TÉCNICO LISBOA

# DEEP STRUCTURED LEARNING

## HOMEWORK 3

André Godinho Number: 84006

# QUESTION 1

1.

First of all, let's tackle **translation** and **shifting invariance**. From the previous homework, on the one hand we understood that the pairwise features were "more tolerant" to shifting and translation than without this feature engineering because it exploited a bit the pixel correlation among them. On the other hand, convolutional layers take **translation** and **shifting invariance** to a whole new level. First, by applying convolution: when an input image moves to one direction, so does its feature layer produced by the convolution. Moreover, if an image has a vertical edge for example, and this vertical edge is translated, the feature layer won't detect it until it matches it. Once it happens, there will be a neuron that will fire maximally because there is a weight vector **W** that will be aligned with a portion of the input image **X**. Thus, the feature layer will have an activation moved to that direction.

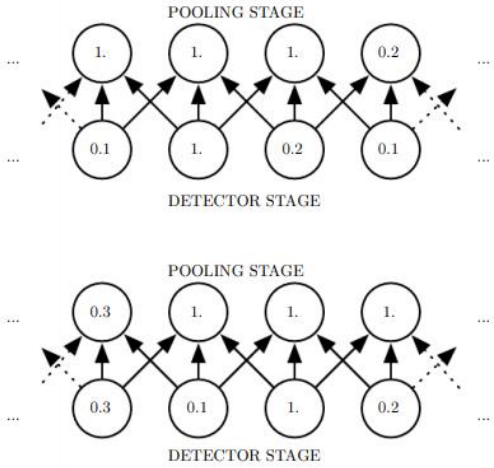



FIG.1 – MAX POOLING ENHANCES SHIFT INVARIANCE (IMAGE TAKEN FROM [1])

Observing Fig.1. we can see the impact of max pooling in **shifting invariance**, after the input been shifted to the right by one pixel. In other words, every value of the detector stage is different (has been shifted to the right) but the pooling stage layer just shifted to the right by one pixel because each max pooling unit is sensible to the maximum value of the neighbor units of each unit of the detect stage. Of course, this only works if the image is not translated too much.

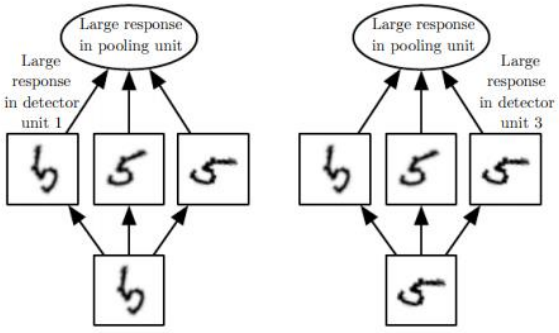



FIG.2 – MAX POOLING ENHANCES ROTATION INVARIANCE (IMAGE TAKEN FROM [1])

Secondly, the convolutional layers also more **rotation invariant** than the pairwise features from the previous homework. This happens because of the max pooling layer.
By observing Fig.2. we can see exactly that. We have three filters that can detect a hand-written 5, but they have different orientations. In the detector stage, there will be a neuron with a higher value for the handwritten 5 that matches the input. Trough max pooling we can detect this unit. Notice that this only works if the feature channels are adjacent, so that the rotated hand-written 5 don't pass to the next layer.

2.

```
class CNN(nn.Module):

    def __init__(self, n_classes, **kwargs):
        super(CNN, self).__init__()

        self.layer1 = nn.Sequential(
            # in_ch = 1 (greyscale) ; out_ch = 16 ; kernel size = 5 = F ;
 padding = (F-1)/2 = 2 ; stride = 1
            nn.Conv2d(1 , 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.layer2 = nn.Sequential(
            # in_ch = 16 (previous layer) ; out_ch = 32 ; kernel size = 7
 = F ; padding = (F-1)/2 = 3 ; stride = 1
            nn.Conv2d(16, 32, kernel_size = 7, stride = 1, padding = 3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.output_layer = nn.Linear(4 * 2 * 32, n_classes)
```

```
    def forward(self, x, **kwargs):
        x = x.view(x.shape[0],1,16,8) # [64, 1, 16, 8]
        out = self.layer1(x) # [61, 16, 8, 4]
        out = self.layer2(out) # [64, 32 , 4, 2]
        out = out.reshape(out.size(0), -1) # [64, 256]
        out = self.output_layer(out) # [64, 26]
        return out
```

