TÉCNICO LISBOA

DEEP STRUCTURED LEARNING

HOMEWORK 2

André Godinho Number: 84006

QUESTION 1

1.

```python
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
    """
    optimizer.zero_grad()

    outputs = model(X)
    loss = criterion(outputs, y)
    loss_value = loss.item()
    loss.backward()
    optimizer.step()
    return loss_value
```

```python
class LogisticRegression(nn.Module):

    def __init__(self, n_classes, n_features, **kwargs):
        """
        n_classes (int)
        n_features (int)
        """
        super(LogisticRegression, self).__init__()
        self.linear_layer = nn.Linear(n_features, n_classes)

    def forward(self, x, **kwargs):
        """
        x (batch_size x n_features): a batch of training examples
        """
        return self.linear_layer(x)
```

The best result produced a test accuracy of **75%**. The hyperparameters were tuned by doing *Grid Search* and can be found on table 1. Furthermore, the training loss and validation accuracy can be observed in the Fig. 1 and Fig. 2.

TABLE 1 – BEST HYPERPARAMETERS TUNING FOR LOGISTIC REGRESSION

Hyperparameter	Learning rate	Regularization constant	Optimizer
Value	0.01	0	SGD

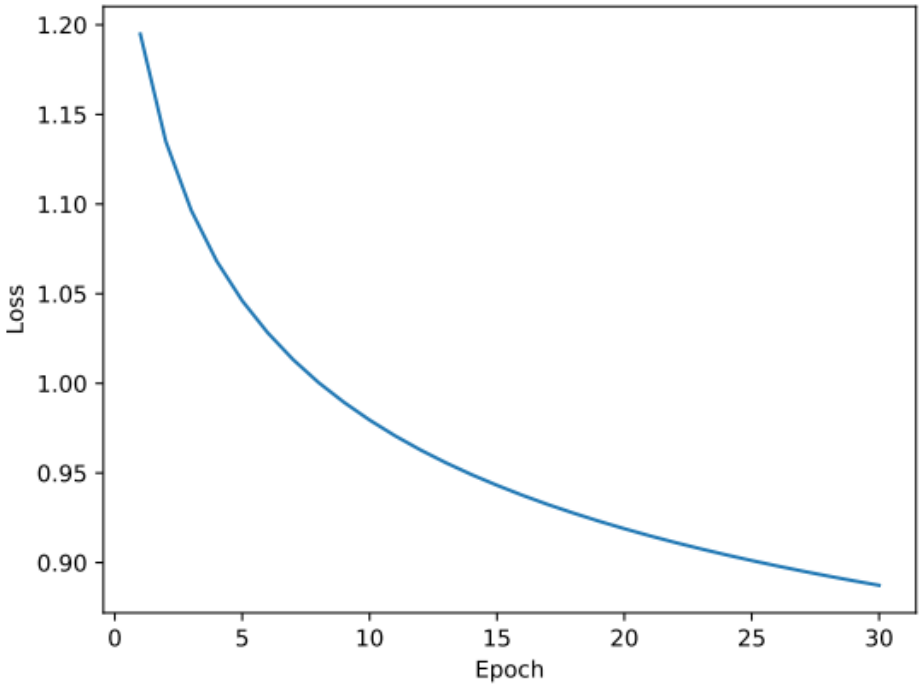

FIG.1 – TRAINING LOSS OF LOGISTIC REGRESSION WITH THE TUNED HYPERPARAMETERS

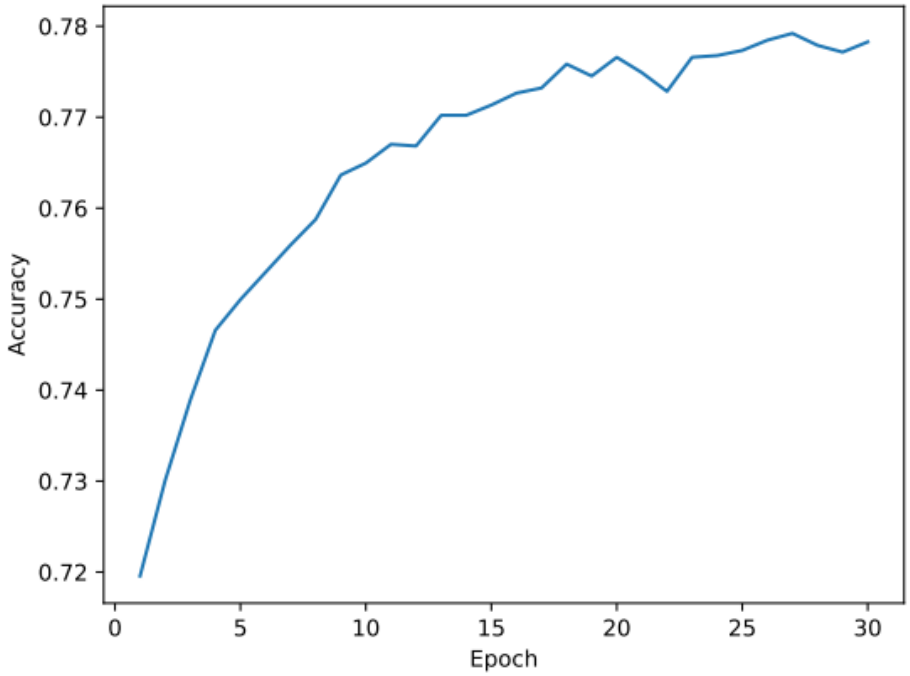

FIG.2 – VALDIATION ACCURACY OF LOGISTIC REGRESSION WITH THE TUNED HYPERPARAMETERS

The training loss decreased smoothly, while the accuracy test tended to increase but oscillated a bit.

All these results can be checked in the folder **Q1.1. results.**

