DATA SCIENCE II: Machine Learning MTH 9899 Baruch College

Lecture 6: Neural Network Topologies

Adrian Sisser

May 10, 2017

Outline

1. Training Tricks
   - Dropout
   - Batch Normalization
2. Neural Network Topologies
   - RNNs

Outline

1. Training Tricks
   - Dropout
   - Batch Normalization
2. Neural Network Topologies
   - RNNs

Dropout

Dropout is a very effective technique to avoid overfitting in deep NNs. The idea is to randomly 'dropout' certain units from an NN during training. By randomly selecting which units to drop out, we avoid overfitting and learning spurious patterns from the input data.

$$r \sim \text{Bernouli}(p)$$

$$h_{\text{out},i} = \begin{cases} \mathcal{A}(XW_i + B_i) & \text{for} \quad r = 0 \\ 0 & \text{for} \quad r = 1 \end{cases}$$

- We pick probability p that a given unit is 'dropped' from a given. Usually 0.5 is a good choice.
- We can think of all of this as us training random subset networks of the original
- When we go to do a prediction, we use all nodes, but scale them by the dropout probability, p.

Outline

1. Training Tricks
   - Dropout
   - Batch Normalization
2. Neural Network Topologies
   - RNNs

Batch Normalization is another technique to speed up covnergence of NN training. It is based on the idea that while we normalize overall input values to be $N \sim (0, 1)$, each minibatch might not have the same distribution, and $N \sim (0, 1)$ might not be ideal. Let X be the input matrix for a layer, l and μ_i and σ_i represent the mean and sd of the ith column. We will transform every input as:

$$X' = XW + B$$
$$X''_i = \gamma_i \frac{(X'_i - \mu_i)}{\sigma_i} + \beta_i$$
$$h_{\text{out}} = \mathcal{A}(X'')$$

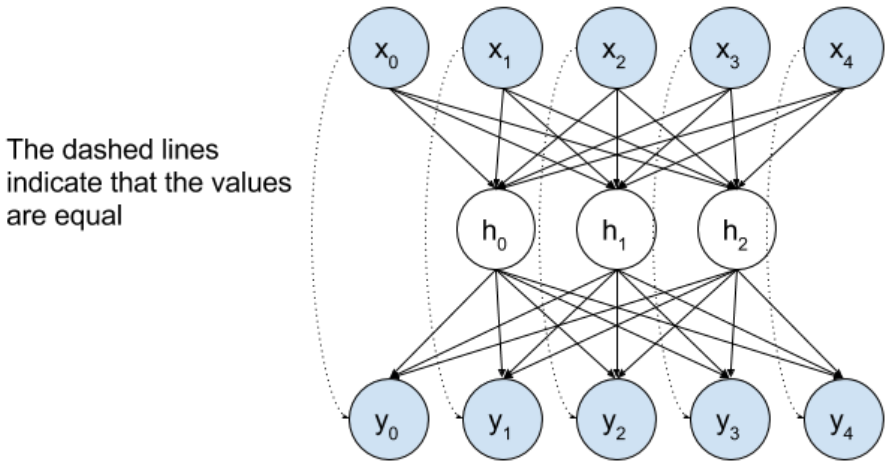
Here, we have transformed each column of X'' to be $N \sim (\beta, \gamma)$.

It is important to note that γ and β are learned parameters here - they help us figure out the optimal scale for training each layer. A few other notes:

- We can trivially extend backpropagation to allow us to learn β and γ
- This is a fairly new technique published in 2015 by Google.
- The initial paper shows training networks more than 10x faster for the same quality
- This improvement combined with existing techniques allowed a significantly lower error rates on ImageNet Classification.

Autoencoders

Autoencoders are a simple type of feedforward network. The goal is to train a simple 1 layer hidden network with the same data for x and y, and so learn to 'recreate' the input. The hope is to learn high-level features that describe the input.



What would an autoencoder with a small h and a linear activation function be doing?

PCA!!

By choosing more interesting functions other than linear, we can get more interested compressed representations.

