CSC413Coding_A3

haoyanjiang

January 12, 2021

1

1.1

```python
class MyGRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MyGRUCell, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        # ------------
        # FILL THIS IN
        # ------------
        ## Input linear layers
        self.Wiz = nn.Linear(input_size, hidden_size)
        self.Wir = nn.Linear(input_size, hidden_size)
        self.Wih = nn.Linear(input_size, hidden_size)

        ## Hidden linear layers
        self.Whz = nn.Linear(hidden_size, hidden_size)
        self.Whr = nn.Linear(hidden_size, hidden_size)
        self.Whh = nn.Linear(hidden_size, hidden_size)

    def forward(self, x, h_prev):
        """Forward pass of the GRU computation for one time step.

        Arguments
            x: batch_size x input_size
            h_prev: batch_size x hidden_size

        Returns:
            h_new: batch_size x hidden_size
        """

        # ------------
        # FILL THIS IN
```

```
        # ------------
        z = torch.sigmoid(self.Wiz(x)+self.Whz(h_prev))
        r = torch.sigmoid(self.Wir(x)+self.Whr(h_prev))
        g = torch.tanh(self.Wih(x)+torch.mul(r, self.Whh(h_prev)))
        h_new = torch.mul((1 - z),g) + torch.mul(z,h_prev);
        return h_new
```

1.2

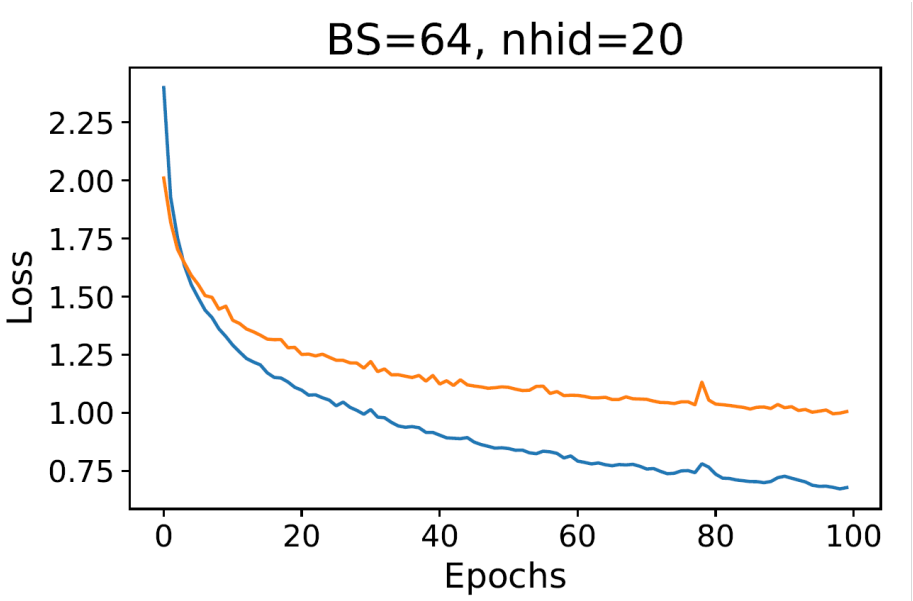


Figure 1: Training and Validation Cross-Entropy Loss for GRU

1.3

Testing results:
source: the air conditioning is working
translated: ehay airay onfichicationcay isshay odinggay-asingway
source: this has a lot of errors
translated: istray astay away othay ofway erachtay
source: wubba lubba dub dub
translated: undbray urfonay undtay undtay
source: the fish listened intently to what the frogs had to say
translated: ehay isthay ilertedway intentway otay athay ehay ongonsay aryday otay ayway

This model fails significantly where
1) when the leading character is an vowel
2) Also, slightly with the leading consonant pairs such as ”sh” and ”ch”,but close enough
3) Fails both on long words

2

2.1

$$\tilde{\alpha}_i^{(t)} = f\left(Q_t, K_i\right) = W_2(max(0, W_1[Q_t; K_i] + b_1)) + b_2$$
$$\alpha_i^{(t)} = softmax(\tilde{\alpha}_i^{(t)})$$
$$c_t = \sum_{i=0}^{T} \alpha_i^{(t)} V_i$$