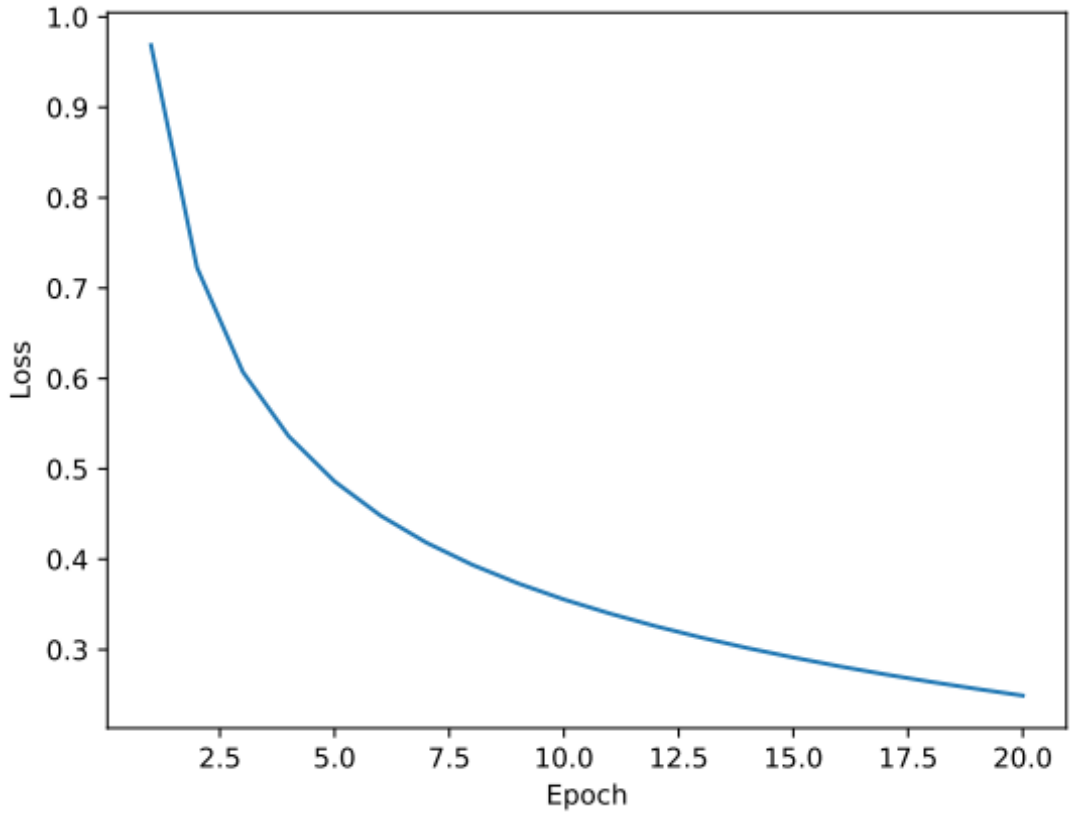



FIG.3 – TRAINING LOSS OF THE CNN IMPLEMENTED WITH 20 EPOCHS

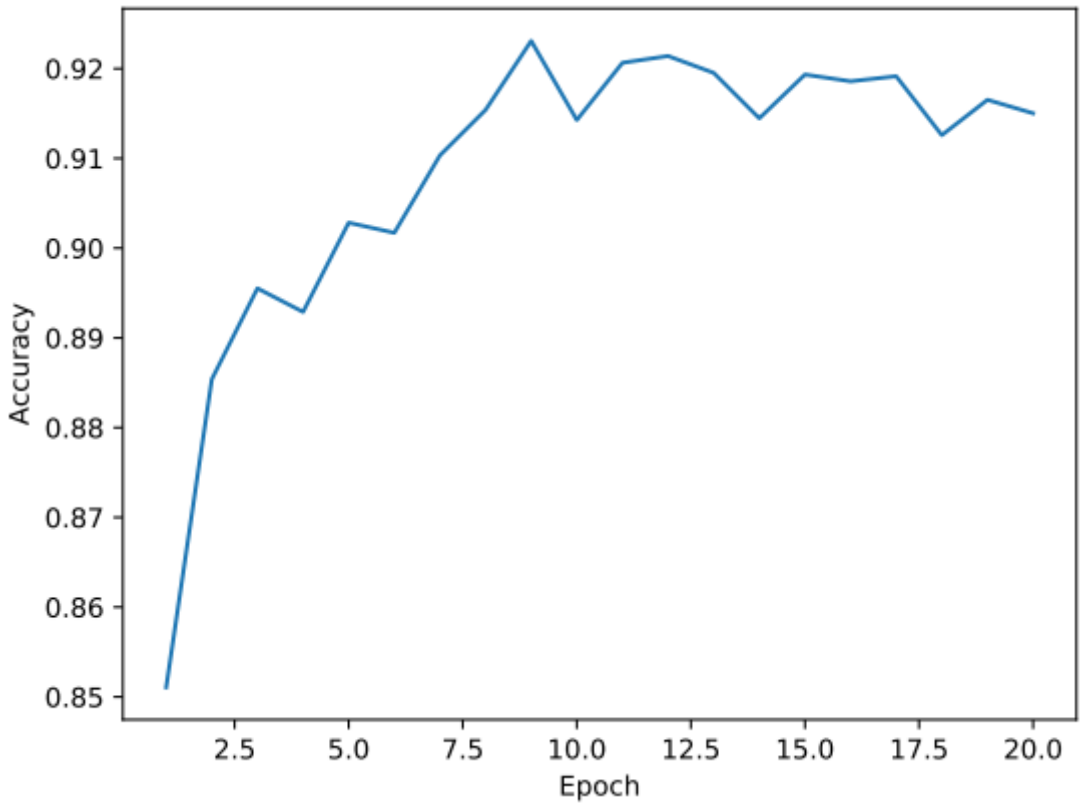



FIG.4 – ACCURACY OF THE CNN IMPLEMENTED WITH 20 EPOCHS

**The final test accuracy was 90.12%.**

3.

It was implemented a function that draws all the channels of the first and second **convolution layer**. Then, this function was invoked in *main* after training the CNN.

```
def plot_kernels(tensor, name, heatmap=0, greyscale=0, num_cols=8):
    num_kernels = tensor.shape[0]
    num_rows = 1+ num_kernels // num_cols
    fig = plt.figure(figsize=(num_cols,num_rows))

    for i in range(num_kernels):
        if heatmap:
            sns.set()
            if i == 0:
                cbar_ax = fig.add_axes([.91, .3, .03, .4])

            ax1 = fig.add_subplot(num_rows,num_cols,i+1)

            sns.heatmap(tensor[i][0,:,:], ax=ax1,
                        cbar=i == 0,
                        xticklabels=0, yticklabels=0,
                        cbar_ax=None if i else cbar_ax)
        else:
            ax1 = fig.add_subplot(num_rows,num_cols,i+1)
            if greyscale == 1:
                ax1.imshow(tensor[i][0,:,:], cmap='gray')
            else:
                ax1.imshow(tensor[i][0,:,:])
            ax1.axis('off')
            ax1.set_xticklabels([])
            ax1.set_yticklabels([])

    if heatmap:
        fig.tight_layout(rect=[0, 0, .9, 1])
        ax1.figure.savefig('%s heatmap.pdf' %(name), bbox_inches ='tight'
)
    else:
        plt.savefig('%s plot.pdf' %(name), bbox_inches ='tight')
```

The code below was the invocation in *main*.

```
        conv1_weights = model.layer1[0].weight.data.numpy() # cnn first l
ayer weights (tensor values)
        plot_kernels(conv1_weights, 'first layer grey', greyscale = 1)

        conv2_weights = model.layer2[0].weight.data.numpy() # cnn second
layer weights (tensor values)
        plot_kernels(conv2_weights, 'second layer grey', greyscale = 1)

        plot_kernels(conv1_weights,'first layer', heatmap = 1)
        plot_kernels(conv2_weights,'second layer', heatmap = 1)
```

