# Deep learning Homework 1

**Teodor Janez Podobnik (63210493)**

## 1 Introduction

Homework is divided into two parts: PenPaper and Programming part. The goal of the first part is to simulate forward and backward pass, while at least an update of weight $w_{11}$ in the first layer is calculated. In the programming part a neural network is designed in Python using additional features such as L2 Regularization, Stochastic Gradient Descent and Adam optimizer. At last, result using different parameters are compared and evaluated.

## 2 Backpropagation (Pen-Paper part)

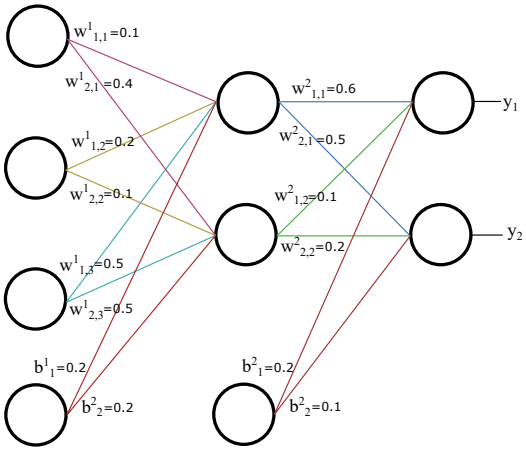



Figure 1: Example neural network

We are given a simple neural network with weights $w_{jk}$ and biases $b_k$ shown in Figure 1. For the input x = [0.5, 0.1, 0.3], the ground truth y = [0.9, 0.1], the loss function L and the learning rate = 0.1. All neurons use the sigmoid activation function. Calculation of forward and backward and the result of $w_{11}$ is summarized in the Figure 2.

## 3 Network implementation

Innitially we design a forward (Figure 3) and backward(Figure 4) pass, where in all layers a sigmoid activation function is used expect in the last layer where a softmax function is used.

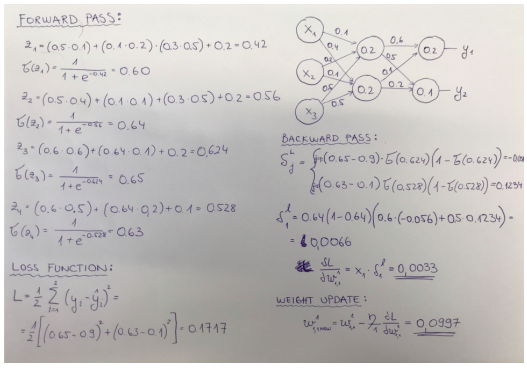



Figure 2: Example neural network

```
def forward_pass(self, input):
    # input = numpy array of dimensions [n0 x m], where n is the number of examples in the mini batch and
    # n0 is the number of input attributes
    Zs = []
    As = [input]
    for w,b in zip(self.weights, self.biases):
        Zs.append(np.dot(w, As[-1])+b)
        As.append(sigmoid(Zs[-1]))
    output = As[-1] = softmax(Zs[-1]) # Update last layer to use softmax activation function

    return output, Zs, As
```

Figure 3: Forward pass in python

```
def backward_pass(self, output, target, Zs, activations):
    '''
    calculate errors for w and b in each layer
    '''
    gb = [np.zeros(b.shape) for b in self.biases]
    gw = [np.zeros(w.shape) for w in self.weights]

    # last layer softmax activation
    delta = softmax_dLdZ(output, target) / len(target)
    gb[-1] = delta
    gw[-1] = np.dot(delta, activations[-2].transpose())

    for l in range(2, self.num_layers):
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sigmoid_prime(Zs[-l]) / len(target) # target je enakega size-a kot mini batch
        gb[-l] = delta
        gw[-l] = np.dot(delta, activations[-l-1].transpose())

    return gw, gb
```

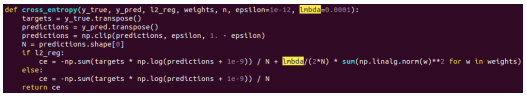
Figure 4: Backward pass in python

## 4 Regularization

In order to prevent training data overfitting and bad accuracy on validation dataset we add L2 regularization method. In practical terms this means adding an additional terms in update weights (Figure 5) function and Cross-Entropy Loss function (Figure 6).

```
self.weights[i] = (1-(eta*lmbda/n))*self.weights[i] - eta * gw[i]
self.biases[i] = self.biases[i] - eta * gb[i]
```

