Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 1

Name: Kleio Fragkedaki, *Student ID:* 2729842

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.

Answer To begin with, the local derivative of $\frac{\partial Loss}{\partial y_i}$ is:

$$\frac{\partial Loss}{\partial y_i} = \frac{\partial(\sum_i l_i)}{\partial y_i} = \begin{cases} -\frac{1}{y_c} & \text{, if i=c where c is the true class} \\ 0 & \text{, otherwise} \end{cases}$$

Using the derivation rule of $(\frac{f}{g})' = \frac{f'*g - f*g'}{g^2}$ and the equation $y_i = \frac{\exp o_i}{\sum_j \exp o_j}$, the local derivative of $\frac{\partial y_i}{\partial o_j}$ is:

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial(\frac{\exp o_i}{\sum_j \exp o_j})}{\partial o_j} = \begin{cases} \frac{\partial(\exp o_i * (\sum_j \exp o_j)^{-1})}{\partial o_i} = \frac{\exp o_i * \sum_j \exp o_j - \exp o_i * \exp o_i}{(\sum_j \exp o_j)^2} = y_i(1 - y_i) & \text{, if } i = j \\ \frac{\partial(\exp o_i * (\sum_j \exp o_j)^{-1})}{\partial o_j} = \frac{0 * \sum_j \exp o_j - \exp o_i * \exp o_j}{(\sum_j \exp o_j)^2} = -y_i * y_j & \text{, if } i \neq j \end{cases}$$

Question 2

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

Answer To calculate the derivative of $\frac{\partial l}{\partial o_i}$, we can use the derivatives of Question 1 and the chain rule as follows:

$$\frac{\partial(Loss)}{\partial o_j} = \frac{\partial(Loss)}{\partial y_i}\frac{\partial y_i}{\partial o_j} \begin{cases} -\frac{1}{y_c} * y_c(1 - y_c) = y_c - 1 & \text{, if } i = j \text{ and } i = c \\ -\frac{1}{y_c} * (-y_c * y_j) = y_j & \text{, if } i \neq j \text{ and } i = c \\ 0 & \text{, otherwise} \end{cases}$$

As shown above, the derivative $\frac{\partial l}{\partial o_i}$ is not strictly necessary for a neural network. If we have the two derivatives calculated in Question 1, we can apply the chain rule and calculate the derivative of the loss with respect to o_j.

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.*

Answer The network was split into more fine-grained modules, as shown in Figure 2. The relevant code can be seen in Appendix A, where two functions were created, one for the forward pass and one for the backward pass. In this implementation, the only functions used are plain Python and the `math` package. The derivatives on all weights (if we initialize the values as given) are shown in table 1.

Question 4

Implement a training loop for your network and show that the training loss drops as training progresses.

Table 1: The derivatives on all weights (including the biases)

Gradients	**Value**
W	[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
V	[[-0.44039853898894116,0.44039853898894116], [-0.44039853898894116,0.44039853898894116], [-0.44039853898894116,0.44039853898894116]]
b	[0.0, 0.0, 0.0]
c	[-0.5, 0.5]

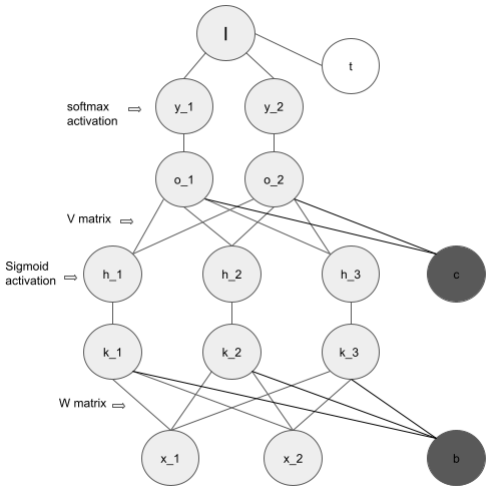


Figure 1: Fine-grained Network for Question 3.

