Recurrent Neural Network (CNN)
Building Effective RNNs

M. R. Hasan
Deep Learning
Readings

• Geron: 15
• Zhang: 8
• Chollet: 6
What We Will Cover

• Limitations of Feedforward Neural Network (FNN) for processing text data
Building Effective RNNs

- We will discuss two topics that contribute towards building effective RNNs.
  - Increase The Representational Power of RNNs
  - Preventing Overfitting in RNNs
Increase Representational Power of RNNs

• So far we have discussed how to train RNNs for very long input sequence.
• In addition to using weight initialization, proper activation, normalization, we introduced special RNN architectures such as GRU and LSTM.
• The next topic we discuss is how to create powerful representations using RNNs for solving complex problems.
• We discuss two techniques.
  - Deep RNNs
  - Bidirectional RNNs
Deep RNNs

• All types of RNN architectures that we have discussed so far are based on a single unidirectional hidden layer.
• However, a single layer is quite limited in its capability to offer enough flexibility for modeling any function of interactions between the hidden state and the input sequence.
Deep RNNs

- Complex problems require us to build flexible models so that powerful representations can be created.
- In feedforward neural networks we did this by increasing the capacity of the model via stacking hidden layers.
- It enabled the network to learn more powerful representations.
- It was possible because each layer in a feedforward NN is connected to the next layer through nonlinearity.
- Thus the key technique is to stack computational units via nonlinearity.
Deep RNNs

• How do we do this to increase capacity in RNNs?
• In other words, how do we stack the computational blocks in RNNs?
• One approach is to use multi-layer perceptron (MLP) inside the RNN cell. More specifically, we can include many hidden layers in the FC networks in a RNN cell (e.g., LSTM).
Deep RNNs

• Another approach is to stack multiple layers of RNN on top of each other.
• It will provide us a more flexible model because of the combination of multiple layers.
• Then, we could learn from input sequence in a hierarchical fashion.
• For example, lower layers may learn short-term dynamics and higher layers can focus on longer-term dynamics.
Deep RNNs

• Below we present the latter approach of stacking multiple hidden layers to build deep RNNs.
Deep RNNs

- We describe the deep RNN architecture.

Let's say that there are $L$ hidden layers.

We denote the layer number using superscript $l$. 
Deep RNNs

- The hidden state at layer $l = 1$ is computed as previously by using a minibatch input $X_t$ at timestep $t$ and hidden state at the previous timestep $H_{t-1}$:

$$H_{t}^{(1)} = f_1(X_t, H_{t-1})$$

Then, the hidden states in the subsequent layers are computed:

$$H_{t}^{(l)} = f_l(H_{t}^{(l-1)}, H_{t-1}^{(l)})$$
Deep RNNs

- Finally, the output layer is only based on the hidden state of hidden layer $L$.\[ O_t = g(H_t^{(L)}) \]

We can use any RNN architecture to implement this model, e.g., simple RNN, GRU, LSTM.
Deep RNNs

- See the TensorFlow implantation of two Deep RNN architectures:
  - Many-to-One
  - Many-to-Many

https://github.com/rhasanbd/Recurrent-Neural-Networks/blob/main/RNN-5-Building%20Effective%20RNNs.ipynb
Bidirectional RNNs

• So far, we have discussed unidirectional RNN models. Using these models, we can infer the next token based on the previous sequence of tokens.

• For example, based on the sequence “The sky is”, an RNN model can learn the next word “blue”.

• However, in some scenarios, the past sequence does not provide enough context to infer the next token.
Bidirectional RNNs

• Consider the following two sentences.
  - The sky is (BLUE) on a sunny day
  - The sky is (DARK) at night.
• In these two examples, information on the color of the sky is encoded in the latter part of the sequences, after the color token.
• The uni-directional RNN models perform poorly on this type of inference.
Bidirectional RNNs

- One approach to overcome this limitation of the unidirectional RNNs is to build bidirectional RNNs (BRNNs), such that we can use information of both the past and future to make an inference.
Bidirectional RNNs

- Following figure shows the architecture of a bidirectional RNN model with a single hidden layer.
- In this hidden layer, during forward propagation, information passes in two directions:
  - from the start token to the end token, and then from the end token to the start token.
Bidirectional RNNs

- The BRNN model was first proposed in 1997.
- There are various possible architectures of BRNN: https://www.cs.toronto.edu/~graves/ijcnn_2005.pdf
BRNN: Computing the Hidden States

- We show the calculation of the hidden states in a single-layer BRNN.
- At each timestep there will be two hidden states: forward $\overrightarrow{H}_t$ and backward $\overleftarrow{H}_t$.

\[
\overrightarrow{H}_t = \phi(\mathbf{x}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{H}_{t-1} \mathbf{W}_{hh}^{(f)}) + \overrightarrow{b}_h
\]
\[
\overleftarrow{H}_t = \phi(\mathbf{x}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{H}_{t-1} \mathbf{W}_{hh}^{(b)}) + \overleftarrow{b}_h
\]
BRNN: Computing the Hidden States

• Here the size of the minibatch input $X_t$ is $m \times d$ and the size of a hidden state is $m \times h$.

• We have 4 weight matrices and two bias vectors for forward and backward processing.

