TÉCNICO LISBOA

Instituto Superior Técnico

Aprendizagem Profunda / Deep Learning

MEEC

Deep Learning Homework 2, 2022/2023

Students:
Afonso Brito Caiado Alemão | 96135
Rui Pedro Canário Daniel | 96317

Group **8**

Teachers:
André Martins, Francisco Melo, Gonçalo Correia,
João Santinha

January 15, 2023

Contents

0 Contribution of each member

Each member of the group contributed to the whole realization of this project.

1

1.1

In this section we consider the convolutional neural network in Fig. 1, used to classify images $x \in \mathbb{R}^{H \times W}$ into 3 possible classes.

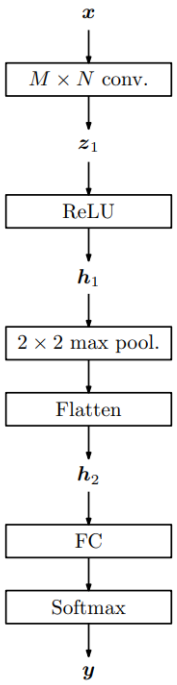


Figure 1: Architecture of the convolutional neural network.

The network has a convolutional layer with a single $M \times N$ filter, a stride of 1 and no padding, and a ReLU non linearity. This layer is followed by a 2×2 max pooling layer with a stride of 2 and no padding, and an output layer with softmax activation.

W denote the filter weights, z is the result of the convolution (given by (1)) and h is given by (2). The flattened versions of x and z, are x' and z', respectively, given by (3).

$$z = \text{conv}(W, x) \tag{1}$$

$$h = \text{ReLU}(z) \tag{2}$$

$$x' = \text{vec}(x) \ , \quad z' = \text{vec}(z) \tag{3}$$

1.1.1 (a)

In order to get the dimension of $z = z_1$, we will analyze the output dimensions of the convolutional layer.

In the input $x \in \mathbb{R}^{N_{1x} \times N_{1y}} = \mathbb{R}^{H \times W}$ we obtain $N_{1x} = H$ and $N_{1y} = W$. The convolutional layer has a single $M \times N = F_{1x} \times F_{1y}$ filter, so we obtain $F_{1x} = M$ and $F_{1y} = N$. That layer has a stride of 1: $(S_{1x}, S_{1y}) = (1, 1)$ and no padding, so its output is $z \in \mathbb{R}^{M_{1x} \times M_{1y}}$ with M_{1x} and M_{1y} given by (4) and (5).

$$M_{1x} = \left\lfloor \frac{N_{1x} - F_{1x}}{S_{1x}} + 1 \right\rfloor = H - M + 1 = H' \tag{4}$$

$$M_{1y} = \left\lfloor \frac{N_{1y} - F_{1y}}{S_{1y}} + 1 \right\rfloor = W - N + 1 = W' \tag{5}$$

Then, the dimension of z is $z \in \mathbb{R}^{(H-M+1) \times (W-N+1)} = \mathbb{R}^{H' \times W'}$.

1.1.2 (b)

In this question we show that there is a matrix $M \in \mathbb{R}^{H'W' \times HW}$, with H' given by (6) and W' given by (7), such that $z' \in \mathbb{R}^{H'W'}$ is given by (3) and (8). From last question $z \in \mathbb{R}^{H' \times W'}$. We also achieve a general expression for each element (i, j) of M.

$$H' = H - M + 1 \tag{6}$$

$$W' = W - N + 1 \tag{7}$$

$$z' = Mx' \tag{8}$$

In mathematics, the vectorization of a matrix is a linear transformation which converts the matrix into a column vector. Specifically, the vectorization of a $m \times n$ matrix A, denoted vec(A), is the $mn \times 1$ column vector obtained by stacking the columns of the matrix A on top of one another and is given by (9).

$$\text{vec}(A) = [a_{11}, \ldots, a_{m1}, a_{12}, \ldots, a_{m2}, \ldots, a_{1n}, \ldots, a_{mn}]^{\text{T}} \tag{9}$$

$x \in \mathbb{R}^{H \times W}$ is given by (10), $x' \in \mathbb{R}^{HW}$ is given by (11), $W \in \mathbb{R}^{M \times N}$ is given by (12) and $M \in \mathbb{R}^{H'W' \times HW}$ is given by (13). Then, by (1), (3) and (8), z' can be written as in (14), with c and d given by (15) (the symbol % means division remainder).

$$x = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1W} \\ x_{21} & x_{22} & \ldots & x_{2W} \\ \vdots & \vdots & \ddots & \vdots \\ x_{H1} & x_{H2} & \ldots & x_{HW} \end{bmatrix} \tag{10}$$

$$x' = \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{H1} \\ x_{12} \\ x_{22} \\ \vdots \\ x_{H2} \\ \vdots \\ x_{1W} \\ \vdots \\ x_{HW} \end{bmatrix} \tag{11}$$

$$W = \begin{bmatrix} W_{11} & W_{12} & \dots & W_{1N} \\ W_{21} & W_{22} & \dots & W_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ W_{M1} & W_{M2} & \dots & W_{MN} \end{bmatrix} \tag{12}$$

$$M = \begin{bmatrix} M_{11} & M_{12} & \dots & M_{1(HW)} \\ M_{21} & M_{22} & \dots & M_{2(HW)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(H'W')1} & M_{(H'W')2} & \dots & M_{(H'W')(HW)} \end{bmatrix} \tag{13}$$

$$z' = \begin{bmatrix} \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{mn} \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{(m+1)n} \\ \vdots \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{(m+H'-1)n} \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{m(n+1)} \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{(m+1)(n+1)} \\ \vdots \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{(m+H'-1)(n+1)} \\ \vdots \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{m(n+W'-1)} \\ \vdots \\ \sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn}x_{(m+H'-1)(n+W'-1)} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{HW} M_{1j}x_{cd} \\ \sum_{j=1}^{HW} M_{2j}x_{cd} \\ \vdots \\ \sum_{j=1}^{HW} M_{(H'W')j}x_{cd} \end{bmatrix} \tag{14}$$

$$c = H - (H(W+1) - j)\%H\ , \quad d = \left\lceil \frac{j}{H} \right\rceil \tag{15}$$