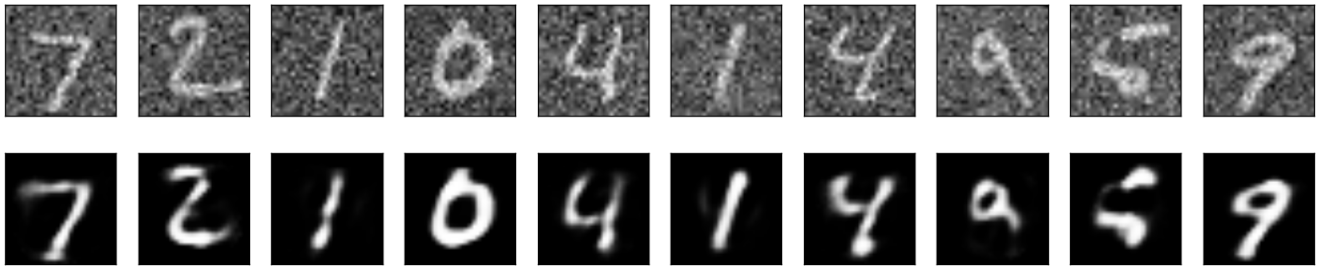
We have a few ways to constrain our Autoencoder.

- We can control the size of the hidden layer. By making h smaller than the input, we force the autoencoder to only learn essential features of the data.
- We can force the autoencoder to be *small*, ie $\mathcal{L}' = \mathcal{L} + \|W\|_2$.
- We can force the autoencoder to be *sparse*, ie $\mathcal{L}' = \mathcal{L} + \|W\|_1$.

Let's look at an example. We'll train an autoencoder on MNIST data. A few notes:

- We'll be looking at 28 x 28 grayscale images.
- We are shrinking these 784 pixels down to only 32 hidden units
- We trained for 50 epochs, using cross-entropy loss on a pixel-by-pixel basis.

Autoencoders are also very good at getting rid of noise:



There are a few twists on Autoencoders:

- Denoising Autoencoders - We deliberately 'corrupt' our input data and train on that.
- Deep Autoencoders - We can train our autoencoder using more than one hidden layer.
- Stacked Autoencoders - We can train a deep network by layering autoencoders together, where we use the hidden vector representation of each autoencoder as the input(/output) of the next layer.

Outline

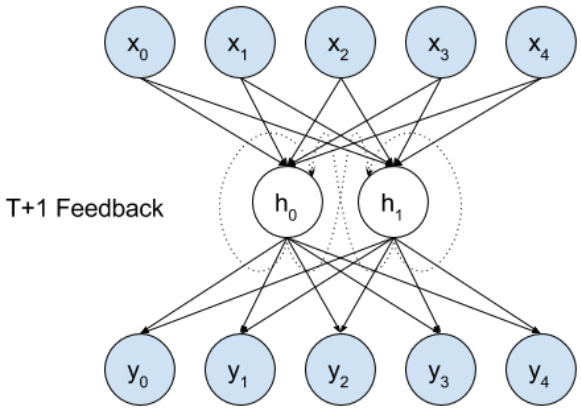
1. Training Tricks
   - Dropout
   - Batch Normalization
2. Neural Network Topologies
   - RNNs

Recurrent NNs

So far, the networks we have talked about have been straightforward "Feed-Forward Networks". That is to say, they have been an directed acyclic graph, where information only flows in one direction. It's natural to wonder how we can use NNs to learn about time-based patterns. A few examples would be:

- Financial Time Series - We are trying to fit the future return of a stock to the past n days return.
- Video - We want to identify what's happening in a frame of video - it's natural to want to know what prior frames looked like.
- Text - If we want to predict the next word in a paragraph, we need to know the prior words.
- Speech - To recognize a sentence, it's useful to know the words that come before ... and after.

Now, we will talk about Recurrent NNs (RNNs). The basic idea is that values from our hidden layers are fed back in at the next time cycle. This allows us to model past inputs.



