FYS-3033 Assignment 1

vho023

March 2022

1 Task 1

1.1 a)

The outputsize of a convolution is given by the equation:

$$\frac{I - K + 2P}{S} + 1 = O$$

This equation might be rewritten to calculate kernel size in stead of output size. This equation will be:

$$K = -OS + 2P + S + 1$$

By using this equation we can find the kernel size to be 5 in both directions in both the convolutional layers.

1.2 b)

All paramaters of the model is shown in table 1

Layer	Input Size	Output Size	Weights	Biases	Total Paramaters
Input layer	-	1,28,28	-	-	-
Conv 1	N,1,28,28	6,24,24	6x5x5	6	156
Pool 1	6,24,24	6,12,12	-	-	-
Conv 2	6,12,12	16,8,8	5x5x6x16	16	2416
Pool 2	16,8,8	16,4,4	-	-	-
Fully connected 1	16,4,4	N,120	256x120	120	30840
Fully connected 2	N,120	N,84	120x84	84	10164
Out	N,84	N,10	84x10	10	850
Total:	-	-	44190	236	44426

Table 1: Inputshape, outputshape and paramaters of each layer in the model

1.3 c)

The benefit of using convolutions when working with grid strucured data is that the paramaters of a convolution kernel can be reused over the whole image. This leads to much less paramaters that needs to be trained, which makes the training of the network faster.

1.4 d)

The relu layer is implemented by setting all negative values of the input to zero and letting all other values be the same. Numpy vectorization is used to increase this effciency.

1.5 e)

Because, we do not have a layer that flattens the input of the fully connected layers this has to be done. After the input is flattend matrix multiplication is used to calculate the output of the layer. Backpropogation is done in much the same way to calculate the gradients and update the weights and biases.

1.6 f)

The forward pass of the convolution layer simply convolves the filters with the input. To make this process faster the implementation is done as a matrix multiplication of vectorized versions of the kernel and the input. The same optimization technique is used in the backpropogation of this layer. When calculating the gradients with regards to the weights the error from the previous layer is used as a kernel and convolved with the original input. The same holds for calculating the gradient wrt. the input however, now the error from the previous layer is convolved with the original kernels.

1.7 g)

The maxpooling is implemented in the same way as the convolution layer, however instead of multiplying with a kernel, the largest value in the grid is choosen as the output value. There are no paramaters to maxpooling which means that the backpropogation only needs to revert the error back to a proper shape before passing it further back in the model.

1.8 h)

