TÉCNICO LISBOA

DEEP STRUCTURED LEARNING

HOMEWORK 4

André Godinho Number: 84006

ACKNOWLEDGEMENTS

For this assignment it was discussed some insights with Francisco Caldas and Bernardo Cortez. Also, some inspiration was taken from the **References** implementations.

QUESTION 1

1.

The target vocabularies were considered using the train set which has a higher number of word pairs. Considering the special symbols: *START, STOP* and *UNK* (we did not pad) there is a total of 50 symbols corresponding to the Arabic vocabulary (source) and 42 symbols corresponding to the English vocabulary (target).

2.

To implement this question, we toke inspiration from **[1]** and **[2]**. There are some considerations to be stated. There are 3 different .py files: *vanSeq2seq.py*, *vanClasses.py* and *readInput.py*. The first one has the whole pipeline for questions Q1. (b) and Q1. (c), the second has the classes implemented for the *Vanilla sequence-to-sequence model* and the third file has a class and functions necessary for reading the input files: *trainar-eng.txt*, *testar-eng.txt* and *validar-eng.txt*.

In this report we will only show for this question the classes implemented for the *Vanilla sequence-to-sequence model.* Please check the other *.py* files to check how the input is read, the conversion to word embeddings and the rest of the pipeline. We ran the whole pipeline with 20 epochs.

```
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size) # embbedin
g size = hidden size
        self.lstm = nn.LSTM(input_size=hidden_size, hidden_size=hidden_si
ze)

        self.hidden = self.init_hidden()

    def forward(self, src):

        # src = [src len, batch size]
```

```
        embedded = self.embedding(src) # [src len, batch size, # embbedin
g size = hidden size]

        output, (hidden, cell) = self.lstm(embedded)

        #outputs = [src len, batch size, hid dim = emb dim]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer
        return hidden, cell

    def init_hidden(self):
        h0 = torch.zeros(1, 1, self.hidden_size)
        c0 = torch.zeros(1, 1, self.hidden_size)

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

class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size, biggest_word):
        super().__init__()
        self.hidden_size = hidden_size # embbeding size = hidden size
        self.output_size = output_size
        self.biggest_word = biggest_word

        self.embedding = nn.Embedding(output_size, hidden_size)

        self.lstm = nn.LSTM(input_size=hidden_size, hidden_size=hidden_si
ze)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden, cell):

        #input = [batch size]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        input = input.unsqueeze(0)  #input = [1, batch size]

        embedded = self.embedding(input)
        output, (hidden, cell) = self.lstm(embedded, (hidden, cell))

        #output = [seq len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        prediction = self.out(output.squeeze(0))
```

```
        #prediction = [batch size, output dim]

        return prediction, hidden, cell

    def init_hidden(self):
        h0 = torch.zeros(1, 1, self.hidden_size)
        c0 = torch.zeros(1, 1, self.hidden_size)

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

For the *Seq2Seq* class take in mind that during training phase we decode using the *target length* and for evaluating phase we decode using the 2 * length of the biggest target or until a <EOS> is decoded. Also, we are using *teacher forcing*.

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use ground-
truth inputs 75% of the time

        batch_size = trg.shape[1]
        trg_vocab_size = self.decoder.output_size

        #tensor to store decoder outputs
        if self.training is True:
            trg_len = trg.shape[0]
            outputs = torch.zeros(trg_len, batch_size, trg_vocab_size)
        else:
            outputs = torch.zeros(2*self.decoder.biggest_word, batch_size
, trg_vocab_size)

        #last hidden state of the encoder is used as the initial hidden s
tate of the decoder
        hidden, cell = self.encoder(src)

        #first input to the decoder is the <sos> tokens
```

```
        input = trg[0,:]

        if self.training is True:
            loop_condition = trg_len
        else:
            loop_condition = 2*self.decoder.biggest_word

        for t in range(1, loop_condition):

            #insert input token embedding, previous hidden and previous c
ell states
            #receive output tensor (predictions) and new hidden and cell
states
            output, hidden, cell = self.decoder(input, hidden, cell)

            #place predictions in a tensor holding predictions for each t
oken
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            if self.training is False and top1.item() == EOS_token:
                break

            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        outputs = outputs[:t+1]
        return outputs
```