Answer I implement a loop over the synthetic dataset for 10 epochs, and train the network of Question 5 , as indicated in Appendix B. To initialize the weights, the python package *random* was used in order to assign normally distributed random values, while the bias weights (b and c) were set to 0. To calculate the loss, I use the Stochastic Gradient Descent algorithm, and compute the loss over one instance at a time. In Figure B, the average training and validation loss per epoch is presented, and as shown, both drop as training progresses.

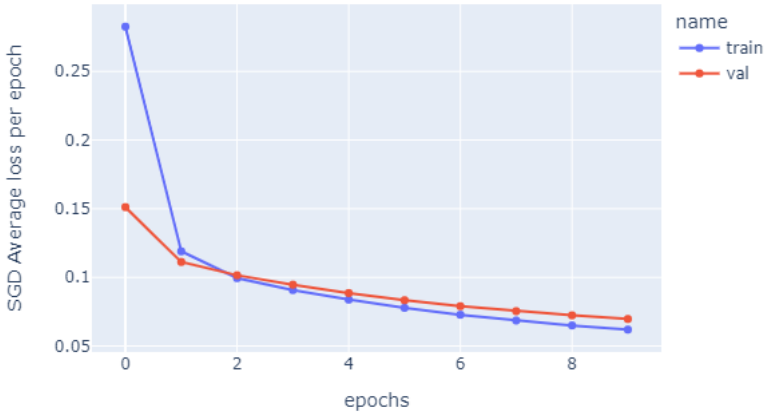


Figure 2: Average of the training loss and the validation loss per epoch. SGD algorithm was used to calculate the loss, and the average over 60000 instances, for the training set, and over 10000 instances for the validation set per epoch is presented.

Question 5

Implement a neural network for the MNIST data. Use two linear layers as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10.

Answer The Neural Network implemented can be found in Appendix C. The code snippet includes the vectorized version of the forward and backward pass, as well as a function to train the Network, update the weights, and compute the loss and accuracy. As shown in the code snippet, the Neural network includes two linear layers as previously and with standard normal initialized weights, but in this case with a hidden layer size of 300 (which is included as an input in the *train* function), one sigmoid activation, and one stoftmax activation over the output layer of size 10.

Question 6

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.*

Answer The snippet code presented in Question 5 works for Mini-batch, Batch and Stochastic Gradient Descent. In general, different versions were implemented, but the version presented in this report performs each multiplication in a single `numpy` call. To train the network with different batch sizes, there is a variable called *batch_size* that needs to be adjusted (code in Appendix C.7). [1]

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.*

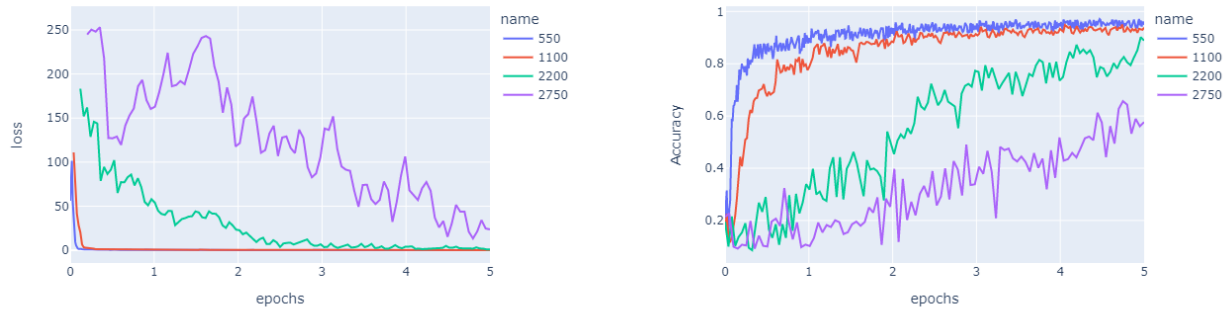
In this section, analysis is conducted using the MNIST data. The data were first normalized to values between 0 and 1, and then the normalized values were fed into the Network, as described in Questions 5 and 6. The Network was trained with different settings for five epochs (as mentioned), and the results of each experiment are presented below.