2.

```
class FeedforwardNetwork(nn.Module):
    def __init__(
            self, n_classes, n_features, hidden_size, layers,
            activation_type, dropout, **kwargs):
        """
        n_classes (int)
        n_features (int)
        hidden_size (int)
        layers (int)
        activation_type (str)
        dropout (float): dropout probability
        """
        super(FeedforwardNetwork, self).__init__()

        # hidden layers
        linear_layer1 = nn.Linear(n_features, hidden_size)
        # output layer
        output_layer = nn.Linear(hidden_size, n_classes)

        if activation_type == 'tanh':
            self.feedforward = nn.Sequential(
                linear_layer1,
                nn.Tanh(),
                nn.Dropout(dropout),
                output_layer)

        elif activation_type == 'relu':
            self.feedforward = nn.Sequential(
                linear_layer1,
                nn.ReLU(),
                nn.Dropout(dropout),
                output_layer)
```

```
def forward(self, x, **kwargs):
        """
        x (batch_size x n_features): a batch of training examples
        """
        return self.feedforward(x)
```

The best result produced a test accuracy of **88.5%.** The hyperparameters were tuned by doing *Grid Search* and can be found on table 2. Furthermore, the training loss and validation accuracy can be observed in the Fig. 3 and Fig. 4.

TABLE 2 – BEST HYPERPARAMETER TUNING FOR FEEDFORWARD NEURAL NETWORK

Hyperparameter	Learning rate	Hidden size	Dropout probability	Activation function	Optimizer
Value	0.001	200	0.3	ReLU	Adam

