CmpE597 - Homework 1

March 5, 2018

Mine Melodi Çalkan - 2015705009

Multilayer Perceptron Network Implementation

The cross entropy error

$E = -\sum_{i=1}^{n}(t_i log(y_i) + (1-t_i)log(1-y_i)) \quad (1)$

where t is the target vector, y is the output vector.

The output prediction is always between zero and one. Training corresponds to minimizing the negative log-likelihood of the data.

Outputs are computed by applying the sigmoid function to the weighted sums of the hidden layer activations.

$y_i = \frac{1}{1+e^{-z_i}}$ where $z_i = \sum_{j=1} h_j w_{ji}$

The derivative of the error with respect to each weight connecting the hidden units to the output units using the chain rule:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ji}}$$

- $\frac{\partial E}{\partial y_i} = \frac{-t_i}{y_i} + \frac{1-t_i}{1-y_i} = \frac{y_i - t_i}{y_i(1-y_i)}$
- $\frac{\partial y_i}{\partial z_i} = y_i(1-y_i)$
- $\frac{\partial z_i}{\partial w_{ji}} = h_j$

The gradients of the error with respect to the output weights:

$$\frac{\partial E}{\partial w_{ji}} = (y_i - t_i)h_j$$

And $\frac{\partial E}{\partial z_i} = \frac{y_i - t_i}{y_i(1-y_i)} y_i(1-y_i) = (y_i - t_i)$

Backpropagation algorithm for gradients with respect to the hidden layer weights

$\frac{\partial E}{\partial w_{kj}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{kj}^l}$

z_j^1 is the weighted input sum at hidden unit j, and $h_j = \frac{1}{1+e^{-z_j^1}}$ is the activation at unit j.

- $\frac{\partial E}{\partial z_j^l} = \sum_{i=1} \frac{\partial E}{\partial z_i}\frac{\partial z_i}{\partial h_j}\frac{\partial h_j}{\partial z_j^l} = \sum_{i=1}(y_i - t_i)(w_{ji})(h_j(1-h_j))$

$\frac{\partial E}{\partial h_j} = \sum_{i=1} \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial x_j} = \sum_{i=1} \frac{\partial E}{\partial y_i} y_i(1-y_i)w_{ji}$
Then $ w^1_{kj}$
connecting input unit k to hidden unit j has gradient
$\frac{\partial E}{\partial w^l_{kj}} = \frac{\partial E}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{kj}} = \sum_{i=1}(y_i - t_i)w_{ji}(h_j(1-h_j))x_k$
** RmsProp **

RMSProp is a method that computes individual adaptive learning rates for different parameters based on the average of the recent magnitudes of the gradients for the weights.
$g_t^2 = 0.9g_{t-1}^2 + 0.1g_t^2$
$\theta_{t+1} = \theta_t - \frac{\rho}{\sqrt{g^2+\epsilon}}g_t$

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
def initialize_network(number_hidden_units):
    w_hidden = 2*np.random.random((2,number_hidden_units)) - 1
    w_output = 2*np.random.random((number_hidden_units,1)) - 1
    r_hidden = np.zeros((2,number_hidden_units))
    r_output = np.zeros((number_hidden_units,1))
    return w_hidden,w_output,r_hidden,r_output
```

In [3]:
```
def f(x):
    '''stable sigmoid'''
    max_x = max(0, np.max(x))
    rebased_x = x - max_x
    res = 1 / (1+np.exp(-rebased_x))

    return res
```

In [4]:
```
def cost(y, t):
    return - np.sum(np.multiply(t, np.log(y)) + np.multiply((1-t), np.log(1-y)))
```

In [12]:
```
#stochastic gradient descent with RmsProp
def backprop_sgd_rms(epochs,inputs,targets,number_hidden_units,lr_init):
    X=inputs
    y=targets
    weight_hidden,weight_output,r_hidden,r_output=initialize_network(number_hidden_uni
    alpha=lr_init
    ro=0.9
    errors=[]
    hws=[]
    hunits=[]
    oweights=[]
    for j in range(epochs):
        e=0
        h_list=[]
        for i in range(4):
            hidden_layer=f(np.dot(X[i], weight_hidden))
            output_layer = f(np.dot(hidden_layer, weight_output))
```