Batch Analysis In this subsection, analysis is conducted in order to pick a proper batch size for all the experiments performed. To do so, four models were trained with different batch sizes, namely 550, 1100, 2200, and 2750, and the batch training data of each are plotted in Figure 3. The experiment was conducted with a fixed learning rate of 0.01. As can be noticed from both plots, when the batch size decreases, the network converges much faster. This makes sense, since the lower the batch size (closer to the Batch Gradient Descent), the more instances in a batch, and thus the model is more certain about the direction of the update. Based on this analysis, the batch size of 550 is used in all experiments.

1. In Figure 4a, one can see the training and validation loss per epoch, while in Figure 4b the training and validation accuracy. As shown on the left plot, both learning curves decrease, with the validation loss starting from a higher value than the training loss, and ending up with a smaller value. This indicates that the validation loss does not generalize that well. On the right graph, training, both batch and total average values, and testing accuracy are displayed. The training batch accuracy quickly increases and stabilizes close to the training (computed after each epoch) and the testing values. To make this experiment a learning rate of 0.01 and a batch of 550 were chosen.

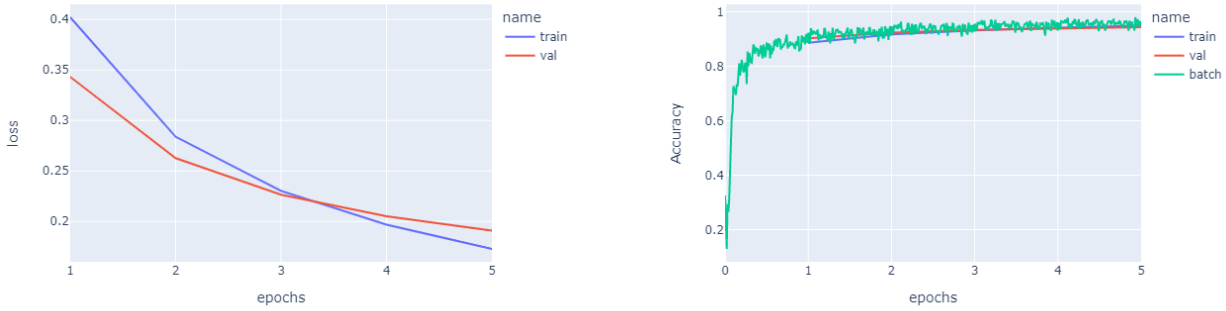
2. The Neural Network was trained three times with randomly initialized weights. In Figure 5, an average and a standard deviation of the loss and accuracy objectives are presented. As can be seen, the objectives depend on the initialized weights when training begins, but the NN quickly converges to almost the same values. In other words, the final performance does not depend on the initialization of the weights. This experiment was conducted with a learning rate of 0.01 and a batch size of 550.

[1] For complete overview of the code, please refer to the Jupyter notebook provided along with the report, all the relevant code and plots are presented there for all questions.



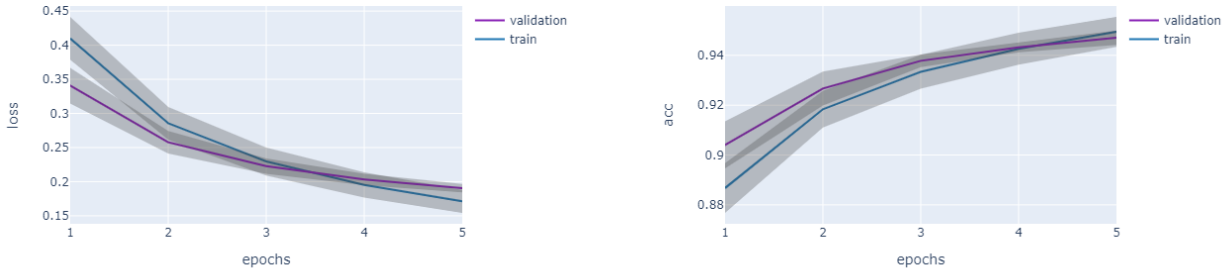
(a) Loss Curve (b) Accuracy over epochs

Figure 3: Test on different batch size values with a learning rate of 0.01.



