TÉCNICO LISBOA

Deep Learning

MEIC-A

Homework 2 [EN]

Authors:

Catarina Bento (93230) catarina.c.bento@tecnico.ulisboa.pt
David Belchior (95550) davidbelchior@tecnico.ulisboa.pt

Group 32

2022/2023 – 1st Semester, Q2

Contents

Question 1

1.1.

a)

Given that $z = conv(w, x)$, where $w \in \mathbb{R}^{M \times N}$ and $x \in \mathbb{R}^{H \times W}$, the convolution's stride is 1 and there's no padding, $z \in \mathbb{R}^{H' \times W'}$, where $H' = H - M + 1$ and $W' = W - N + 1$. We'll be using a lowercase w for the convolution matrix, to avoid confusion with W, the dimension.

b)

For $i \in \{1, \cdots, H'\}$ and $j \in \{1, \cdots, W'\}$, the convolution operation can be defined as follows:

$$\begin{gathered} z = conv(W, x) \\ z_{i,j} = \sum_{k=1}^{M} \sum_{l=1}^{N} W_{k,l}\ x_{i+k-1,\ j+l-1} \end{gathered} \tag{1}$$

Let us transform z into $z' = vec(z)$:

$$z'_a = z_{\lfloor \frac{a-1}{W'} \rfloor + 1,\ (a-1)\ mod\ W' + 1}, a \in \{1, \cdots, H'W'\} \tag{2}$$

This yields a direct correspondence between a and (i, j) which we can apply into the context of our problem:

$$z'_a = \sum_{k=1}^{M} \sum_{l=1}^{N} w_{kl}\ x_{\lfloor \frac{a-1}{W'} \rfloor + k,\ (a-1)\ mod\ W' + l} \tag{3}$$

Now, we need to transform x into $x' = vec(x)$. This time, it's useful to define x' w.r.t. x's original parameters, instead of the other way around, like we did in (2):

$$x_{i,j} = x'_{W(i-1)+j}, i \in \{1, \cdots, H\}, j \in \{1, \cdots, W\} \tag{4}$$

This can then be applied to our already existing equation as follows:

$$z'_a = \sum_{k=1}^{M} \sum_{l=1}^{N} w_{k,l}\ x'_{W(\lfloor \frac{a-1}{W'} \rfloor + k - 1) + (a-1)\ mod\ W' + l} \tag{5}$$

All that's left to do is find a closed-form solution for our matrix $M \in \mathbb{R}^{H'W' \times HW} : z' = Mx'$. For $i \in \{1, \cdots, H'W'\}$ and $j \in \{1, \cdots, HW\}$:

$$M_{i,j} = \begin{cases} w_{k,l} & \text{if } \exists k \in \{1, \cdots, M\}, l \in \{1, \cdots, N\} : j = W(\lfloor \dfrac{i-1}{W'} \rfloor + k - 1) + (i-1)\ mod\ W' + l \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

It's important to point out that, for a given i, there's injectivity between (k, l) and j, otherwise different values of j could repeat the same W_{kl} coefficient, cancelling the desired equivalence between (5) and (6). This happens because, since w is a convolution matrix over x, $l \leq N \leq W$, so an increment in k represents an increase in j by a value bigger than l's maximum value.

This concludes the proof, $Q.E.D.$

c)

The total number of parameters in the original network is given by the number of parameters of the convolutional layer and the number of parameters of the fully connected layer which precedes the softmax activation, since no other stage contains learnable parameters.

For a 2D convolutional layer with a single MN filter, a stride of 1 and no padding, the number of parameters, is defined by:

$$P_{conv} = MN \tag{7}$$

Before defining the number of the parameters in the fully connected layer, we must now the dimensions of the weight matrix which represents it.