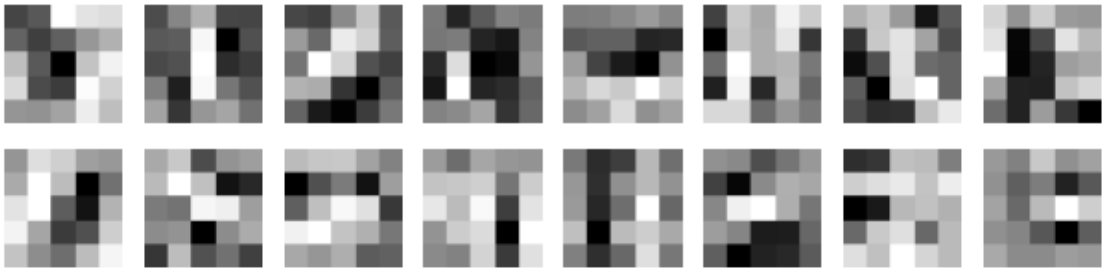


FIG.5 – GREYSCALE OF THE 5X5 FILTERS OF THE 16 CHANNELS OF THE FIRST LAYER

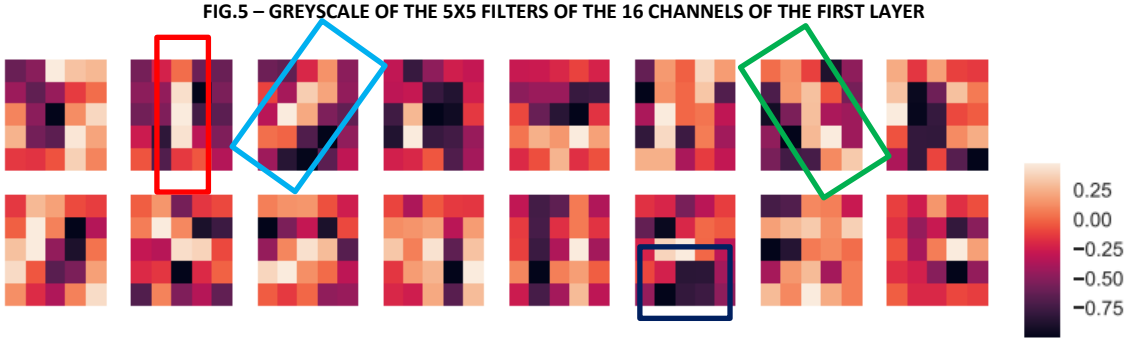



FIG.6 - HEATMAP OF THE 5X5 FILTERS OF THE 16 CHANNELS OF THE FIRST LAYER

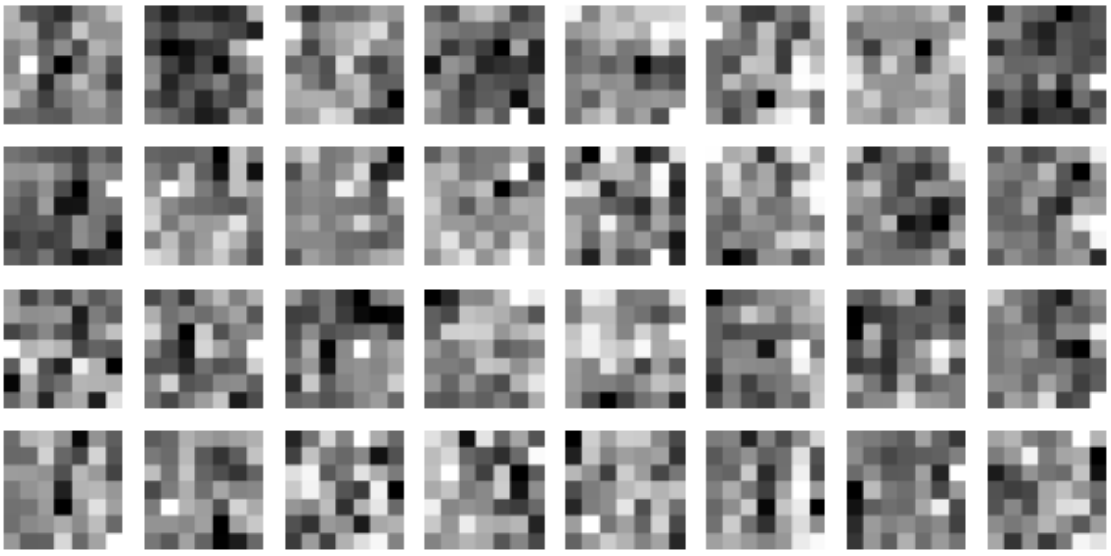


FIG.7 - GREYSCALE OF THE 7X7 FILTERS OF THE 32 CHANNELS OF THE SECOND LAYER

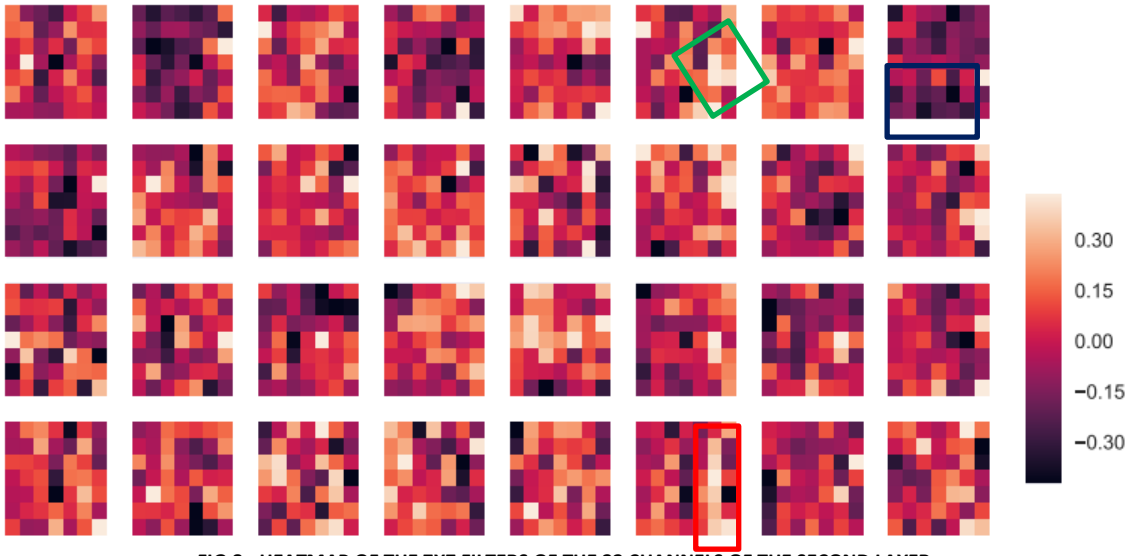


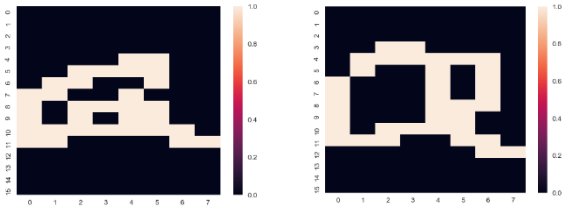

FIG.8 - HEATMAP OF THE 7X7 FILTERS OF THE 32 CHANNELS OF THE SECOND LAYER

Theoretically, on the one hand, first layer filters will detect mostly horizontal, vertical and diagonal lines. In other words, they are mostly used for detecting edges in an image. For example, the CNN could identify edges by learning a filter that acts as an edge detector.
On the other hand, second layer filters will detect aggregations of the first layer filters. In other words, they try to give more information than the previous layer. For instance, they might detect corners.

Bear in mind the heatmap scale, lighter colors correspond to a positive weight in the kernel.
In Fig.4, we can observe diagonal lines highlighted by the light blue and green rectangles. As well as a vertical line highlighted by the red rectangle.
In Fig.8, we can observe some aggregations of the first layers. For example, the same shape of the red and green rectangle in Fig.4 are like the lines in the Fig.8 green and

red rectangles. Besides that, we also have some nearby weights that have low values in correspondent positions of the weights, like is shown with the dark blue rectangles.
Overall, regarding the Fig.4 to Fig.8, it may be difficult to decipher what these filters are identifying because the images have a very low resolution (16x8 pixels). However, their convolutions on the input character images resulted in a 90.12% accuracy in the test set.

We would also highlight the fact that some of the input letters are very weird, which will harm even more the deciphering of these weights. Check Fig. 9, for example, that represent two samples of the letter *a*.



(A) (B)

**FIG.9 – (A): LETTER A; (B): LETTER A**

4.

In this task, we kept the same hyperparameters of Question 1.2:

1. Optimizer – Adam
2. Learning rate = 0.001

We did several tests with higher number of convolution layers but keeping only 2 max pooling in the first and last layer since we already have 32 channels with **4x2 pixels before the output layer.**
So, we kept adding layers with convolution (same number of channels and kernel size) and ReLU between the first and last layer.

**Constructor of the CNN model of trial 1 (3 layers)**

```
class CNN(nn.Module):

