A Tutorial on Deep Learning
Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks

Quoc V. Le
qvl@google.com
Google Brain, Google Inc.
1600 Amphitheatre Pkwy, Mountain View, CA 94043

October 20, 2015

1 Introduction

In the previous tutorial, I discussed the use of deep networks to classify nonlinear data. In addition to their ability to handle nonlinear data, deep networks also have a special strength in their flexibility which sets them apart from other tranditional machine learning models: we can modify them in many ways to suit our tasks. In the following, I will discuss three most common modifications:

- Unsupervised learning and data compression via autoencoders which require modifications in the loss function,
- Translational invariance via convolutional neural networks which require modifications in the network architecture,
- Variable-sized sequence prediction via recurrent neural networks which require modifications in the network architecture.

The flexibility of neural networks is a very powerful property. In many cases, these changes lead to great improvements in accuracy compared to basic models that we discussed in the previous tutorial.

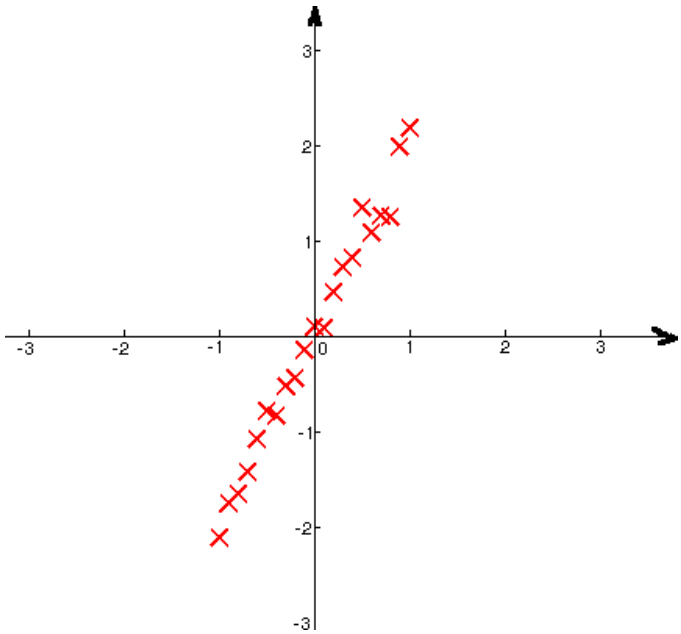
In the last part of the tutorial, I will also explain how to parallelize the training of neural networks. This is also an important topic because parallelizing neural networks has played an important role in the current deep learning movement.

2 Autoencoders

One of the first important results in Deep Learning since early 2000 was the use of Deep Belief Networks [15] to pretrain deep networks. This approach is based on the observation that random initialization is a bad idea, and that pretraining each layer with an unsupervised learning algorithm can allow for better initial weights. Examples of such unsupervised algorithms are Deep Belief Networks, which are based on Restricted Boltzmann Machines, and Deep Autoencoders, which are based on Autoencoders. Although the first breakthrough result is related to Deep Belief Networks, similar gains can also be obtained later by Autoencoders [4]. In the following section, I will only describe the Autoencoder algorithm because it is simpler to understand.

2.1 Data compression via autoencoders

Suppose that I would like to write a program to send some data from my cellphone to the cloud. Since I want to limit my network usage, I will optimize every bit of data that I am going to send. The data is a collection of data points, each has two dimensions. A sample of my data look like the following:



Here, the red crosses are my data points, the horizontal axis is the value of the first dimension and the vertical axis is the value of the second dimension.

Upon visualization, I notice is that the value of the second dimension is approximately twice as much as that of the first dimension. Given this observation, we can send only the first dimension of every data point to the cloud. Then at the cloud, we can compute the value of the second dimension by doubling the value of the first dimension. This requires some computation, and the compression is lossy, but it reduces the network traffic by 50%. And since network traffic is what I try to optimize, this idea seems reasonable.

So far this method is achieved via visualization, but the bigger question is can we do this more systematically for high-dimensional data? More formally, suppose we have a set of data points $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ where each data point has many dimensions. The question becomes whether there is a general way to map them to another set of data points $\{z^{(1)}, z^{(2)}, ..., z^{(m)}\}$, where z's have lower dimensionality than x's and z's can faithfully reconstruct x's.

To answer this, notice that in the above process of sending data from my cellphone to the cloud has three steps:

1. Encoding: in my cellphone, map my data $x^{(i)}$ to compressed data $z^{(i)}$.
2. Sending: send $z^{(i)}$ to the cloud.
3. Decoding: in the cloud, map from my compressed data $z^{(i)}$ back to $\tilde{x}^{(i)}$, which approximates the original data.

To map data back and forth more systematically, I propose that z and $\tilde{x}$ are functions of their inputs, in the following manner:

$$z^{(i)} = W_1 x^{(i)} + b_1$$
$$\tilde{x}^{(i)} = W_2 z^{(i)} + b_2$$

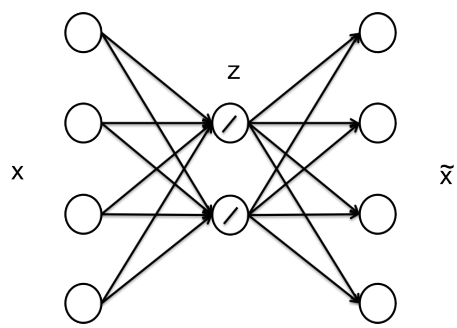
If $x^{(i)}$ is a two-dimensional vector, it may be possible to visualize the data to find W_1, b_1 and W_2, b_2 analytically as the experiment above suggested. Most often, it is difficult to find those matrices using visualization, so we will have to rely on gradient descent.

As our goal is to have $\tilde{x}^{(i)}$ to approximate $x^{(i)}$, we can set up the following objective function, which is the sum of squared differences between $\tilde{x}^{(i)}$ and $x^{(i)}$:

$$\begin{aligned} J(W_1, b_1, W_2, b_2) &= \sum_{i=1}^{m} \left(\tilde{x}^{(i)} - x^{(i)} \right)^2 \\ &= \sum_{i=1}^{m} \left(W_2 z^{(i)} + b_2 - x^{(i)} \right)^2 \\ &= \sum_{i=1}^{m} \left(W_2 \big(W_1 x^{(i)} + b_1 \big) + b_2 - x^{(i)} \right)^2 \end{aligned}$$

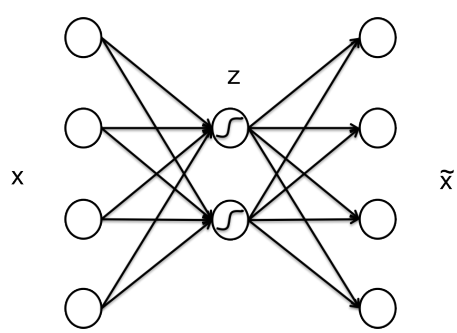
which can be minimized using stochastic gradient descent.

