Neural Networks and their Applications

2022 Spring

Assignment 3

2002387 Mahmoud Aboud Nada *Instructor: Dr. Hazem Abbas*

§ Home Work 3 §

Problem 1:

(1)

1.
$h_2 = \tanh(2*10+2*h_1)$
$h_1 = \tanh(1*10+1*1)$
$h_1 = 0.999$
$h_2 = \tanh(2*10+2*0.999)$
$h_2 = 1$
$y_2 = F(h_2)$
$y_2 = 1$

2.
$L_t = \Sigma(\hat{y}-y)^2$
$y_1 = 0.999$
$y_2 = 1$
$L_t = (1-5)^2 + (0.999-1)^2$
$L_t = 32$

3.
$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y_1}}{\partial h_1} * \frac{\partial h_1}{\partial h_0}$
$\frac{\partial L}{\partial \hat{y}} = 2*1 = 2$
$\frac{\partial \hat{y_1}}{\partial h_1} =$
$\frac{\partial h_1}{\partial h_0} =$

4.
$\frac{\partial L}{\partial w_h} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y_1}}{\partial h_1} * \frac{\partial h_1}{\partial h_0} * \frac{\partial h_0}{\partial w_h}$

(2) In theory, RNNs are absolutely capable of handling such "long-term dependencies.", Sadly, in practice, Recurrent Neural Network don't seem to be able to learn them. This problem is called the Vanishing gradient problem. The neural network updates the weight using the gradient descent algorithm. The gradients grow smaller when the network progress down to lower layers, this can be seen in the formula $h_t = \tanh(W_{hh}h_{t-1}, W_{xh}X_t)$ each step depends on all the previous steps and if any of those previous steps accidentally gets very low it will significantly affect all of its successors and the gradients will stay constant meaning there is no space for improvement. The model learns from a change in the gradient. This change affects the network's output. However, if the difference in the gradients is tiny, the network will not learn anything and so there is no difference in the output. Therefore, a network facing a vanishing gradient problem cannot converge towards a good solution.

(3) We can use GRU over vanilla RNN when we need to learn from long sequences or "Long-term dependencies" because GRU has the update gate which works as the forget and input LSTM gates.

(4) **Advantage:** for long sequences it helps lower the computation power and the memory needed to save all the sequence data.

Disadvantage: in TBTT we divide each sequence, if we split sequences into small splits we may not capture all important sequence related features.

(5)

1. RNN is able to learn temporal dependencies between the letters that form the words in sentences, while a feedforward NN will not consider the sequence information.

2. Since this is an RNN we need to preprocess its input in form of sequences not a single example each time, we will trun our data into words and each word is considered a sequence of characters, that would be more relevant as we are dealing with a problem that maps characters not words.

3. The RNN will receive as input the preprocessed cipher text, character by character, and output the predictions character by character. Accordingly, our defined RNN will have a character-level many-to-many (matching) architecture.

4. Assuming we have a dataset that has many sentences, we will split each sentence into single word, and consider each word as an example of certain sequence length, make an example

5. When defining the model, we specify the size of the input layer so that the model knows what length to expect. we'll convert all the sequences to the same length by adding padding to the shorter sequences, or truncating the longer sequences.

6. Since there is a 1-to-1 mapping between characters, the input at each timestep is a single character. The characters need to be individually isolated and converted to lowercase. Punctuation marks such as commas and periods should map to themselves.

Afterward, these tokens need to be mapped to integers. Every character will be assigned an integer value, non-letters included. Assuming that the only non-letters are commas, spaces, and periods.

Then we apply the padding or truncating process to keep identical sequence length across dataset.

7.

```python
import tensorflow as tf

input_layer = tf.keras.Input(input_shape)
rnn = tf.keras.layers.SimpleRNN(32, activation="relu",
 ↪ return_sequences=True)(input_seq)
logits =
 ↪ tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(plaintext_vocab_size))(rnn)
out = tf.keras.layers.Activation("softmax")(logits)
model = tf.keras.Model(inputs=input_seq, outputs=out)
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.optimizers.Adam(learning_rate),
              metrics=["accuracy"])
```

8. The RNN outputs vectors of numbers which need to be converted back to characters.

Each vector is a categorical distribution over the possible unique characters. To get the prediction, we choose the argmax of each vector. Then, we map that index back to the original index by reversing the original mapping that was performed in tokenization process.