Course: Deep Learning **(14.11.2021)**

Submission Assignment #1

Coordinator: Jakub Tomczak *Name:* Bartłomiej Boczek, *Netid:* bbk205

1 Question answers

Question 1 *Work out the local derivatives of both,in scalar terms. Show the derivation.Assume that the target class is given as an integer value.*
In this section I will work out the derivation of the **cross entropy loss** function. The loss is defined as follows:

$$loss = \sum_i l_i$$

$$l_i = \begin{cases} -\log y_c & \text{if } c = i \\ 0 & \text{otherwise} \end{cases}$$

Work out the derivative:

$$\frac{\partial l}{\partial y_i} = \begin{cases} -\frac{1}{y_c} & \text{if } c = i \\ 0 & \text{otherwise} \end{cases}$$

In the following section derivation of the **softmax function** will be presented. The softmax function is defined as:

$$y_i = \frac{e^{o_i}}{\sum_{j=0}^{nclass} e^{o_j}}$$

Because of the sum in the softmax formula, y_i depends on all o_j, not just o_i, it is helpful to work out and separately cases where $i = j$ and $i \neq j$.

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial \frac{e^{o_i}}{\sum_{j=0}^{nclass} e^{o_j}}}{\partial o_j}$$

Quotient rule is need to calculate the derivative of softmax, because it is defined as a fraction.

- $i = j$

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial \frac{e^{o_i}}{\sum_{j=0}^{nclass} e^{o_j}}}{\partial o_j} = \frac{e^{o_i} * \sum_{j=0}^{nclass} e^{o_j} - e^{o_j} * e^{o_i}}{[\sum_{j=0}^{nclass} e^{o_j}]^2} = \frac{e^{o_i}}{\sum_{j=0}^{nclass} e^{o_j}} \frac{(\sum_{j=0}^{nclass} e^{o_j} - e^{o_j})}{\sum_{j=0}^{nclass} e^{o_j}} = y_i(1 - y_i)$$

- $i \neq j$

  In this case the derivative $\frac{\partial e^{o_i}}{\partial o_j} = 0$, thus the equation looks like this:

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial \frac{e^{o_i}}{\sum_{j=0}^{nclass} e^{o_j}}}{\partial o_j} = \frac{0 * \sum_{j=0}^{nclass} e^{o_j} - e^{o_j} * e^{o_i}}{[\sum_{j=0}^{nclass} e^{o_j}]^2} = \frac{-e^{o_j} * e^{o_i}}{[\sum_{j=0}^{nclass} e^{o_j}]^2} = -y_j y_i$$

Finally the equation for the local derivative of the softmax function looks as follows:

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_i(1 - y_i) & \text{if } i = j \\ -y_j y_i & \text{if } i \neq j \end{cases}$$

Question 2*2 *Work out the derivative* $\frac{\partial l}{\partial o_i}$ *. Why is this not strictly necessary for a neural2network, if we already have the two derivatives we worked out above?*

Question 3 *Implement the network drawn in the image below, including the weights. Perform one forward pass, up to the loss on the target value, and one backward pass. Show the relevant code in your report. Report the derivatives on all weights (including biases). Do not use anything more than plain python and the math package.*