Thus, it is possible to write $M \in \mathbb{R}^{H'W' \times HW}$ as given in (16) which concludes the proof that there is a matrix $M \in \mathbb{R}^{H'W' \times HW}$, such that $z' = Mx'$.

$$M = \begin{bmatrix} W_{11} & W_{21} & \dots & W_{M1} & 0 & \dots & \dots & 0 & W_{12} & W_{22} & \dots & W_{M2} & 0 & \dots & \dots & 0 & \dots & W_{1N} & W_{2N} & \dots & W_{MN} & 0 & \dots & \dots & 0 & \dots & 0 & 0 & \dots & & & 0 \\ 0 & W_{11} & W_{21} & \dots & W_{M1} & 0 & \dots & 0 & 0 & W_{12} & W_{22} & \dots & W_{M2} & 0 & \dots & 0 & \dots & 0 & W_{1N} & W_{2N} & \dots & W_{MN} & 0 & \dots & 0 & \dots & 0 & & & & & \\ \vdots & & & & & & & \vdots & \vdots & & & & & & & \vdots & & \vdots & & & & & & & \vdots & & \vdots & & & & \ddots & \vdots \\ 0 & \dots & \dots & 0 & W_{11} & W_{21} & \dots & W_{M1} & 0 & \dots & \dots & 0 & W_{12} & W_{22} & \dots & W_{M2} & \dots & 0 & \dots & \dots & 0 & W_{1N} & W_{2N} & \dots & W_{MN} & \dots & 0 & \dots & & & 0 \\ 0 & 0 & \dots & & & & & 0 & W_{11} & W_{21} & \dots & W_{M1} & 0 & \dots & \dots & 0 & \dots & W_{1(N-1)} & W_{2(N-1)} & \dots & W_{M(N-1)} & 0 & \dots & \dots & 0 & \dots & 0 & 0 & \dots & & & 0 \\ 0 & & & & & & & 0 & 0 & W_{11} & W_{21} & \dots & W_{M1} & 0 & \dots & 0 & \dots & 0 & W_{1(N-1)} & W_{2(N-1)} & \dots & W_{M(N-1)} & 0 & \dots & 0 & \dots & 0 & & & & & \\ \vdots & & & & & \ddots & \vdots & \vdots & & & & & & & & \vdots & & \vdots & & & & & & & \vdots & & \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & & & 0 & 0 & \dots & \dots & 0 & W_{11} & W_{21} & \dots & W_{M1} & \dots & 0 & \dots & \dots & 0 & W_{1(N-1)} & W_{2(N-1)} & \dots & W_{M(N-1)} & \dots & 0 & & \dots & & & 0 \\ \vdots & & & & & & \vdots & \vdots & & & & & & & & \vdots & & \vdots & & & & & & & \vdots & & \vdots & & & & & \vdots \\ 0 & 0 & \dots & & & & 0 & 0 & 0 & & \dots & & & & & 0 & \dots & 0 & 0 & & \dots & & & & & 0 & \dots & W_{1N} & W_{2N} & \dots & W_{MN} & 0 & \dots & \dots & 0 \\ 0 & & & & & & 0 & 0 & & & & & & & & 0 & \dots & 0 & & & & & & & & 0 & \dots & 0 & W_{1N} & W_{2N} & \dots & W_{MN} & 0 & \dots & 0 \\ \vdots & & & & & \ddots & \vdots & \vdots & & & & & & \ddots & \vdots & \vdots & & & & & & & & \ddots & \vdots & & \vdots & & & & & \ddots & \vdots \\ 0 & & \dots & & & & 0 & 0 & & & \dots & & & & & 0 & \dots & 0 & & & & & & & & 0 & \dots & 0 & \dots & \dots & 0 & W_{1N} & W_{2N} & \dots & W_{MN} \end{bmatrix} \tag{16}$$

The general expression for element (i, j) of M is given by (17), with a given by (18) and b by (19).

$$M_{ij} = \begin{cases} W_{ab}, & \text{if } 1 \le a \le M \text{ and } 1 \le b \le N \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

$$a = j - H\left\lfloor \frac{j-1}{H} \right\rfloor - (i-1) + H'\left\lfloor \frac{i-1}{H'} \right\rfloor \tag{18}$$

$$b = \left\lceil \frac{j}{H} \right\rceil - \left\lfloor \frac{i-1}{H'} \right\rfloor \tag{19}$$

1.1.3 (c)

From last questions, the output from the convolutional layer $z_1 \in \mathbb{R}^{H' \times W'}$. So, after the ReLU, $h_1 \in \mathbb{R}^{H' \times W'}$.

Then, we get the dimension of the output of the max pooling layer, z_2. From its input $h_1 \in \mathbb{R}^{N_{2x} \times N_{2y}} = \mathbb{R}^{H' \times W'}$ we obtain $N_{2x} = H'$ and $N_{2y} = W'$.

