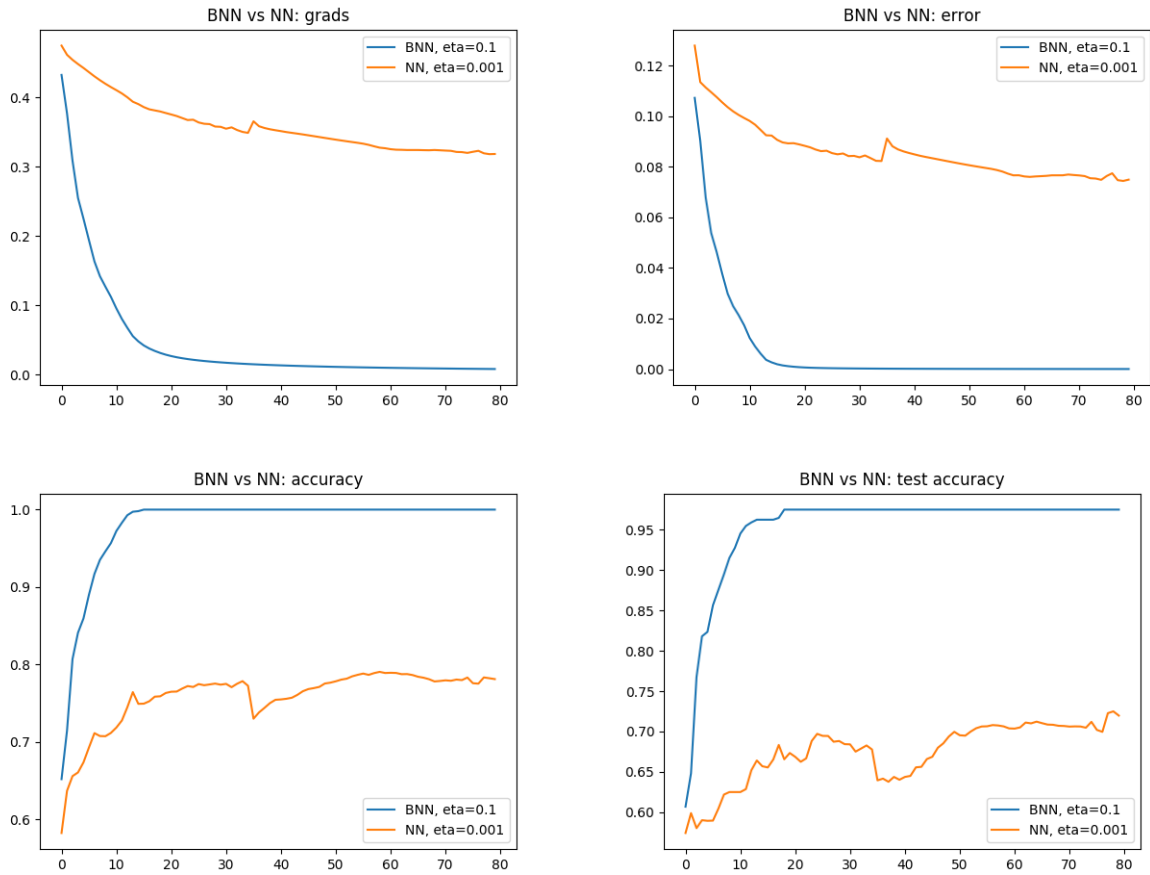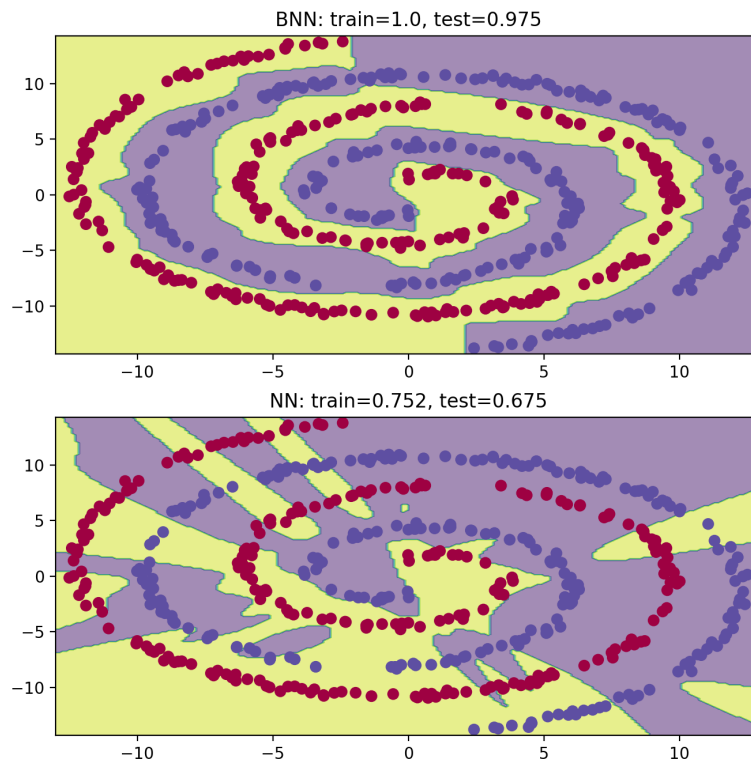
Figure 9: BetterNeuralNetwork vs Neural Netwotk

BNN: train=1.0, test=0.975

NN: train=0.752, test=0.675

3.11   Save and Load

You can save and load the network state by calling .save(file name) .load(file name). The following snippet shows hot to load the previous network.

```python
model = BetterNeuralNetwork()
# create layers
model.add_input_layer(2, 20, act.tanh, act.dtanh)
model.add_hidden_layer(15, act.tanh, act.dtanh)
model.add_hidden_layer(15, act.tanh, act.dtanh)
model.add_output_layer(1)
# load prev weights
model.load('test')
```

References

[1] Elad Hazan John Duchi and Yoram Singer. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.* 2011.

[2] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* 2017.