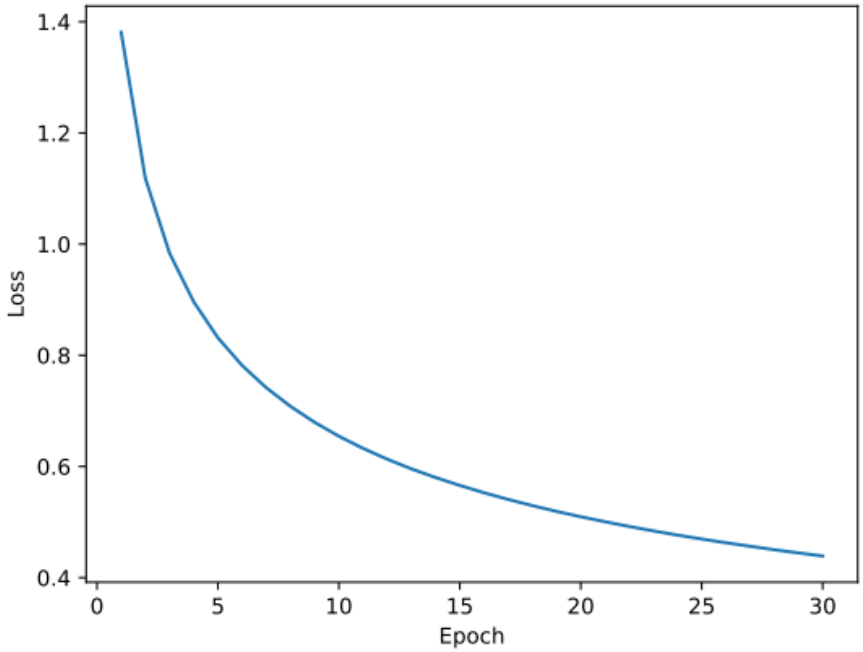

FIG.3 - TRAINING LOSS OF FEEDFORWARD NEURAL NETWORK WITH THE TUNED HYPERPARAMETERS

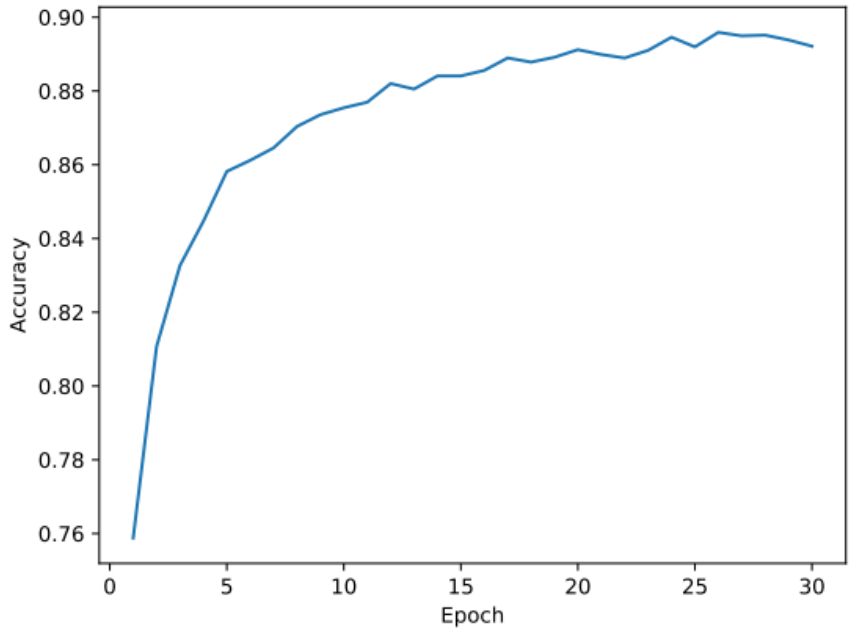

FIG.4 - VALDIATION ACCURACY OF FEEDFORWARD NEURAL NETWORK WITH THE TUNED HYPERPARAMETERS

Like the Logistic regression plots, the training loss decreased smoothly, while the accuracy test tended to increase but oscillated a bit. However, in the feedforward neural network the obtained accuracy was much higher.

All the grid search results can be observed in the folder **Q1.2. results.**

3.

Code of a feedforward neural network with 2 layers

```
class FeedforwardNetwork(nn.Module):
    def __init__(
            self, n_classes, n_features, hidden_size, layers,
            activation_type, dropout, **kwargs):
        """
        n_classes (int)
        n_features (int)
        hidden_size (int)
        layers (int)
        activation_type (str)
        dropout (float): dropout probability
        """
        super(FeedforwardNetwork, self).__init__()

        self.hidden = nn.ModuleList()

        # hidden layers
        linear_layer1 = nn.Linear(n_features, hidden_size)
        linear_layer2 = nn.Linear(hidden_size, hidden_size)
        # output layer
        output_layer = nn.Linear(hidden_size, n_classes)

        if activation_type == 'tanh':
            self.feedforward = nn.Sequential(
                linear_layer1,
                nn.Tanh(),
                nn.Dropout(dropout),
                linear_layer2,
                nn.Tanh(),
                nn.Dropout(dropout),
                output_layer)

        elif activation_type == 'relu':
            self.feedforward = nn.Sequential(
                linear_layer1,
                nn.ReLU(),
                nn.Dropout(dropout),
                linear_layer2,
                nn.Tanh(),
```

```
                nn.Dropout(dropout),
                output_layer)
```