This particular architecture is also known as a linear *autoencoder*, which is shown in the following network architecture:



In the above figure, we are trying to map data from 4 dimensions to 2 dimensions using a neural network with one hidden layer. The activation function of the hidden layer is linear and hence the name linear autoencoder.

The above network uses the linear activation function and works for the case that the data lie on a linear surface. If the data lie on a nonlinear surface, it makes more sense to use a nonlinear autoencoder, e.g., one that looks like following:



If the data is highly nonlinear, one could add more hidden layers to the network to have a deep autoencoder.

Autoencoders belong to a class of learning algorithms known as *unsupervised* learning. Unlike supervised algorithms as presented in the previous tutorial, unsupervised learning algorithms do not need labeled information for the data. In other words, unlike in the previous tutorials, our data only have x's but do not have y's.

2.2 Autoencoders as an initialization method

Autoencoders have many interesting applications, such as data compression, visualization, etc. But around 2006-2007, researchers [4] observed that autoencoders could be used as a way to "pretrain" neural networks.

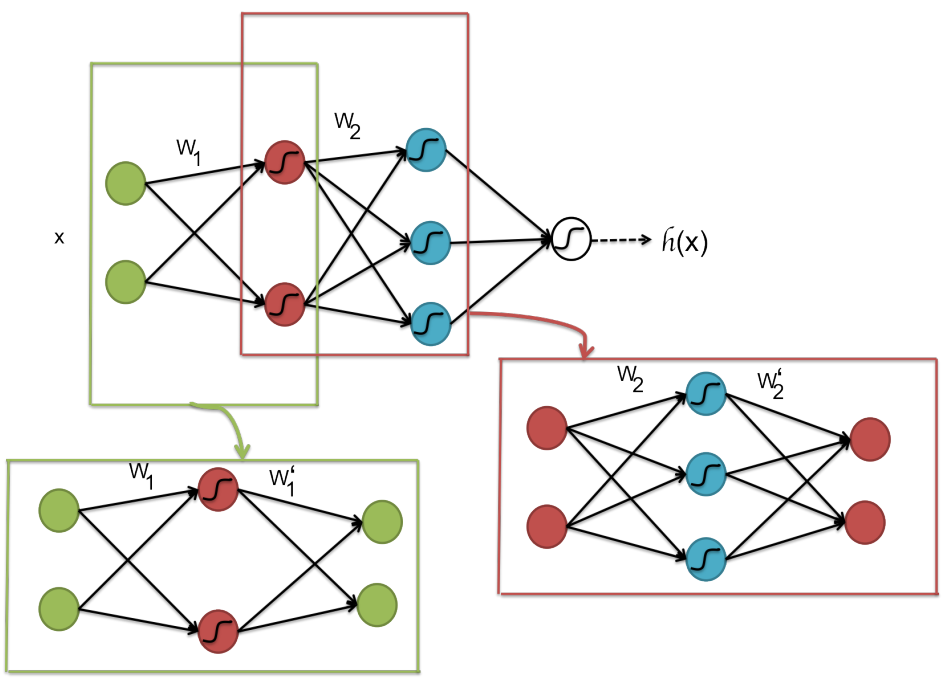
Why? The reason is that training very deep neural networks is difficult:

- The magnitudes of gradients in the lower layers and in higher layers are different,
- The landscape or curvature of the objective function is difficult for stochastic gradient descent to find a good local optimum,
- Deep networks have many parameters, which can remember training data and do not generalize well.

The goal of pretraining is to address the above problems. With pretraining, the process of training a deep network is divided in a sequence of steps:

- Pretraining step: train a sequence of shallow autoencoders, greedily one layer at a time, using unsupervised data,
- Fine-tuning step 1: train the last layer using supervised data,
- Fine-tuning step 2: use backpropagation to *fine-tune* the entire network using supervised data.

While the last two steps are quite clear, the first step needs needs some explanation, perhaps via an example. Suppose I would like to train a relatively deep network of two hidden layers to classify some data. The parameters of the first two hidden layers are W_1 and W_2 respectively. Such network can be pretrained by a sequence of two autoencoders, in the following manner:



More concretely, to train the red neurons, we will train an autoencoder that has parameters W_1 and W_1'. After this, we will use W_1 to compute the values for the red neurons for all of our data, which will then be used as input data to the subsequent autoencoder. The parameters of the decoding process W_1' will be discarded. The subsequent autoencoder uses the values for the red neurons as inputs, and trains an autoencoder to predict those values by adding a decoding layer with parameters W_2'.

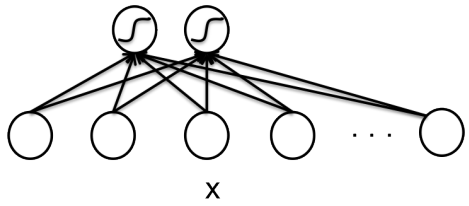
Researchers have shown that this pretraining idea improves deep neural networks; perhaps because pretraining is done one layer at a time which means it does not suffer from the difficulty of full supervised learning. From 2006 to 2011, this approach gained much traction as a scientific endeavor because the brain is likely to unsupervised learning as well. Unsupervised learning is also more appealing because it makes use of *inexpensive* unlabeled data. Since 2012, this research direction however has gone through a relatively quiet period, because unsupervised learning is less relevant when a lot of labeled data are available. Nonetheless, there are a few recent research attempts to revive this area, for example, using variational methods for probabilistic autoencoders [24].

3 Convolutional neural networks

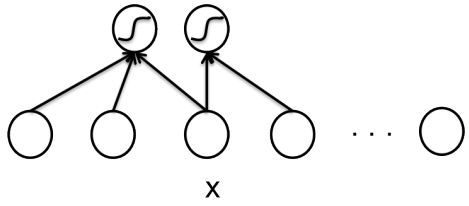
Since 2012, one of the most important results in Deep Learning is the use of convolutional neural networks to obtain a remarkable improvement in object recognition for ImageNet [25]. In the following sections, I will discuss this powerful architecture in detail.