The max pooling layer has kernel $2 \times 2 = F_{2x} \times F_{2y}$, so we obtain $F_{2x} = 2$ and $F_{2y} = 2$. That layer has a stride of 2: $(S_{2x}, S_{2y}) = (2, 2)$ and no padding, so its output is $z_2 \in \mathbb{R}^{M_{2x} \times M_{2y}}$ with M_{2x} and M_{2y} given by (20) and (21).

$$M_{2x} = \left\lfloor \frac{N_{2x} - F_{2x}}{S_{2x}} + 1 \right\rfloor = \left\lfloor \frac{H'}{2} \right\rfloor \tag{20}$$

$$M_{2y} = \left\lfloor \frac{N_{2y} - F_{2y}}{S_{2y}} + 1 \right\rfloor = \left\lfloor \frac{W'}{2} \right\rfloor \tag{21}$$

Then, the dimension of z_2 is $z_2 \in \mathbb{R}^{\left\lfloor \frac{H'}{2} \right\rfloor \times \left\lfloor \frac{W'}{2} \right\rfloor}$. So, after the flatten layer $h_2 \in \mathbb{R}^{\left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor}$.

First, we obtain the number of parameters in the network. We ignore the bias terms. We need $F_{1x} \cdot F_{2x} = MN$ parameters for the single filter. This gives $1 \cdot MN + 0 = MN$ parameters. With a padding of zero and stride of 1, the output after this convolution layer will be $z_1 \in \mathbb{R}^{H' \times W'}$ as verified above. The max-pooling layer with kernel size 2×2, stride of 2 and no padding has no parameters. Finally, the output layer with softmax activation takes as input vectors of dimension $h_2 \in \mathbb{R}^{\left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor}$ and maps them to a vector of dimension 3 because there are 3 possible classes, $y \in \mathbb{R}^3$, leading to $\left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor \cdot 3 + 0 = 3 \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor$ parameters. Therefore, we have a total of $MN + 3 \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor$ parameters.

Then, we obtain the number of parameters in a network where the convolutional and max pooling layers are replaced by a fully connected layer yielding an output with the same dimension as h_2. With a fully connected layer with hidden size equal to $\left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor$, we need $N_{1x} \cdot N_{1y} \cdot \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor + 0 = HW \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor$ parameters in the first layer, and $\left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor \cdot 3 + 0 = 3 \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor$ parameters in the second layer, which gives a total of $(HW + 3) \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor$ parameters.

Therefore, as usually $HW \left\lfloor \frac{H'}{2} \right\rfloor \left\lfloor \frac{W'}{2} \right\rfloor >> MN$, we verify that the network where the convolutional and max pooling layers are replaced by a fully connected layer has many more parameters, as it does not do the parameter sharing done by the CNN. In sections 2.1, 2.2 and 2.3 we discuss this topic.

1.2

In this question, we assume that, instead of using convolutions, the same $H \times W$ image $x \in \mathbb{R}^{H \times W}$ is flattened into a sequence $x' \in \mathbb{R}^{HW}$ of length $L = HW$ given by (3) and goes through a single-head self-attention layer with 1×1 projection matrices $W_Q = W_K = W_V = 1$, without any positional encodings.

By projecting the matrix to a lower dimension we obtain $Q \in \mathbb{R}^{L \times d_Q}$, $K \in \mathbb{R}^{L \times d_K}$ and $V \in \mathbb{R}^{L \times d_V}$, given by (22) and (23).

$$\begin{cases} Q = x' W_Q \\ K = x' W_K \\ V = x' W_V \end{cases} \tag{22}$$

$$Q = K = V = x' \tag{23}$$

First, we compute query-key affinity scores using scaled dot-product attention in (24). This scale by length of key vector in order to solve the problem that as d_K gets large, the variance of $q^T k$ increases, and the softmax gets very peaked, hence its gradient gets smaller. In this case, $K \in \mathbb{R}^{HW \times 1}$, so $d_K = 1$.

$$S = \frac{QK^T}{\sqrt{d_k}} = x'(x')^T \tag{24}$$

Then, we obtain the attention probabilities P as a function of x' by (25). Finally, the attention output Z as a function of x' is given by (26).

$$P = \text{softmax}(S) = \text{softmax}\Big(x'(x')^T\Big) \tag{25}$$

$$Z = PV = \text{softmax}\Big(x'(x')^T\Big)x' \tag{26}$$

2 Image classification with CNNs

In this exercise, we implement a convolutional neural network to perform classification using the Kuzushiji-MNIST dataset. We use a deep learning framework with automatic differentiation: Pytorch.

2.1

CNN have fewer free parameters than a fully-connected network with the same input size and the same number of classes (assuming it is a network for classification), as we proved in 1.1.3. This is due to CNN parameter tying/sharing that allows to reduce the number of parameters to be learned and deal with arbitrary long, variable-length, sequences. CNN is more time and memory efficient than FC network: using fewer parameters allows the increase of a deep CNN with a huge number of layers and neurons which is not possible in FC network.

The number of parameters in a FC network highly increases as the number of image pixels and hidden layers increase. This requires a lot of computation power. The number of parameters gets very large even for small networks because FC networks have a parameter between every two neurons in the successive layers.

CNN can create a very large network but with less number of parameters than FC networks. Rather than assigning a single parameter between every two neurons, in a CNN model, a single parameter may be given to a block or group of neurons since in image analysis each pixel is highly correlated to pixels surrounding it (spatial structure exists). Instead of taking the weighted sum of all the inputs in order to output one neuron we are taking the weighted sum of fewer inputs.

2.2

Although fully connected networks make no assumptions about the input, they tend to perform less and aren't good for feature extraction. They have a higher number of weights to train that results in high training time while on the other hand CNNs are trained to identify and extract the best features from the images for the problem at hand with relatively fewer parameters to train: a CNN extracts local spatial features from the input and combines the local spatial features to higher-order features that are used to linearly separate different image types.