Code of a feedforward neural network with 3 layers

```
class FeedforwardNetwork(nn.Module):
    def __init__(
            self, n_classes, n_features, hidden_size, layers,
            activation_type, dropout, **kwargs):
        """
        n_classes (int)
        n_features (int)
        hidden_size (int)
        layers (int)
        activation_type (str)
        dropout (float): dropout probability
        """
        super(FeedforwardNetwork, self).__init__()

        self.hidden = nn.ModuleList()

        # hidden layers
        linear_layer1 = nn.Linear(n_features, hidden_size)
        linear_layer2 = nn.Linear(hidden_size, hidden_size)
        linear_layer3 = nn.Linear(hidden_size, hidden_size)
        # output layer
        output_layer = nn.Linear(hidden_size, n_classes)

        if activation_type == 'tanh':
            self.feedforward = nn.Sequential(
                linear_layer1,
                nn.Tanh(),
                nn.Dropout(dropout),
                linear_layer2,
                nn.Tanh(),
                nn.Dropout(dropout),
                linear_layer3,
                nn.Tanh(),
                nn.Dropout(dropout),
                output_layer)

        elif activation_type == 'relu':
            self.feedforward = nn.Sequential(
                linear_layer1,
                nn.ReLU(),
                nn.Dropout(dropout),
                linear_layer2,
                nn.ReLU(),
```

```
            nn.Dropout(dropout),
            linear_layer3,
            nn.ReLU(),
            nn.Dropout(dropout),
            output_layer)
```

The best test accuracy was obtained for the feedforward neural network with 2 layers. The result was **83.2%** for the given hyperparameters. Furthermore, the training loss and validation accuracy can be observed in the Fig. 5 and Fig. 6.