3.1 Using local networks for high dimensional inputs

In all networks that you have seen so far, every neuron in the first hidden layer connects to all the neurons in the inputs:



This does not work when x is high-dimensional (the number of bubbles is large) because every neuron ends up with many connections. For example, when x is a small image of 100x100 pixels (i.e., input vector has 10,000 dimensions), every neuron has 10,000 parameters. To make this more efficient, we can force each neuron to have a small number of connections to the input. The connection patterns can be designed to fit some structure in the inputs. For example, in the case of images, the connection patterns are that neurons can only look at adjacent pixels in the input image:

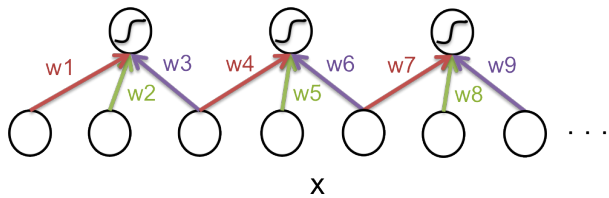


We can extend this idea to force local connectivity in many layers, to obtain a deep locally connected network. Training with gradient descent is possible because we can modify the backpropagation algorithm to deal with local connectivity: in the forward pass, we can compute the values of neurons by assuming that the empty connections have weights of zeros; wheareas in the backward pass, we do not need to compute the gradients for the empty connections.

This kind of networks has many names: local networks, locally connected networks, local receptive field networks. The last name is inspired by the fact that neurons in the brain are also mostly locally connected, and the corresponding terminology in neuroscience/biology is "local receptive field."

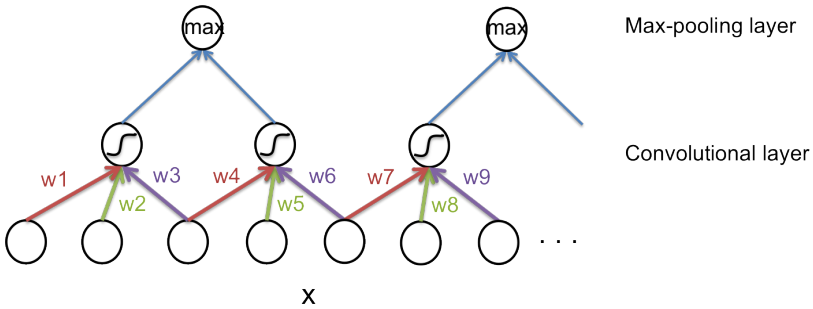
3.2 Translational invariance with convolutional neural networks

In the previous section, we see that using locality structures significantly reduces the number of connections. Even further reduction can be achieved via another technique called "weight sharing." In weight sharing, some of the parameters in the model are constrained to be *equal* to each other. For example, in the following layer of a network, we have the following constraints $w_1 = w_4 = w_7$, $w_2 = w_5 = w_8$ and $w_3 = w_6 = w_9$ (edges that have the same color have the same weight):



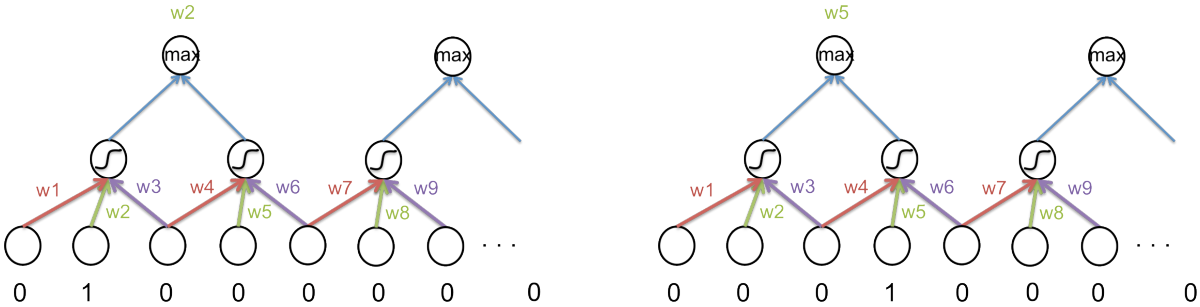
With these constraints, the model can be quite compact in terms of number of actual parameters. Instead of storing all the weights from w_1 to w_9, we only need to store w_1, w_2, w_3. Values for other connections can then be derived from these three values.

This idea of sharing the weights resembles an important operation in signal processing known as convolution. In convolution, we can apply a "filter" (a set of weights) to many positions in the input signals. In practice, this type of networks also comes with another layer known as the "max-pooling layer." The max-pooling layer computes the max value of a selected set of output neurons from the convolutional layer and uses these as inputs to higher layers:



An interesting property of this approach is that the output of the max-pooling neurons are invariant to shifts in the inputs. To see this, consider the following two examples: $x_1 = [0, 1, 0, 0, 0, 0, 0, ...]$ and $x_2 = [0, 0, 0, 1, 0, 0, 0, ...]$ which can be thought of as two 1D input images, each with one white dot. Two images are the same, except that in x_2, the white dot gets shifted two pixels to the right compared to x_1.

We can see that as the dot moves 2 pixels to the right, the value of the first max pooling neuron changes from w_2 to w_5. But as $w_2 = w_5$, the value of the neuron is unchanged:



This property, that the outputs of the system are invariant to translation, is also known as translational invariance. Translational invariant systems are typically effective for natural data (such as images, sounds

etc.) because a major source of distortions in natural data is typically translation.

This type of networks is also known as convolutional neural networks (sometimes called convnets). The max pooling layer in the convolutional neural networks is also known as the subsampling layer because it dramatically reduces the size of the input data. The max operation can sometimes be replaced by the average operation.

Many recent convolutional neural networks also have another type of layers, called Local Contrast Normalization (LCN) [19]. This layer operates on the outputs of the max-pooling layer. Its goal is to subtract the mean and divide the standard deviation of the incoming neurons (in the same manner with max-pooling). This operation allows brightness invariance, which is useful for image recognition.

We can also construct a deep convolutional neural networks by treating the outputs of a max-pooling layer (or LCN layer) as a new input vector, and adding a new convolutional layer and a new max-pooling layer (and maybe a LCN layer) on top of this vector.

Finally, it is possible to modify the backpropagation algorithm to work with these layers. For example, for the convolutional layer, we can do the following steps:

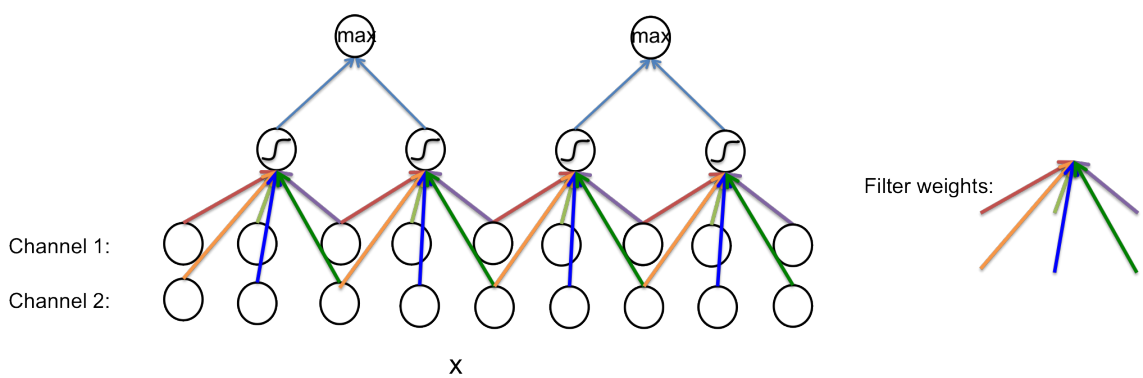
- In the forward pass, perform explicit computation with all the weights, $w_1, w_2, ..., w_9$ (some of them are the same),
- In the backward pass, compute the gradient $\frac{\partial J}{\partial w_1}, ..., \frac{\partial J}{\partial w_9}$, for all the weights,
- When updating the weights, use the average of the gradients from the shared weights, e.g., $w_1 = w_1 - \alpha(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7})$, $w_4 = w_4 - \alpha(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7})$ and $w_7 = w_7 - \alpha(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7})$

For the max-pooling layer, in the forward pass, we need to remember what branch gives the max value, so that in the backward pass, we only compute the gradient for that branch.

3.3 Convolutional neural networks with multi-channel inputs

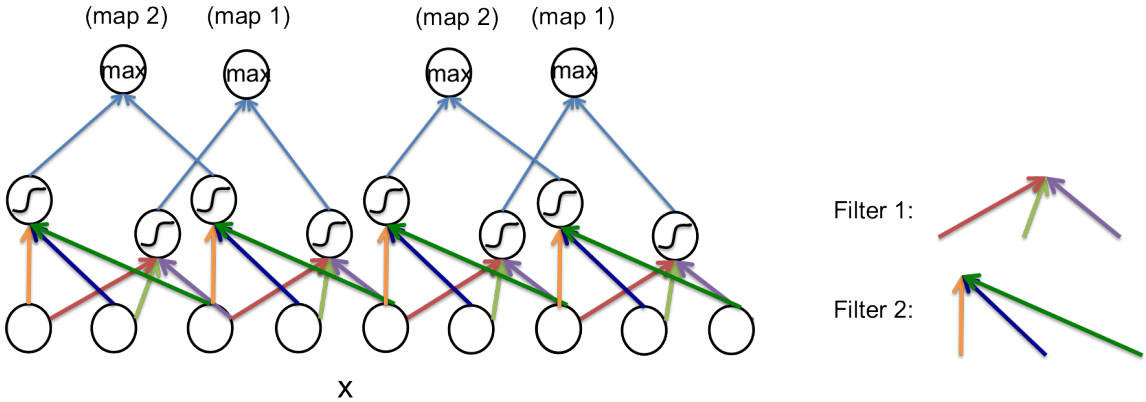
Images typically have multiple channels (for example, Red Green Blue channels). It is possible to modify the convolutional architecture above to work with multiple channel inputs. The modification is essentially to have a filter that looks at multiple channels. The weights are often not shared across channel.

The following figure demonstrates a convolutional architecture for an image with two channels:



3.4 Convolutional neural networks with multiple maps

The current convolutional architecture has only one filter per position of the input. We can extend that to have many filters per location. For instance, at one location we can have two filters looking at exactly the same input (to keep things simple, we only have one input channel):



Each set of output produced by each filter is called a "map." In the example above, the outputs of the two filters create two maps. Map 1 is created by the first filter and map 2 is created by the second filter. Finally, to pass information forward, we can treat the output as an image with multiple channels where an output map is an input channel.

3.5 Some practical considerations when implementing convolutional neural networks

- In contrast to what was discussed above, images are typically two-dimensional. It's easy to modify the above architecture to deal with two-dimensional inputs: each filter has two dimensions. If the input images have many channels, then each filter is essentially three-dimensional: row-column-channel.
- So far, a convnet architecture only considers input with fixed size, for example, all images have to be 100x100 pixels. In reality, however, images may have many sizes. To deal with that, it is typical to crop the images at the center and convert all images to the desired size.
- Many state-of-the-art convnets are sequences of processing blocks where each block is a combination of convolution, max pooling, LCN. Finally, there is one or two fully connected hidden layers that connects the output of the last LCN units to the classifier. Dropout (see previous tutorial) is usually applied at these fully connected layers.
- Many libraries have fast convolution operations, whose implementation may dynamically decide whether to convert the convolution step to an FFT operation or not. For example, in MATLAB, there are *conv* for 1D convolution and *conv2* for 2D convolution, which convert the the convolution operation to FFT, depending on the size of the image and the filter. These operations are typically faster than implementing the explicit for loop to compute the forward pass and backward pass.

Convolutional neural networks are undergone rapid developments. Since the breakthrough work of [25], many novel architectures have been proposed, which result in improvements in object recognition performances [35, 38]. As it is difficult to implement and train a good convolutional neural network, I recommend using some existing software packages. An excellent implementation of state-of-the-art convolutional models and training recipes is Caffe (`http://caffe.berkeleyvision.org/`) [21].

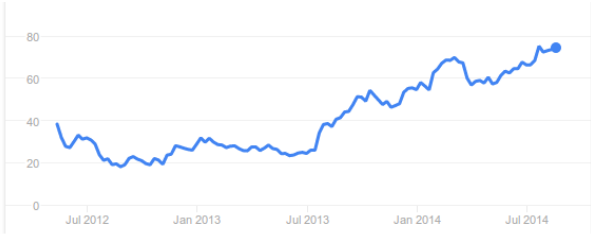
4 Sequence prediction with recurrent neural networks

In the following sections, I will discuss a simple modification to neural networks such that they can work with time series data.

4.1 Sequence prediction

Suppose I would like to predict the stock prices of tech companies tomorrow. As a start, I search for the stock history of several companies and analyze them. The following are the performances of two example companies, A and B:

Company A



Company B

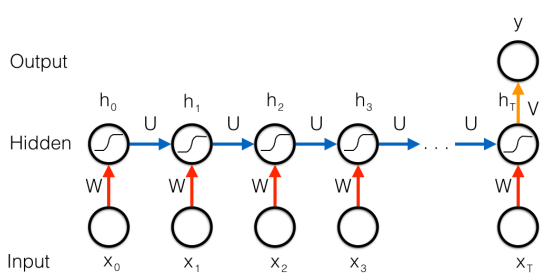
My goal is to build a model to predict the stock market of any company given their historical data. The training data is the stock history that I've collected. So, what is input x and output y? To simplify matters, I decide that for a given company, the input x is all the stock prices up to yesterday, and output y is the stock price of today. So essentially, the input x is $[x_0, x_1, ..., x_T]$, where x_i is the stock price of day i-th since IPO and the output y is the stock price of today. Our goal is to train a model to predict y given all the x's. Note that this is an oversimplistic model to predict stock prices, as it does not have access to other information such as founders, news etc.