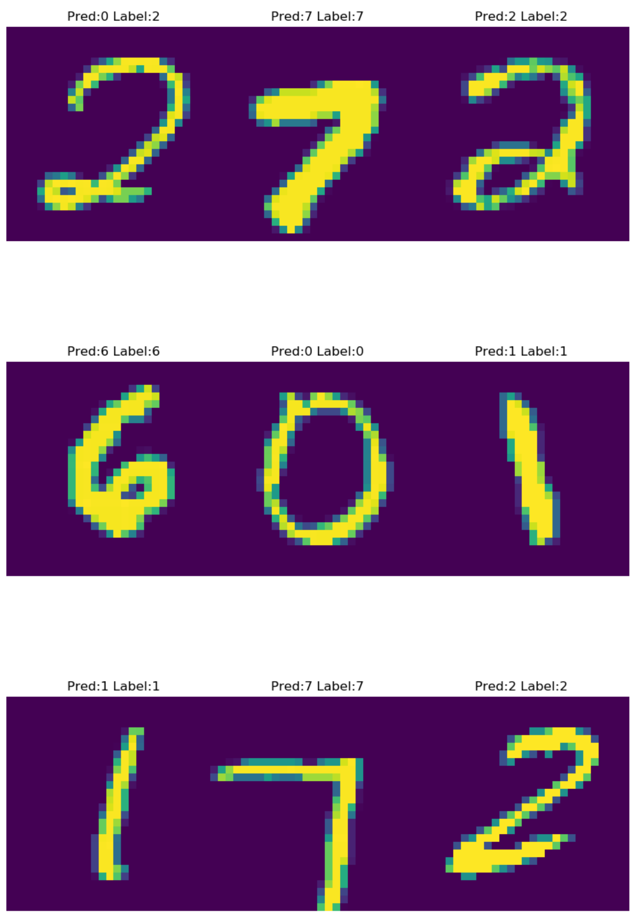

Figure 1: Predictions on random samples from the mnist dataset made by the model

The results of the network after beeing trained for 100 epochs is shown in figures 1 and 2. Figure 1 shows random samples of the mnist dataset and the predictions made by the network on these samples. However, a more relevant figure is figure 2 which shows how the loss evolves as the epochs increase. From this figure it is clear that

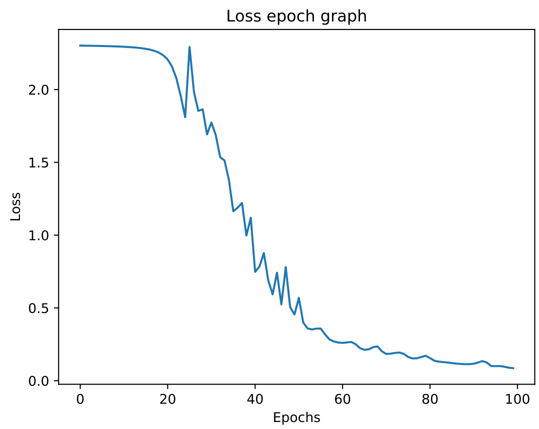

Figure 2: Evolution of avarage loss over each epochs

the network starts to converge around epoch 80. This means that the training could possibly be stopped a bit earlier without loosing to much performance. As this data is only based on training data and not on test data it is hard to say wether or not any overfitting has occured.

2 Task 2

2.1 a)

The issue of vanishing gradient appears when several layers use a activation function where the derivative has a high likelihood of beeing small. When the error is propogated backwards through the layers the gradient will move towards zero making learning infeasable. In RNN's this is espescially an issue because the gradient is propogated back in time and multiplied by the same weights over and over again. If the gradient has a singular value less than one, then the vanishing gradient effect will happen, on the other hand if the singular value is greater than one the opposite effect, "exploding gradient", will happen. The exploding gradient effect can be mitigated by capping the maximum value of the gradien. Mitigating the vanishing gradient is not as trivial. LSTM's and GRU address this issue by storing a state in each layer, this state might be forwarded directly without activation which means it will not be affected by the vanishing gradient effect. To make sure this state will not overpower the input, a forget gate is used to learn what information is important to keep in each layer.

2.2 b)

Attention and how it can be used to address the sequence to sequence task? A sequence to sequence task is a task where one sequence is mapped to another sequence. One example of such a task might be translation, where one sequence from a particular language is mapped to a sequence of another language. This problem is usually solved by having some encoder layers, which encodes the original sequence, the output of these layers are then forwarded into the decoder layer. This layer then decodes the sequence into the new sequence. These encode/decode layers are usually represented as RNN's, however, RNN's has the problem of beeing quite limited in memory. Attention can then be used to increase this memory, by storing information between the encoders and decoders.

2.3 c)

The implementation of the lstm layer is done by concatenating the input and the previous state into one large matrix. The same is done for the weights for each of the gates. These two large matricies are multiplied together. This is done so that the calculation can be done in one matrix multiplication. The back propogation is done is much the same way, where the weight gradients is concatenated into one large matrix which means that the gradient wrt. previous state and the gradient wrt. input can be calculated using one matrix multiplication instead of four.