Below there is a code that implements forward calculation of forward and backward pass from the image. Weights of layers are stored in *W1* and *W2* lists, biases in *b1* and *b2* lists. X is the input lists and Y is the output list. All has been implemented using basic math library, in scalar terms. At the end, resulting gradients are printed. In the results V=W2, W=W1, b=b1 and c=b2 to keep consistency with the assignment text.

```
import math
from typing import List

X = [1.0, -1.0]
Y = [1, 0]
b1 = [0.0, 0.0, 0.0]
W1 = [[1.0, 1.0, 1.0], [-1.0, -1.0, -1.0]]
b2 = [0.0, 0.0]
W2 = [[1.0, 1.0], [-1.0, -1.0], [-1.0, -1.0]]

n_input=2
n_hidden = 3
n_out = 2

def sigmoid_scalar(Z: List[float]) -> List[float]:
    return [1 / (1 + math.exp(-x)) for x in Z]

def softmax_scalar(Z: List[float]) -> List[float]:
    return [math.exp(oi) / sum([math.exp(oj) for oj in Z]) for oi in Z]

# Forward pass
# Linear transformation with hidden layer
Z1 = [None] * n_hidden
for j in range(n_hidden):
    Z1[j] = sum(X[i] * W1[i][j] for i in range(n_input)) + b1[j]

# Activations for hidden layer
A1 = sigmoid_scalar(Z1)

# Linear transformation with output layer
Z2 = [None] * n_out
for j in range(n_out):
    Z2[j] = sum(A1[i] * W2[i][j] for i in range(n_out)) + b2[j]

# Activations for output layer
A2 = softmax_scalar(Z2)

# Calculate loss
trg_idx = Y.index(1)
yc = A2[trg_idx]
loss = -math.log(yc)

# Backward pass
# Gradient of activated output layer
dZ2 = [A2[i] - Y[i] for i in range(n_out)]

# Gradient of second layer weights
dW2 = [[None]*n_out for _ in range(n_hidden)]
```

```python
for j in range(n_out):
    for i in range(n_hidden):
        dW2[i][j] = dZ2[j]*A1[i]

# Gradient of second layer bias
db2 = dZ2

# Gradient of first activated output
dZ1 = [None]*n_hidden
for j in range(n_hidden):
    dZ1[j] = A1[j] * (1 - A1[j]) * sum(dZ2[i]*W2[j][i] for i in range(n_out))

# Gradient of first layer weights
dW1 = [[None]*n_hidden for _ in range(n_input)]
for j in range(n_hidden):
    for i in range(n_input):
        dW1[i][j] = dZ1[j]*X[i]

# Gradient of first layer bias
db1 = dZ1

print(f'dW = {dW1}')
print(f'db = {db1}')
print(f'dV = {dW2}')
print(f'dc = {db2}')
```

The resulting derivatives are as follows:

```
dW = [[0.0, 0.0, 0.0], [-0.0, -0.0, -0.0]]
db = [0.0, 0.0, 0.0]
dV = [[-0.44039853898894116, 0.44039853898894116], [-0.44039853898894116,
    0.44039853898894116], [-0.44039853898894116, 0.44039853898894116]]
dc = [-0.5, 0.5]
```

Question 4 *Implement a training loop for your networkand show that the training loss dropsas training progresses.*

1.1 Dataset

For training synthetic dataset provided in the assignment is used. It contains 2 classes, which distribution is presented in the Figure 1

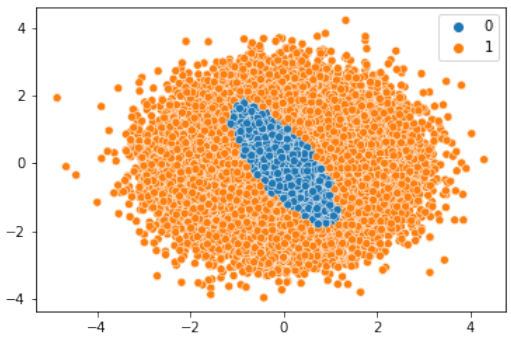

Figure 1: Synthetic data distribution

1.2 Implementation

To implement a training loop additional MLP python class has been added. It stores weights and biases, but also caches activations, input and output used in forward method, so they can be used in backward pass and parameter updates. The interface with the class is very simple. 'forward' method returns a loss of