The challenge is that the inputs are variable-sized: the value of T is different for different companies. For instance, in the above figure, Company B is IPO'ed since June 2012 so its T is larger than that of Company A which is IPO'ed since Many 2014. In fact, none of our solutions so far works with variable-sized inputs. One way to address the variable-sized input problem is to use convolutional neural network where the max pooling is applied to all of the output of the filters below. That way, even though the input is variable-sized, the output is fix-sized (i.e., equal to the number of filters). The problem with this approach is that convolutional neural network, as stated above, is invariant to translation. With such a large max-pooling, losing position information is inevitable.

Translation invariance is acceptable for images because the output of an object recognition system should be invariant to translation. In contrast, in our stock prediction model, this is an undesirable property, because we want to make use of the precise temporal information (the stock price today is likely to be more influenced by the stock price yesterday than 10 years ago). A proper way to deal with variable-sized inputs is to use a Recurrent Neural Network, as described below.

4.2 Recurrent Neural Networks

A recurrent neural network for our stock prediction task should look like this:



Here, as mentioned above, $x_0, x_1, ..., x_T$ are stock prices of a company up to today. $h_0, h_1, ..., h_T$ are the hidden states of the recurrent network. Note also that the bubbles in this figure indicate a layer.

For a recurrent neural network, there are typically three sets of parameters: the input to hidden weights (W), the hidden to hidden weights (U), and the hidden to label weight (V). Notice that all the W's are shared, all the U's are shared and all the V's are shared. The weight sharing property makes our network suitable for variable-sized inputs. Even if T grows, the size of our parameters stay the same: it's only W, U and V. With these notations, the hidden states are recursively computed as:

$$\begin{aligned} f(x) &= Vh_T \\ h_t &= \sigma(Uh_{t-1} + Wx_t), \text{ for } t = T, .., 1 \\ &... \\ h_0 &= \sigma(Wx_0) \end{aligned}$$

We can then minimize our cost function $(y - f(x))^2$ to obtain the appropriate weights. To compute the gradient of the recurrent neural network, we can use backpropagation again. In this case, the algorithm also has another name: Backpropagation through time (BPTT).

Since there are tied-weights in the network, we can apply the same idea in the convolutional neural network: in the forward pass, pretend that the weights are not shared (so that we have weights $W_0, W_1, ..., W_T$ and $U_0, U_1, ..., U_T$). In the backward pass, compute the gradient with respect to all W's and all U's. The final gradient of W is the sum of all gradients for $W_0, W_1, ..., W_T$; and the final gradient of U is the sum of all gradients for $U_0, U_1, ..., U_T$.

4.3 Gradient clipping

In practice, when computing the gradient for the recurrent neural network, one can find that the gradient is either very large or very small. This can cause the optimizer to converge slowly. To speed up training, it is important to clip the gradient at certain values. For example, any dimension of the gradient should be smaller than 1, if the value of a dimension is larger than 1 we should set it to be 1.

4.4 Recurrent Neural Network for Language Modeling

An important contribution of recurrent neural networks is in the area of language modeling [32], an important task in natural language processing. The goal of language modeling is simple, given the previous words in a text document, predict the next word. For example, to predict the word that comes after these contexts:

I go to ...
I play soccer with ...
The plane flew within 40 feet of ...

Unlike many previous tasks, we have to deal with words instead of numbers for the inputs and outputs. It turns out that it is possible to convert a word to a vector, as follows. First, we will construct a dictionary of all possible words in our text data. Every word is then associated with an index, for example, *a* is associated with index 1; *the* is associated with 10,000 and *zzzz* is associated with 20,000 (the last entry in the dictionary). Using this dictionary, every word is represented as a vector of 20,000 dimensions (the size of the dictionary) and all of its dimensions are zeros but one that has a value of 1. The non-zero dimension corresponds to the index of the word in the dictionary. For example, *the* is represented as $[0, 0, ...0, 1, 0, ...0]$ and the index of 1 is 10,000. This representation is also known as the one-of-k encoding. With this numerical representation, we can use recurrent neural network to learn a model to predict the next word.

Interestingly, if we multiply the input with W we essentially select a column of matrix W. The resulting vector is also known as a "word vector." If we visualize the word vectors (maybe with Principal Component Analysis), we will see that words that have similar meanings have similar vectors. This property makes word vectors attractive for many language processing tasks.

4.5 Other methods to learn word vectors

Word vectors are, in a sense, an orthogonal idea to recurrent neural networks. There are many architectures that can learn word vectors, e.g., convolutional neural networks. In fact, one of the best word vector methods in *word2vec* package uses a simple convolutional network which averages the hidden representation of a context to predict the middle word (the CBOW model) or uses the hidden representation of the middle word to predict the surrounding context (the Skipgram model) [30]. These word vectors are better than 1-of-k encoding when plugged into other applications because word vectors keep better semantics of the words. One can also extend word vectors to paragraph vectors [27], which can represent a document as a vector.

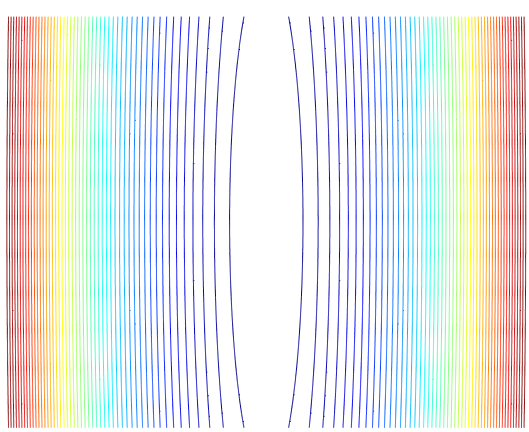
That said, if the purpose is to predict the next word given the previous words, then recurrent neural networks usually work better.

4.6 Long Short Term Memory Networks

Backpropagation through time (BPTT) for recurrent neural networks is usually difficult due to a problem known as vanishing/exploding gradient [16]: the magnitude of the gradient becomes extremely small or extremely large towards the first or the last time steps in the network. This problem makes the training of recurrent neural network challenging, especially when there are a lot of long term dependencies (the output prediction is influenced by long-distance pieces of information in the inputs).

Let's first talk about the consequence of this. Suppose we want to predict the movie rating after reading a one-sentence English review. Since most English sentences begin with the word "the" and since the gradient becomes small towards the beginning of the sentence, the word vector for "the" will hardly be changed during the optimization. This is an undesirable property as we would like fair updates for all the words in the document.