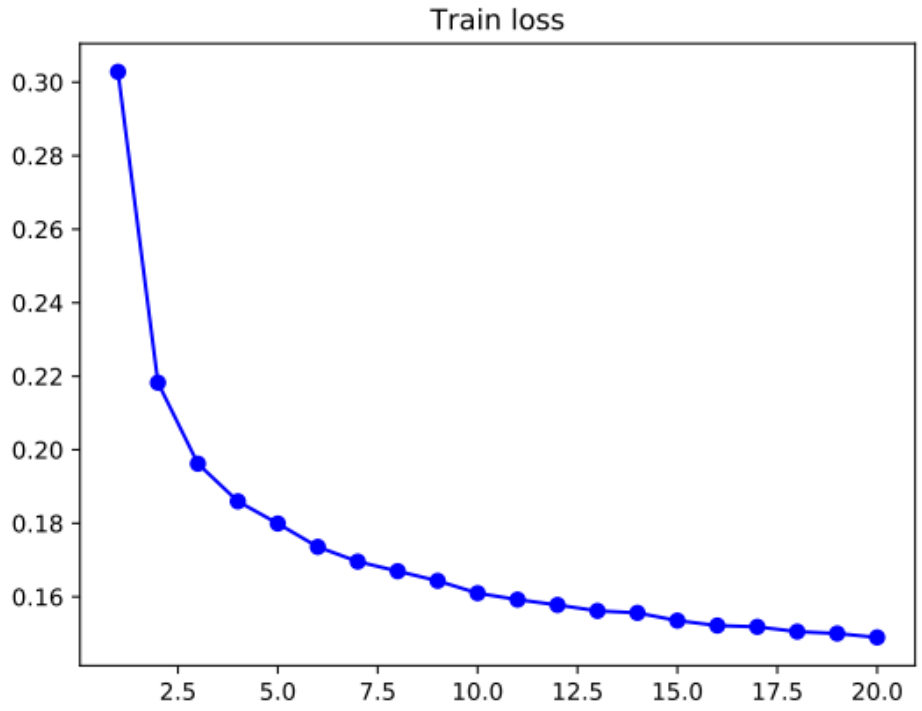

FIG.1 – VANILLA SEQUENCE-TO-SEQUENCE MODEL TRAIN LOSS

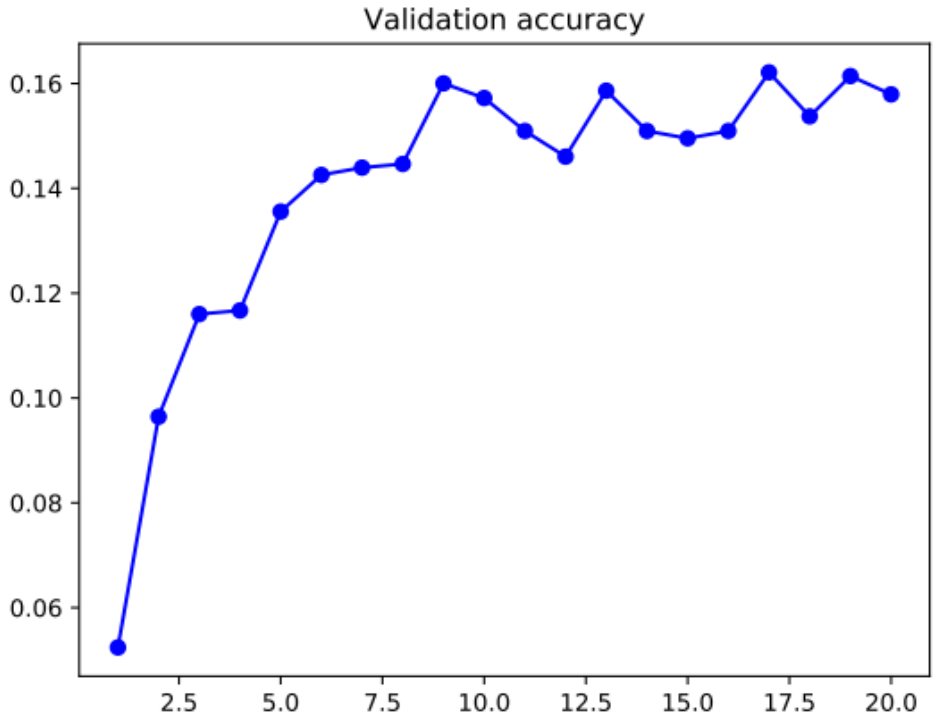

FIG.2 - VANILLA SEQUENCE-TO-SEQUENCE MODEL VALIDATION ACCURACY

The test accuracy obtained was 14.34%.