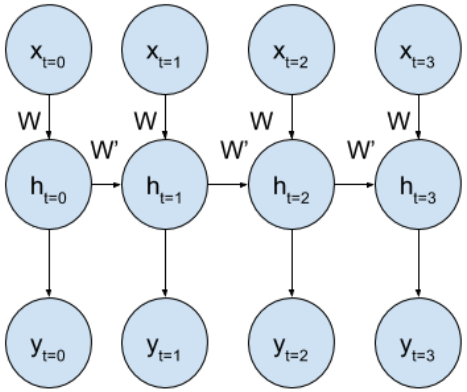
Simple RNN

So one obvious question is, how do we train this? The network has a loop in it!

The idea here is to *unroll* our network across time. We will keep the same W matrix at each time.

For simplicity, we have compressed each layer of each time step into a single 'node'. The subscripts represent a time index.

NOTE: W MATRIX IS THE SAME IN EACH STEP

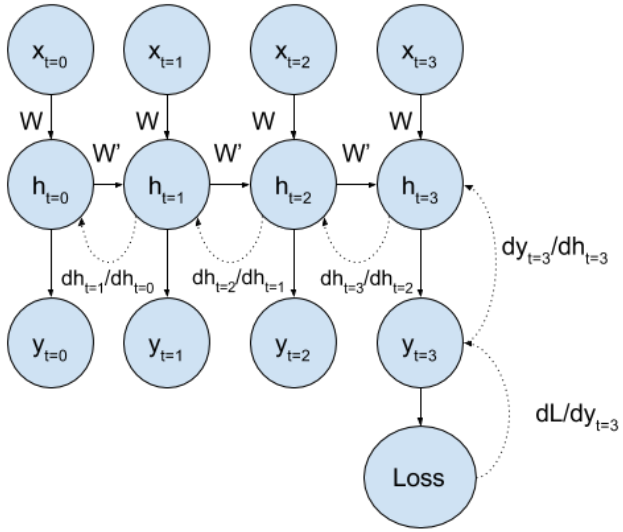


BPTT

Now that we've unrolled our network, we can again do a simple backpropagation algorithm. We call this Backpropagation Through Time or BPTT.

For simplicity, we have compressed each layer of each time step into a single 'node'. The subscripts represent a time index.

NOTE: W MATRIX IS THE SAME IN EACH STEP



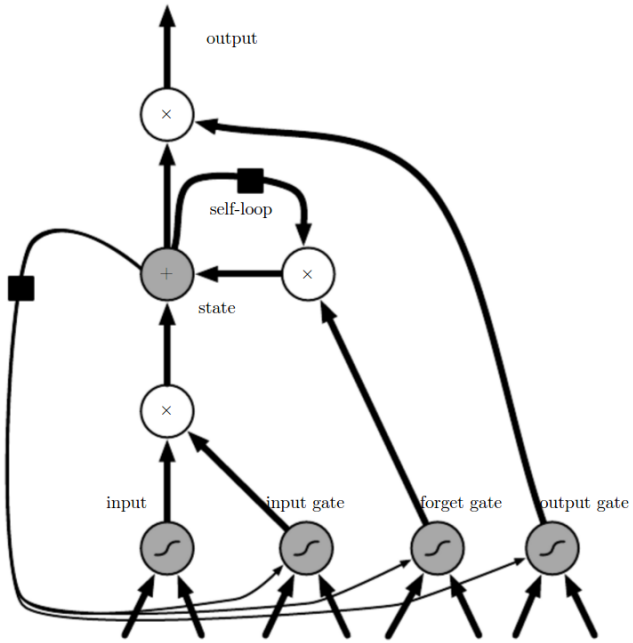
There is one big problem with the simple RNN model that we've discussed so far ... **it doesn't work**!

In practice, this model of RNNs suffers from the same Vanishing/Exploding Gradient problem that we spoke about. If we think of a simple model that is trying to learn the next word in a paragraph, we could easily need a memory going back 20+ words. As we already know, this is extremely hard.

