Notes on Deep learning basics

Author: Jung-Shen Kao

December, 2018

Contents

List of Tables

List of Figures

Chapter 1

Neural network

1.1 Elements of Neural Network

In machine learning(ML) , we are given dataset X,Y and the goal is to fit a function $\mathbb{F} : X \to Y$ that is approximately the same as the true function $\mathbb{G} : X \to Y$. Neural network(NN), a special approach of ML, is powerful to do this when $\mathbb{G}$ are highly nonlinear. We first fix our notations in the following: [1]

Input data set : $X = [x^{(1)}x^{(2)}....x^{(m)}]$ which is $n_0 \times m$ matrix
Output data set : $Y = [y^{(1)}y^{(2)}....y^{(m)}]$ which is $n_L \times m$ matrix

The data set are feeding into a Neural network which we denote by

NN Model parameters weights : $w^{[\ell]}$ which is a set of $n_l \times n_{\ell-1}$ matrices.
NN Model parameters bias : $b^{[\ell]}$ which is a set of $n_\ell \times 1$ matrices.
For the above model parameters, ℓ =1,2,...,L

And the feeding process at ℓ-th layer can be described by

[1] We use square superscript [] for layer label and round superscript for data label() , i.e. $[\ell]$ means ℓth layer, and (d) means dth data in the data set

Weighted + bias : $Z^{[\ell]} = w^{[\ell]}X + b^{[\ell]}$, where $Z^{[\ell]} = [z^{[\ell](1)}, z^{[\ell](2)}, ..., z^{[\ell](m)}]$
Neurons (by activations g) : $A^{[\ell]} = g^{[\ell]}(Z^{[\ell]})$ which is $n_\ell \times m$ matrix

Here is an important remarks : we denote number of neurons in ℓ-th layer by n_ℓ. It can be seen that zeroth layer is input data and L-th layer is output data, and the number of datas is m.

With this in hand, which is rather abstract, we can now practically quantify the performance of model upon which we can improve this model.

First we measure the error of model on one single data, say $x^{(d)}$, This quantity can be defined by an error function called Loss Function L[2]

$$L = L(a^{[L](d)}, y^{(d)}, w, b)$$

It's important to note that in the arguments of L, we briefly use w and b instead of $w^{[\ell]}$ and $b^{[\ell]}$ because L is the performance of the model and it's the accumulated contributions of each layer that is of directly relevance.

With L measure the error of the model on a single data $x^{(d)}$, we can define the Cost function to measure the error of the model on whole data set.

$$J(w, b) = \frac{1}{m}\sum_{d=1}^{m} L = \frac{1}{m}\sum_{d=1}^{m} L(a^{[L](d)}, y^{(d)}, w, b)$$

Note that J only depends on properties of model, i.e. parameters of model, since we've summed error over the whole data set.

ML's goal is to minimize J with respect to W,b. Next we introduce one of the major algorithms in ML called gradient descent(GD).

[2]Most commonly used Loss function are square error , cross entropy error which will be applied later for specific examples.

1.2 Gradient Descent Algorithm

In this section, we'll write out all indices explicitly, in contrast to the previous section where we compressed all indices except to layer index and data index.

To minimize J(w,b), GD simply updates parameters w and b by a "small" amount dw and db in each iteration step in the following way

$$w \Rightarrow w = w - \alpha \left(\frac{\partial J}{\partial w} \right)$$

$$b \Rightarrow b = b - \alpha \left(\frac{\partial J}{\partial b} \right)$$

Note that we use brief notation w and b which correspond to L layer with $n_\ell \times n_{\ell-1}$ and $n_\ell \times 1$ elements for each layer. By indices written explicitly, the above equation is

$$w^{[\ell]}_{i_\ell, i_{\ell-1}} \Rightarrow w^{[\ell]}_{i_\ell, i_{\ell-1}} - \alpha \cdot \left(\frac{\partial J}{\partial w^{[\ell]}_{i_\ell, i_{\ell-1}}} \right)$$

$$b^{[\ell]}_{i_\ell} \Rightarrow b^{[\ell]}_{i_\ell} - \alpha \cdot \left(\frac{\partial J}{\partial b^{[\ell]}_{i_\ell}} \right)$$

Thus we have to take partial derivative respect to every single element of them. We do $\frac{\partial J}{\partial w}$ first,

$$\begin{aligned} \left(\frac{\partial J}{\partial w} \right)^{[\ell]}_{i,j} &= \frac{\partial}{\partial w^{[\ell]}_{i,j}} \left[\frac{1}{m} \sum_{d=1}^{m} L(a^{[L](d)}, y^{(d)}, w, b) \right] \\ &= \frac{1}{m} \sum_{d=1}^{m} \frac{\partial}{\partial w^{[\ell]}_{i,j}} L^{(d)}(a^{[L]}, y, w, b) \end{aligned}$$

use chain-rule for the data-wise derivatives and the equation becomes [3]