Convolutional layers have translation equivariance due to the same filters being applied to all regions of the image. By translating the image (horizontally or vertically), the response of this layer is also translated by the same amount (ignoring boundary effects). Combining with pooling layers, it is approximately achieved invariance to small translations of the input. On the other hand, convolution layers are not (in general) equivariant to rotations or scaling transformations. Pooling layers also allow subsampling to reduce temporal/spacial scale and computation.

A fully-connected network can learn a lot of unnecessary details and patterns that are specific to the training data, rather than learning the more general patterns that are useful for making predictions on new data.

Images and patterns representing letters and numbers have a very well defined spatial structure with well-known low-level characteristics like corners, edges, lines, and other small features and that allow to capture higher-level characteristics.

So, since CNN can capture a well defined spatial structure and CNN is (approximately) invariant to translations of the input (predicting the same label as the untranslated image), CNN usually achieves better generalization on images and patterns representing letters and numbers than a fully-connected network.

2.3

A CNN extracts local spatial features from an image and combines the local spatial features to higher-order features. The higher-order features are then used to linearly separate different image types. So if the input is from a source composed of independent sensors, i.e, has no spatial structure, CNN cannot extract local spatial features from the input.

In a CNN model, a single parameter may be given to a block or group of neurons since in image analysis with spatial structure each pixel is highly correlated to pixels surrounding it. So as we verify in 2.2, CNN take advantage of local spatial coherence of images. This means that they are able to reduce the number of operation needed to process an image by using convolution on patches of adjacent pixels (adjacent pixels together are meaningful). CNN also use pooling layers, which downscale the image. This is possible because, throughout the network, we retain features that are organized spatially like an image, and thus downscaling makes sense (reducing the size of the image). On independent inputs it is not possible to downscale a vector, as there is no coherence between an input and the one next to it.

A CNN model applied to the input from a source composed of independent sensors can disturb the data from the different sensors, and relevant information may be lost. For example, downscaling does not make sense. In this case, the convolution on patches of adjacent pixels and pooling operations in a CNN may not be as effective at capturing the relevant features and patterns in the data.

Fully connected networks don't use the correlation between their inputs, so their performance won't be affected by the lack of spatial structure in the input. They may be able to achieve similar or even better performance.

Then, in this case, a CNN may not achieve better generalization than a fully connected network.

2.4

We implement a simple convolutional network with the following structure:

- A convolution layer with 8 output channels, a kernel of size 5×5, stride of 1, and padding chosen to preserve the original image size.
- A rectified linear unit activation function.
- A max pooling with kernel size 2×2 and stride of 2.
- A convolution layer with 16 output channels, a kernel of size 3×3, stride of 1, and padding of zero.
- A rectified linear unit activation function.
- A max pooling with kernel size 2×2 and stride of 2.
- An affine transformation with 600 output features.
- A rectified linear unit activation function.
- A dropout layer with a dropout probability of 0.3.
- An affine transformation with 120 output features.

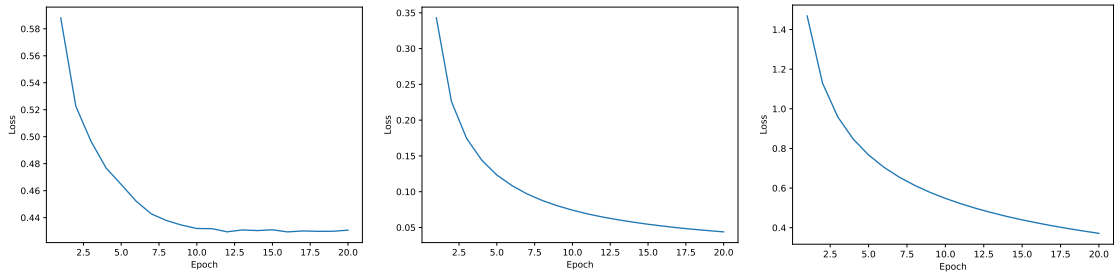
- A rectified linear unit activation function.
- An affine transformation with the number of classes followed by an output `LogSoftmax` layer.

Our code implementation is reported below.

```
class CNN(nn.Module):

    def __init__(self, dropout_prob): # used dropout prob = 0.3
        """
        The __init__ should be used to declare what kind of layers and other
        parameters the module has. For example, a CNN module has convolution,
        max pooling, activation, linear, and other types of layers. For an
        idea of how to us pytorch for this have a look at
        https://pytorch.org/docs/stable/nn.html
        """
        super(CNN, self).__init__()

        # padding chosen to preserve the original image size: pad size = (F-1)/2 = 2
        self.conv1 = nn.Conv2d(1, 8, kernel_size=(5,5), stride=1, padding="same")
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3,3), stride=1, padding="valid")

        self.max1 = nn.MaxPool2d(kernel_size=(2,2), stride=2)
        self.max2 = nn.MaxPool2d(kernel_size=(2,2), stride=2)

        self.drop = dropout_prob

        # The number of input features = number of output channels x
        # output width x output height
        self.fc1 = nn.Linear(576, 600)
        self.fc2 = nn.Linear(600, 120)
        self.fc3 = nn.Linear(120, 10) # 10 possible classes

    def forward(self, x):
        """
        x (batch_size x n_channels x height x width): a batch of training
        examples

        Every subclass of nn.Module needs to have a forward() method. forward()
        describes how the module computes the forward pass. This method needs
        to perform all the computation needed to compute the output from x.
        This will include using various hidden layers, pointwise nonlinear
        functions, and dropout. Don't forget to use logsoftmax function before
        the return

        One nice thing about pytorch is that you only need to define the
        forward pass -- this is enough for it to figure out how to do the
        backward pass.
        """
        batch_size = x.shape[0]
        # Batch size = 8, images 784->(28x28) =>
        #     x.shape = [8, 1, 28, 28]
        x = x.view(batch_size, 1, -1, 28)

        x = self.conv1(x)
        # Convolution preserving the original image size and 8 output channels
        #     x.shape = [8, 8, 28, 28]
        x = F.relu(x)

        x = self.max1(x)
        # Max pooling 2x2, stride of 2 =>
        #     x.shape = [8, 8, 14, 14] since Mi = (28 - 2) / 2 + 1 = 14

        x = self.conv2(x)
        # Convolution with 3x3 filter without padding, stride = 1
```

