Intro to NLP and Deep Learning - 236605

Tutorial 6 – LSTM

- More about the vanishing & exploding gradient problem
- LSTM Introduction
- LSTM architecture

Tomer Golany – tomer.golany@gmail.com
Vanishing Gradients
Vanishing Gradients problem:

- Remind that we Train Neural Networks with stochastic gradient descent algorithm. That means calculating the gradient of our loss function based on each of the trainable weights.

- In many networks we might begin to notice something odd:
  - the weights closer to the end of the network change a lot more than those at the beginning

- And the deeper the network, the less and less the beginning layers change.

- This is problematic, because our weights are initialized randomly. If they're barely moving, they're never going to reach the right values, or it'll take them years.
Vanishing Gradients problem:

- To illustrate this we trained a simple fully connected network to classify MNIST images.
- Here, we can see how the gradients change over time for a network with one input layer and two hidden layers:
Vanishing Gradients problem:

- Notice how the first layer's gradients are much lower than the third layer's, which means those weights are changing by a much smaller amount.

- If we add more layers, the difference only gets more dramatic.

- The whole rest of the network is affected by what comes out of the first layer.

- So if those **first weights are totally wrong**, our network is not going to perform well.
Let’s imagine we have a 3-layer network, initialized with some set of weights and activations. For simplicity, let’s envision that each layer has one neuron:

The output of each neuron is a result of activation function on the multiplication of $W_n \times X$.

Where $f$ is an activation function such as **sigmoid** or **tanh**.
Why do gradients vanish?

- At the end of this network we end up with a **loss**, or a measure the difference between what we expected to see and what our network actually outputted. (such as cross-entropy or mean squares)

- The loss function is always a function of the output of the last layer, in our case the output from Z3.

- Now let’s start **backpropagating**. First let’s improve W3, the input weight of the **last hidden layer**:
We need to calculate the gradient of the loss with respect to $W_3$:

\[
\frac{\partial \text{Loss}}{\partial W_3} = \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot \frac{\partial f(z_3)}{\partial W_3} = \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot f'(z_3) \cdot W_3
\]

Now let’s skip ahead to calculating how to change our first input weight of the network:
We need to calculate the gradient of the loss with respect to $W_1$:

$$\frac{\partial \text{Loss}}{\partial W_1} = \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot \frac{\partial f(z_3)}{\partial f(z_2)} \cdot \frac{\partial f(z_2)}{\partial f(z_1)} \cdot \frac{\partial f(z_1)}{\partial W_1}$$

$$= \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot f'(z_3) \cdot W_3 \cdot f'(z_2) \cdot W_2 \cdot f'(z_1) \cdot W_1$$

Notice how many more terms of the form $f'(z) \cdot W$ we’re multiplying together to get the gradient here.

Typical initialization of weights is from a Gaussian with **mean zero and standard deviation one**, which will yield mostly weights of **magnitude less than one**.
Why do gradients vanish?

- If $f(x)$ is a sigmoid function, which is quite common, it’s derivative will always be less than 0.25.

- That’s a lot of small numbers being multiplied together, yielding a really, really small number.

- You could also imagine the opposite scenario, where if our weights happened to reach larger values, we’d be multiplying a lot of big numbers, and the gradient would explode rather than vanish.

- Now let’s take a look at how we can Overcome those problems.
How to solve to vanishing gradient problem
The vanishing gradient was arising from multiplying lots of $f'(z) \cdot W$ terms.

This gives us some insight into why certain activation functions might work better than others for combatting this problem.

while the derivative of a **sigmoid function** is < 0.25 everywhere, making each term even smaller, the **derivative of the ReLU function** is **one** at every point above zero, creating a more stable network.
Activation functions

- ReLU Derivative
- Sigmoid Derivative
- Tanh Derivative
Clipping Gradients

- A simple solution for **Exploding** gradients: just scale them down whenever they pass above a certain threshold.

- It is very easy to identify exploding gradient in a running program:

  - our gradients will become NaN (not a number) and your program will crash.

- Clipping the gradients at a pre-defined threshold 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.
LSTM – Long Short Term Memory
LSTM

- LSTM (Long Short-Term Memory Networks) are a special subset of RNNs that are able to deal with remembering information for **much longer periods of time**

- LSTMs were designed to combat **vanishing gradients** through a **gating mechanism**.