```
                output_layer_delta = (output_layer - y[i][0])
                e+=cost(output_layer,y[i][0])
                deriv_hidden =(hidden_layer * (1-hidden_layer))
                hidden_layer_delta = output_layer_delta.dot(weight_output.T) * deriv_hidde
                gradient_output = [[x*output_layer_delta[0]] for x in hidden_layer]
                #adjust dimensions and shapes
                dim_adjusted_x=np.array([[x] for x in X[i]])
                dim_adjusted_hidden=np.array([[x] for x in hidden_layer_delta]).T
                gradient_hidden =  dim_adjusted_x.dot(dim_adjusted_hidden)
                ##rms equations
                r_output = (ro * r_output) + ((1-ro)* (np.array(gradient_output)**2))
                r_hidden = (ro * r_hidden) + ((1-ro)* (gradient_hidden**2))

                weight_output -= (alpha/(np.sqrt(r_output+0.000001))) * np.array(gradient_
                weight_hidden -= (alpha/(np.sqrt(r_hidden+0.000001))) * gradient_hidden
                if j%1000==0:
                    hws.append(weight_hidden)

                    hunits.append(hidden_layer)
                    oweights.append(weight_output)
                h_list.append(hidden_layer)
            errors.append(e)
        return weight_hidden,weight_output,errors
```

In [24]:
```
def train_test(test_data,number_hidden_units,epochs, lr_init=0.5):
    X = np.array([ [0,0],[1,1],[0,1],[1,0]])
    y = np.array([[0,0,1,1]]).T
    ##train
    hiddenWeight,outputWeight,errs=backprop_sgd_rms(epochs,X,y,number_hidden_units,lr_
    ##test
    z1=np.dot(test_data,hiddenWeight)
    h1=1/(1+np.exp(-z1))
    z2=np.dot(h1,outputWeight)
    h2=1/(1+np.exp(-z2))
    ##threshold for output probabilities
    res = [1 if i>=0.5 else 0 for i in h2]
    return res, h2,errs,hiddenWeight,outputWeight
```

In [37]:
```
test_data=np.array([[0,0],[0,1],[1,0],[1,1]])
```

In [38]:
```
returned=train_test(test_data,2,2000)
```

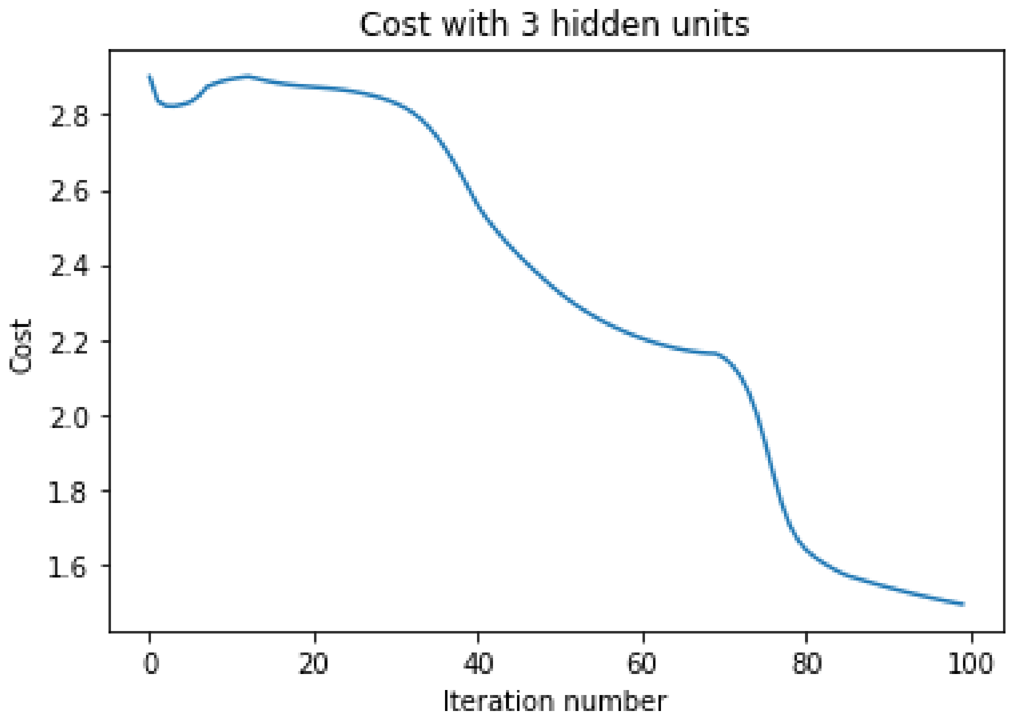
In [39]:
```
import matplotlib.pyplot as plt
plt.plot(returned[2])
plt.ylabel('Cost')
plt.xlabel('Iteration number')
plt.title('Cost with 2 hidden units')
plt.show()
```


```
In [40]: for i in range(4):
             print('expected', y_test[i])
             print('output', returned[0][i])
```