In terms of optimization, this means that the gradient magnitude in some certain dimension is extremely small, and in some other dimension is extremely large. In such case, the curvature of the objective function in the parameter space looks like a valley, as illustrated by a contour plot of a function with two variables:



A high learning rate is bad because it causes overshooting in some dimension. Most often, we have to set the learning rate extremely small to avoid overshooting and this makes learning extremely slow.

What is the cause? The reason for this vanishing/exploding gradient problem is due to the use of sigmoidal activation functions in recurrent networks. As the error derivatives are backpropagated backwards, it has to be multiplied by the derivative of the sigmoid (or tanh) function which can saturate quickly. One can imagine that the ReLU activation function can help here as its derivative allows for better gradient flow. However, another problem arises as we use ReLU as the activation function: in the forward prop computation, if we are not careful with the initialization of the U matrix, the hidden values can explode or vanish depending on whether the eigenvalues of U are bigger or smaller than 1. Attempts have been made to fix this problem by using identity as part of the transfer function [31, 26] with certain success.

Perhaps the most successful attempt to improve the learning of recurrent networks to date is Long Short Term Memory (LSTM) Recurrent Networks [17, 12]. The idea behind LSTM is to modify the architecture of recurrent networks to allow the error derivatives to flow better. In this tutorial, I will use the formulation of [13] with small modifications to explain the construction of LSTM.

At the heart of an LSTM is the concept of memory cells which act as an integrator over time. Suppose that input data at time t is x_t and the hidden state at the previous timestep is h_{t-1}, then the memory cells at time t have values:

$$m_t = \alpha \odot m_{t-1} + \beta \odot f(x_t, h_{t-1})$$

Where $\odot$ is an element-wise multiplication between two vectors. Just think of m_t as a linearly weighted combination between m_{t-1} and f. A nice property of m_t is that it is computed additively and not associated with any nonlinearity. So if the error derivatives cannot pass through the function f at time step t (to subsequently flow through h_{t-1}), it has an opportunity to propagated backward further through m_{t-1}. In other words, it allows another path for the error derivatives to flow. To prevent m from exploding, f is often associated with a sigmoid/tanh function.

Built on this construct, the hidden state of the network at time t can be computed as following:

$$h_t = \gamma \odot \tanh(m_t)$$

Note again that both h_t and m_t will be used as inputs to the next time steps, and that means the gradient has more opportunities to flow through different paths.

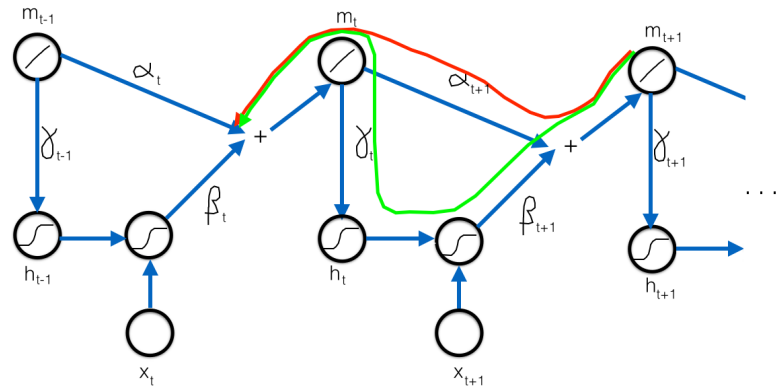
The variables α, β and γ are often called "gates" as they modulate the contribution of input, memory, and output to the next time step. At a time step t, they are computed as:

$$\begin{aligned}
\alpha(t) &= g_1(x_t, h_{t-1}, m_{t-1}) \\
\beta(t) &= g_2(x_t, h_{t-1}, m_{t-1}) \\
\gamma(t) &= g_3(x_t, h_{t-1}, m_t)
\end{aligned}$$

Usually these functions can be implemented as follows:

$$\begin{aligned}
\alpha(t) &= \sigma(W_{x\alpha}x_t + W_{h\alpha}h_{t-1} + W_{m\alpha}m_{t-1} + b_\alpha) \\
\beta(t) &= \sigma(W_{x\beta}x_t + W_{h\beta}h_{t-1} + W_{m\beta}m_{t-1} + b_\beta) \\
\gamma(t) &= \sigma(W_{x\gamma}x_t + W_{h\gamma}h_{t-1} + W_{m\gamma}m_t + b_\gamma) \\
f(x_t, h_{t-1}) &= \tanh(W_{xm}x_t + W_{hm}h_{t-1} + b_m)
\end{aligned}$$

A simplified figure that shows gradient flowing from timestep $t+1$ to t should look like this:



In the above figure, the green and the red paths are the two paths that gradient can flow back from m_{t+1} to m_t. I want to emphasize that m_t is linearly computed which means the gradient can continue to flow through m_t as well. Hence the green path, which generates nonlinear ouputs, is a "difficult" path for gradient to flow; whereas the red path, which only generates linear functions, is an "easy" path for gradient to flow.

Under this construction, the following properties can be achieved with LSTM:

- It is possible to set up the connections initially such that the LSTM can behave like a convnet, thereby allowing gradients to flow to all timesteps,
- It is also possible to set up the connections initially to recover standard Recurrent Networks. So in the very worst case, we don't lose anything,
- Units in LSTM are mostly bounded and numerically stable. In particular, even though m_t allows values to be added, the added values are bounded between -1 and 1, which is rather small. For example, if we have a very long sequence in our training data with a million timesteps, the value of m_t is bounded between -10^6 and 10^6, which is not too bad.

5 Sequence output prediction with Recurrent Neural Networks

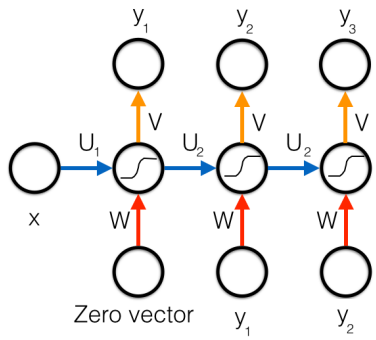
In the above sections, the prediction outputs have always been a scalar (classification, and regression) or a fixed-length vector (autoencoders). But this is restrictive since in many applications the desired outputs can be a variable-length sequence. For example, in the task of machine translation, we would like to predict the target sentence from the source sentence; or in the task of automatic image captioning, we would like to predict the caption given the input image.

A recent advance in Deep Learning is to use a recurrent network as a "dynamic" classifier. By "dynamic," I mean that the classifier can predict more than just fixed-length vectors, which have been the mainstream of machine learning for the past decades. Instead, it can be used to predict a variable-length sequence. This framework of using Recurrent Networks as a variable-sized predictor was proposed in different forms by [22, 7, 37]. In the following, I will use the formulation of [37] to describe the method.