(a) Training & Validation Loss over epochs (b) Training (for both batch and total average), & Validation Accuracy over epochs

Figure 4: Accuracy and Loss curve for training and validation data with a learning rate of 0.01 and batch size 550.

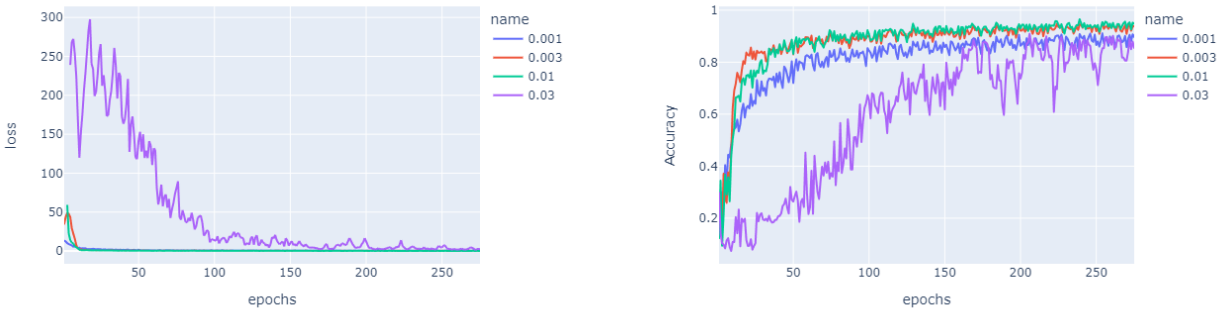


(a) Loss Curve (b) Accuracy over epochs

Figure 5: Average and Standard deviation of three model runs with learning rate 0.01, batch size 550, and randomly initialized weights.

3. To analyze how the learning rate influences the final performance, we train four models with different learning rates, namely 0.01, 0.001, 0.03, and 0.003. Figure 7 shows the batch data, with a batch size of 550, per epoch for each different model trained. As can be noticed in Figure 6b, both the lowest learning rate of 0.001, and the highest learning rate of 0.03 perform worst than the learning rate values in between. It is also interesting to highlight that all lines follow a similar trend, they increase at the beginning and stabilize after a

number of epochs. Regarding the loss curves of the different models, similar results are recorded, but with the learning rate of 0.03 to perform noticeably worse than all others, while the learning rate of 0.001 follows.



(a) Loss Curve (b) Accuracy over epochs

Figure 6: The batch data of models with different learning rates. The batch size used is fixed at 550 for all the models.

4. Based on the previous experiments, the final set of hyperparameters chosen is a batch size of 550, a learning rate of 0.01, an epoch number of 5, and normally distributed random initialization of the weights. To report the final accuracy and loss, the full training data (60000 instances) with the canonical test set (10000 instances) was loaded. The model was trained with the selected hyperparameters, and the final accuracy is around 93%, as can be seen in Table 2.

Table 2: The final loss and accuracy for the full training and test data.

Objective	**Final Train Data**	**Final Test Data**
Loss	0.17979	0.24034
Accuracy	0.94697	0.9318

(a) Loss Curve (b) Accuracy over epochs

Figure 7: Training and test data per epoch of the final model, with a batch size of 550 and a learning rate of 0.01.