    def __init__(self, n_classes, **kwargs):
        super(CNN, self).__init__()

        self.layer1 = nn.Sequential(
            # in_ch = 1 (greyscale) ; out_ch = 16 ; kernel size = 5 = F ;
 padding = (F-1)/2 = 2 ; stride = 1
            nn.Conv2d(1 , 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
```

```
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU()
        )
        self.layer3 = nn.Sequential(
            # in_ch = 16 (previous layer) ; out_ch = 32 ; kernel size = 7
= F ; padding = (F-1)/2 = 3 ; stride = 1
            nn.Conv2d(16, 32, kernel_size = 7, stride = 1, padding = 3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.output_layer = nn.Linear(4 * 2 * 32, n_classes)
```

**Constructor of the CNN model of trial 2 (4 layers)**

```
class CNN(nn.Module):

    def __init__(self, n_classes, **kwargs):
        super(CNN, self).__init__()

        self.layer1 = nn.Sequential(
            # in_ch = 1 (greyscale) ; out_ch = 16 ; kernel size = 5 = F ;
 padding = (F-1)/2 = 2 ; stride = 1
            nn.Conv2d(1 , 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU()
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU()
        )

        self.layer4 = nn.Sequential(
            # in_ch = 16 (previous layer) ; out_ch = 32 ; kernel size = 7
= F ; padding = (F-1)/2 = 3 ; stride = 1
            nn.Conv2d(16, 32, kernel_size = 7, stride = 1, padding = 3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.output_layer = nn.Linear(4 * 2 * 32, n_classes)
```

## Constructor of the CNN model of trial 3 (5 layers)

```
class CNN(nn.Module):

    def __init__(self, n_classes, **kwargs):
        super(CNN, self).__init__()

        self.layer1 = nn.Sequential(
            # in_ch = 1 (greyscale) ; out_ch = 16 ; kernel size = 5 = F ;
 padding = (F-1)/2 = 2 ; stride = 1
            nn.Conv2d(1 , 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU()
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU()
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU()
        )
        self.layer5 = nn.Sequential(
            # in_ch = 16 (previous layer) ; out_ch = 32 ; kernel size = 7
 = F ; padding = (F-1)/2 = 3 ; stride = 1
            nn.Conv2d(16, 32, kernel_size = 7, stride = 1, padding = 3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )
        self.output_layer = nn.Linear(4 * 2 * 32, n_classes)
```

TABLE 1 – TEST SET ACCURACY WITH DIFFERENT ARCHITECTURES

| Trial | Number of layers | Test accuracy |
|---|---|---|
| 0 | 2 | 90.12% (Q1.2.) |
| 1 | 3 | 90.53% |
| **2** | **4** | **91.35%** |
| 3 | 5 | 90.88% |

The architecture with higher **test accuracy** was the one in trial 2 with 4 layers by achieving **91.35%** accuracy.