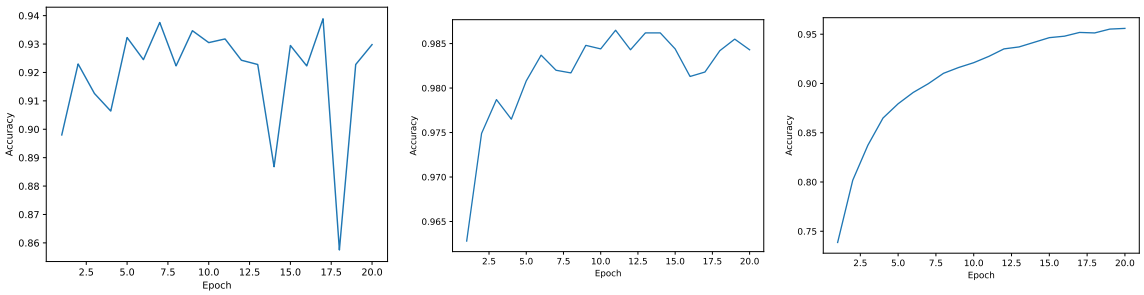
```
        # and 16 output channels =>
        #     x.shape = [8, 16, 12, 12] since Mi = (14 - 3) / 1 + 1 = 12
        x = F.relu(x)

        x = self.max2(x)
        # Max pooling 2x2, stride of 2 =>
        #     x.shape = [8, 16, 6, 6] since Mi = (12 - 2) / 2 + 1 = 6

        x = x.view(-1, 576) # 8 * 16 * 6 * 6 / 8 = 576
        # Reshape =>
        #     x.shape = [8, 576]

        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training, p=self.drop)
        x = F.relu(self.fc2(x))

        x = self.fc3(x)
        x = F.log_softmax(x, dim=1)

        return x



def train_batch(X, y, model, optimizer, criterion, **kwargs):
    """
    X (n_examples x n_features)
    y (n_examples): gold labels
    model: a PyTorch defined model
    optimizer: optimizer used in gradient step
    criterion: loss function

    To train a batch, the model needs to predict outputs for X, compute the
    loss between these predictions and the "gold" labels y using the criterion,
    and compute the gradient of the loss with respect to the model parameters.

    Check out https://pytorch.org/docs/stable/optim.html for examples of how
    to use an optimizer object to update the parameters.

    This function should return the loss (tip: call loss.item()) to get the
    loss as a numerical value that is not part of the computation graph.
    """
    model.train()
    # clear the gradients
    optimizer.zero_grad()
    # compute the model output
    yhat = model(X)
    # calculate loss
    loss = criterion(yhat, y)
    # credit assignment
    loss.backward()
    # update model weights
    optimizer.step()

    return loss.item()
```

We train the model for 20 epochs, with batch size equal to 8, l_2 decay equal to 0, and using `Adam` optimizer. We tune only the learning rate on the validation data, using the following values $\{0.00001, 0.0005, 0.01\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each leaning rate in Fig. 2 and in Fig. 3.



(a): Learning rate = 0.01. (b): Learning rate = 0.0005. (c): Learning rate = 0.00001.

Figure 2: Training loss as a function of the epoch number for different values of learning rate.



(a): Learning rate = 0.01. (b): Learning rate = 0.0005. (c): Learning rate = 0.00001.

Figure 3: Validation accuracy as a function of the epoch number for different values of learning rate.

The results are reported in Table 1.

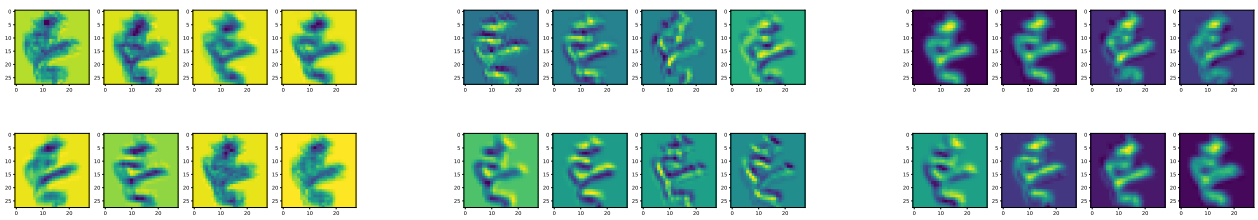
Table 1: Results of the model for `Adam` tuning the learning rate.

Learning Rate	Final training loss	Final validation accuracy	Final accuracy on the test set
0.00001	0.3715	0.9559	0.8896
0.0005	**0.0438**	**0.9843**	**0.9598**
0.01	0.4307	0.9298	0.8404

So in terms of final validation accuracy, the **tuned learning rate** is equal to **0.0005** and the final accuracy on the test set is equal to 0.9598.