Given an input $z \in \mathbb{R}^{H' \times W'}$, the output will have dimensions $H'' \times W''$, where:

$$\begin{aligned} H'' &= \lfloor \frac{H' - 2}{2} \rfloor + 1 = \lfloor \frac{H'}{2} \rfloor \\ W'' &= \lfloor \frac{W' - 2}{2} \rfloor + 1 = \lfloor \frac{W'}{2} \rfloor \end{aligned} \tag{8}$$

After flattening, this gives us a vector with dimensions $H''W'' = \frac{H'W'}{4}$.

As such, the fully connected layer will have a $3 \times H''$ dimension, so the number of parameters will be:

$$P_{conn} = 3H''W'' \tag{9}$$

Finally, we can obtain the total number of parameters:

$$P = P_{conv} + P_{conn} = MN + 3H''W'' = MN + 3\lfloor \frac{H - M + 1}{2} \rfloor \lfloor \frac{W - N + 1}{2} \rfloor \tag{10}$$

If we replaced the convolutional layer with another fully connected layer (and assuming x is flattened before passing through it), that layer's matrix's dimensions would be $HW \times H''W''$. As such, this layer would have the following number of parameters:

$$P'_{conn} = HWH''W'' \tag{11}$$

The final number of parameters would be, in that case:

$$P' = P'_{conv} + P_{conn} = HWH''W'' + 3H''W'' = (HW + 3)\lfloor \frac{H - M + 1}{2} \rfloor \lfloor \frac{W - N + 1}{2} \rfloor \tag{12}$$

This expression contains 4th degree terms, unlike the original one, which is limited to 2nd degree terms. This represents a much higher growth rate for higher dimensionality inputs, which also brings higher computational costs to train the model.

1.2.

Since the sequence's length is HW, the input matrix is defined as $x' \in \mathbb{R}^{HW}$. The query, key and value matrices are then defined as follows:

$$\begin{aligned} Q &= x'W_Q = x' \\ K &= x'W_K = x' \\ V &= x'W_V = x' \end{aligned} \tag{13}$$

The attention probability matrix is given by:

$$P = \frac{softmax(QK^T)}{\sqrt{Columns(Q)}} = softmax(x'x'^T) \in \mathbb{R}^{HW \times HW} \tag{14}$$

Finally, the output vector is given by:

$$Z = PV = softmax(x'x'^T)x' \in \mathbb{R}^{HW} \tag{15}$$

Question 2

2.1.

A convolutional neural network (CNN) can be defined as a neural network with a special connectivity structure which has at least one convolution layer. This layer, just like in fully connected networks, uses a weight matrix to transform a given input. However, unlike the latter, this matrix "slides" through the whole input, applying the convolution operation to submatrices of the original input. This allows this kind of network to have fewer parameters than fully connected networks while having the same number of classes and features, as we could confirm, for example, in exercise 1.1.c), where we concluded that a convolutional layer requires a much smaller number of parameters than a regular fully connected layer for an output vector with the same dimensionality.

2.2.

By taking advantage of sparse/local connectivity, parameter tying/sharing (for example, getting local context from a neighbouring pixel on an image) and the reuse of the weight matrix across the whole input, the filter is able to detect the same patterns in different parts of an image/pattern, unlike fully connected networks, where, due to their inability to share the same parameters across the image to train them, a minor change in the input (for example, a rotation or reflection of the image) can easily be mislabeled.

This implies that, normally, CNNs are better at generalising classification on images and patterns than a fully connected network.

2.3.

CNNs take the lead if compared to fully connected networks when the input is comprised of an image or a pattern. For example, an orange, which is characterised for being round and, well, orange, can be found anywhere in the image because the weight matrix, if trained to detect it, is the same for any position in the image.

However, if we're dealing with independent sensors, there's no context in the input, so there's no advantage in using a CNN versus a regular neural network, if we strictly consider the generalisation ability.

2.4.

This exercise is an application of a CNN using PyTorch, a Python library dedicated to machine learning. For further details on the exact calculations done by the module, please refer to that section or to the module's documentation [1].