Figure 5: Update weights L2 update

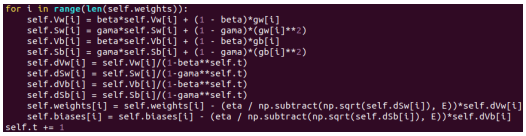


```
def cross_entropy(y_true, y_pred, l2_reg, weights, n, epsilon=1e-12, lmbd=0.0001):
    targets = y_true.transpose()
    predictions = y_pred.transpose()
    predictions = np.clip(predictions, epsilon, 1. - epsilon)
    N = predictions.shape[0]
    if l2_reg:
        ce = -np.sum(targets * np.log(predictions + 1e-9)) / N + lmbd/(2*N) * sum(np.linalg.norm(w)**2 for w in weights)
    else:
        ce = -np.sum(targets * np.log(predictions + 1e-9)) / N
    return ce
```

Figure 6: Cross Entropy function L2 update

## 5 Optimizer

To improve the basic Stochastic Gradient descent we add Adam optimizer which uses Momentum update and adaptive scaling that further improve convergence of weights to their optimal values.

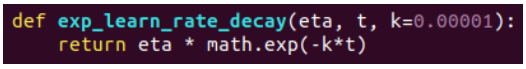


```
for i in range(len(self.weights)):
    self.Vw[i] = beta*self.Vw[i] + (1 - beta)*gw[i]
    self.Sw[i] = gama*self.Sw[i] + (1 - gama)*(gw[i]**2)
    self.Vb[i] = beta*self.Vb[i] + (1 - beta)*gb[i]
    self.Sb[i] = gama*self.Sb[i] + (1 - gama)*(gb[i]**2)
    self.dVw[i] = self.Vw[i]/(1-beta**self.t)
    self.dSw[i] = self.Sw[i]/(1-gama**self.t)
    self.dVb[i] = self.Vb[i]/(1-beta**self.t)
    self.dSb[i] = self.Sb[i]/(1-gama**self.t)
    self.weights[i] = self.weights[i] - (eta / np.subtract(np.sqrt(self.dSw[i]), E))*self.dVw[i]
    self.biases[i] = self.biases[i] - (eta / np.subtract(np.sqrt(self.dSb[i]), E))*self.dVb[i]
self.t += 1
```

Figure 7: Adam Optimizer

## 6 Learning Rate Decay

Last but not least we add Learning rate decay that makes the learning step smaller over time.



```
def exp_learn_rate_decay(eta, t, k=0.00001):
    return eta * math.exp(-k*t)
```

Figure 8: Learning rate decay

## 7 Results

The following section summarizes result of performed tests using different parameters and network architecture.

| Experiment | Schedule | k | L2 reg. | Lambda | Optimizer | LR | Epoch | Architecture | Cls. Acc. |
|---|---|---|---|---|---|---|---|---|---|
| Exp. 1 | None | None | No | None | SGD | 0.01 | 20 | 100-100 | 46% |
| Exp. 2 | None | None | Yes | 0.0001 | SGD | 0.01 | 20 | 200-100 | 49% |
| Exp. 3 | Exponent. | 0.0001 | Yes | 0.0001 | Adam | 0.001 | 20 | 250-150 | 48.8% |
| Exp. 4 | Exponent. | 0.000001 | Yes | 0.0001 | Adam | 0.0001 | 40 | 250-150 | 50.5% |
| Exp. 5 | Exponent. | 0.000001 | Yes | 0.0001 | SGD | 0.0001 | 40 | 250-150 | 47.8% |
| Exp. 6 | None | None | No | None | Adam | 0.0001 | 40 | 250-150 | 46.1% |
| Exp. 7 | None | None | Yes | 0.0001 | SGD | 0.01 | 40 | 200-100 | 50.7% |

Table 1: Table of random experiments. Not actual results or suggestions.

Column architecture only denotes hidden layer sizes, while input layer is the size of the train data feature and output layer is of size 10. On all experiments using Adam optimizer I was using values $\beta = 0.9$ and $\gamma = 0.999$.

As it can be seen from the table, I achieved the best result in the Experiment 7 using SGD. In general the accuracy were always better when I was using SGD in comparison to Adam optimizer. By adjusting learning speed I didn't experience any significant changes in result, expect for the fact that in some case I was able to avoid overflows during calculations if the learning rate decay was higher, in other words, higher value of $k$. Using L2 regularization Loss was higher for training data, but lower for validation data - which is good and expected.