```
expected 0
output 0
expected 1
output 0
expected 1
output 0
expected 0
output 0
```

```
In [30]: returned=train_test(test_data,3,100,0.1)
```

```
In [31]: import matplotlib.pyplot as plt
         plt.plot(returned[2])
         plt.ylabel('Cost')
         plt.xlabel('Iteration number')
         plt.title('Cost with 3 hidden units')
         plt.show()
```




```
In [35]: y_test=[0,1,1,0]
```

```
In [36]: for i in range(4):
             print('expected', y_test[i])
             print('output', returned[0][i])
```

```
expected 0
output 0
expected 1
output 0
expected 1
output 1
expected 0
output 0
```

Momentum

SGD with momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$
$\theta = \theta - v_t$

```
In [48]: import numpy as np
         X = np.array([ [0,0],[0,1],[1,0],[1,1]])
```

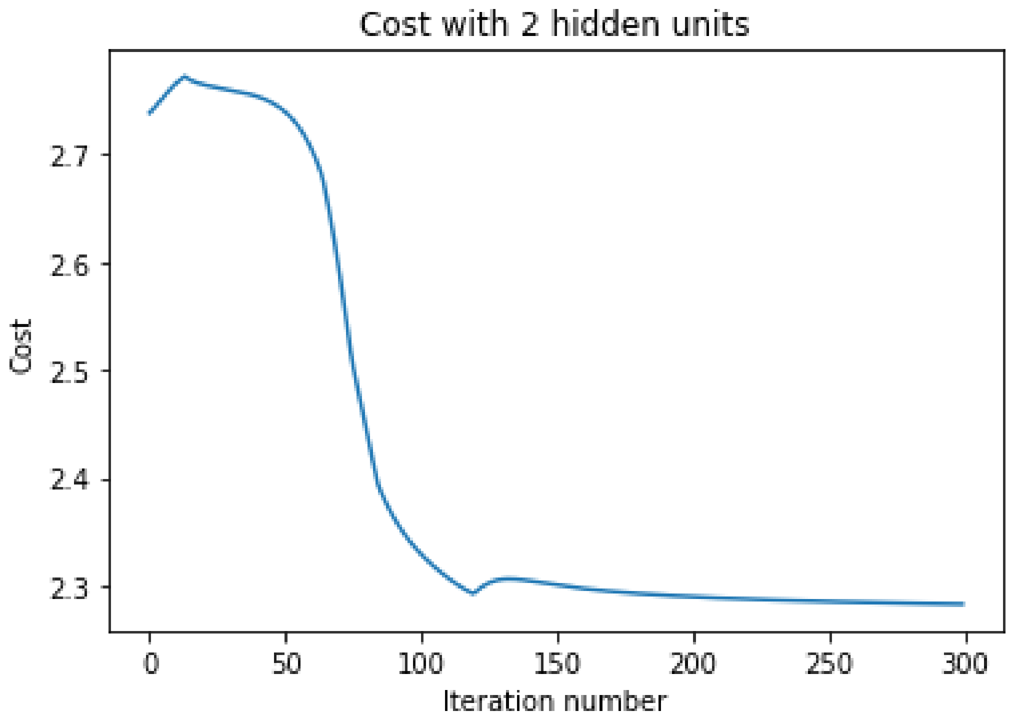
```
y = np.array([[0,1,1,0]]).T
def initialize_network_m(number_hidden_units=2):
    w_hidden = 2*np.random.random((2,number_hidden_units)) - 1
    w_output = 2*np.random.random((number_hidden_units,1)) - 1
    velocity_hidden = np.zeros((2,number_hidden_units))
    velocity_output = np.zeros((number_hidden_units,1))
    return w_hidden,w_output,velocity_hidden,velocity_output


#stochastic gradient descent with momentum
def backprop_sgd_m(epochs,number_hidden_units,lr_init=0.1):
    weight_hidden,weight_output,velocity_hidden,velocity_output=initialize_network_m(
    alpha=lr_init
    errors=[]
    for j in range(epochs):
        e=0
        for i in range(4):
            hidden_layer =f(np.dot(X[i], weight_hidden))
            output_layer = f(np.dot(hidden_layer, weight_output))
            output_layer_delta = (output_layer - y[i][0])
            e+=cost(output_layer,y[i][0])
            deriv_hidden =(hidden_layer * (1-hidden_layer))
            hidden_layer_delta = output_layer_delta.dot(weight_output.T) *deriv_hidde
            gradient_output = [[alpha*x*output_layer_delta[0]] for x in hidden_layer]
            #adjust dimensions and shapes
            dim_adjusted_x=np.array([[x] for x in X[i]])
            dim_adjusted_hidden=np.array([[x] for x in hidden_layer_delta]).T
            gradient_hidden =  alpha*dim_adjusted_x.dot(dim_adjusted_hidden)