Appendix

A Scalar Backpropagation Code

Source Code 1: Scalar Forwardpass

```
def forward_pass(W, V, X, t_index, b, c):
    k, h = [0., 0., 0.], [0., 0., 0.]
    o, y = [0., 0.], [0., 0.]

    # linear
    for j in range(3):
        for i in range(2):
            k[j] += W[i][j] * X[i]
        k[j] += b[j]

    # sigmoid activation
    for i in range(3):
        h[i] = (1+math.exp(-k[i]))**(-1)

    # linear
    for i in range(2):
        for j in range(3):
            o[i] += h[j] * V[j][i]
        o[i] += c[i]

    #softmax activation
    sum_exp_output = sum([math.exp(output) for output in o])
    for i in range(2):
        y[i] = math.exp(o[i])/ sum_exp_output

    # compute loss
    loss = (-1*math.log(y[t_index]))

    context = W, V, X, t_index, b, c, k, h, o, y
    return loss, context
```

Source Code 2: Scalar Backwardpass

```
def backward_pass(context, alpha=0.01):
    W, V, X, t_index, b, c, k, h, o, y = context
    dy, do = [0., 0.], [0., 0.]
    dV, dW = [[0., 0.], [0., 0.], [0., 0.]], [[0., 0. , 0.], [0., 0., 0.]]
    dh, dk = [0. , 0., 0.], [0. , 0., 0.]
    db = [0., 0., 0.]

    # compute d(Loss)/dy[i]
    for i in range(2):
        if i == t_index: dy[i] = -1/y[i]
        else: dy[i] = 0

    # compute d(Loss)/do[i]
    for i in range(2):
        for j in range(2):
            if i != j and i == t_index:
                do[j] += dy[i]*(-1)*y[i]*y[j]
            elif i == j and i == t_index:
                do[j] += dy[i]*y[i]*(1-y[i])
```

```
            elif i != t_index:
                do[j] += 0

    # compute d(Loss)/dV[j][i] and d(Loss)/dh[i]
    for i in range(2):
        for j in range(3):
            dV[j][i] = do[i]*h[j]
            dh[j] += do[i]*V[j][i]
    dc = do

    # compute d(Loss)/dk[i]
    for j in range(3):
        dk[j] = dh[j] * h[j] * (1-h[j])

    # compute d(Loss)/dW[i][j] and d(Loss)/db[j]
    for j in range(3):
        for i in range(2):
            dW[i][j] = dk[j] * X[i]
        db[j] = dk[j]

    # update W and b parameters
    for j in range(3):
        for i in range(2):
            W[i][j] += -alpha*dW[i][j]
        b[j] += -alpha*db[j]

    # update V and c parameters
    for i in range(2):
        for j in range(3):
            V[j][i] += -alpha*dV[j][i]
        c[i] += -alpha*dc[i]

    results = W, V, b, c
    gradients = dy, do, dh, dk, dV, dW, db, dc
    return gradients, results
```

B Implement training loop using the synthetic dataset

Source Code 3: Loop over the synthetic dataset

```
W, V, b, c = initialize_weights(is_random=True)

# save loss values for train and validation
avg_loss_train, avg_loss_val = [], []
loss_train_epoch1, loss_val_epoch1 = [], []
epochs = 10
alpha = 0.01 # learning rate

for epoch in range(epochs):
    # train data
    loss_train, loss_val = [], []
    for X, t_index in zip(xtrain, ytrain):
        loss, context = forward_pass(W, V, list(X), t_index, b, c)

        gradients, results = backward_pass(context, alpha)
        W, V, b, c  = results

        loss, context = forward_pass(W, V, list(X), t_index, b, c)
```

```
        loss_train.append(loss)

    # validation data
    for X, t_index in zip(xval, yval):
        loss, context = forward_pass(W, V, list(X), t_index, b, c)
        loss_val.append(loss)

    avg_loss_train.append(sum(loss_train)/len(loss_train))
    avg_loss_val.append(sum(loss_val)/len(loss_val))
```

C Vectorized Network

Source Code 4: Helper Functions

```
def update_weights(W, V, b, c, gradients, alpha=0.01):
    dV, dW, db, dc = gradients
    W = W - alpha*dW
    b = b - alpha*db
    V = V - alpha*dV
    c = c - alpha*dc

    return W, V, b, c

def compute_loss_acc(y, t_index):
    # pick the y value that corresponds to y_c
    loss = np.mean(-np.log(y[np.arange(y.shape[0]), t_index]))
    preds = list(np.argmax(y, axis=1))
    accuracy = np.mean(preds == t_index)
    return loss, accuracy
```