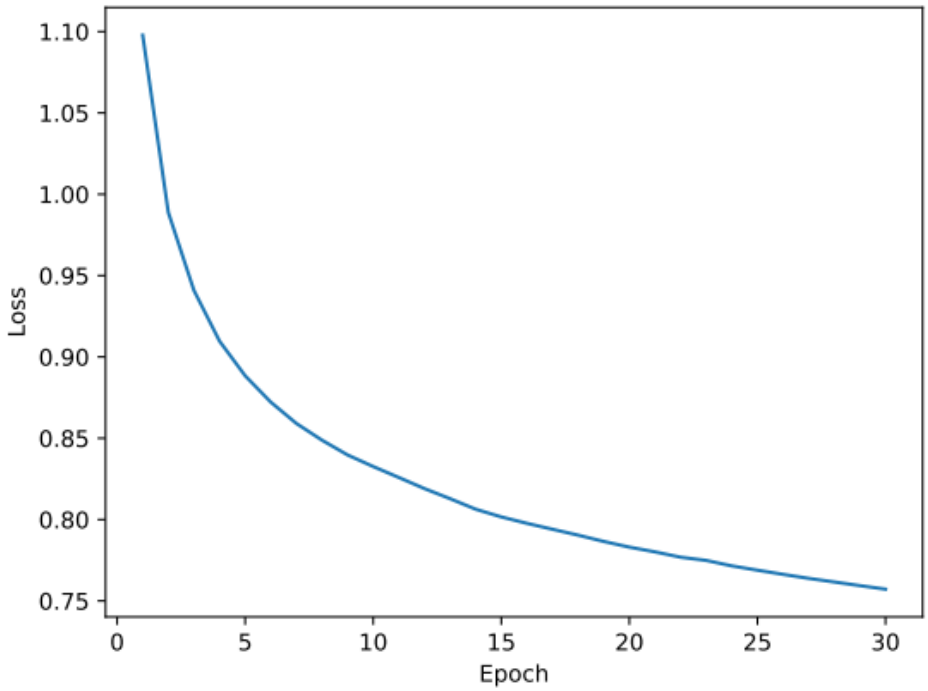

FIG.5 - TRAINING LOSS OF FEEDFORWARD NEURAL NETWORK WITH THE GIVEN HYPERPARAMETERS

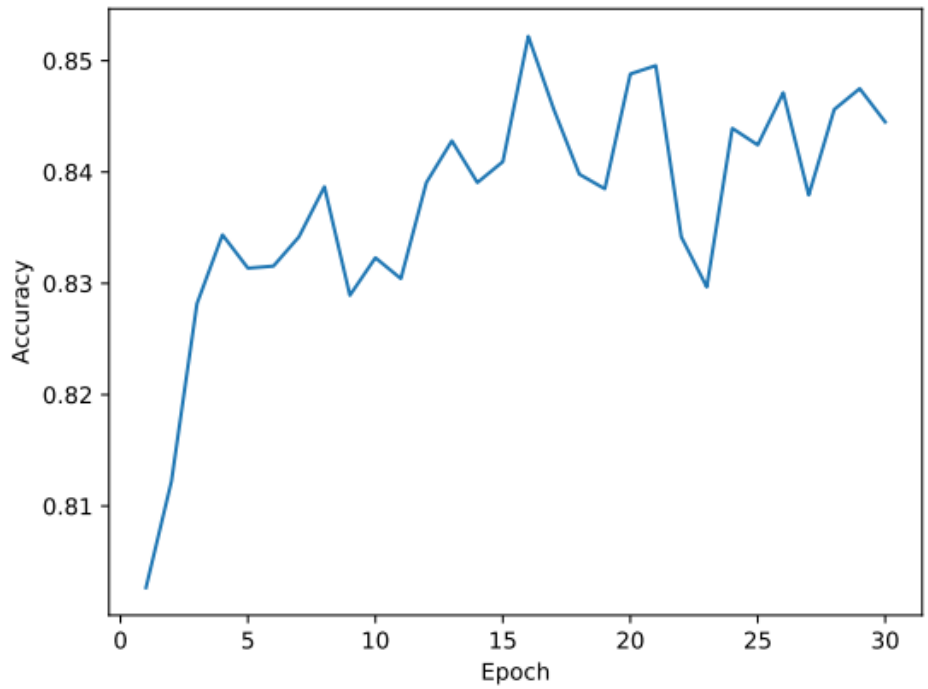

FIG.6 - VALDIATION ACCURACY OF FEEDFORWARD NEURAL NETWORK WITH THE GIVEN HYPERPARAMETERS

For both the feedforward neural networks (with 2 and 3 layers) with the given hyperparameters the validation accuracy oscillated more throughout the epochs than the feedforward neural network with 1 layer and the best hyperparameters. We can observe this by comparing Fig.4 and Fig. 6. The train loss had a similar behavior as before, decreasing smoothly throughout the epochs (Fig.3 and Fig.5).

All the results can be observed in the folder **Q1.3. results.**

QUESTION 2

1.(A)

The objective in this task is to predict the most likely weather for the past week. Since this corresponds to a sequence of the hidden states, we should use the *Viterbi algorithm*, which is a dynamic programming algorithm that finds the most likely sequence of hidden states. Its implementation is presented below.

```
def viterbi(initial_scores, transition_scores, final_scores, emission_sco
res):
    """Computes the viterbi trellis for a given sequence.
    Receives:
    - Initial scores: (num_states) array
    - Transition scores: (length-1, num_states, num_states) array
    - Final scores: (num_states) array
    - Emission scores: (length, num_states) array.
    Your solution should return:
    - best_path: (length) array containing the most likely state sequence
    """

    backptr = []
    best_path = []
    scores = []

    scores.append(initial_scores + emission_scores[0])

    # calculate scores and backpointers
    for i in range(1,emission_scores.shape[0]):
        max_cur = np.max(transition_scores[i-1] + scores[i-1], axis = 1)
        arg_max_cur = np.argmax(transition_scores[i-1] + scores[i-
1], axis = 1)

        scores.append(emission_scores[i] + max_cur)
        backptr.append(arg_max_cur)

    # backward

    # get last state
    best_path.insert(0, int(np.argmax(final_scores + scores[-1])))
```

```
    # get previous states
    for item in reversed(backptr):
        best_path.insert(0, item[best_path[0]])

    return best_path
```

```
Sequence given by Viterbi: Rainy -> Rainy -> Rainy -> Sunny -> Sunny -> Sunny -> Sunny
Sequence given by posterior: Rainy -> Rainy -> Rainy -> Windy -> Sunny -> Sunny -> Sunny
```

FIG.7 – OUTPUT OF THE PROGRAM HW2-Q2.PY THAT CONTAINS THE MOST LIKELY WEATHER FOR THE LAST WEEK ACCORDING TO THE VITORBI ALGORITHM

1.(B)

TABLE 3 – SEQUENCE GIVEN BY FORWARD-BACKWARD ALGORITHM

Day of the week	Bet
Monday, Oct 8	Rainy
Tuesday, Oct 9	Rainy
Wednesday, Oct 10	Rainy
Thursday, Oct 11	Windy
Friday, Oct 12	Sunny
Saturday, Oct 13	Sunny
Sunday, Oct 14	Sunny