            velocity_output = 0.9 * velocity_output + gradient_output
            velocity_hidden = 0.9 * velocity_hidden + gradient_hidden

            weight_output -= velocity_output
            weight_hidden -= velocity_hidden
        errors.append(e)
    return weight_hidden,weight_output,errors
```

In [52]:
```
def test_m(test_data,number_hidden_units,epochs):
    hiddenWeight,outputWeight,residuals=backprop_sgd_m(epochs,number_hidden_units)
    _hiddenLayer=f(np.dot(test_data,hiddenWeight))
    _outputLayer=f(np.dot(_hiddenLayer,outputWeight))
    ##threshold for output probabilities
    results = [1 if i>=0.5 else 0 for i in _outputLayer]
    return results,_outputLayer,residuals,hiddenWeight
```

In [53]:
```
test_data=np.array([[0,0],[0,1],[1,0],[1,1]])
```

In [54]:
```
result, outputlayer, residual,hidden=test_m(test_data,2,300)
```

In [55]:
```
import matplotlib.pyplot as plt
plt.plot(residual)
plt.ylabel('Cost')
plt.xlabel('Iteration number')
plt.title('Cost with 2 hidden units')
plt.show()
```



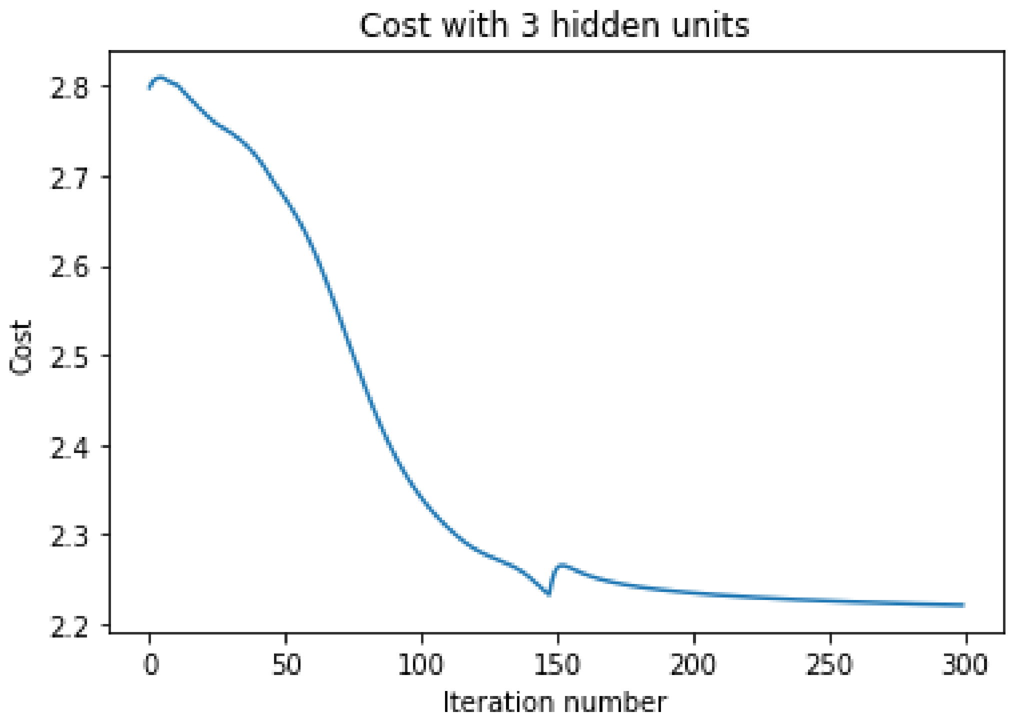
In [58]:
```
for i in range(4):
    print('expected', y_test[i])
    print('output', result[i])
```

```
expected 0
output 0
expected 1
output 0
expected 1
output 1
expected 0
output 0
```

In [59]:
```
result, outputlayer, residual,hidden=test_m(test_data,3,300)
```

In [60]:
```
import matplotlib.pyplot as plt
plt.plot(residual)
```

```
plt.ylabel('Cost')
plt.xlabel('Iteration number')
plt.title('Cost with 3 hidden units')
plt.show()
```




```
In [61]: for i in range(4):
             print('expected', y_test[i])
             print('output', result[i])
```

```
expected 0
output 0
expected 1
output 0
expected 1
output 1
expected 0
output 0
```