Source Code 5: Vectorized Forward Pass

```
def forward_pass_vectorized(X, W, V, b, c):
    # linear
    k = np.dot(X, W) + b

    # sigmoid activation
    h = sigmoid(k)

    # linear
    o = np.dot(h, V) + c

    # softmax activation
    exp_o = np.exp(o)
    y = exp_o/np.sum(exp_o, axis=1, keepdims=True)

    context = k, h, o, y
    return context
```

Source Code 6: Vectorized Backward Pass

```
def backward_pass_vectorized(X, t_index, V, context):
    k, h, o, y = context

    # compute d(Loss)/do[i]
```

```
    # https://davidbieber.com/snippets/2020-12-12-derivative-of-softmax-and-the-softmax-cross-entropy-loss/
    t = convert_onehotencoding(y, t_index)
    do = (y-t)

    # compute d(Loss)/dV[j][i] and d(Loss)/dh[i]
    dV = (h.T) @ do
    dh = do @ V.T
    dc = np.sum(do, axis=0)

    # compute d(Loss)/dk[i]
    dk = np.multiply(np.multiply(dh, h), (1-h))

    # compute d(Loss)/dW[i][j] and d(Loss)/db[j]
    dW = X.T @ dk
    db = np.sum(dk, axis=0)

    return dV, dW, db, dc
```

Source Code 7: Train Function - Batched Implementation

```
# train data
def train(x_train, t_index, x_val, t_val, num_mcls, nr_hlayers, epochs, alpha, batch_size=1, plot_batches=True):
    (W, V, b, c) = initialize_weights_vector(x_train.shape[1], nr_classes=num_mcls, nr_hlayers=nr_hlayers)

    # In this case mini batch becomes same as batch gradient descent
    if batch_size > x_train.shape[0]:
        batch_size = x_train.shape[0]

    num_batches = round(x_train.shape[0] / batch_size)
    batches, y_batches = np.array_split(x_train, num_batches), np.array_split(t_index, num_batches)

    epochs_loss_train, epochs_loss_val, epochs_batch_loss = [], [], []
    epochs_acc_train, epochs_acc_val, epochs_batch_acc = [], [], []

    for epoch in tqdm(range(epochs), desc='epochs'):
        for (batch_X, batch_t_index) in tqdm(zip(batches, y_batches), total=len(y_batches), desc='train', position=0):
            context = forward_pass_vectorized(batch_X, W, V, b, c)
            gradients = backward_pass_vectorized(batch_X, batch_t_index, V, context)

            (W, V, b, c) = update_weights(W, V, b, c, gradients, alpha=alpha)

            if plot_batches:
                # loss and accuracy for batch data
                context = forward_pass_vectorized(batch_X, W, V, b, c)
                _, _, _, pred = context
                loss, acc = compute_loss_acc(pred, batch_t_index)
                epochs_batch_loss.append(loss)
                epochs_batch_acc.append(acc)

        # train
        context = forward_pass_vectorized(x_train, W, V, b, c)
        _, _, _, pred_train = context
        loss_train, acc_train = compute_loss_acc(pred_train, t_index)
        epochs_loss_train.append(loss_train)
        epochs_acc_train.append(acc_train)

        # validation
        context = forward_pass_vectorized(x_val, W, V, b, c)
        _, _, _, pred_val = context
        loss_val, acc_val = compute_loss_acc(pred_val, t_val)
```

```
        epochs_loss_val.append(loss_val)
        epochs_acc_val.append(acc_val)

        print("Epoch {}: Train Loss: {} - Train Accuracy: {}".format(
            epoch, round(epochs_loss_train[epoch], 5), round(epochs_acc_train[epoch], 5)))
        print("Epoch {}: Validation Loss: {} - Validation Accuracy: {}".format(
            epoch, round(epochs_loss_val[epoch], 5), round(epochs_acc_val[epoch], 5)))

    return epochs_loss_train, epochs_acc_train, \
           epochs_loss_val, epochs_acc_val, \
           epochs_batch_loss, epochs_batch_acc
```