On the one hand we have *Viterbi algorithm* that can obtain the most likely sequence; however, this algorithm is greedy because it makes decisions according to a maximum probability in the current state without looking ahead. Once taking a decision, even though it was a bad call it can't go back. On the other hand, we have *forward-backward algorithm* that minimizes the expected cost. In other words, it minimizes the errors of predictions in the whole sequence.
Thus, to maximize the amount of money gained in the bet, we should use the sequence given by the *forward-backward algorithm*, which is the one expressed in the Table 3.

2.

John surfing two days straight, corresponds to two observed variables of activities being *Surf* one after the other. *Surf* is not a hidden variable.
In hidden Markov models we do not deal with probabilities like $P(x_i|x_{i-1})$. Thus, we cannot accommodate this information directly into our model.
An idea to try adding this extra piece of knowledge in a Markov model might be giving a higher weight to the probability $P(x_i = Surf \mid y_i = Sunny)$ or $P(x_i = Surf \mid y_i = Windy)$ so that if John surfed in one day that was *Sunny* or *Windy*, and the next day would also be *Sunny* or *Windy*, then John would likely do surf. On the contrary we would give a smaller weight to $P(x_i = Surf|y_i = Rainy)$ which would decrease the probability of John surfing if the weather was *Rainy*. So, John would be likely to do *Surf* if the weather was *Windy* or *Sunny* and very unlikely to *Surf* if it rained.

QUESTION 3

All the results can be observed in the folder **Q3 results.**

1.

In this task we implemented the Structured Perceptron. The code is a bit long, therefore we will some explanation throughout it.

```
class StructuredPerceptron(LinearSequenceModel):
    def update_weight(self, xseq, yseq, **kwargs):
        """
        xseq (list): list of np.arrays, each of size (n_feat)
        yseq (list): list of class labels (int)
        """
        L = len(yseq)

        phi_start = np.zeros(self.n_classes)
        phi_start[yseq[0]] = 1
        initial_scores = np.multiply(self.W_start, phi_start)
```

In these lines of code, we define the initial scores to enter in the *Viterbi* algorithm.

```
        phi_stop = np.zeros(self.n_classes)
        phi_stop[yseq[-1]] = 1
        final_scores = np.multiply(self.W_stop, phi_stop
```

Now, we do the same for the final scores.

```
        phi_transition = np.zeros((L-1, self.n_classes, self.n_classes))
        s_t = np.zeros((L-1,self.n_classes,self.n_classes))
        for i in range(L-1):
            phi_transition[i][yseq[i+1],yseq[i]] = 1
            s_t[i] = np.dot(self.W_bigrams, phi_transition[i])
```

Next, we define the $\varphi^{(t)}\left((x,i),(y^{'},y)\right), i = 1 \dots L-1$, that will be needed to calculate each $s^{(t)}_{y^{'},y,i}$.

```
        s_u = np.dot(np.array(xseq), self.W_unigrams.T)

        yseq_hat = viterbi(initial_scores, s_t, final_scores, s_u)
```

Since $\varphi^{(u)}\big((x,i),y\big) = xseq[i]$, we can obtain $s^{(u)}_{y,i}$ efficiently by doing that matrix product. Now, we everything we need to obtain the predicted sequence given by the *Viterbi* algorithm.

```
        if yseq_hat != yseq:
            for t in range(L):
                if yseq_hat[t] != yseq[t]:
                    self.W_unigrams[yseq_hat[t]] -= xseq[t]
                    self.W_unigrams[yseq[t]] += xseq[t]
```

```
                    if t == 0:
                        self.W_start[yseq_hat[t]] -= 1
                        self.W_start[yseq[t]] += 1

                    else:
                        self.W_bigrams[yseq_hat[t],yseq_hat[t-1]] -
= phi_transition[t-1][yseq_hat[t],yseq_hat[t-1]]
                        self.W_bigrams[yseq[t],yseq[t-
1]] += phi_transition[t-1][yseq[t], yseq[t-1]]

                        if t == L-1:
                            self.W_stop[yseq[t]] += 1
                            self.W_stop[yseq_hat[t]] -= 1
            return 1
        return 0
```

Finally, without forgetting the start and stop symbols, we just need to update the weights if there is an error in the prediction of each character according to the slide 74 of the lecture 5 with all the previous defined "φ".
The accuracy results are shown on the next page.