  - Forward Weight Matrices and Bias Vector
    
    $W_{xh}^{(f)}$: size $d \times h$
    
    $W_{hh}^{(f)}$: size $h \times h$
    
    $b_h^{(f)}$: size $1 \times h$

  - Backward Weight Matrices and Bias Vector
    
    $W_{xh}^{(b)}$: size $d \times h$
    
    $W_{hh}^{(b)}$: size $h \times h$
    
    $b_h^{(b)}$: size $1 \times h$

\[
\begin{align*}
\rightarrow H_t &= \phi(X_t W_{xh}^{(f)} + \rightarrow H_{t-1} W_{hh}^{(f)} + b_h^{(f)}) \\
\leftarrow H_t &= \phi(X_t W_{xh}^{(b)} + \leftarrow H_{t-1} W_{hh}^{(b)} + b_h^{(b)})
\end{align*}
\]
BRNN: Computing the Hidden States

• Note that we need to concatenate the forward hidden states $H_t^f$ and the backward hidden states $H_t^b$ for obtaining the hidden state $H_t$.
• Due to concatenations, its size will be $m \times 2h$.

This hidden state is used as input to the output layer that computes the output as follows:

$$O_t = H_t W_{ho} + b_o$$

Also note that in a BRNN, the forward and backward directions can have different number of hidden units.
When Should We Use BRNNs?

• BRNNs require information from both ends of the sequence to estimate the output.

• Thus, for making predictions, a BRNN will have to use information from both future and past observations to predict the current one.

• Let’s say that we are building a language model that needs to predict the next word in a sequence based only on the previous words (e.g., speech recognition), then BRNNs won’t be suitable.

• Because we don't have access to future data when we make inference.

• Thus, BRNNs will perform poorly in such scenarios.
When Should We Use BRNNs?

• But for many natural language processing (NLP) tasks BRNNs turn out to be quite useful.
• For example, filling in missing words, annotating tokens (e.g., for named entity recognition).
• The NLP tasks at which BRNNs excel at is building embeddings is a self-supervised manner.
When Should We Use BRNNs?

• To encode the words in a vocabulary, we can train a BRNN, which uses both the past and future context to learn powerful embeddings.

• These embeddings can then be used in downstream supervised tasks such as sentiment classification.

• This way BRNNs are used to create deep contextualized language models.

• https://arxiv.org/abs/1802.05365
When Should We Use BRNNs?

- Due to their usefulness in many NLP tasks, BRNNs are known as the swiss army knife of NLP.
- But it’s a very expensive piece of tool to afford.
- Let’s see why.
BRNN: Computational Cost

- BRNNs pass signals first in the forward direction, and then based on the outcome of the forward hidden states, a BRNN computes the backward hidden states.
- Finally, both the forward and backward hidden states are concatenated to create the complete hidden states for a single layer.
BRNN: Computational Cost

• Due to the dependency of the backward hidden states on the forward states, the loss gradients need to be propagated through a very long dependency chain.
• As a consequence, BRNNs are exceedingly slow to train.
• This computational cost blows up in case of deep BRNNs.
• This is the reason why the NLP community deflected from BRNNs irrespective of their representations prowess.
BRNN: Implementation

• See the TensorFlow implementation of BRNN:
RNN: Preventing Overfitting

- We have discussed several techniques to increase the representational power of RNNs.
- The main idea is to build large models either by stacking many hidden layers or by passing information through both the forward and backward directions.
- This will essentially increase the model’s capacity.
- But at the same time, the model will become prone to overfitting.
RNN: Preventing Overfitting

• Previously we used the dropout regularization in feedforward neural networks to prevent overfitting.
• In dropout regularization, the noisy correlations are broken by randomly assigning zeros to some input units.
• However, applying dropout in RNNs is not straightforward.
• In RNNs we have two types of input to produce the output.
  - Input data
  - Past hidden state
RNN: Preventing Overfitting

- Two weight matrices are associated with these two data types:
  - Weights of the input layer
  - Weights of the recurrent layers
RNN: Preventing Overfitting

• While applying dropout to the input weights is useful, applying dropout to the recurrent weights is not straightforward.

• If we drop some units in the recurrent layer randomly, then it will be detrimental for learning.

• Because the same units are applied to each input in a sequence.

• Randomly switching this units on and off will disrupt the loss propagation and will deteriorate the learning.

• Thus, stochastic on-off is not a useful dropout solution for RNNs.
RNN: Preventing Overfitting

• In 2015, Yarin Gal (as part of his PhD thesis) proposed a technique to apply dropout in the recurrent layers of RNNs.
• The main idea is that the same dropout mask (the same pattern of dropped units) should be applied to the recurrent units at every timestep, instead of a dropout mask that varies randomly from timestep to timestep.
RNN: Preventing Overfitting

• What’s more, in order to regularize the representations formed by the recurrent gates of layers such as GRU and LSTM, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a recurrent dropout mask).

• This technique is effective because using the same dropout mask at every timestep allows the network to properly propagate its learning error through time.

• On the other hand, a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.
Recurrent Dropout: Implementation

• Recurrent Dropout: TensorFlow Implementation
• Every recurrent layer in Keras has two dropout-related arguments:
  - dropout, a float specifying the dropout rate for input units of the layer
  - recurrent_dropout, specifying the dropout rate of the recurrent units
• Since networks being regularized with dropout always take longer to fully converge, we need to train the network for twice as many epochs.