5.1 The basics

Suppose during training, we are presented with an input vector x (a hidden layer in convolutional or recurrent networks), and a target output sequence $y_1 y_2 y_3$. We would like to train a recurrent network to go from x to $y_1 y_2 y_3$.

We know that we can use a recurrent network to predict y_1 from x, but what is the input of y_2? Well, that brings us to the first trick of this paradigm. The trick is that the input of y_2 can just simply be y_1. This means that during training we feed the ground-truth targets as inputs to the next step prediction. This is good because during training, we see the ground-truth targets anyway. Hence, the model architecture can look like this:



(In the above figure, I use a "Zero vector" as input to the first step of the network to keep it simple and consistent with later steps; the first set of weights U_1 and other set of weights U_2 can be different.)

The problem is that during inference, we will not see the ground-truth targets. So what can we use as inputs to make a prediction as the second step? Well, we can do something rather greedy: we simply take the best prediction of the previous timestep as the input to the current timestep. This is sometimes called "greedy search."

The problem with greedy search, as its name suggested, is that it may not be optimal. More concretely, the joint probability of the output sequence may not be highest possible. To do better than greedy search, one can do a "full search:" we use the recurrent networks to compute the joint probability of every possible output sequence. This can guarantee to find the best possible sequence with a large computation cost.

Greedy search and full search lie at two extremes. One is very fast but not optimal, the other is extremely slow but optimal. To be practical, maybe we can do something in between. This brings us to another trick: left-to-right "beam search." In beam search, we can keep a list of k possible sequences sorted by the joint probability (which is computed by multiplying the output probability of each prediction in the sequence produced so far). We will generate output predictions (or "decode") from left to right starting from the first output. During the decode procedure, any sequence that does not belong to the top-k highest joint probability will be removed from our list of candidates. This beam search procedure, proposed first in [37], completes the basics of sequence output prediction. Even though it's not guaranteed for the beam search to achieve the optimal sequence, in practice, the generated sequences in the top-k list are generally very good.

The notion of when to stop producing the output sequence (halting) is interesting and worth some of attention as well. In practice, we process our data to always have a termination symbol at the end of the sequence (i.e., y_3 = "eos" or end-of-sequence symbol). This way, when the beam search arrives at the termination symbol, we will terminate that particular beam.

5.2 The attention model

What's missing from the above analysis is the fact that the input vector x is static but with gradient descent, we can always adjust x by backpropagating through x to the input network. As such it feels alright if the input network, that gives rise to x, takes some other fixed-length vectors as inputs. But a problem arises if the input network takes some variable-length sequences as well. The problem is that we have to push all variable-length information into a fixed-length vector x. A small dimension for x works well for short sequences but less so for longer sequences and vice versa.

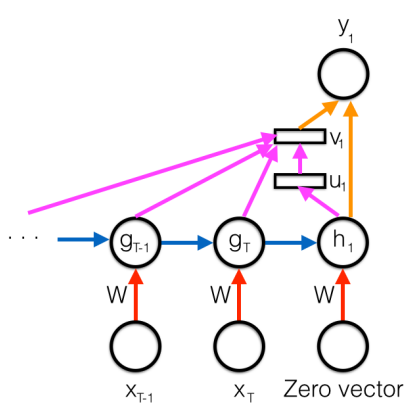
Perhaps a better model is to allow the output recurrent network to pay attention to the certain part of the input network so that it doesn't have to rely on this bottleneck in x. The modification to our model goes as follows. At every time step in the output, we will predict an auxiliary hidden state called u; this u vector will be used to compute the dot product with the hidden states at all timesteps in the input network. These dot products are used as weights to combine all hidden states into a vector v. This vector can be then used as additional inputs to our prediction at the current time step.

More concretely, let's assume that we are at step τ in our output recurrent network. Suppose also that the input sequence is represented by another recurrent network, whose the input sequence has T steps $(x_1, ..., x_T)$. The hidden states of the output network is denoted by h and the hidden states of the input network is denoted by g. Here's the math:

$$
\begin{aligned}
u_\tau &= \Theta h_\tau \\
\beta_t &= u_\tau^T g_t \quad \text{for all t = 1..T} \\
\alpha_t &= \frac{\exp(\beta_t)}{\sum_{t=1}^{T} \exp(\beta_t)} \quad \text{for all t = 1..T} \\
v_\tau &= \sum_{t=1}^{T} \alpha_t g_t \\
[v_\tau, h_\tau] &\rightarrow y_\tau
\end{aligned}
$$

In the last equation, the arrow means that we use the combined vector $[v_\tau, h\tau]$ to predict y_τ. The vector α is also called the alignment vector. Due to the softmax equation in computing α's, their values are bounded between $[0, 1]$. Note also that α's and β's should be indexed by τ but I remove that for simplicity, and readers should understand that these temporary variables are computed differently at different step of τ.

Pictorially, the attention model looks like this:



(I simplified the figure by not drawing α and β, and also assuming that $\tau = 1$, the first step in the output sequence.)

These equations cover the basics, but other variations exist, e.g., having a nonlinearity for u_τ or using v_τ as additional inputs to the next timesteps. Most importantly, during training, all of these variables are adjusted by gradient descent together with other variables in the networks.

In addition to the aforementioned dynamic memory access argument, the attention model can also be interpreted as a way to form shortcut connections to parts of the input data. In any case, it has been shown to be a very effective mechanism for input sequence, output sequence prediction problems. This method was first proposed by [1].

5.3 Applications and Extensions

This idea of using recurrent networks as sequence output prediction and beam search during inference (with and without attention model) paves the way for numerous applications, e.g., in machine translation [29, 20], image captioning [40, 41], parsing [39], and speech recognition [6, 2].

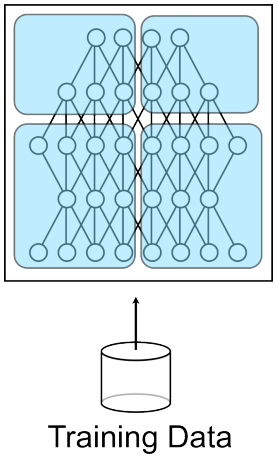
An interesting property of the attention model is that it is a differentiable pooling mechanism of the input data. This differentiable attention mechanism has now been used as pooling mechanism for facts in a database (Memory Networks) [36] and written content in an augmented memory (Neural Turing Machines) [14].

Another interpretation of the attention model is that it allows an $O(T)$ computation per prediction step. So the model itself has $O(T^2)$ total computation (assuming the lengths of input and output sequences are roughly the same). With this interpretation, an alternative approach to the attention model is to lay out the input and output sequences in a grid structure to allow $O(T^2)$ computation. This idea is called Grid-LSTM and was first proposed by [23].

