Intro to NLP and Deep Learning - 236605

Tutorial 5 – RNN

- Introduction to RNN
- Understanding RNN architecture by examples
- Training RNNs
- Backpropagation through time algorithm
- Vanishing Gradient problem

Tomer Golany – tomer.golany@gmail.com
Introduction to RNN
What are RNNs?

- The idea behind RNNs is to make use of **sequential information**

- In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that’s a very bad idea.

- If you want to predict the next word in a sentence, you better know which words came before it.

- RNNs are called **recurrent** because they perform the same task for every element of a sequence, with the output being depended on the previous computations.

Another way to think about RNNs is that they have a "**memory**" which captures information about what has been calculated so far.
What are RNNs?

- Here is what a typical RNN looks like:
Understanding RNN by an Example

- Lets say that our task is:
  - Given a sequence of words – predict the next word in the sequence: (Text Generation)
Understanding RNN by an Example

- **Input:** A list of documents / raw text contained in a csv file:
  - In our example we will work with 15,000 reddit comments from a [dataset available on Google’s BigQuery](https://www.google.com/bigquery).
First step:
- Usually the data will **not be clean** and we will need to do some pre-processing to get our data into the right format.
- We will go through some common pre-processing steps later.

Second Step:
- Build a vocabulary from all the words that appear in the text input.
- Decide how to represent each word – word embedding such as word2vec / Glove or one hot vector.
- For this example I will use one-hot vector representation, but usually we will train a word embedding.
Third step – Creating the dataset:

– In this step we need to create labeled examples and separate our data to train/test/validation sets.
– For simplicity, let’s say that our data contains only Two sentences and that we want to predict a sequence of 3 words.
– That means, given an input of 2 words we want our neural network to predict the third word.

- Size of the Vocabulary: |V| = 11 (There are 11 unique words in the two sentences)

| the quick brown fox jumps over the lazy dog | the = (1,0,0,…,0) |
| the White fox walked away | quick = (0,1,0,…,0) |
| | brown = (0,0,1,…,0) |
First observation: The output of our network is dependent not only on one data sample. There is a dependency between our samples.
Designing our network:

- **First try**: Regular Feed-Forward Network:

  Problem! The output is predicted only based on 1 word. We saw that predicting the next word in a sequence is depended in all the words that came before it.
Designing our network:

- **Second Try**: Regular Feed-Forward Network, but this time the size of the input is: \(|V| \times \text{sequence length}

- That means we will feed to our network the whole sequence at the first layer, and it will predict the next word based on that.

- In our example \(|V| = 11\) and the sequence length is 2, so the input vector will be of size 22.

```
the = (1,0,0,0)
white = (0,1,0,0)
```

```
x1
... x10
```

```
x2
... x22
```

```
Label - fox
```

```
the = (1,0,0,0)
white = (0,1,0,0)
```
Understanding RNN by an Example

- Designing our network:
  - Best approach – **Recurrent neural network** !!

See on board the whole architecture
More about the architecture of the RNN:

- $X_t$ is the input at time step $t$. For example, $X_1$ could be a one-hot vector corresponding to the second word of a sentence.
- $S_t$ is the hidden state at time step $t$. It's the “memory” of the network. $S_t$ is calculated based on the previous hidden state and the input at the current step:
  \[ s_t = f(Ux_t + Ws_{t-1})k. \]
- $f$ is an activation function such as ReLU or tanh.
- $S(-1)$, which is required to calculate the first hidden state, is typically initialized to all zeros.
- $O_t$ is the output of step $t$. For example, if we wanted to predict the next word in a sentence, it would be a vector of probabilities across our vocabulary:
  \[ o_t = \text{softmax}(Vs_t). \]
More about the architecture of the RNN:

- **Important** Notice:
  - Unlike a traditional deep neural network, which uses different parameters at each layer, a RNN, shares the same parameters - U, V, W across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs.
  - This greatly reduces the total number of parameters we need to learn.

- The above diagram has outputs at each time step, but depending on the task this may not be necessary. For example, when predicting the sentiment of a sentence we may only care about the final output, not the sentiment after each word. Similarly, we may not need inputs at each time step.

- The main feature of an RNN is its hidden state, which captures some information about a sequence.
Training RNNs
Training RNNs

- Training a RNN is similar to training a traditional Neural Network.
- We also use the **backpropagation algorithm**, but with a little **twist**

- Because the parameters are **shared by all time steps** in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.