2.2

```
def forward(self, inputs, annotations, hidden_init):
        """Forward pass of the attention-based decoder RNN.

        Arguments:
            inputs: Input token indexes across a batch for all the time
                step. (batch_size x decoder_seq_len)
            annotations: The encoder hidden states for each step of the
                input.
                        sequence. (batch_size x seq_len x hidden_size)
            hidden_init: The final hidden states from the encoder, across
                a batch. (batch_size x hidden_size)

        Returns:
            output: Un-normalized scores for each token in the
                vocabulary, across a batch for all the decoding time
                steps. (batch_size x decoder_seq_len x vocab_size)
            attentions: The stacked attention weights applied to the
                encoder annotations (batch_size x encoder_seq_len x
                decoder_seq_len)
        """

        batch_size, seq_len = inputs.size()
        embed = self.embedding(inputs) # batch_size x seq_len x
            hidden_size

        hiddens = []
        attentions = []
        h_prev = hidden_init
        for i in range(seq_len):
            # ------------
            # FILL THIS IN - START
            # ------------
            embed_current = embed[:,i,:] # Get the current time step,
                across the whole batch
            context, attention_weights = self.attention(h_prev,
                annotations, annotations) # batch_size x 1 x hidden_size
```

```
            embed_and_context =
                torch.cat([context.reshape_as(embed_current),
                embed_current], dim=1) # batch_size x (2*hidden_size)
            h_prev = self.rnn(embed_and_context, h_prev) # batch_size x
                hidden_size
            # ------------
            # FILL THIS IN - END
            # ------------

            hiddens.append(h_prev)
            attentions.append(attention_weights)

        hiddens = torch.stack(hiddens, dim=1) # batch_size x seq_len x
            hidden_size
        attentions = torch.cat(attentions, dim=2) # batch_size x seq_len
            x seq_len

        output = self.out(hiddens) # batch_size x seq_len x vocab_size
        return output, attentions
```

2.3

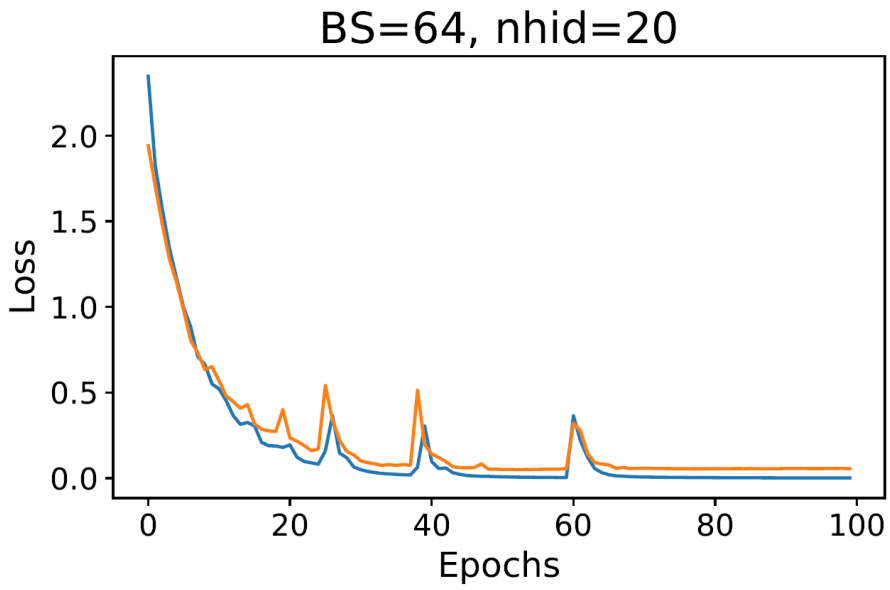


Figure 2: Training and Validation Cross-Entropy Loss for Additive attention

2.4

Testing results:
source: the air conditioning is working
translated: etay airway onditioningcay isway orkingway
source: this has a lot of errors
translated: isthay ashay away otlay ofway errorsway
source: wubba lubba dub dub
translated: ubbaway ubbay-aglay ubday ubday
source: the fish listened intently to what the frogs had to say
translated: etay ishfay istenedlay intentlyway otay atwhay etay ogsfray adhay otay aysay

The test for sentance: "the air conditioning is working", for example, RNN with attention does much better than without attention. The target is: "ethway airway onditioningcay isway orkingway". RNN with attention predict all correctly except "the", while RNN without attention predicts all incorrectly. In all, the attention in RNN significantly improve the accuracy. This model has minor problems with leading consonant pairs translation.