forward pass. ‘backward‘ calculates gradients and updates neural network weights. In my implementation I use one-hot encoding - i.e. the output vector is binary and has 1 at the position of true class. Weights are initialized with normal distribution and it uses stochastic gradient descent - loss is calculated over one instance at a time.

```
import math
from random import gauss
from datasets import load_synth
from typing import List

def sigmoid_scalar(Z: List[float]) -> List[float]:
    return [1 / (1 + math.exp(-x)) for x in Z]

def softmax_scalar(Z: List[float]) -> List[float]:
    return [math.exp(oi) / sum([math.exp(oj) for oj in Z]) for oi in Z]

class MLP:
    def __init__(self, n_input, n_hidden, n_out, lr) -> None:
        self.n_input, self.n_hidden, self.n_out = n_input, n_hidden, n_out
        self.lr = lr

        self.b1 = [0.0 for _ in range(n_hidden)]
        self.W1 = [[gauss(0,1) for _ in range(n_hidden)] for _ in range(n_input)]
        self.b2 = [0.0 for _ in range(n_out)]
        self.W2 = [[gauss(0,1) for _ in range(n_out)] for _ in range(n_hidden)]

        self.A1 = None
        self.A2 = None
        self.X = None
        self.Y = None

    def forward(self, X, Y):
        Z1 = [None] * self.n_hidden
        for j in range(self.n_hidden):
            Z1[j] = sum(X[i] * self.W1[i][j] for i in range(self.n_input)) + self.b1[j]

        A1 = sigmoid_scalar(Z1)

        # Linear transformation with output layer
        Z2 = [None] * self.n_out
        for j in range(self.n_out):
            Z2[j] = sum(A1[i] * self.W2[i][j] for i in range(self.n_out)) + self.b2[j]

        # Activations for output layer
        A2 = softmax_scalar(Z2)

        # Calculate loss
        trg_idx = Y.index(1)
        yc = A2[trg_idx]
        loss = -math.log(yc)

        # Cache for backward
        self.X = X
        self.Y = Y
        self.A1 = A1
        self.A2 = A2
        return loss

    def backward_and_update(self):
        # Backward pass
        # Gradient of activated output layer
        dZ2 = [self.A2[i] - self.Y[i] for i in range(self.n_out)]

        # Gradient of second layer weights
        dW2 = [[None]*self.n_out for _ in range(self.n_hidden)]
        for j in range(self.n_out):
            for i in range(self.n_hidden):
                dW2[i][j] = dZ2[j]*self.A1[i]

        # Gradient of second layer bias
        db2 = dZ2

        # Gradient of first activated output
```

```
        dZ1 = [None]*self.n_hidden
        for j in range(self.n_hidden):
            dZ1[j] = self.A1[j] * (1 - self.A1[j]) * sum(dZ2[i]*self.W2[j][i] for i in
    range(self.n_out))

        # Gradient of first layer weights
        dW1 = [[None]*self.n_hidden for _ in range(self.n_input)]
        for j in range(self.n_hidden):
            for i in range(self.n_input):
                dW1[i][j] = dZ1[j]*self.X[i]

        # Gradient of first layer bias
        db1 = dZ1

        for j in range(self.n_hidden):
            for i in range(self.n_input):
                self.W1[i][j] -= self.lr * dW1[i][j]
            self.b1[j] -= self.lr * db1[j]

        for j in range(self.n_out):
            for i in range(self.n_hidden):
                self.W2[i][j] -= self.lr * dW2[i][j]
            self.b2[j] -= self.lr * db2[j]


n_input=2
n_hidden = 3
n_out = 2
epochs = 5
lr = 0.001

(xtrain, ytrain), (xval, yval), num_cls = load_synth()

n_train = len(xtrain)
one_hots = [[0]* n_input for _ in range(n_train)]
for i, yi in enumerate(ytrain):
    one_hots[i][yi] = 1

mlp = MLP(n_input, n_hidden, n_out, lr)

losses = list()
for epoch in range(epochs):
    epoch_losses = list()
    for x, y in zip(xtrain, one_hots):
        loss = mlp.forward(x,y)
        epoch_losses.append(loss)
        mlp.backward_and_update()
    avg_loss = sum(epoch_losses)/len(epoch_losses)
    print(f'Epoch: {epoch}, loss: {avg_loss}')
    losses.append(avg_loss)
```

As can be observe in Figure 2 the training loss drops from epoch to epoch.

Question 5 *Implement a neural network for the MNISTdata. Use two linear layers asbefore, with a hidden layer size of 300, a sigmoid activation, and a softmax activation overthe output layer, which has size 10.*

1.3 Dataset

For this task MNIST dataset has been used. Dataset contains 70000 samples with handwritten digits. In the experiments a network for digits classification will be trained. 60000 are in the training/validation sets and 10000 are used for testing. Data is randomly shuffled before training, both in batched and SGD cases.