- For example, in order to calculate the gradient at $t = 4$, we would need to backpropagate 3 steps and sum up the gradients.
- This is called **Backpropagation Through Time** (BPTT).
Let’s quickly recap the basic equations of our RNN:

\[ s_t = \tanh(Ux_t + Ws_{t-1}) \]
\[ \hat{y}_t = \text{softmax}(Vs_t) \]

Our loss function will usually be cross-entropy loss:

- \( y_t \) is the correct word at time step \( t \), and \( \hat{y}^t(t) \) is the prediction
- We typically treat the full sequence (sentence) as one training example, so the total error is just the sum of the errors at each time step (word).
Backpropagation Through Time

$E_0$  $E_1$  $E_2$  $E_3$  $E_4$

$s_0$  $s_1$  $s_2$  $s_3$  $s_4$

$x_0$  $x_1$  $x_2$  $x_3$  $x_4$
Our goal is to calculate the gradients of the error with respect to our parameters U,V,W and then learn good parameters using gradient descent.

Just like we sum up the errors, we also sum up the gradients at each time step for one training example:

\[ \frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W} \, . \]

To calculate these gradients we use the chain rule as we did with feed-forward networks. That’s the backpropagation algorithm when applied backwards starting from the error.
For the previous example, let's calculate the gradient of:

\[
\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial V} + \frac{\partial E_3}{\partial \hat{z}_3} \frac{\partial \hat{z}_3}{\partial V} = (\hat{y}_3 - y_3) \otimes s_3
\]

- \(z_3 = V \ast S_3\), and \(\otimes\) is the outer product of two vectors.

- We can see that the gradient based on \(V\) only depends on the values at the current time step, \(\hat{y}_3, y_3, s_3\).

- If you have these, calculating the gradient for \(V\) is a simple matrix multiplication.
Vanishing Gradients Problem

- The story is different for $\frac{\partial E_3}{\partial W}$ (and for U).
- To see why, we write out the chain rule, just as before:

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial W}$$

- Now, note that $s_3 = \tanh(Ux_t + Ws_2)$ depends on $S_2$ which depends on $W$ and $S_1$

- So if we take the derivative with respect to $W$, we can’t simply treat $S_2$ as a constant! We need to apply the chain rule again and what we really have is this:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

- We sum up the contributions of each time step to the gradient
Vanishing Gradients Problem

- In other words, because W is used in every step up to the output we care about, we need to backpropagate gradients from t=3 to t = 0:

- Note that this is exactly the same as the standard backpropagation algorithm that we use in deep Feedforward Neural Networks. The key difference is that we sum up the gradients for W at each time step.
- In a traditional NN we don’t share parameters across layers, so we don’t need to sum anything.
Vanishing Gradients Problem

- Let’s take a closer look at the gradient we calculated before:

\[
\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial y_3} \frac{\partial y_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}
\]

- Note that \( \frac{\partial s_3}{\partial s_k} \) is a chain rule in itself. For example, \( \frac{\partial s_3}{\partial s_1} = \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \).

- Also note that because we are taking the derivative of a vector function with respect to a vector, the result is a matrix (called the Jacobian matrix).

- The Jacobian matrix elements are all the pointwise derivatives.
Vanishing Gradients Problem

- We can rewrite the gradient:

\[
\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial y_3} \frac{\partial y_3}{\partial s_3} \left( \prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W}
\]

- It turns out (see paper at the references) that the **2-norm**, which you can think of it as an absolute value, of the above Jacobian matrix has an upper bound of 1.

- his makes intuitive sense because our tanh activation function maps all values into a range between -1 and 1, and the derivative is bounded by 1:
Vanishing Gradients Problem

- The tanh and sigmoid function have derivatives of 0 at both ends. They approach a flat line. When this happens we say the corresponding neurons are saturated.

- They have a zero gradient and drive other gradients in previous layers towards 0

- Thus, with small values in the matrix and multiple matrix multiplications he gradient values are shrinking exponentially fast, eventually vanishing completely after a few time steps

- Gradient contributions from “far away” steps become zero, and the state at those steps doesn’t contribute to what you are learning:

  - **You end up not learning long-range dependencies.**
Exploding Gradients Problem

– It is easy to imagine that, depending on our activation functions and network parameters, we could get exploding instead of vanishing gradients
– if the values of the Jacobian matrix are large.
– Indeed, that’s called the exploding gradient problem.

– The reason that vanishing gradients have received more attention than exploding gradients is two-fold:
  • For one, exploding gradients are obvious. Your gradients will become NaN (not a number) and your program will crash.
  • Secondly, clipping the gradients at a pre-defined threshold (See referenced paper) is a very simple and effective solution to exploding gradients. Vanishing gradients are more problematic because it’s not obvious when they occur or how to deal with them.
References