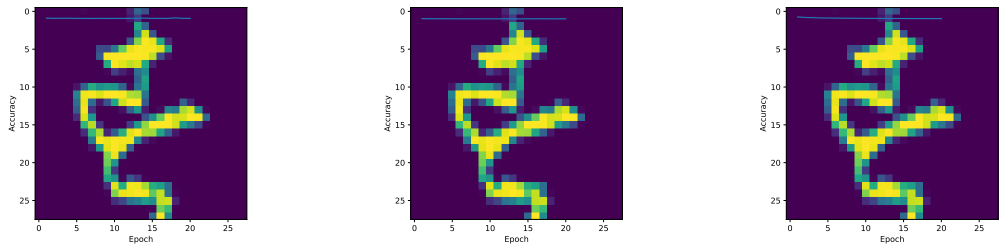
2.5

The plot the activation maps of the first convolutional layer and the original training example, are represented for each leaning rate in Fig. 4 and in Fig. 5.



(a): Learning rate = 0.01. (b): Learning rate = 0.0005. (c): Learning rate = 0.00001.

Figure 4: Activation maps of the first convolutional layer for different values of learning rate.



(a): Learning rate = 0.01. (b): Learning rate = 0.0005. (c): Learning rate = 0.00001.

Figure 5: The original training example for different values of learning rate.

Through the analysis of activation maps of the first convolutional layer against the original training examples, we verify that the features from the first convolutional layer highlight (capture) low-level characteristics like corners, edges, lines, and other small features.

On the other hand, features from subsequent convolutional layers capture higher-level characteristics: higher layers have more semantic representations such as objects or scenes.

3 Character-level Machine Translation

In this section we approach machine translation that is the problem of automatically translating a sequence of tokens from a source language into a target language. This task is usually addressed with an encoder-decoder model.

Typically, the encoder is based on LSTMs or self-attentions that process and summarises the source sentence into relevant feature representations. The representations are then passed to a target language decoder, commonly an LSTM model or a masked self-attention, which generates the respective translation of the source sentence token by token given the previous tokens and the encoder representations.

3.1

We implement a character-level machine translation model from Spanish to English: a model that translates a sequence of characters (a sentence in Spanish) into another sequence of characters (a sentence in English).

The evaluation metric is the mean error rate, which calculates the Levenshtein distance between the prediction and the true target, divided by the true length of the sequence.

3.1.1 (a)

In order to implement a character-level machine translation model using an encoder-decoder architecture with an autoregressive LSTM as the decoder and a Bidirectional LSTM in the encoder, we implement the method `forward()` of both the `Encoder` and the `Decoder` in `models.py`. The dropout layer is only applied to the embeddings and to the final outputs, in both the encoder and the decoder.