3.

The code for this question is activated with the flag variable *reverse_source* of the *readInput.py*. Please check the functions of this file.

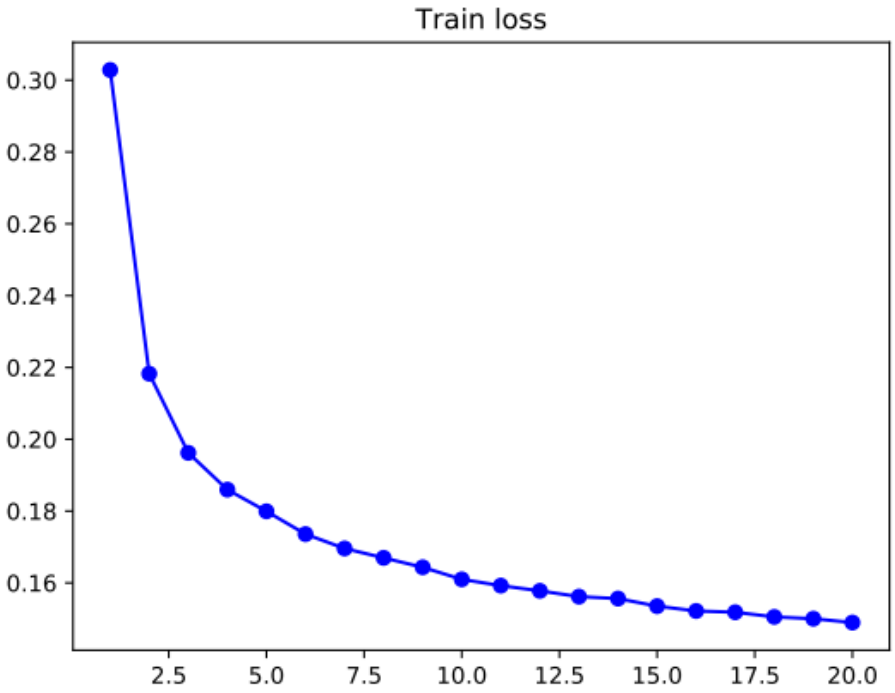

FIG.3 - VANILLA SEQUENCE-TO-SEQUENCE MODEL TRAIN LOSS REVERTING THE SOURCE STRING

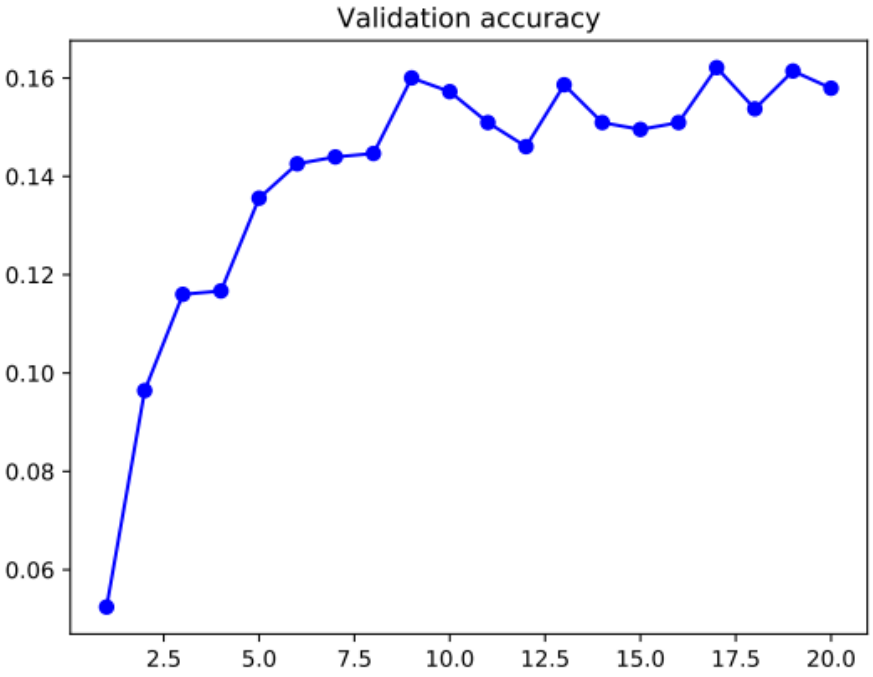

FIG.4 - VANILLA SEQUENCE-TO-SEQUENCE MODEL VALIDATION ACCURACY REVERTING THE SOURCE STRING

The test set accuracy was higher by reverting the source string. We obtained an accuracy of 15.66%.

4.

To implement this question, we toke inspiration from **[3]**. There are some considerations to be stated. There are 3 different .py files: attn*Seq2seq.py*, attn*Classes.py* and *readInput.py*. The first one has the whole pipeline for question Q1. (d), the second has the classes implemented for the *Vanilla sequence-to-sequence model* with *Attention mechanism* and the third file has a class and functions necessary for reading the input files: *trainar-eng.txt*, *testar-eng.txt* and *validar-eng.txt*.

In this report we will only show for this question the classes implemented for the *Vanilla sequence-to-sequence model* with *Attention mechanism.* Please check the other *.py* files to check how the input is read, the conversion to word embeddings and the rest of the pipeline. We ran the whole pipeline with 20 epochs.

For the *BidirectEncoder* class take in mind that we defined it as a bidirectional LSTM.

```
class BidirectEncoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size) # embbedin
g size = hidden size

        self.lstm = nn.LSTM(input_size=hidden_size, hidden_size=hidden_si
ze // 2, bidirectional=True)

        self.hidden = self.init_hidden()

    def forward(self, src):

        # src = [src len, batch size]
        embedded = self.embedding(src) # [src len, batch size, # embbedin
g size = hidden size]

        output, hidden = self.lstm(embedded)

        #outputs = [src len, batch size, hid dim = emb dim * n directions
]
        #hidden[0] = [n layers * n directions, batch size, hid dim]
        #hidden[1] = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer
        return output, hidden

    def init_hidden(self):
```

```
        h0 = torch.zeros(1, 1, self.hidden_size)
        c0 = torch.zeros(1, 1, self.hidden_size)

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

```
class AttnDecoder(nn.Module):
    def __init__(self, hidden_size, output_size, biggest_word):
        super().__init__()
        self.hidden_size = hidden_size # embbeding size = hidden size
        self.output_size = output_size
        self.max_length = biggest_word

        self.embedding = nn.Embedding(output_size, hidden_size)

        self.fc_hidden = nn.Linear(self.hidden_size, self.hidden_size, bi
as=False)
        self.fc_encoder = nn.Linear(self.hidden_size, self.hidden_size, b
ias=False)
        self.weight = nn.Parameter(torch.FloatTensor(1, hidden_size))
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_s
ize)
        self.lstm = nn.LSTM(self.hidden_size*2, self.hidden_size, batch_f
irst=True)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):

        #input = [batch size]
        #hidden[0] = [n layers * n directions, batch size, hid dim]
        #hidden[1] = [n layers * n directions, batch size, hid dim]
        #encoder_outputs = [src len, batch size, hid dim = emb dim * n_di
rections]

        encoder_outputs = encoder_outputs.squeeze()
        input = input.unsqueeze(0)  #input = [1, batch size]
        embedded = self.embedding(input)

        # Calculating Alignment Scores
        x = torch.tanh( self.fc_hidden(hidden[0]) + self.fc_encoder(encod
er_outputs) )
        w_alignment = self.weight.unsqueeze(2)
        alignment_scores = x.bmm(w_alignment)

        # Softmaxing alignment scores to get Attention weights
        attn_weights = F.softmax(alignment_scores.view(1,-1), dim=1)
```

```
        # Multiplying the Attention weights with encoder outputs to get t
he context vector
        context_vector = torch.bmm(attn_weights.unsqueeze(0),encoder_outp
uts.unsqueeze(0))

        # Concatenating context vector with embedded input word
        output = torch.cat((embedded[0], context_vector[0]), 1).unsqueeze
(0)