The train loss and validation accuracy plot are like the ones in Fig.2 and Fig.3, so we won't append them to the report. However, the plots for all the trials can be observed in the directory **Q1.4. results**.

# QUESTION 2

1.

```
class BILSTM(nn.Module):

    def __init__(self, input_size, output_size, batch_size = 1, n_layers
= 1, **kwargs):
        super(BILSTM, self).__init__()

        self.input_size = self.hidden_size = input_size # 128
        self.output_size = output_size # 26
        self.n_layers = n_layers # 1
        self.batch_size = batch_size # 1

        self.layer1 = nn.Sequential(
            nn.Linear(self.input_size, self.input_size),
            nn.Tanh()
        )

        self.layer2 = nn.LSTM(input_size=self.input_size, hidden_size=sel
f.hidden_size, bidirectional=True)

        self.output_layer = nn.Linear(self.hidden_size * 2, self.output_s
ize)

        self.hidden = self.init_hidden()

    def forward(self, x, **kwargs):
        out = self.layer1(x)

        out = out.view(out.shape[0], self.batch_size, out.shape[1])
        out, lstm_hidden = self.layer2(out, self.hidden)

        out = self.output_layer(out)
        return out

    def init_hidden(self):
        h0 = torch.zeros(self.n_layers * 2, self.batch_size, self.hidden_
size)
```

```
        c0 = torch.zeros(self.n_layers * 2, self.batch_size, self.hidden_
size)

        return (Variable(h0), Variable(c0))
```

```
def train_epoch(model, X_train, y_train, optimizer, criterion):
    loss_value = 0

    for xseq, yseq in zip(X_train, y_train):
        optimizer.zero_grad()

        outputs = model(xseq)
        outputs = outputs.view(outputs.shape[0], outputs.shape[2]) # [L x
 26]
        loss = criterion(outputs, yseq)
        loss_value += loss.item()

        # backprop
        loss.backward()
        optimizer.step()

    return loss_value

def predict(model, X):
    """X (n_examples x n_features)"""

    scores = model(X)
    predicted_labels = scores.argmax(dim=-1)
    return predicted_labels

def evaluate(model, X, y, decision=0):
    """Evaluate model on data."""
    n_correct = 0
    n_total = 0

    for xseq, yseq in zip(X, y):
        model.eval()

        yseq_hat = predict(model, xseq)
        if decision == 0: # #mistakes = # different characters between ys
eq and yseq_hat
            n_correct += (sum([yseq[t] == yseq_hat[t] for t in range(len(
yseq))])).item()
            n_total += len(yseq)

        else: # at least 1 character different between yseq and yseq_hat
            n_correct += not(False in torch.eq(yseq.T,yseq_hat.T))
            n_total += 1
```

```
        model.train()
    return n_correct / n_total
```