2.4 d)

Output of RNN after 196 epochs

```
*****************
EPOCH 196
*****************
Loss: 0.15651827021469314
*****************
Prediction from input (just next char):
usand curbs
Of more strong link asunder than can
*****************
Generated character is fed as input to next time step:
uthe'
Whold become such a bready
Trough you do you wise rest.

COMIUS:
You king o' the most Corpens of the would hang to ving, for heady would you have,
And were I any their countens to mards, come
O the sonat, then with being answer'd,
And a gaartry's prestnves me pauthy's whol



________________________________________________________
Executed in  481,10 mins    fish           external
   usr time  477,28 mins  1292,00 micros  477,28 mins
   sys time    4,76 mins   570,00 micros    4,76 mins
```

3 Appendix

```python
import numpy as np
import utils


def update_param(dx, learning_rate=1e-2):
    """
    Implementation of standard gradient descent algorithm.
    """
    return learning_rate * dx

def update_param_adagrad(dx, mx, learning_rate=1e-2):
    """
    Implementation of adagrad algorithm.
    """
    return learning_rate * dx / np.sqrt(mx+1e-8)

def sigmoid(x):
    """
    A numerically stable version of the logistic sigmoid function.
    """
    pos_mask = (x >= 0)
    neg_mask = (x < 0)
    z = np.zeros_like(x)
    z[pos_mask] = np.exp(-x[pos_mask])
```

```
    z[neg_mask] = np.exp(x[neg_mask])
    top = np.ones_like(x)
    top[neg_mask] = z[neg_mask]
    return top / (1 + z)

class Layers():
    def __init__(self):
        """
        store: used to store variables and pass information from forward to backward pass.
        """
        self.store = None

class FullyConnectedLayer(Layers):
    def __init__(self, dim_in, dim_out):
        """
        Implementation of a fully connected layer.

        dim_in: Number of neurons in previous layer.
        dim_out: Number of neurons in current layer.
        w: Weight matrix of the layer.
        b: Bias vector of the layer.
        dw: Gradient of weight matrix.
        db: Gradient of bias vector
        """
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.w = np.random.uniform(-1, 1, (dim_in, dim_out)) / max(dim_in, dim_out)
        self.b = np.random.uniform(-1, 1, (dim_out,)) / max(dim_in, dim_out)
        self.dw = None
        self.db = None

    def forward(self, x):
        """
        Forward pass of fully connencted layer.

        x: Input to layer (either of form Nxdim_in or in tensor form after convolution NxCxHx
        store: Store input to layer for backward pass.
        """
        self.store = x
        #Make sure to flatten input in case input is from a conv or maxpool layer
        reshaped_x = x.reshape(x.shape[0], -1)

        out = reshaped_x @ self.w + self.b

        return out

    def backward(self, delta):
        """
        Backward pass of fully connencted layer.

        delta: Error from succeeding layer
        dx: Loss derivitive that that is passed on to layers below
        store: Store input to layer for backward passs
        """
        #Flatten input
        reshaped_x = self.store.reshape(self.store.shape[0], -1)
```

```python
        dx = delta @ self.w.T
        self.dw = reshaped_x.T @ delta
        self.db = np.sum(delta, axis=0)

        # Upades the weights and bias using the computed gradients
        self.w -= update_param(self.dw)
        self.b -= update_param(self.db)
        return dx.reshape(self.store.shape)


class ConvolutionalLayer(Layers):
    def __init__(self, filtersize, pad=0, stride=1):
        """
        Implementation of a convolutional layer.

        filtersize = (C_out, C_in, F_H, F_W)
        w: Weight tensor of layer.
        b: Bias vector of layer.
        dw: Gradient of weight tensor.
        db: Gradient of bias vector
        """
        self.filtersize = filtersize
        self.pad = pad