        # Passing the concatenated vector as input to the LSTM cell
        output, hidden = self.lstm(output, hidden)


        #output = [seq len, batch size, hid dim * n directions]
        #hidden[0] = [n layers * n directions, batch size, hid dim]
        #hidden[1] = [n layers * n directions, batch size, hid dim]

        prediction = self.out(output.squeeze(0))

        #prediction = [batch size, output dim]

        return prediction, hidden, attn_weights

    def init_hidden(self):
        h0 = torch.zeros(1, 1, self.hidden_size)
        c0 = torch.zeros(1, 1, self.hidden_size)

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

For the *Seq2seq* class take in mind that the first hidden state of the decoder is initialized with zeros. Regarding training and evaluating phase the same is applied as before.

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use ground-
truth inputs 75% of the time
```

```
        batch_size = trg.shape[1]
        trg_vocab_size = self.decoder.output_size

        #tensor to store decoder outputs
        if self.training is True:
            trg_len = trg.shape[0]
            outputs = torch.zeros(trg_len, batch_size, trg_vocab_size)
        else:
            outputs = torch.zeros(2*self.decoder.max_length, batch_size,
trg_vocab_size)

        #last hidden state of the encoder is used as the initial hidden s
tate of the decoder
        encoder_outputs, h = self.encoder(src)

        #first input to the decoder is the <sos> tokens
        input = trg[0,:]

        if self.training is True:
            loop_condition = trg_len
        else:
            loop_condition = 2*self.decoder.max_length

        for t in range(1, loop_condition):
            if t == 1:
                hidden = self.decoder.init_hidden()

            #insert input token embedding, previous hidden and previous c
ell states
            #receive output tensor (predictions) and new hidden and cell
states
            output, hidden, attn_weights = self.decoder(input, hidden, en
coder_outputs)

            #place predictions in a tensor holding predictions for each t
oken
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            if self.training is False and top1.item() == EOS_token:
                break