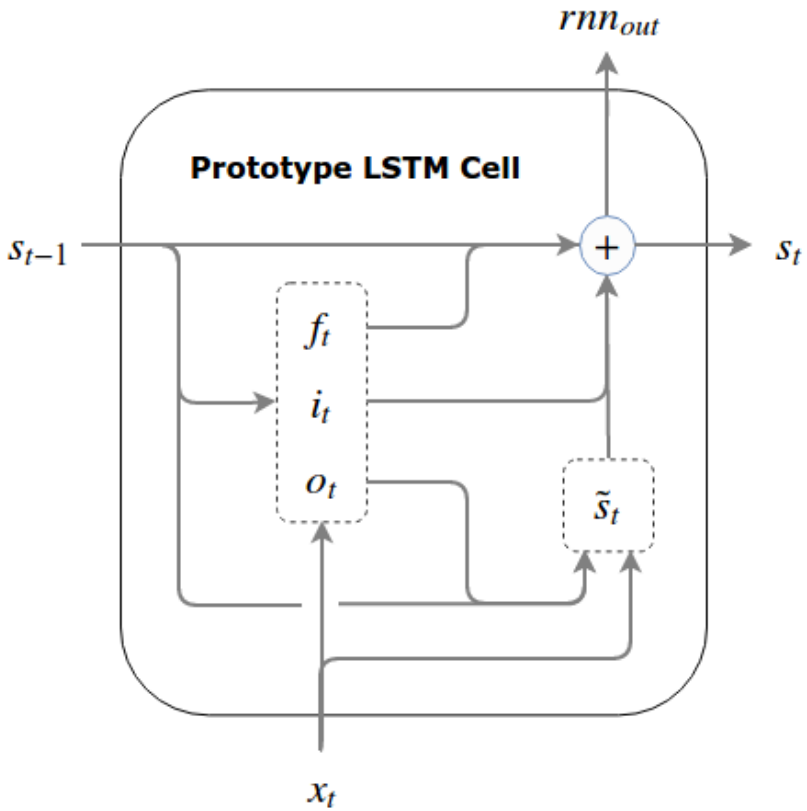
$$\frac{\partial L}{\partial h_{t=0}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_{t=3}}\frac{\partial h_{t=3}}{\partial h_{t=2}}\frac{\partial h_{t=2}}{\partial h_{t=1}}\frac{\partial h_{t=1}}{\partial h_{t=0}}$$

LSTM

One solution that was developed to solve these issues with simple RNNs is the confusingly named LSTM - Long Short Term Memory model. LSTMs look incredibly complicated, but can be understood with a few minutes of thinking about it.



[1] Source: http://www.deeplearningbook.org/contents/rnn.html


rnn_{out}
Prototype LSTM Cell
s_{t-1}
+
s_t
f_t
i_t
o_t
$\tilde{s}_t$
x_t

LSTM

Below, we will use c to represent the 'state' of the cell, and H to represent hidden layer values.

Forget Gate The forget gate controls whether or not we discard the prior memory state, c:

$$f_t = \sigma(X_t W_{f,x} + H_{t-1} W_{f,H} + b_f)$$

Input Gate The input gate controls how much the new incoming information gets incorporated into the memory state:

$$i_t = \sigma(X_t W_{i,x} + H_{t-1} W_{i,H} + b_i)$$

LSTM

Output Gate The output gate controls whether or not the current memory state is output:

$$o_t = \sigma(X_t W_{o,x} + H_{t-1} W_{o,H} + b_o)$$

Memory Update The memory cell is updated based on how much we 'forget' and how much we accept new 'input':

$$c_t = f_t \circ c_{t-1} + i_t \circ \mathcal{A}_{\text{tanh}}(X_t W_{c,x} + H_{t-1} W_{c,H} + b_c)$$

Hidden Layer Output The hidden layer output is based on the 'output' gate and the new state:

$$h_t = o_t \circ \mathcal{A}_{\text{tanh}}(c_t)$$

So how do LSTMs help? Well, the hidden state going out is a function of the current memory cell, c. As long as the forget get, f, stays near one, we keep our memory state around.
This is much easier than the vanishing gradients of a 'vanilla' RNN. We also add in the new changes to c, rather than multiply, so they propagate backwards better.