        self.stride = stride
        self.w = np.random.normal(0, 0.1, filtersize)
        self.b = np.random.normal(0, 0.1, (filtersize[0],))
        self.dw = None
        self.db = None

    def forward(self, x):
        """
        Forward pass of convolutional layer.

        x_col: Input tensor reshaped to matrix form.
        store_shape: Save shape of input tensor for backward pass.
        store_col: Save input tensor on matrix from for backward pass.

        This implementation is heavily based upon the implementation found in
        https://github.com/parasdahal/deepnet/blob/master/deepnet/layers.py
        However, this is mostly for optimization purposes
        """
        N, C, H, W = x.shape
        F, C, HH, WW = self.filtersize

        Wout = int((W - self.filtersize[3]+2*self.pad)/self.stride+1)
        Hout = int((H - self.filtersize[2]+2*self.pad)/self.stride+1)

        self.store = (utils.im2col_indices(x, HH, WW, self.pad, self.stride),(N,C,H,W))
        col_w = self.w.reshape(F, HH*WW*C)
        out = col_w@self.store[0] + np.expand_dims(self.b,axis=1)
        out = out.reshape(F,Hout, Wout, N).transpose(3,0,1,2)

        return out

    def backward(self, delta):
```

```python
        """
        Backward pass of convolutional layer.

        delta: gradients from layer above
        dx: gradients that are propagated to layer below

        This implementation is heavily based upon the implementation found in
        https://github.com/parasdahal/deepnet/blob/master/deepnet/layers.py
        However, this is mostly for optimization purposes
        """
        ########################################################
        ######## REPLACE NEXT PART WITH YOUR SOLUTION ########
        ########################################################
        x,(N, C, H, W) = self.store
        F, C, HH, WW = self.filtersize

        Wout = int((W - self.filtersize[3]+2*self.pad)/self.stride+1)
        Hout = int((H - self.filtersize[2]+2*self.pad)/self.stride+1)

        #Update bias
        self.db = np.sum(delta, axis=(0,2,3)).reshape(F)

        #Reshape delta so that it can be used in vectorized convolution
        delta_flat = delta.transpose(1,2,3,0).reshape(F,N*Wout*Hout)

        #Create column of weights
        col_w = self.w.reshape(F, HH*WW*C)

        #Find delta input
        dx = col_w.T @ delta_flat
        dx = utils.col2im_indices(dx, (N,C,H,W), HH, WW, self.pad, self.stride)

        #Find delta weights
        self.dw = (delta_flat @ x.T).reshape(self.w.shape)

        # Updates the weights and bias using the computed gradients
        self.w -= update_param(self.dw)
        self.b -= update_param(self.db)

        return dx


class MaxPoolingLayer(Layers):
    """
    Implementation of MaxPoolingLayer.
    pool_r, pool_c: integers that denote pooling window size along row and column direction
    stride: integer that denotes with what stride the window is applied
    """
    def __init__(self, pool_r, pool_c, stride):
        self.pool_r = pool_r
        self.pool_c = pool_c
        self.stride = stride

    def forward(self, x):
        """
        Forward pass.
```

```
        x: Input tensor of form (NxCxHxW)
        out: Output tensor of form NxCxH_outxW_out
        N: Batch size
        C: Nr of channels
        H, H_out: Input and output heights
        W, W_out: Input and output width
        """
        N, C, H, W = x.shape

        #Calculate output shape
        Hout = (H - self.pool_r) // self.stride + 1
        Wout = (W - self.pool_c) // self.stride + 1

        #Reshape all channels into individuall images
        x = x.reshape(N * C, 1, H, W)
        #Create a list of (pool_c*pool_r, C*H*W) which can be used to find max
        x_col = utils.im2col_indices(x, self.pool_c,self.pool_r, 0, self.stride)
        idx = np.argmax(x_col, axis=0)

        self.store = (idx, x_col.shape, (N,C,H,W))