2.5

The RNN with attention trains slower than RNN without attention since RNN with attention requires two parts of calculation in sequence, which takes much more computation power, and not calculate in parallel.

3

3.1

Code for forward function of class **ScaledDotAttention**

```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor.
            (batch_size x (k) x hidden_size)
        keys: The encoder hidden states for each step of the input
            sequence. (batch_size x seq_len x hidden_size)
        values: The encoder hidden states for each step of the input
            sequence. (batch_size x seq_len x hidden_size)

    Returns:
        context: weighted average of the values (batch_size x k x
            hidden_size)
        attention_weights: Normalized attention weights for each encoder
            hidden state. (batch_size x seq_len x 1)

        The output must be a softmax weighting over the seq_len
            annotations.
    """

    # ------------
    # FILL THIS IN
    # ------------
    batch_size, seq_len, _ = keys.size()
    q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        #batch_size x (k) x hidden_size
    k = self.K(keys).view(batch_size, -1, self.hidden_size) #batch_size
        x seq_len x hidden_size
    v = self.V(values).view(batch_size, -1, self.hidden_size)
        #batch_size x seq_len x hidden_size
    unnormalized_attention = torch.bmm(k, q.transpose(1,2)) *
        self.scaling_factor #batch_size x seq_len x k
    attention_weights = self.softmax(unnormalized_attention) #batch_size
        x seq_len x k
    context = torch.bmm(attention_weights.transpose(1,2), v) #batch_size
        x k x hidden_size
    return context, attention_weights
```

3.2

Code for forward function of class **CausalScaledDotAttention**

```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor.
            (batch_size x (k) x hidden_size)
        keys: The encoder hidden states for each step of the input
            sequence. (batch_size x seq_len x hidden_size)
        values: The encoder hidden states for each step of the input
            sequence. (batch_size x seq_len x hidden_size)

    Returns:
        context: weighted average of the values (batch_size x k x
            hidden_size)
        attention_weights: Normalized attention weights for each encoder
            hidden state. (batch_size x seq_len x 1)

        The output must be a softmax weighting over the seq_len
            annotations.
    """

    # ------------
    # FILL THIS IN
    # ------------
    batch_size, seq_len, _ = keys.size()
    q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        #batch_size x (k) x hidden_size
    k = self.K(keys).view(batch_size, -1, self.hidden_size) #batch_size
        x seq_len x hidden_size
    v = self.V(values).view(batch_size, -1, self.hidden_size)
        #batch_size x seq_len x hidden_size
    unnormalized_attention = torch.bmm(k, q.transpose(1,2)) *
        self.scaling_factor #batch_size x seq_len x k
    mask =
        self.neg_inf*torch.tril(torch.ones_like(unnormalized_attention),
        diagonal=-1)
    attention_weights = self.softmax(unnormalized_attention + mask)
    context = torch.bmm(attention_weights.transpose(1,2), v) #batch_size
        x k x hidden_size
    return context, attention_weights
```

3.3

Code for forward function of class **TransformerEncoder**

```
def forward(self, inputs):
    """Forward pass of the encoder RNN.

```

```
    Arguments:
        inputs: Input token indexes across a batch for all time steps in
            the sequence. (batch_size x seq_len)

    Returns:
        annotations: The hidden states computed at each step of the
            input sequence. (batch_size x seq_len x hidden_size)
        hidden: The final hidden state of the encoder, for each sequence
            in a batch. (batch_size x hidden_size)
    """

    batch_size, seq_len = inputs.size()
    # ------------
    # FILL THIS IN - START
    # ------------
    encoded = self.embedding(inputs) # batch_size x seq_len x hidden_size
    # Add positinal embeddings from self.create_positional_encodings.
        (a'la https://arxiv.org/pdf/1706.03762.pdf, section 3.5)
    encoded =
        encoded+self.create_positional_encodings(seq_len).unsqueeze(0)

    annotations = encoded

    for i in range(self.num_layers):
      new_annotations, self_attention_weights =
          self.self_attentions[i](annotations, annotations, annotations)
          # batch_size x seq_len x hidden_size
      residual_annotations = annotations + new_annotations
      new_annotations = self.attention_mlps[i](residual_annotations)
      annotations = residual_annotations + new_annotations
    # ------------
    # FILL THIS IN - END
    # ------------

    # Transformer encoder does not have a last hidden layer.
    return annotations, None
```