Our code implementation is reported below.

```
class Attention(nn.Module):
    def __init__(
        self,
        hidden_size,
    ):

        super(Attention, self).__init__()
        "Luong et al. general attention (https://arxiv.org/pdf/1508.04025.pdf)"
        self.linear_in = nn.Linear(hidden_size, hidden_size, bias=False)
        self.linear_out = nn.Linear(hidden_size * 2, hidden_size)

    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)
        # src_lengths: (batch_size)

        # we will need to use this mask to assign float("-inf") in the attention scores
        # of the padding tokens (such that the output of the softmax is 0
```

```
        # in those positions)
        # src_seq_mask: (batch_size, max_src_len)
        # the "~" is the elementwise NOT operator
        src_seq_mask = ~self.sequence_mask(src_lengths)
        ################################################
        # TODO: Implement the forward pass of the attention layer
        # Hints:
        # - Use torch.bmm to do the batch matrix multiplication
        #    (it does matrix multiplication to each sample in the batch)
        # - Use torch.softmax to do the softmax
        # - Use torch.tanh to do the tanh
        # - Use torch.masked_fill to do the masking of the padding tokens
        ################################################

        # Linear transformation of the query
        query = self.linear_in(query)

        # Compute bilinear attention weights with masking of the padding tokens
        attention = torch.bmm(query, encoder_outputs.transpose(1, 2))
        src_seq_mask = torch.reshape(src_seq_mask,
           (src_seq_mask.shape[0],1,src_seq_mask.shape[1]))
        attention = attention.masked_fill(src_seq_mask, -float("inf"))
        attention = torch.softmax(attention, 2)

        # Compute the context vector
        c = torch.bmm(attention, encoder_outputs)

        # Compute the attention layer output
        attn_out = torch.cat([query, c], dim=2)
        attn_out = torch.tanh(self.linear_out(attn_out))

        ################################################
        # END OF YOUR CODE
        ################################################
        # attn_out: (batch_size, max_tgt_len, hidden_dim)
        # TODO: Uncomment the following line when you implement the forward pass
        return attn_out

    def sequence_mask(self, lengths):
        """
        Creates a boolean mask from sequence lengths.
        """
        batch_size = lengths.numel()
        max_len = lengths.max()
        return (
            torch.arange(0, max_len)
            .type_as(lengths)
            .repeat(batch_size, 1)
            .lt(lengths.unsqueeze(1))
        )


class Encoder(nn.Module):
    def __init__(
        self,
        src_vocab_size,
        hidden_size,
        padding_idx,
        dropout,
    ):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size // 2
        self.dropout = dropout
```

```
87
88          self.embedding = nn.Embedding(
89              src_vocab_size,
90              hidden_size,
91              padding_idx=padding_idx,
92          )
93          self.lstm = nn.LSTM(
94              hidden_size,
95              self.hidden_size,
96              bidirectional=True,
97              batch_first=True,
98          )
99          self.dropout = nn.Dropout(self.dropout)
100
101     def forward(
102         self,
103         src,
104         lengths,
105     ):
106         # src: (batch_size, max_src_len)
107         # lengths: (batch_size)
108         ################################################
109         # TODO: Implement the forward pass of the encoder
110         # Hints:
111         # - Use torch.nn.utils.rnn.pack_padded_sequence to pack the padded sequences
112         #   (before passing them to the LSTM)
113         # - Use torch.nn.utils.rnn.pad_packed_sequence to unpack the packed sequences
114         #   (after passing them to the LSTM)
115         ################################################
116
117         emb = self.embedding(src)
118         if self.training:
119             emb = self.dropout(emb)
120
121         emb = pack(emb, lengths.cpu(), batch_first=True, enforce_sorted=False)
122         enc_output, final_hidden = self.lstm(emb)
123         enc_output, _ = unpack(enc_output, batch_first=True)
124         final_hidden = self._reshape_hidden(final_hidden)
125
126         if self.training:
127             enc_output = self.dropout(enc_output)
128
129         ################################################
130         # END OF YOUR CODE
131         ################################################
132         # enc_output: (batch_size, max_src_len, hidden_size)
133         # final_hidden: tuple with 2 tensors
134         # each tensor is (num_layers * num_directions, batch_size, hidden_size)
135         # TODO: Uncomment the following line when you implement the forward pass
136         return enc_output, final_hidden
137
138     def _merge_tensor(self, state_tensor):
139         forward_states = state_tensor[::2]
140         backward_states = state_tensor[1::2]
141         return torch.cat([forward_states, backward_states], 2)
142
143     def _reshape_hidden(self, hidden):
144         """
145         hidden:
146             num_layers * num_directions x batch x self.hidden_size // 2
147             or a tuple of these
148         returns:
149             num_layers
```

```
        """
        assert self.lstm.bidirectional
        if isinstance(hidden, tuple):
            return tuple(self._merge_tensor(h) for h in hidden)
        else:
            return self._merge_tensor(hidden)

class Decoder(nn.Module):
    def __init__(
        self,
        hidden_size,
        tgt_vocab_size,
        attn,
        padding_idx,
        dropout,
    ):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.tgt_vocab_size = tgt_vocab_size
        self.dropout = dropout

        self.embedding = nn.Embedding(
            self.tgt_vocab_size, self.hidden_size, padding_idx=padding_idx
        )

        self.dropout = nn.Dropout(self.dropout)
        self.lstm = nn.LSTM(
            self.hidden_size,
            self.hidden_size,
            batch_first=True,
        )

        self.attn = attn

    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)

        ##############################################
        # TODO: Implement the forward pass of the decoder
        # Hints:
        # - the input to the decoder is the previous target token,
        #   and the output is the next target token
        # - New token representations should be generated one at a time, given
        #   the previous token representation
        # and the previous decoder state
        # - Add this somewhere in the decoder loop when you implement the attention mechani
```

```
        # if self.attn is not None:
        #     output = self.attn(
        #         output,
        #         encoder_outputs,
        #         src_lengths,
        #     )
        ################################################

        if self.training:
            # tgt = (SOS, sequence, EOS)
            # input = (SOS, sequence)
            emb = self.embedding(tgt[:, :-1])
        else:
            emb = self.embedding(tgt)

        if self.training:
            emb = self.dropout(emb)

        outputs, dec_state = self.lstm(emb, dec_state)

        if self.training:
            outputs = self.dropout(outputs)

        # apply attention between source context and query from
        # decoder RNN
        if self.attn is not None:
            outputs = self.attn(outputs, encoder_outputs, src_lengths)

        ################################################
        # END OF YOUR CODE
        ################################################
        # outputs: (batch_size, max_tgt_len, hidden_size)
        # dec_state: tuple with 2 tensors
        # each tensor is (num_layers, batch_size, hidden_size)
        # TODO: Uncomment the following line when you implement the forward pass

        return outputs, dec_state
```

We train the model for 50 epochs, with learning rate of 0.003, a dropout rate of 0.3, a hidden size of 128, and a batch size of 64. In this question we do not use an attention mechanism to the decoder. The plot of validation error rate over epochs is represented in Fig. 6 and some prediction examples are in Fig. 7.

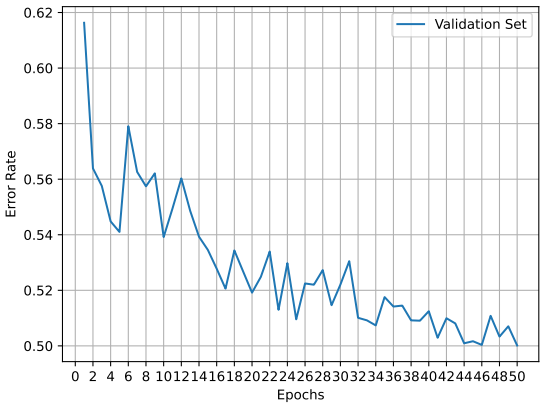


Figure 6: Validation error rate as a function of the epoch number without an attention mechanism to the decoder.

```
Validation Error Rate of the model: 0.5001
Test Error Rate of the model: 0.5048
true src: "venios ."
true tgt: "come with us ."
pred tgt: "come on ."
true src: "lo prometiste ."
true tgt: "you promised ."
pred tgt: "you should me ."
true src: "no tengo blanca ."
true tgt: "i m broke ."
pred tgt: "i have no fat ."
Final validation error rate: 0.5001
Test error rate: 0.5048
```

Figure 7: Prediction examples for the model without an attention mechanism to the decoder.

The final error rate in the validation set is equal to 0.5001 and the final error rate in the test set is equal to 0.5048.