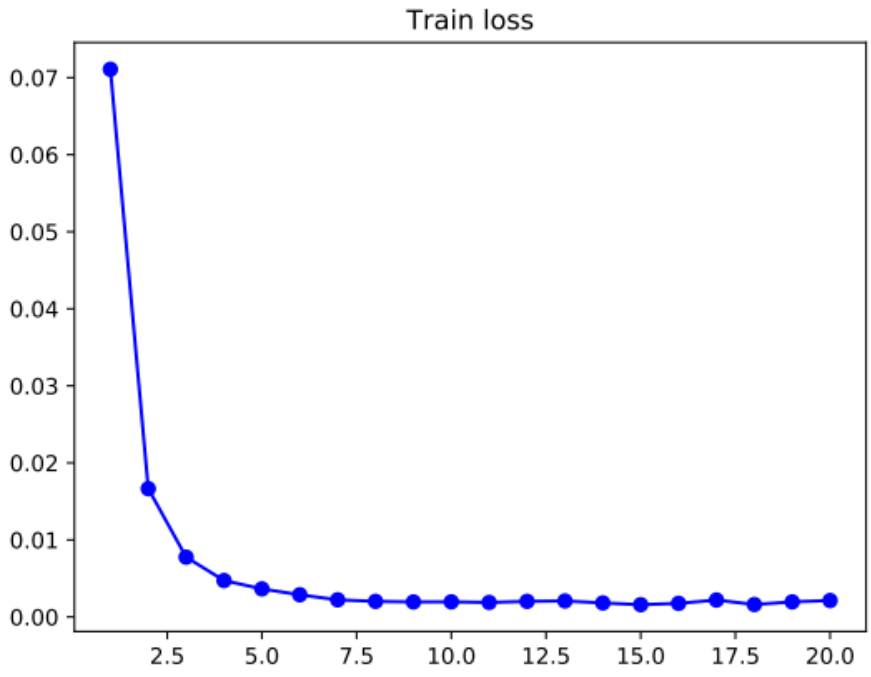



FIG.10 – TRAIN LOSS OF THE BILSTM WITH 20 EPOCHS

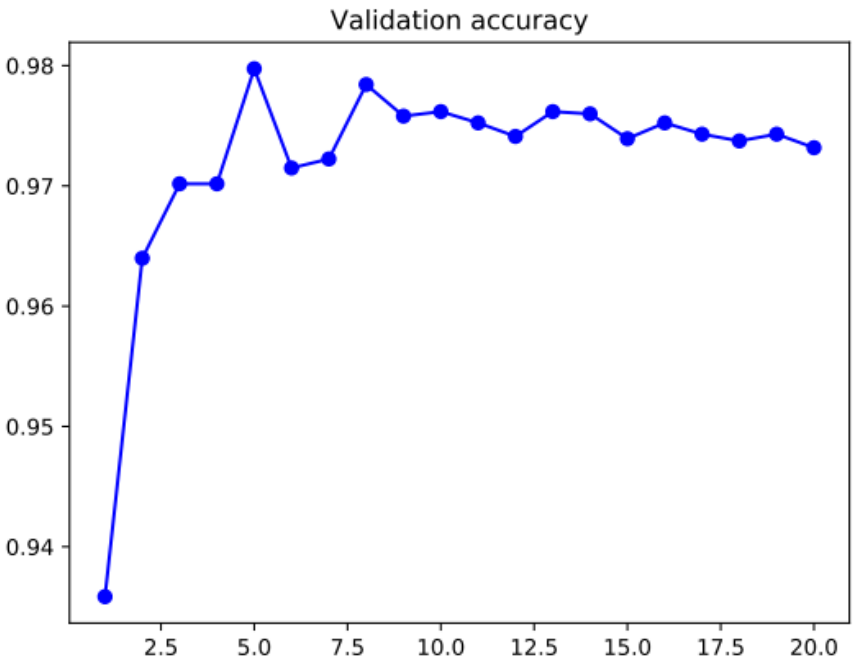



FIG.11 – VALIDATION ACCURACY OF THE BILSTM WITH 20 EPOCHS

It is important to notice that we measured the test accuracy by considering as errors the characters that were not equal in the predicted and correct sequence.
The **final test accuracy was 97.01%.**

2.

It is possible to combine these two models. We should use the BILSTM to compute scores for the emissions instead of using the feature-based linear model. For the transitions, we can have an indicator features that looks at the that bigrams or compute scores from the same recurrent layer. We also need to replace the softmax output layer by a CRF output layer. In training time, we plug the *Forward-backward algorithm* in the gradient backpropagation and in run time we use the *Viterbi algorithm* to predict the most likely sequence of tags.

BILSTMs are extremely good at identifying patterns from data, but still, each prediction is done in isolation and not as part of a sequence. In other words, BILSTMs benefit from recognizing patterns in the surrounding input features, while structured learning models like CRF benefit from the knowledge about neighboring predictions.
On the one hand, BILSTMs lack output structure because some transitions might not even be possible and still, they might be predicted. While CRFs can prevent this. On the other hand, BILSTMs work better to grasp patters of the input data. Thus, in NLP problems, for example, they can make predictions of words that have context in the whole sequence.

3.

```
class BILSTM(nn.Module):

    def __init__(self, input_size, output_size, batch_size = 1, n_layers
= 1, **kwargs):
        super(BILSTM, self).__init__()

        self.input_size = self.hidden_size = input_size # 128
        self.output_size = output_size # 26
        self.n_layers = n_layers # 1
        self.batch_size = batch_size # 1

        self.layer1 = nn.Sequential(
            # in_ch = 1 (greyscale) ; out_ch = 16 ; kernel size = 5 = F ;
 padding = (F-1)/2 = 2 ; stride = 1
            nn.Conv2d(1 , 16, kernel_size = 5, stride = 1, padding = 2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2)
        )

        self.layer2 = nn.Sequential(
```

```
            nn.Linear(8 * 4 * 16, self.input_size),
            nn.Tanh()
        )

        self.layer3 = nn.LSTM(input_size=self.input_size, hidden_size=sel
f.hidden_size, bidirectional=True)

        self.output_layer = nn.Linear(self.hidden_size * 2, self.output_s
ize)

        self.hidden = self.init_hidden()

    def forward(self, x, **kwargs):
        x = x.view(x.shape[0],1,16,8) # [L, 1, 16, 8]
        out = self.layer1(x)

        out = out.reshape(out.size(0), -1)
        out = self.layer2(out)

        # Dimensions of output of neural network is (seq_len, batch , hid
den_dim)
        out = out.view(out.shape[0], self.batch_size, out.shape[1])
        out, lstm_hidden = self.layer3(out, self.hidden)

        out = self.output_layer(out)
        return out

    def init_hidden(self):
        h0 = torch.zeros(self.n_layers * 2, self.batch_size, self.hidden_
size)
        c0 = torch.zeros(self.n_layers * 2, self.batch_size, self.hidden_
size)

        return (Variable(h0), Variable(c0))
```