3.4

Code for forward function of class **TransformerDecoder**

```
def forward(self, inputs, annotations, hidden_init):
        """Forward pass of the attention-based decoder RNN.

        Arguments:
            inputs: Input token indexes across a batch for all the time
                step. (batch_size x decoder_seq_len)
            annotations: The encoder hidden states for each step of the
```

```
            input.
                    sequence. (batch_size x seq_len x hidden_size)
        hidden_init: Not used in the transformer decoder
    Returns:
        output: Un-normalized scores for each token in the
            vocabulary, across a batch for all the decoding time
            steps. (batch_size x decoder_seq_len x vocab_size)
        attentions: The stacked attention weights applied to the
            encoder annotations (batch_size x encoder_seq_len x
            decoder_seq_len)
    """

    batch_size, seq_len = inputs.size()
    embed = self.embedding(inputs) # batch_size x seq_len x
        hidden_size

    embed = embed + self.positional_encodings[:seq_len]
    encoder_attention_weights_list = []
    self_attention_weights_list = []
    contexts = embed
    for i in range(self.num_layers):
      # ------------
      # FILL THIS IN - START
      # ------------
      new_contexts, self_attention_weights =
          self.self_attentions[i](contexts, contexts, contexts) #
          batch_size x seq_len x hidden_size
      residual_contexts = contexts + new_contexts
      new_contexts, encoder_attention_weights =
          self.encoder_attentions[i](residual_contexts, annotations,
          annotations) # batch_size x seq_len x hidden_size
      residual_contexts = residual_contexts + new_contexts
      new_contexts = self.attention_mlps[i](residual_contexts)
      contexts = residual_contexts + new_contexts

      # ------------
      # FILL THIS IN - END
      # ------------

      encoder_attention_weights_list.append(encoder_attention_weights)
      self_attention_weights_list.append(self_attention_weights)

    output = self.out(contexts)
    encoder_attention_weights =
        torch.stack(encoder_attention_weights_list)
    self_attention_weights = torch.stack(self_attention_weights_list)

    return output, (encoder_attention_weights,
        self_attention_weights)
```

3.5

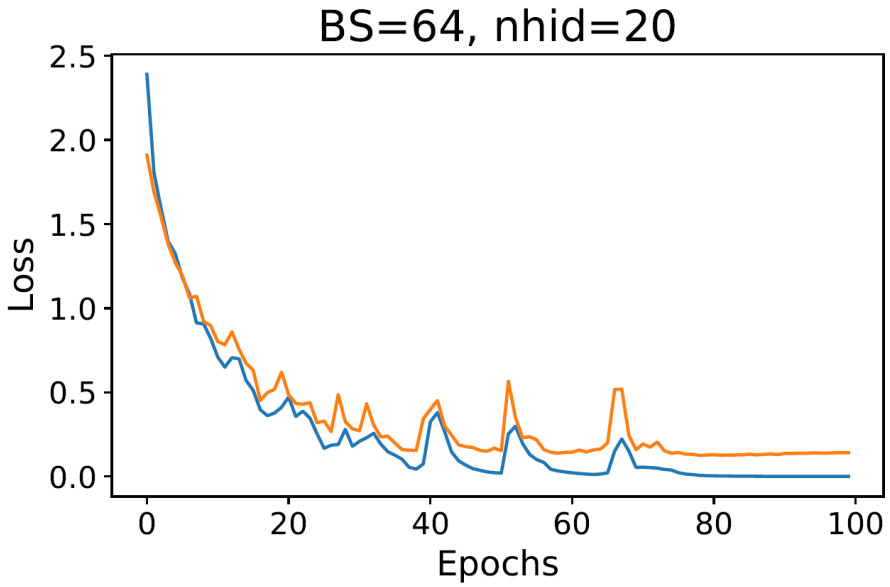


Figure 3: Training and Validation Cross-Entropy Loss for Transformer model with Causal Scaled Dot Product

Testing results:
source: the air conditioning is working
translated: ehay airway onditioningcay isway orkingway
source: this has a lot of errors
translated: isthhEOSy ashhy awway ottay ofway errorsway
source: wubba lubba dub dub
translated: ubbay ubbay uddy uddy
source: the fish listened intently to what the frogs had to say
translated: ehay issay istenedday intentlyway oty atwway ehay oosfrayy adday oty aay

As we can see, the testing result is worse than GRU with attention, but with much faster speed and good enough accuracy. The training result is better that GRU with attention, while has worse testing result indicates that transformer is possibly overfit. The model can mostly predicts correctly following the three rules: exactly correct for vocabularies with leading vowel, almost correct to vocabularies with leading consonants and leading consonant pairs.