3.1.2 (b)

In this question we add an attention mechanism to the decoder (bilinear attention), which weights the contribution of the different source characters, according to relevance for the current prediction.

Bilinear Attention is an attention mechanism defined as follows. The score between the i-th source character and the target character at the current time step, s_i, is given by (27), where z is given by (28), in which q is the hidden state of the decoder at the current time step (the query), W_q is a weight matrix, and h_i is the hidden state of the encoder at the i-th time step.

The attention weights p are given by (29) where the softmax is applied in the dimension of the source sequence size. This is equivalent to applying softmax to the score vectors.

$$s_i = z^T h_i \tag{27}$$

$$z = W_q^T q \tag{28}$$

$$p = \text{softmax}(s) \tag{29}$$

Then, the context vector c is given by (30).

$$c = \sum_{i=1}^{T_x} p_i h_i \tag{30}$$

Finally, the attention layer output, a_o, is used as the final representation at the current time step in the decoder and is computed by (31), where W_o is a weight matrix and $[q; c]$ is the concatenation of the query and the context vector.

$$a_o = \tanh\left(W_o^T [q; c]\right) \tag{31}$$

We implement the `forward()` method of the `Attention` class and slightly modify the `Decoder` implementation to account for the attention mechanism. Our code implementation is reported in section 3.1.1.

We use the same hyperparameters as in the previous exercise. We train the model for 50 epochs, with learning rate of 0.003, a dropout rate of 0.3, a hidden size of 128, and a batch size of 64. In this question we use a bilinear attention mechanism to the decoder. The plot of validation error rate over epochs is represented in Fig. 8 and some prediction examples are in Fig. 9.

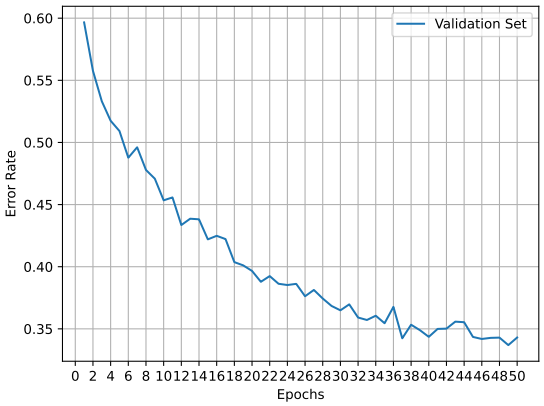


Figure 8: Validation error rate as a function of the epoch number with a bilinear attention mechanism to the decoder.

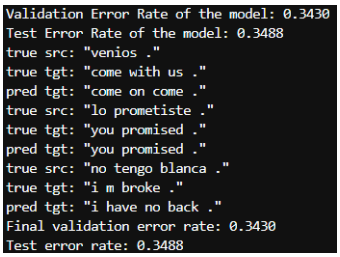


Figure 9: Prediction examples for the model with a bilinear attention mechanism to the decoder.

The final error rate in the validation set is equal to 0.3430 and the final error rate in the test set is equal to 0.3488. We can then conclude that the model performance improved with the addition of the bilinear attention mechanism to the decoder.

3.1.3 (c)

In order to improve the performance of the vanilla character-level machine translation model, that uses a bilinear attention mechanism to the decoder, without changing the architecture, we can use several techniques described below.

Use a larger beam size allow the decoder to consider more potential translations at each time step, which may lead to better translations being produced. In addition, we can test the model performance with different beam search algorithms.

Use pre-trained word embeddings such as *GloVe* or *fastText* that have been trained on large datasets and can provide the model with additional contextual information.

We can attempt different loss functions that may allow the model to learn a more robust translation function.

It is also possible to try different attention mechanisms to the decoder such as additive attention or multiplicative attention that may allow the decoder to attend to different parts of the input sequence more effectively.

On the other hand, it is also possible to use more conventional techniques: fine-tune the hyperparameters of the model, such as the learning rate and the batch size; increase the size (number of parameters) of the model that allows it to capture more complex patterns in the data; use a larger dataset for training; and add regularization in order to prevent the model from overfitting to the training data.

We can also use data augmentation by applying transformations to the input and output sequences, that improves model generalization. There are data augmentation techniques for NLP such as randomly swapping the positions of the words/characters in the sentence, randomly removing words/characters from the sentence and randomly replacing words with their synonyms.

Finally it is possible to ensemble multiple models: combine N independently trained models and obtaining a "consensus" such as averaging their predictions during decoding. This can often lead to improved robustness because it reduces the dispersion (variance) of the predictions.

References

[1] Deep Learning Slides - Slides of Theoretical Classes 2022/2023, 1st Semester (MEEC), by A. Martins, F. Melo, M. Figueiredo.

[2] 1st Semester 2022/2023, Deep Learning (IST, 2022-23), Homework 2, by André Martins, Francisco Melo, Gonçalo Correia, João Santinha.

[3] Deep Learning Practical Lectures - 2022/2023, 1st Semester (MEEC).

[4] Medium: "Fully Connected vs Convolutional Neural Networks".
Available: `https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5`.

[5] Towards Data Science: "Convolutional Layers vs Fully Connected Layers".
Available: `https://towardsdatascience.com/convolutional-layers-vs-fully-connected-layers-364f05ab460b`.

[6] Medium: "CNN (Convolution Neural Network)".
Available: `https://medium.com/analytics-vidhya/cnn-convolution-neural-network-17cc89802234`.

[7] Quora: "Why are convolutional neural networks better than other neural networks in processing data such as images and video?".
Available: `https://bit.ly/3QGDvPK`.

[8] Wikipedia: Vectorization (mathematics).
Available: `https://en.wikipedia.org/wiki/Vectorization_(mathematics)`.