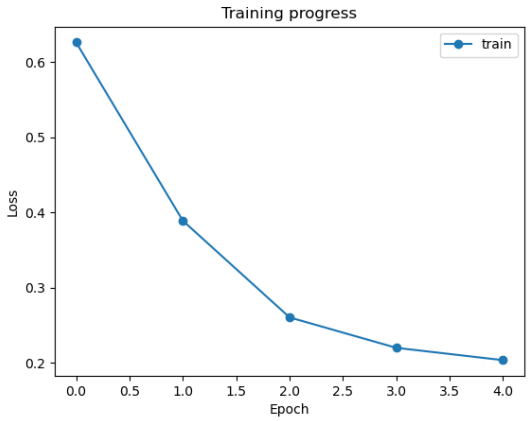

Figure 2: Training progress of scalar version of Neural Network

1.4 Implementation

In this section I will present the solution to transform scalar version from previous paragraph into vectorized version that works using *numpy* library. In *numpy* matrix operations are well optimized. What is more mathematical functions can be applied to whole matrices, without the need to do it element-wise and using for loops.

Similarly to previous implementations, in *forward* pass inputs, outputs and activations are caches, so they can be used in *backward* pass. What is more, minibatch gradient descent is implemented by accumulation of gradients from multiple iterations. When update of parameters is performed, average gradients from the minibatch are used. Training loop looks the same as in the previous paragraph.

Training progress can be seen in Figure 3. As we can see training loss is going down from epoch to epoch. Training is stable, because mini-batch of 8 was used.

```python
import numpy as np

def sigmoid(x: np.ndarray) -> np.ndarray:
    return 1 / (1 + np.exp(-x))

def softmax(x: np.ndarray) -> np.ndarray:
    exp = np.exp(x)
    return exp / exp.sum()

class MLP:
    def __init__(self, n_input: int, n_hidden: int, n_out: int, lr: float, minibatch: int
    ) -> None:
        self.n_input, self.n_hidden, self.n_out = n_input, n_hidden, n_out
        self.lr = lr

        self.b1 = np.zeros(n_hidden)
        self.W1 = np.random.normal(size=(n_input, n_hidden))
        self.b2 = np.zeros(n_out)
        self.W2 = np.random.normal(size=(n_hidden, n_out))

        self.minibatch = minibatch
        self.batch_counter = 0

        # minibatch gradient accumulators
        self.dW1 = 0
        self.dW2 = 0
        self.db1 = 0
        self.db2 = 0

        # cache
        self.A1 = None
        self.A2 = None
        self.X = None
        self.Y = None

```

```python
    def forward(self, X: np.ndarray, Y: np.ndarray) -> float:
        Z1 = X.dot(self.W1) + self.b1
        self.A1 = sigmoid(Z1)
        Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = softmax(Z2)

        loss = np.sum(-Y * np.log(self.A2))

        self.X = X
        self.Y = Y
        return loss

    def backward_and_update(self):
        dZ2 = self.A2 - self.Y
        dW2 = self.A1[:, np.newaxis].dot(dZ2[np.newaxis, :])
        db2 = dZ2

        dZ1 = self.A1 * (1 - self.A1) * self.W2.dot(dZ2)
        dW1 = self.X[:, np.newaxis].dot(dZ1[np.newaxis, :])
        db1 = dZ1

        self.dW1 += dW1
        self.dW2 += dW2
        self.db1 += db1
        self.db2 += db2
        self.batch_counter += 1
        if self.batch_counter % self.minibatch == 0:
            # Perform update
            self.W1 -= self.lr * self.dW1 / self.minibatch
            self.W2 -= self.lr * self.dW2 / self.minibatch
            self.b1 -= self.lr * self.db1 / self.minibatch
            self.b2 -= self.lr * self.db2 / self.minibatch

            self.zero_grad()

    def zero_grad(self):
        self.dW1 = 0
        self.dW2 = 0
        self.db1 = 0
        self.db2 = 0
```