- To understand what this means, let’s look at how a LSTM calculates a hidden state $S_t$:

\[
\begin{align*}
    i &= \sigma(x_t U^i + s_{t-1} W^i) \\
    f &= \sigma(x_t U^f + s_{t-1} W^f) \\
    o &= \sigma(x_t U^o + s_{t-1} W^o) \\
    g &= \tanh(x_t U^g + s_{t-1} W^g) \\
    c_t &= c_{t-1} \odot f + g \odot i \\
    s_t &= \tanh(c_t) \odot o
\end{align*}
\]
First, notice that a LSTM layer is just another way to compute a hidden state.

Previously, we computed the hidden state as:

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

The inputs to this unit were $x_t$, the current input at step $t$ and $s_{t-1}$ the previous hidden state.

The output was a new hidden state $s_t$.

A LSTM unit does the exact same thing, just in a different way! This is key to understanding the big picture.
- You can essentially treat LSTM unit as a black box.
- Given the current input and previous hidden state, they compute the next hidden state in some way.
let’s try to get an intuition for how a LSTM unit computes the hidden state:

- i, f, o are called the **input, forget and output gates**, respectively.

- Note that they have the exact same equations, just with different parameter matrices.

- They care called gates because the **sigmoid function squashes** the values of these vectors between 0 and 1.

- And by multiplying them elementwise with another vector you define how much of that other **vector you want to “let through”**
The **input gate** defines how much of the **newly computed state for the current input** you want to let through.

\[ i = \sigma(x_t U^i + s_{t-1} W^i) \]

The **forget gate** defines how much of the **previous state** you want to let through.

\[ f = \sigma(x_t U^f + s_{t-1} W^f) \]

The **output gate** defines how much of the **internal state** you want to expose to the **external network** (higher layers and the next time step).

\[ o = \sigma(x_t U^o + s_{t-1} W^o) \]

The **Tanh** function is applied to the **internal state**.

\[ \text{Tanh}(C_t) \]

\[ i = \sigma(x_t U^i + s_{t-1} W^i) \]

\[ f = \sigma(x_t U^f + s_{t-1} W^f) \]

\[ o = \sigma(x_t U^o + s_{t-1} W^o) \]

\[ g \]

\[ \text{Tanh}(C_t) \]

\[ \text{St} \]

\[ Ct-1 \]

All the gates have the same dimensions.
\[ g = \tanh(x_t U^g + s_{t-1} W^g) \]

- **g** is a “candidate” hidden state that is computed based on the **current input** and the **previous hidden state**.

- It is exactly the same equation we had in our vanilla RNN, we just renamed the parameters U and W.

- However, instead of taking g as the new hidden state as we did in the RNN, we will use the **input gate** from above to pick some of it.

\[
\begin{align*}
    i &= \sigma(x_t U^i + s_{t-1} W^i) \\
    f &= \sigma(x_t U^f + s_{t-1} W^f) \\
    o &= \sigma(x_t U^o + s_{t-1} W^o) \\
    g &= \tanh(x_t U^g + s_{t-1} W^g) \\
    c_t &= c_{t-1} \circ f + g \circ i \\
    s_t &= \tanh(c_t) \circ o
\end{align*}
\]
- \( C_t \) is the **internal memory** of the unit.

- It is a combination of the previous memory \( C_{t-1} \) multiplied by the **forget gate**, and the newly computed hidden state \( g \), multiplied by the input gate.

- Intuitively it is a combination of how we want to combine previous memory and the new input.

- We could choose to ignore the old memory completely (forget gate all 0’s) or ignore the newly computed state completely (input gate all 0’s),

- But most likely we want something in between these two extremes.
Given the memory Ct-1 we finally compute the output hidden state St by multiplying the memory with the output gate.

Not all of the internal memory may be relevant to the hidden state used by other units in the network.
Intuitively, plain RNNs could be considered a special case of LSTMs.

If you fix the input gate all 1’s, the forget gate to all 0’s (you always forget the previous memory) and the output gate to all one’s (you expose the whole memory) you almost get standard RNN.

There’s just an additional tanh that squashes the output a bit.

The gating mechanism is what allows LSTMs to explicitly model long-term dependencies. By learning the parameters for its gates, the network learns how its memory should behave.
References


– https://github.com/harinisuresh/VanishingGradient/blob/master/Vanishing%20Gradient%20Example.ipynb

– http://harinisuresh.com/2016/10/09/lstms/