            #if teacher forcing, use actual next token as next input
```

```
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        outputs = outputs[:t+1]
        return outputs
```

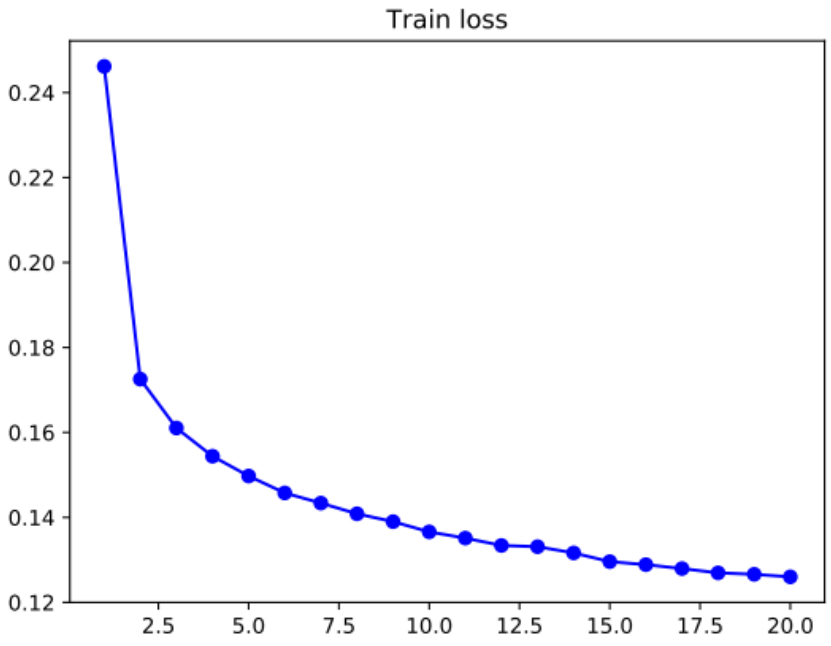

FIG.5 - VANILLA SEQUENCE-TO-SEQUENCE MODEL TRAIN LOSS WITH ATTENTION MECHANISM

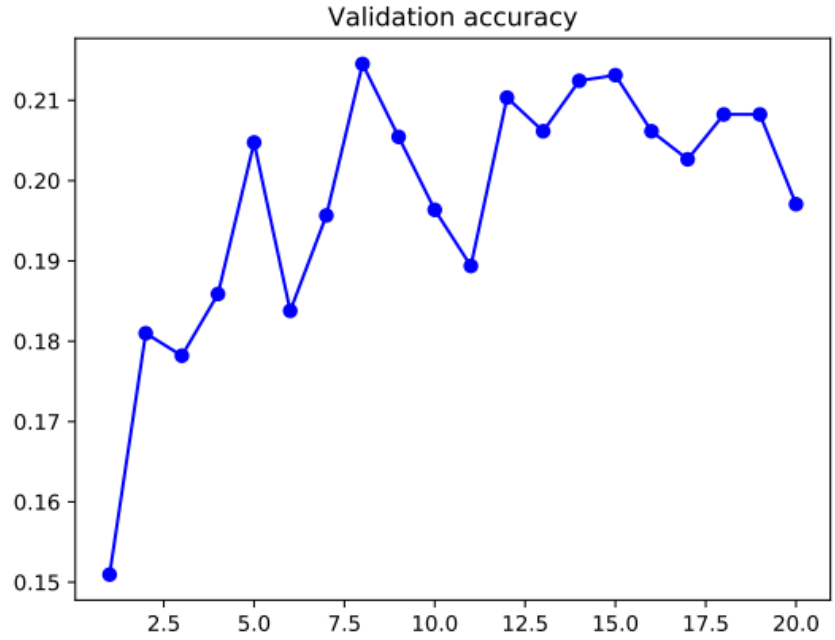

FIG.6 - VANILLA SEQUENCE-TO-SEQUENCE MODEL VALIDATION ACCURACY WITH ATTENTION MECHANISM

The test set accuracy was higher than the previous tests by achieving an accuracy of 18.55%.

QUESTION 2

1.

$$E[f(x)] = \int f(x) \cdot p_\theta(x)\, dx \quad (\mathbf{1})$$

$$\nabla_\theta \log p_\theta(x) = \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)} \leftrightarrow \nabla_\theta p_\theta(x) = \nabla_\theta \log p_\theta(x) \cdot p_\theta(x) \quad (\mathbf{2})$$

Since f is a function independent of θ, the gradient of **(1)** with respect to θ is,

$$\nabla_\theta\, E[f(x)] = \nabla_\theta \int f(x) \cdot p_\theta(x)\, dx = \int f(x) \cdot \nabla_\theta\, p_\theta(x)\, dx \quad (\mathbf{3})$$

By applying **(2)** on **(3)** we get,

$$\nabla_\theta\, E[f(x)] = \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx = E[f(x) \cdot \nabla_\theta \log p_\theta(x)]$$

Where the expectation is with respect to $p_\theta(x)$.

2.

By using **(2)** we get that,

$$E[\nabla_\theta \log p_\theta(x)] = \int \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx$$
$$= \int \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)} \cdot p_\theta(x)\, dx = \int \nabla_\theta p_\theta(x)\, dx = \nabla_\theta \int p_\theta(x)\, dx = \nabla_\theta \cdot 1$$
$$= 0$$

And,

$$E[(f(x) - b) \cdot \nabla_\theta \log p_\theta(x)] = \int (f(x) - b) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx$$
$$= \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx - \int b \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx =$$
$$= \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx - \int b \cdot \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)} \cdot p_\theta(x)\, dx =$$
$$= \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx - \int b \cdot \nabla_\theta p_\theta(x)\, dx =$$
$$= \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx - b \cdot \nabla_\theta \int p_\theta(x)\, dx =$$
$$= \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx - b \cdot \nabla_\theta \cdot 1 =$$
$$= \int f(x) \cdot \nabla_\theta \log p_\theta(x) \cdot p_\theta(x)\, dx = E[f(x) \cdot \nabla_\theta \log p_\theta(x)], \forall_b \in \mathcal{R}$$

By using b (baseline) what's left is the advantage, the amount of how much the current action is better than we'd usually do in that state. The advantage has lower variance since the baseline compensates for the variance introduced by being in different states. Thus, improving variance of sample-based estimates.

REFERENCES

[1] NLP from scratch: Translation with a sequence to sequence network and attention

[2] Sequece to sequence Learning with Neural Networks

[3] Attention Mechanism