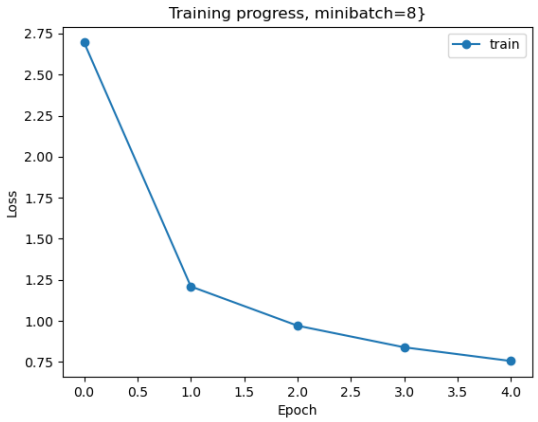

Figure 3: Training progress of vectorized version of Neural Network trained on MNIST with minibatch=8

Question6 *Work out the vectorized version of a batched forward and backward. That is,work out how you can feed your network a batch of images in a single 3-tensor, and still perform each multiplication by a weight matrix, the addition of a bias vector, computation of the loss, etc. in a single numpy call.*

To implement batched forward and backward descent one needs to decide the dimension for batch. I decided to make the first dimension of the 3-d tensor, a *batch dimension*, thus the input vector has the size of *batch* x 784. The first dimension means batch in every tensor operation during forward and backward pass. Update is performed on average gradient of the batch.

Below crucial set of code is presented. Tensors can be multiplies with *np.matmul* for tensor operations. During calculation of *dZ1*, reduction of the last excessive dimension is needed with *squeeze(-1)* method.

```
    def forward(self, X: np.ndarray, Y: np.ndarray) -> np.ndarray:
        Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = sigmoid(Z1)
        Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = softmax(Z2)
        loss = np.sum(-Y * np.log(self.A2), axis=0)
        self.X = X
        self.Y = Y
        return loss

    def backward_and_update(self):
        dZ2 = self.A2 - self.Y
        dW2 = np.matmul(self.A1[:, :, np.newaxis], dZ2[:, np.newaxis, :])
        db2 = dZ2
        dZ1 = self.A1 * (1 - self.A1) * np.matmul(self.W2[np.newaxis, :, :], dZ2[:, :, np
    .newaxis]).squeeze(-1)
        dW1 = np.matmul(self.X[:, :, np.newaxis], dZ1[:, np.newaxis, :])
        db1 = dZ1
        self.W1 -= self.lr * dW1.mean(axis=0)
        self.W2 -= self.lr * dW2.mean(axis=0)
        self.b1 -= self.lr * db1.mean(axis=0)
        self.b2 -= self.lr * db2.mean(axis=0)
```

Question 7 *Train the network on MNIST and plot the loss of each batch or instance against the timestep. This is called a learning curve or a loss curve. You can achieve this easily enough with a library like matplotlib in a jupyter notebook, or you can install a specialized tool like tensorboard. We'll leave that up to you.*

1 *Compare the training loss per epoch to the validation loss per epoch. What does the difference tell you?*

In this experiment dataset is split into 2 groups - validation and training. Model is trained only on the training part and that, after each epoch, forward pass is performed in the validation part to compute loss and accuracy of predictions. From the plot we can see that validation loss is lower than training loss after even 15 epochs. Train accuracy is lower than validation accuracy. It might suggest that validation data is not a good representation of the problem and might contains too easy examples, that are not challenging enough for the model.

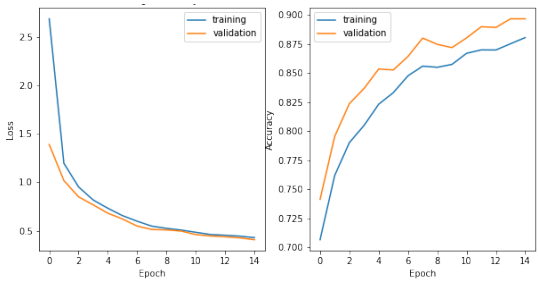

Figure 4: Training vs validation loss and accuracy

2 *Run the SGD method multiple times (at least 3) and plot an average and a standard deviation of the objective value in each iteration(e.g., see :here). What does this tell you?*

With set learning rate to 0.001 and number of epochs to 5, the same experiment but with differently initialized weights was repeated 3 times to determine how stable is the SGD algorithm. As we can see on the graph below, SGD behaves very stable.