3.6

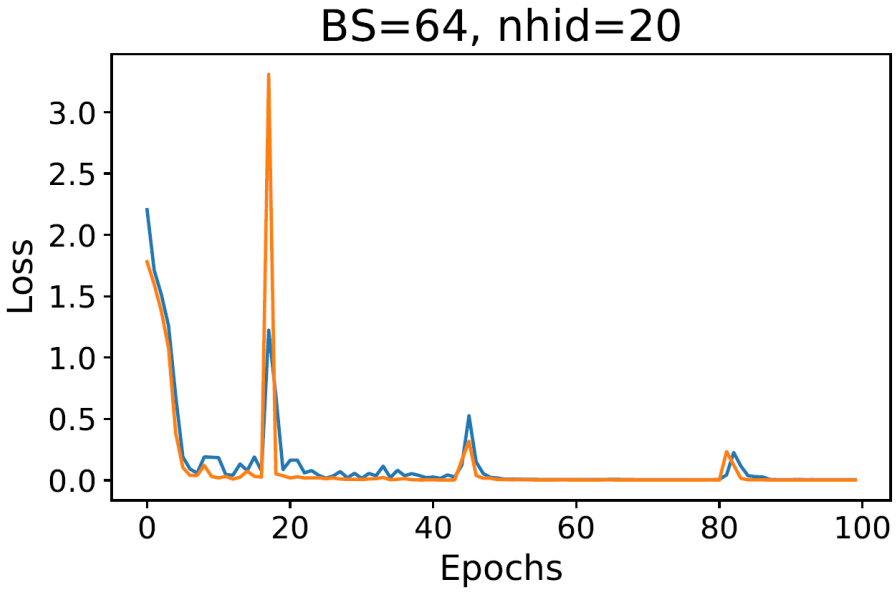


Figure 4: Training and Validation Cross-Entropy Loss for Transformer model without Causal Scaled Dot Product

Testing results:
source: the air conditioning is working
translated:
source: this has a lot of errors
translated: uiiiiiiiiiiiiiiiifi
source: wubba lubba dub dub
translated:
source: the fish listened intently to what the frogs had to say
translated: uueeeeeeeeeeeeeeeee

As we can see, the result is totally messed up, with predictions way off the target. A lot of repetitions and blank results are shown. Almost all rules are violated. This behaves much worse than model with causal model, this may because without causal model, the multi-head attention considered the whole decoder input sequence at each position. This causes the network lacking of the ability for future prediction and cannot foresee the future in testing time.

3.7

Model	Advantage	Disadvantage
Additive Attention	Has higher capacity for training	Training in sequential steps, take more time, lack paralleling designs
Transformer	More efficient with matrix manipulation, high accuracy	Too many parameters, easily overfit