FIG.8 – ACCURACY OF THE STRUCTURED PERCEPTRON

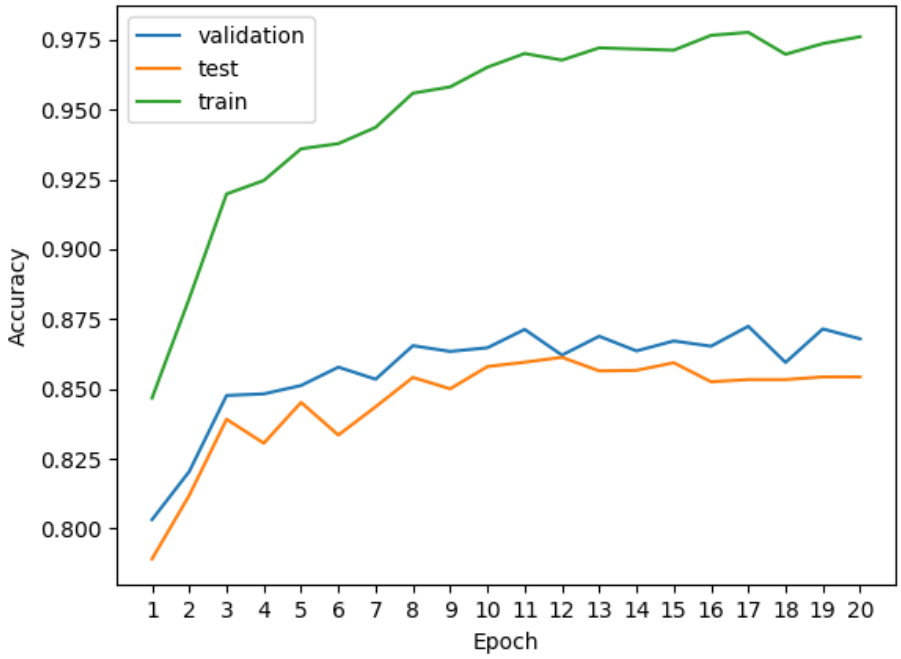

FIG.9 – ACCURACY OF THE PERCEPTRON WITHOUT USING ANY STRUCTURE (HOMEWORK 1)

TABLE 4 – TEST ACCURACY OF PERCEPTRON (HOMEWORK 2) WITH AND WITHOUT STRUCTURE (HOMEWORK1)

	Test set
Accuracy w/ structure	83.8%
Accuracy w/o structure	86,8%
Difference	3%

In the homework 1 we predicted characters using Perceptron. In this homework we predicted sequence of characters using structure perceptron. Even though we are comparing results of different predictions, the results were quite impressive since we got a **small difference of 3% of accuracy** (after 20 epochs) in the test set. Bear in mind that we are not predicting a character, but yes, a sequence of characters, which is a more demanding task.

2.

```
class CRF(LinearSequenceModel):
    def update_weight(self, xseq, yseq, l2_decay=None, learning_rate=None
):
        """
        xseq (list): list of np.arrays, each of size (n_feat)
        yseq (list): list of class labels (int)
        l2_decay (float): l2 regularization constant
        learning_rate (float)
        """
        assert l2_decay is not None and learning_rate is not None

        L = len(yseq)

        phi_start = np.zeros(self.n_classes)
        phi_start[yseq[0]] = 1
```

```
        initial_scores = np.multiply(self.W_start, phi_start)
```

In these lines of code, we define the initial scores to enter in the *Forward-backward* algorithm.

```
        phi_stop = np.zeros(self.n_classes)
        phi_stop[yseq[-1]] = 1
        final_scores = np.multiply(self.W_stop, phi_stop)
```

Now, we do the same for the final scores.

```
        phi_transition = np.zeros((L-1, self.n_classes, self.n_classes))
        s_t = np.zeros((L-1,self.n_classes,self.n_classes))
        for t in range(L-1):
            phi_transition[t][yseq[t+1],yseq[t]] = 1
            s_t[t] = np.dot(self.W_bigrams, phi_transition[t])
```