Listing 1: Initialisation of the CNN.

```
def __init__(self, n_classes, **kwargs):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 8, kernel_size=5, padding=2)
    self.conv2 = nn.Conv2d(8, 16, kernel_size=3)
    self.dropout_prob = dropout_prob
    self.dropout = nn.Dropout2d()
    self.fc1 = nn.Linear(576, 600)
    self.fc2 = nn.Linear(600, 120)
    self.fc3 = nn.Linear(120, n_classes)
```

Listing 2: Forward step of the CNN.

```
def forward(self, x):
    """
    x (batch_size x n_channels x height x width): a batch of training
    examples
    """
    N = x.shape[0]
    x = x.view(N, 1, 28, 28)
    # Batch size = N, images 28x28 => x.shape = [N, 1, 28, 28]

    x = F.max_pool2d(F.relu(self.conv1(x)), 2, stride=2)
    # Convolution with 5x5 filter with padding = 2 (to preserve size) and
    # 8 channels => x.shape = [N, 8, 28, 28] since 28 = 28 - 5 + 2x2 + 1
    # Max pooling with stride of 2 => x.shape = [N, 8, 14, 14]

    x = F.max_pool2d(F.relu(self.conv2(x)), 2, stride=2)
    # Convolution with 3x3 filter without padding and 16 channels =>
    # => x.shape = [N, 16, 12, 12] since 12 = 14 - 3 + 1
    # Max pooling with stride of 2 => x.shape = [N, 16, 6, 6]
```

```
x = x.view(N, 576)
# Reshape => x.shape = [N, 576]

x = F.relu(self.fc1(x))
x = F.dropout(x, training=self.training, p=self.dropout_prob)
x = F.relu(self.fc2(x))
x = F.log_softmax(self.fc3(x), dim=1)
return x
```

After experimenting with three different learning rates (0.00001, 0.0005 and 0.01), we found that the best configuration was the CNN with learning rate = 0.0005, with a final validation accuracy of 95.71%.

Table 1: Final testing accuracies per learning rate with the encoder-decoder model.

Learning rate	**Validation accuracy**
0.00001	89.38%
0.0005	95.71%
0.01	10.00%

As such, the best configuration found was the one with learning rate = 0.0005. The training loss and the validation accuracy, as a function of the epoch number, are as follows:

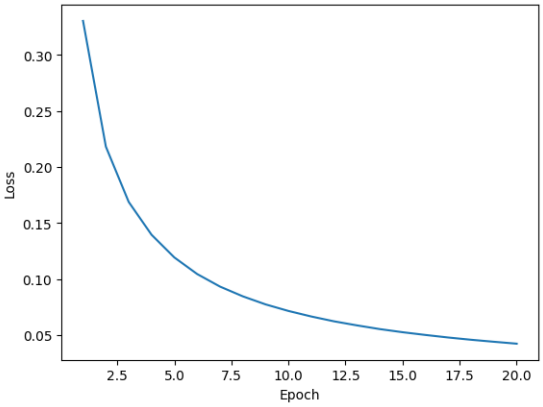

Figure 1: Training loss per epoch.

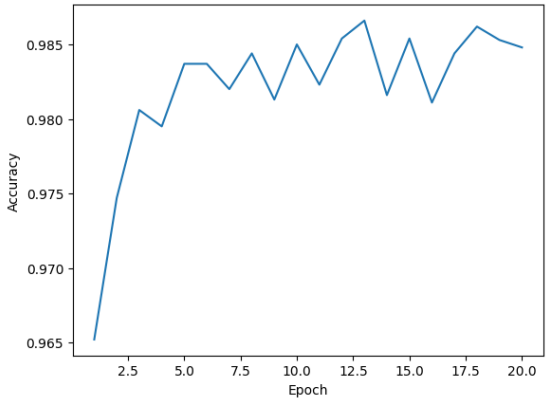

Figure 2: Validation accuracy per epoch.