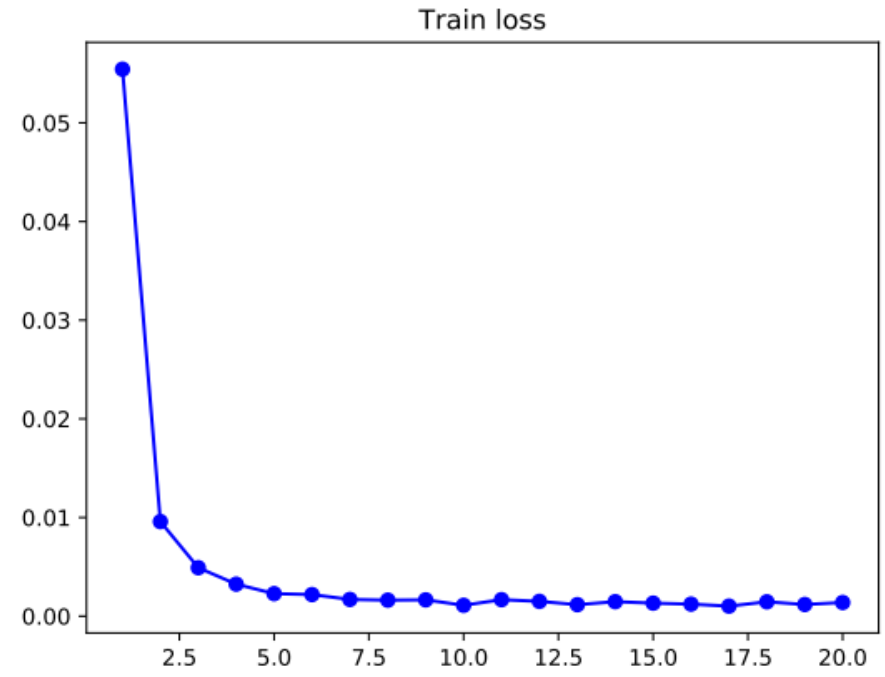



FIG.12 – TRAIN LOSS WITH THE BILSTM WITH CONVOLUTIONAL LAYER AND 20 EPOCHS

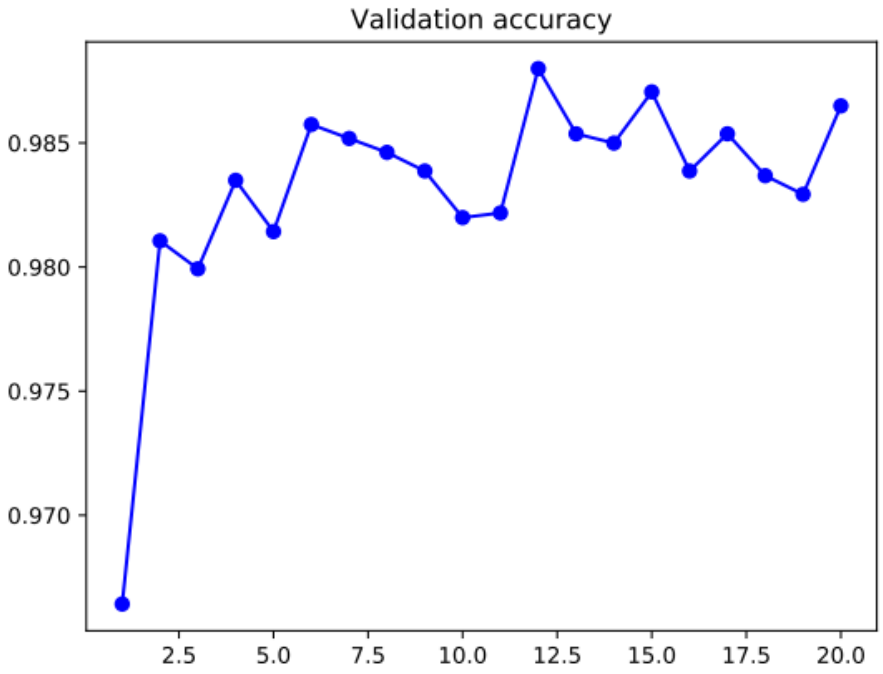



FIG.13 – VALIDATION ACCURACY WITH BILSTM AND CONVOLUTIONAL LAYER WITH 20 EPOCHS

The **final test accuracy was 98.29%.**

# REFERENCES

[1] - Deep Learning. *Ian Goodfellow and Yoshua Bengio and Aaron Courville. 2016*