6 Parallelism with neural networks

Training neural networks takes a long time, especially when the training set is large. It therefore makes sense to use many machines to train our neural networks.

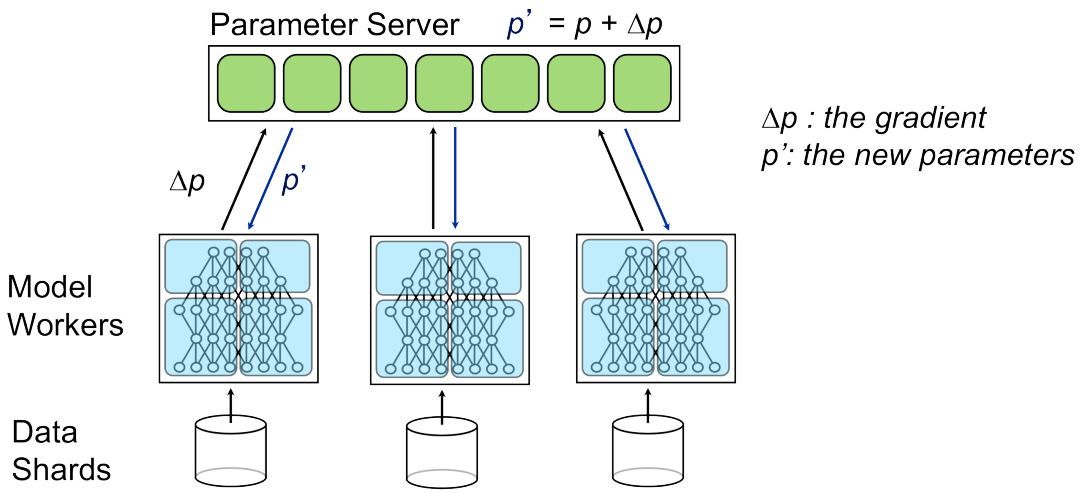
DistBelief [9] is one of the most well-known frameworks for scaling up the training neural networks using many machines. DistBelief has two levels of parallelism: model parallelism and data parallelism. First, every model in DistBelief is partitioned into multiple machines, as suggested by the following figure:



In the above figure, a neural network with four hidden layers and one output layer is partitioned into four machines. There will be communications across machines, but if the network is locally connected, the

communication cost is small compared to the computation cost. This idea of partitioning a model into several machines is known as *model parallelism*.

The second level of parallelism is by partitioning the data into different shards, and training different copies of the model on each shard. To keep them in sync, we can have a set of machines known as the parameter server that stores the parameters. The communication to the parameter server is asynchronous: at every iteration, each copy of the model ("a replica") will compute the gradient on its data, and then send the gradient (Δp) to the parameter server, the parameter server will grab the gradient and update its own parameter copy to arrive at p'. After several updates, the parameter server will broadcast the new parameters to the network. All replicas of the model will replace its parameters with the arriving parameters. This way of parallelizing the training is known as *data parallelism via asynchronous communication*.[1]



The asynchronous communication can be thought of as a soft way to average the gradient without waiting. An advantage of this approach is that it is insensitive to machine slowness: if a replica of the model is slow, it won't delay the entire training.

Maybe less well-known is the third level of parallelism: multi-threading within a machine. In this level, we can allocate several cores to do matrix vector computation, and several cores to do data reading so that once computation is done, data is ready in memory for processing.

7 Recommended readings

Convolutional networks have a long history [11, 28]. They have been used extensively to read handwritten checks and zip codes. Since the work of [25], convolutional neural networks are widely adopted by computer vision researchers.

In this work, I discussed language modeling using recurrent networks. Feedforward networks have also been used for this task with certain success [3] but generally less competitive compared to recurrent networks due to their limitations in handling the word ordering.

This tutorial covers only the very basic optimization strategy: Stochastic Gradient Descent (SGD). It is possible to improve SGD with either Adagrad [10] or Batch Normalization [18]. In Adagrad, we will keep a running norm of the gradient over time and use that to compute a learning rate per dimension. In Batch Normalization, we will compute a running mean and variance of hidden units over time, and use them to "whiten" hidden units over time.

Another topic that I didn't go into great depth but can be useful for neural networks is model ensembling. Model ensembling is a technique where we combine many models by averaging its prediction

[1] The idea of asynchronous SGD was perhaps first proposed by [34].

outputs. These models can be different in terms of random seeds or architectures. This has been used extensively to win competitions (e.g., [25]) or beat hard baselines (i.e., [37]). This technique is also effective for other non deep learning methods.

8 Miscellaneous

Many networks in this tutorial can be difficult to implement. I recommend the readers use Caffe (`http://caffe.berkeleyvision.org/`) [21] or Theano (`http://deeplearning.net/software/theano/`) [5] or Torch7 (`http://torch.ch/`) [8]. To learn word vectors, I recommend *word2vec* (`https://code.google.com/p/word2vec/`) [33].

This tutorial was written as preparation materials for the machine learning summer school at CMU (MLSS'14) with some parts being extended. Videos of the lectures are at `http://tinyurl.com/pduxz2z` .

If you find bugs with this tutorial, please send them to me at qvl@google.com .

9 Acknowledgements

I would like to thank Pangwei Koh, Thai T. Pham and members of the Google Brain team for many insightful comments and suggestions. I am also grateful to many readers who gave various comments and corrections to the tutorial.

References

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. End-to-end attention-based large vocabulary speech recognition. *arXiv preprint arXiv:1508.04395*, 2015.

[3] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[4] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems*, 2007.

[5] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3, 2010.

[6] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.

[7] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[8] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[9] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[10] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[11] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 1980.

[12] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber. Learning precise timing with lstm recurrent networks. *The Journal of Machine Learning Research*, 2003.

[13] A. Graves. Generating sequences with recurrent neural networks. In *Arxiv*, 2013.

[14] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[15] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[16] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*, 2001.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[18] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[19] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *ICCV*, 2009.

[20] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*, 2014.

[21] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[22] N. Kalchbrenner and P. Blunsom. Recurrent continuous translation models. In *EMNLP*, pages 1700–1709, 2013.

[23] N. Kalchbrenner, I. Danihelka, and A. Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

[24] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[26] Q. V. Le, N. Jaitly, and G. E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[27] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014.

[28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[29] T. Luong, I. Sutskever, Q. V. Le, O. Vinyals, and W. Zaremba. Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*, 2014.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[31] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. A. Ranzato. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

[32] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.

[33] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of phrases and their compositionality. In *Advances on Neural Information Processing Systems*, 2013.

[34] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[35] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[36] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

[37] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[39] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton. Grammar as a foreign language. *arXiv preprint arXiv:1412.7449*, 2014.

[40] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[41] K. Xu, J. Ba, R. Kiros, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.