        #Reshape column back into output image
        out = np.reshape(x_col[idx, range(len(idx))],(Hout, Wout, N, C)).transpose(2,3,0,1)

        return out

    def backward(self, delta):
        """
        Backward pass.
        delta: loss derivative from above (of size NxCxH_outxW_out)
        dX: gradient of loss wrt. input (of size NxCxHxW)
        """

        idx, x_col_shape, (N,C,H,W) = self.store
        zeros = np.zeros(x_col_shape)

        delta_flat = delta.transpose(2,3,0,1).reshape(-1)
        zeros[idx, range(len(idx))] = delta_flat
        dx = utils.col2im_indices(zeros, (N*C, 1,H,W), self.pool_c, self.pool_r, 0, self.stri

        return dx.reshape(N,C,H,W)


class LSTMLayer(Layers):
    """
    Implementation of a LSTM layer.

    dim_in: Integer indicating input dimension
    dim_hid: Integer indicating hidden dimension
    wx: Weight tensor for input to hidden mapping (dim_in, 4*dim_hid)
    wh: Weight tensor for hidden to hidden mapping (dim_hid, 4*dim_hid)
    b: Bias vector of layer (4*dim_hid)
    """
    def __init__(self, dim_in, dim_hid):
        self.dim_in = dim_in
        self.dim_hid = dim_hid
```

```
        self.wx = np.random.normal(0, 0.1, (dim_in, 4*dim_hid))
        self.wh = np.random.normal(0, 0.1, (dim_hid, 4*dim_hid))
        self.b = np.random.normal(0, 0.1, (4*dim_hid,))

    def forward_step(self, x, h, c):
        """
        Implementation of a single forward step (one timestep)
        x: Input to layer (Nxdim_in) where N=#samples in batch and dim_in=feature dimension
        h: Hidden state from previous time step (Nxdim_hid) where dim_hid=#hidden units
        c: Cell state from previous time step (Nxdim_hid) where dim_hid=#hidden units
        next_h: Updated hidden state(Nxdim_hid)
        next_c: Updated cell state(Nxdim_hid)
        cache: A tuple where you can store anything that might be useful for the backward pas
        """

        #Reshape X so that it matches h
        x = x.reshape(h.shape[0],-1)
        #Concatenate earlier weights
        hx = np.concatenate((h,x), 1)
        #Concatenate weight matricies
        whwx = np.concatenate((self.wh, self.wx), 0)

        #Multiply input and weight matrix
        out = (hx @ whwx + self.b).T

        #Sigmoid on forget, output and update gates
        sout = sigmoid(out[:3*self.dim_hid,:])
        update_gate = sout[:self.dim_hid,:].T
        forget_gate = sout[self.dim_hid:2*self.dim_hid, :].T
        output_gate = sout[2*self.dim_hid:3*self.dim_hid, :].T
        #Tanh, on update
        tanout = np.tanh(out[3*self.dim_hid:,:]).T
        #Calculate C based on gate outputs
        next_c = c * forget_gate + (update_gate * tanout)
        #Calculate output based on C and outputgate
        next_h = output_gate*np.tanh(next_c)

        cache = (update_gate, forget_gate, output_gate, tanout, next_h, next_c)

        return next_h, next_c, cache

    def backward_step(self, delta_h, delta_c, store):
        """
        Implementation of a single backward step (one timestep)
        delta_h: Upstream gradients from hidden state
        delta_h: Upstream gradients from cell state
        store:
          hn: Updated hidden state from forward pass (Nxdim_hid) where dim_hid=#hidden units
          x: Input to layer (Nxdim_in) where N=#samples in batch and dim_in=feature dimension
          h: Hidden state from previous time step (Nxdim_hid) where dim_hid=#hidden units
          cn: Updated cell state from forward pass (Nxdim_hid) where dim_hid=#hidden units
          c: Cell state from previous time step (Nxdim_hid) where dim_hid=#hidden units
          cache: Whatever was added to the cache in forward pass
        dx: Gradient of loss wrt. input
        dh: Gradient of loss wrt. previous hidden state
        dc: Gradient of loss wrt. previous cell state
```

```
        dwh: Gradient of loss wrt. weight tensor for hidden to hidden mapping