Next, we define the $\varphi^{(t)}\left((x,i),(y^{'},y)\right), i = 1 \dots L-1$, that will be needed to calculate each $s^{(t)}_{y^{'},y,i}$.

```
s_u = np.dot(np.array(xseq), self.W_unigrams.T)

        emission_posteriors, transition_posteriors, log_likelihood = forw
ard_backward(initial_scores, s_t, final_scores, s_u)
        yseq_hat = list(emission_posteriors.argmax(1))
```

Since $\varphi^{(u)}((x,i),y)) = xseq[i]$, we can obtain $s^{(u)}_{y,i}$ efficiently by doing that matrix product. Now, we everything we need to obtain the predicted sequence given by the *Forward-backward* algorithm.

```
        if yseq_hat != yseq:
            for t in range(L):
                if yseq_hat[t] != yseq[t]:
                    self.W_unigrams[yseq[t]] += learning_rate*xseq[t]
                    u_gradient = np.dot(emission_posteriors[:, yseq_hat[t
]].T, xseq)
                    self.W_unigrams[yseq_hat[t]] -
= learning_rate * u_gradient

                    if t == 0:
                        self.W_start[yseq_hat[t]] -
= learning_rate * emission_posteriors[0, yseq_hat[t]]
                        self.W_start[yseq[t]] += learning_rate * emission
_posteriors[0, yseq[t]]

                    else:
```

```
                        self.W_bigrams[yseq[t],yseq[t-
1]] += learning_rate * phi_transition[t-1][yseq[t], yseq[t-1]]
                        t_gradient = np.dot(phi_transition[t-
1], transition_posteriors[t-1])
                        self.W_bigrams[yseq_hat[t],yseq_hat[t-1]] -
= learning_rate * t_gradient.sum()
                        if t == L-1:
                            self.W_stop[yseq[t]] += learning_rate * emiss
ion_posteriors[-1, yseq[t]]
                            self.W_stop[yseq_hat[t]] -
= learning_rate * emission_posteriors[-1, yseq_hat[t]]
            return 1
        return 0
```

Finally, without forgetting the start and stop symbols, we just need to update the weights if there is an error in the prediction of each character according to the slides 77 and 80 of the lecture 5 with all the previous defined "φ".
The accuracy results are shown below.

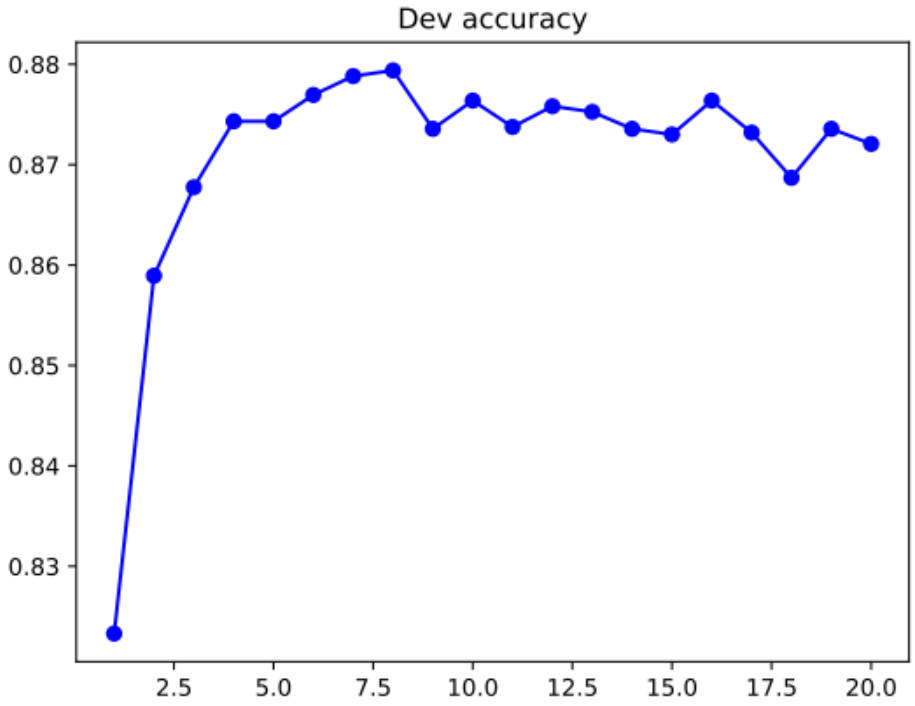

FIG.9 – ACCURACY OF THE CRF