3 *Run the SGD with different learning rates (e.g., 0.001, 0.003, 0.01, 0.03). Analyze how the learning rate value influences the final performance.*

In this experiment training was run one time for different values of learning rate, namely $[0.001, 0.003, 0.01, 0.03]$. We can see in the Figure 6, that the best learning rate in terms training loss was 0.003 and worse was 0.03.

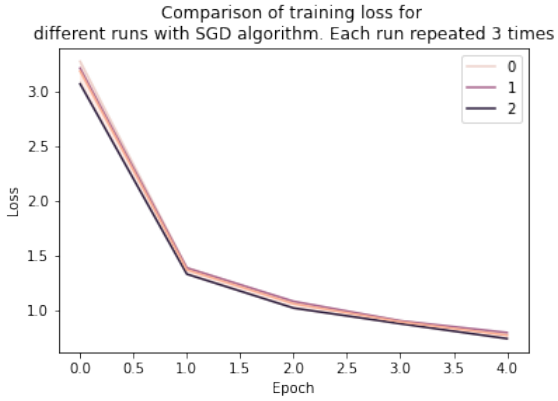

Figure 5: Training loss from 3 SGD runs

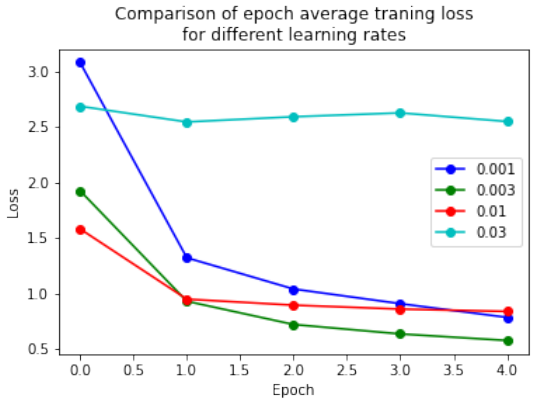

Figure 6: Training loss from 4 different learning rates

4 *Based on these experiments, choose a final set of hyperparameters, load the full training data with the canonical test set, train your model with the chosen hyperparameters and report the accuracy you get.* For the final experiment 0.003 learning rate was used. The model was trained with minibatch=8 for 10 epochs. As we can see after 10 epochs loss is not dropping significantly. After training model was evaluated on the testset. We can see from classification report (Figure 8), that the model performs best in classification of 1 and 6, and worst in detecting 5 and 8.

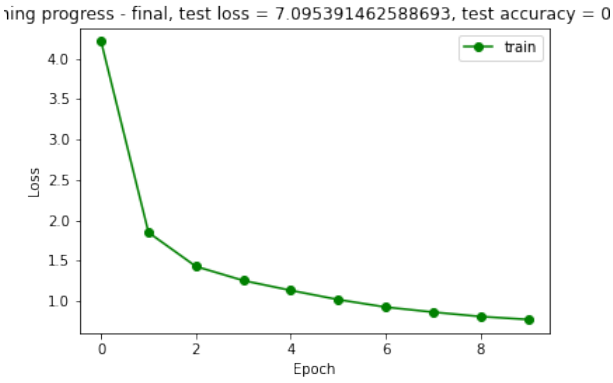

Figure 7: Training loss final experiment.

	precision	recall	f1-score	support
0	0.892604	0.898980	0.895780	980.0000
1	0.949513	0.944493	0.946996	1135.0000
2	0.823587	0.818798	0.821186	1032.0000
3	0.800801	0.792079	0.796416	1010.0000
4	0.879824	0.812627	0.844891	982.0000
5	0.773893	0.744395	0.758857	892.0000
6	0.884298	0.893528	0.888889	958.0000
7	0.840160	0.818093	0.828980	1028.0000
8	0.748538	0.788501	0.768000	974.0000
9	0.757052	0.824579	0.789374	1009.0000
accuracy	0.835700	0.835700	0.835700	0.8357
macro avg	0.835027	0.833607	0.833937	10000.0000
weighted avg	0.836928	0.835700	0.835939	10000.0000

Figure 8: Classification report on the testset

References

Appendix