        dwx: Gradient of loss wrt. weight tensor for input to hidden mapping
        db: Gradient of loss wrt. bias vector
        """
        hn, x, h, cn, c, cache = store
        update_gate, forget_gate, output_gate, tanout,next_h, next_c = cache
        np.zeros((4*self.dim_hid, update_gate.shape[1]))

        #Calculate dcn based on dcn in output and stored state
        dcn = delta_c.copy()
        dcn += delta_h * output_gate * (1-np.tanh(next_c)**2)
        #Derivative of output gate based on state and dcn in output
        doutput = np.tanh(next_c) * delta_h
        dforget_gate = dcn * c
        dc = dcn * forget_gate
        dupdate_gate = dcn  * tanout
        dtanout = dcn * update_gate

        dgates = np.concatenate(
            (
            (1 - update_gate) * update_gate * dupdate_gate,
            (1 - forget_gate) * forget_gate * dforget_gate,
            (1 - output_gate) * output_gate * doutput,
            (1 - np.square(tanout)) * dtanout
            ), axis=1).T

        whwx = np.concatenate((self.wh, self.wx), 0)

        dh = (self.wh @ dgates).T
        dx = (self.wx @ dgates).T

        dwh = (dgates @ h).T
        dwx = (dgates @ x).T

        db = np.sum(dgates, axis=1)

        return dx, dh, dc, dwh, dwx, db


class WordEmbeddingLayer(Layers):
    """
    Implementation of WordEmbeddingLayer.
    """
    def __init__(self, vocab_dim, embedding_dim):
        self.w = np.random.normal(0, 0.1, (vocab_dim, embedding_dim))
        self.dw = None

    def forward(self, x):
        """
        Forward pass.
        Look-up embedding for x of form (NxTx1) where each element is an integer indicating t
        N: Number of words in batch.
        T: Number of timesteps.
        Output: (NxTxE) where E is embedding dimensionality.
        """
        self.store = x
```

```
        return self.w[x,:]

    def backward(self, delta):
        """
        Backward pass. Update embedding matrix.
        Delta: Loss derivative from above
        """
        x = self.store
        self.dw = np.zeros(self.w.shape)
        np.add.at(self.dw, x, delta)
        self.w -= update_param(self.dw)
        return 0


"""
Activation functions
"""
class SoftmaxLossLayer(Layers):
    """
    Implementation of SoftmaxLayer forward pass with cross-entropy loss.
    """
    def forward(self, x, y):
        ex = np.exp(x-np.max(x, axis=1, keepdims=True))
        y_hat = ex/np.sum(ex, axis=1, keepdims=True)
        m = y.shape[0]
        log_likehood = -np.log(y_hat[range(m), y.astype(int)])
        loss = np.sum(log_likehood) / m

        d_out = y_hat
        d_out[range(m), y.astype(int)] -= 1
        d_out /= m

        return loss, d_out

class SoftmaxLayer(Layers):
    """
    Implementation of SoftmaxLayer forward pass.
    """
    def forward(self, x):
        ex = np.exp(x-np.max(x, axis=1, keepdims=True))
        y_hat = ex/np.sum(ex, axis=1, keepdims=True)
        return y_hat


class ReluLayer(Layers):
    """
    Implementation of relu activation function.
    """
    def forward(self, x):
        """
        x: Input to layer. Any dimension.
        """
        ###########################################################
        ######## REPLACE NEXT PART WITH YOUR SOLUTION ########
        ###########################################################
        self.store = x
```

```
        #Relu is the maximum of
        out = np.maximum(0,x)
        ##########################################################
        ##########################################################
        ##########################################################
        return out

    def backward(self, delta):
        """
        delta: Loss derivative from above. Any dimension.
        """
        x = self.store
        #The derivative relu is zero for x < 0 and 1 otherwise
        dx = delta * np.where(x<0, 0,1)
        return dx
```