2.5.

After comparing the original training example returned by our program with the CNN's first convolutional layer's activation maps (shown below), we can find a fairly accurate overlap between the highlighted character and the distinguishable shapes in each of the activation maps. This shows what the model learned to "look for", i.e., emphasise when provided an input.

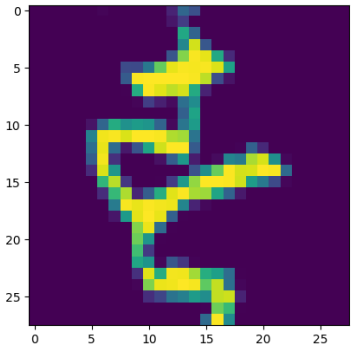

Figure 3: Original training example.

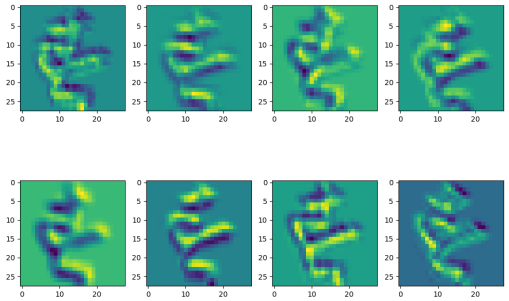

Figure 4: Activation maps for the CNN with the best configuration obtained in 2.4.

Question 3

3.1.

a)

For the encoder and decoder models, we defined each forward step as follows:

Listing 3: Forward step of the encoder.

```
def forward(self, src, lengths):
    # src: (batch_size, max_src_len)
    # lengths: (batch_size)
    emb = self.dropout(self.embedding(src))
    emb = nn.utils.rnn.pack_padded_sequence(emb, lengths, batch_first=True,
        enforce_sorted=False)
    output, final_hidden = self.lstm(emb)
    output, _ = nn.utils.rnn.pad_packed_sequence(output, batch_first=True)
    enc_output = self.dropout(output)
    # enc_output: (batch_size, max_src_len, hidden_size)
    # final_hidden: tuple with 2 tensors
    # each tensor is (num_layers * num_directions, batch_size, hidden_size)
    return enc_output, final_hidden
```

Listing 4: Forward step of the decoder.

```
def forward(self, tgt, dec_state, encoder_outputs, src_lengths):
    # tgt: (batch_size, max_tgt_len)
    # dec_state: tuple with 2 tensors
    # each tensor is (num_layers * num_directions, batch_size, hidden_size)
    # encoder_outputs: (batch_size, max_src_len, hidden_size)
    # src_lengths: (batch_size)
    # bidirectional encoder outputs are concatenated, so we may need to
    # reshape the decoder states to be of size (num_layers, batch_size,
       2*hidden_size)
    # if they are of size (num_layers*num_directions, batch_size, hidden_size)
    if dec_state[0].shape[0] == 2:
        dec_state = reshape_state(dec_state)
    tgt = tgt[:,:-1] if tgt.shape[1] > 1 else tgt
    emb = self.dropout(self.embedding(tgt))
    outputs, dec_state = self.lstm(emb, dec_state)
    if self.attn is not None:
        outputs = self.attn(outputs, encoder_outputs, src_lengths)
    outputs = self.dropout(outputs)
    # outputs: (batch_size, max_tgt_len, hidden_size)
    # dec_state: tuple with 2 tensors
    # each tensor is (num_layers, batch_size, hidden_size)
    return outputs, dec_state
```

The final error rate in the testing set was 50.02%, while the validation error, as a function of the epoch, was plotted as follows:

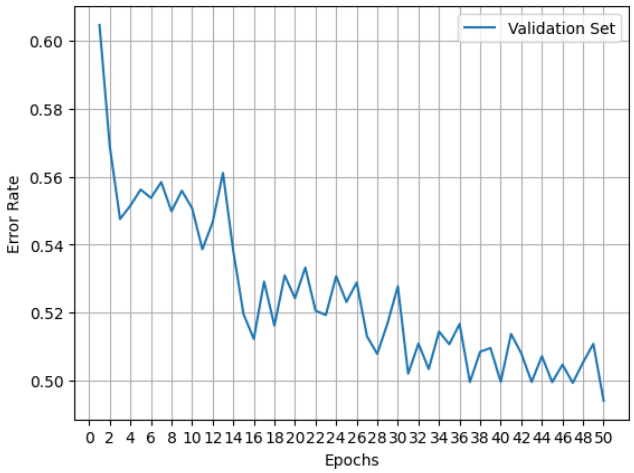

Figure 5: Validation error rate with an encoder-decoder model without attention.

b)

For this exercise, we added to the encoder and decoder defined in 3.1.a), a bilinear attention attention layer, whose forward step can be found below:

Listing 5: Forward step of the attention layer.

```
def forward(self, query, encoder_outputs, src_lengths):
    # query: (batch_size, max_tgt_len, hidden_dim)
    # encoder_outputs: (batch_size, max_src_len, hidden_dim) -> context
    # src_lengths: (batch_size)
    src_seq_mask = ~self.sequence_mask(src_lengths)
    z = self.linear_in(query)
    attn_scores = torch.bmm(z, encoder_outputs.transpose(1, 2))
    attn_scores = attn_scores.masked_fill_(src_seq_mask.unsqueeze(1),
        float("-inf"))
    attn_weights = torch.softmax(attn_scores, 2)
    c = torch.bmm(attn_weights, encoder_outputs)
    attn_out = torch.tanh(self.linear_out((torch.cat([query, c], dim=2))))
    # attn_out: (batch_size, max_tgt_len, hidden_size)
    return attn_out
```

The final error rate in the testing set was 38.08%, while the validation error, as a function of the epoch, was plotted as follows:

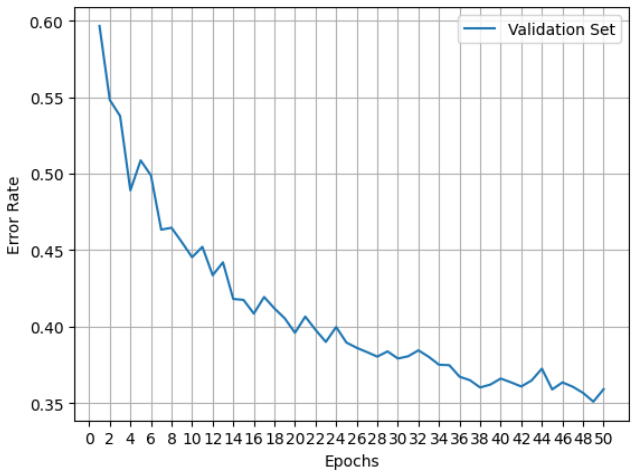

Figure 6: Validation error rate with an encoder-decoder model with an attention layer.

c)

A good way to improve results without changing the model's architecture is beam searching.

Right now, our model only considers the most probable output by choosing the $argmax(output)$. However, this is a greedy approach, since a partial sequence with a lower probability in the decoding process might yield a higher accuracy in the long run, but will never be chosen due to the model only focusing on the most probable output.

To solve this, beam search keeps in memory a set of the k most probable partial sequences (e.g., $k = 4$) to use as input to the next step of the decoder; then, after this step, the new k sequences are obtained from the newly obtained ones.

This isn't a perfect approach, since it's still somewhat greedy and doesn't compute all possible sequences (due to the overwhelmingly high computational cost), but it gives a good trade-off between accuracy and speed.

Contributions

All exercises were done in collaboration between David and Catarina, with an equal contribution between both group members.

References

[1] "PyTorch Documentation." https://pytorch.org/docs/stable/index.html. Accessed: 2023/01/14.