Table 1: Advantages and disadvantages between two models

4

4.1

Change ReLU to Sigmoid activation function

4.2

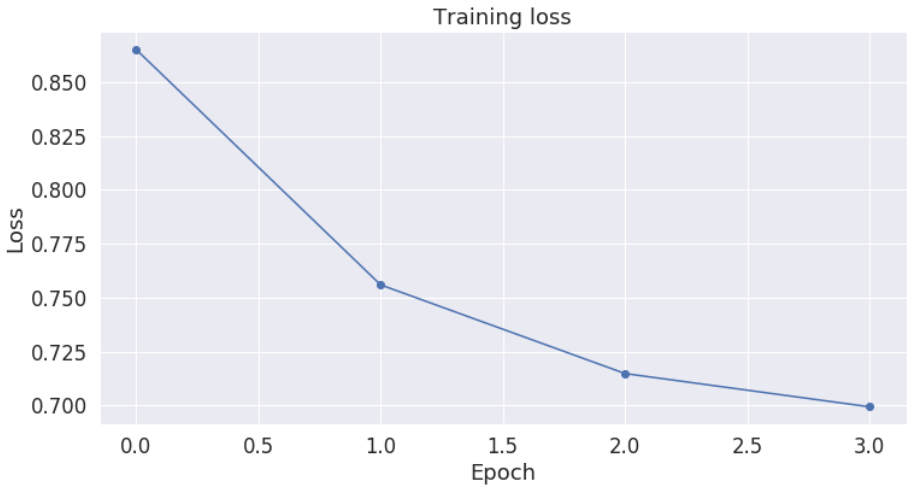


Figure 5: Training Loss for Freeze Bert

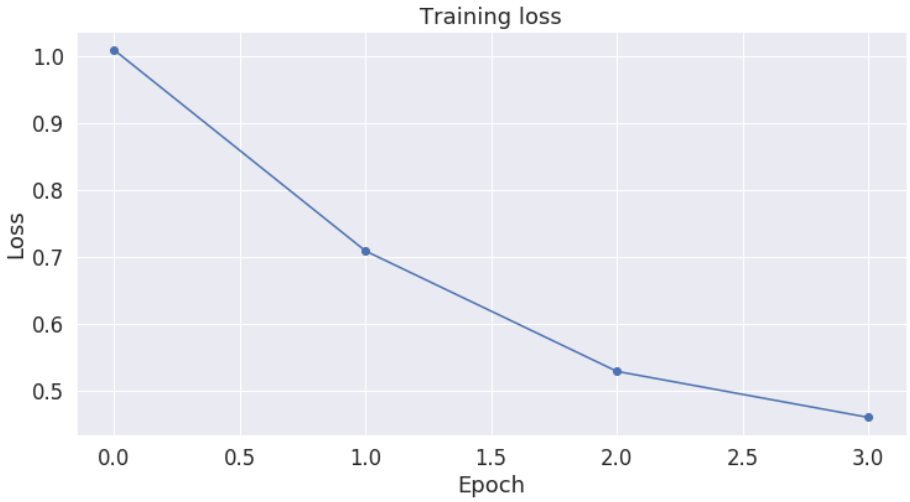


Figure 6: Training Loss for Fine-tuned Bert

	Fine-tuned Bert	Freeze Bert
Negative Accuracy	100.0%	0.0%
0 Accuracy	0.0%	0.0%
Positive Accuracy	99.1%	100.0%

Table 2: Accuracy for Fine-tuned Bert and Freeze Bert under different test

Freeze Bert has same validation accuracy of 73% for all 4 validations, while fine-tuned Bert has a mostly increasing validation accuracy and reaches 98% in the end. Also shown in figure 5 and 6 that Fine-tuned Bert generally has smaller training errors than Freeze Bert, which indicates that Fine-tuned Bert has better generalization than Freeze Bert. Both models fails under 0 results, this may because the answer space for prediction exactly equals 0 is much much smaller than "positive" and "negative". It is almost impossible to have a prediction result exactly equal to 0. A more reasonable solution would be three equal ranges of acceptable values for "positive", 0, and "negative"

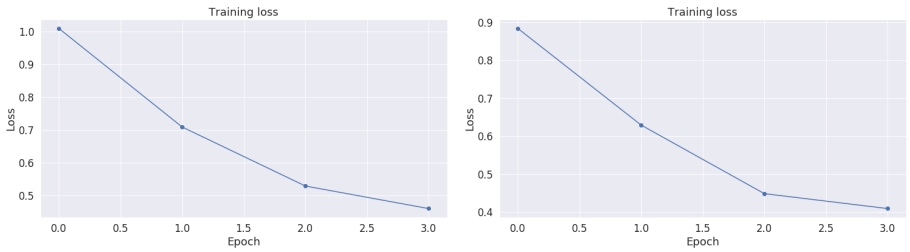
4.3

"one minus one minus one" fine-tuned Bert gives incorrect result, while "1 minus 1 minus 1" gives correct result. This is not because the model 'understand' the second phrase more correctly, but "1 minus 1" gives negative, "one minus one" gives positive. All tests fails on 0 as expected. Same trend is found for "$string(num1) minus string(num1)$" always gives positive result, while "$num1 minus num1$" always gives negative results.

4.4

3. change learning rate of fine-tuned model to $2 * 10^{-4}$

Although the learning rate is increased in a magnitude of 10, the training time for this model is still almost the same as the first one, with much lower training errors but similar validation accuracy. As shown in figure 7



(a) Learning rate of $2 * 10^{-5}$ (b) Learning rate of $2 * 10^{-4}$

Figure 7: Training Loss for Fine-tuned Bert