$$\begin{aligned}\frac{\partial L^{(d)}}{\partial w_{i,j}^{[\ell]}} &= \frac{\partial L^{(d)}}{\partial a_{i_L}^{[L]}}\frac{\partial a_{i_L}^{[L]}}{\partial z_{i_L}^{[L]}}\frac{\partial z_{i_L}^{[L]}}{\partial a_{i_{L-1}}^{[L-1]}}\frac{\partial a_{i_{L-1}}^{[L-1]}}{\partial z_{i_{L-1}}^{[L-1]}}\frac{\partial z_{i_{L-1}}^{[L-1]}}{\partial a_{i_{L-2}}^{[L-2]}}.....\frac{\partial z_{i_{\ell+2}}^{[\ell+2]}}{\partial a_{i_{\ell+1}}^{[\ell+1]}}\frac{\partial a_{i_{\ell+1}}^{[\ell+1]}}{\partial z_{i_{\ell+1}}^{[\ell+1]}}\frac{\partial z_{i_{\ell+1}}^{[\ell+1]}}{\partial a_{i_\ell}^{[\ell]}}\frac{\partial a_{i_\ell}^{[\ell]}}{\partial z_{i_\ell}^{[\ell]}}\frac{\partial z_{i_\ell}^{[\ell]}}{\partial w_{i,j}^{[\ell]}} \\ &= \frac{\partial L^{(d)}}{\partial a_{i_L}^{[L]}}g_{i_L}^{\prime[L]}\frac{\partial z_{i_L}^{[L]}}{\partial a_{i_{L-1}}^{[L-1]}}g_{i_{L-1}}^{\prime[L-1]}\frac{\partial z_{i_{L-1}}^{[L-1]}}{\partial a_{i_{L-2}}^{[L-2]}}.....\frac{\partial z_{i_{\ell+2}}^{[\ell+2]}}{\partial a_{i_{\ell+1}}^{[\ell+1]}}g_{i_{\ell+1}}^{\prime[\ell+1]}\frac{\partial z_{i_{\ell+1}}^{[\ell+1]}}{\partial a_{i_\ell}^{[\ell]}}g_{i_\ell}^{\prime[\ell]}\cdot\delta_{i_\ell,i}\cdot a_j^{[\ell-1]}\end{aligned}$$

in the second line, we use the definition of NN and those terms of the form $\frac{\partial a_{i_\ell}^{[\ell]}}{\partial z_{i_\ell}^{[\ell]}}$ are just the derivative of the ℓ-th layer activation function $g^{[\ell]}$, i.e. $g^{\prime[\ell]}(z_{i_\ell}^{[\ell]})$, and we use the shorthand $g_{i_\ell}^{\prime[\ell]}$ to keep in mind that it's an $n_\ell \times 1$ matrix with index i_ℓ. Also, by definition of NN, the last term $\frac{\partial z_{i_\ell}^{[\ell]}}{\partial w_{i,j}^{[\ell]}}$ reduces to $\delta_{i_\ell,i}\cdot a_j^{[\ell-1]}$ as shown in the second line.

Then we do $\frac{\partial J}{\partial b}$, which is similar to $\frac{\partial J}{\partial w}$ but replacing the last term by $\frac{\partial z_{i_\ell}^{[\ell]}}{\partial b_j^{[\ell]}}$ which equals to $\delta_{i_\ell,j}$

$$\frac{\partial L^{(d)}}{\partial b_j^{[\ell]}} == \frac{\partial L^{(d)}}{\partial a_{i_L}^{[L]}}g_{i_L}^{\prime[L]}\frac{\partial z_{i_L}^{[L]}}{\partial a_{i_{L-1}}^{[L-1]}}g_{i_{L-1}}^{\prime[L-1]}\frac{\partial z_{i_{L-1}}^{[L-1]}}{\partial a_{i_{L-2}}^{[L-2]}}.....\frac{\partial z_{i_{\ell+2}}^{[\ell+2]}}{\partial a_{i_{\ell+1}}^{[\ell+1]}}g_{i_{\ell+1}}^{\prime[\ell+1]}\frac{\partial z_{i_{\ell+1}}^{[\ell+1]}}{\partial a_{i_\ell}^{[\ell]}}g_{i_\ell}^{\prime[\ell]}\cdot\delta_{i_\ell,j}$$

1.2.1 Computing the derivatives - Backward Propagation

In Deep NN, there are terms Forward Propagation and Backward Propagation. These terms are nothing but manipulating the computation blocks by forward and backward directions.

The forward direction is natural for computing from zeroth layer X to final layer $A^{[L]}$ just by sequential matrix multiplications and element-wise multiplications[4].

[3] We've moved the superscript (d) from a^L, y to L for the sake of convenience in later calculations, and will restore it back to all a(also x), y at the end of the calculation of data-wise derivatives

[4] Matrix multiplications account for Z=wA+b and element-wise ones are for A=g(Z).

The backward one is going from $A^{[L]}$ to X which is suitable for gathering computing blocks for derivatives. There are four computing blocks for computer doing this derivative as we can find as follows.

Referring to eq.(?), and imagine this chain can be constructed by several computing blocks. Just like forward propagation, those blocks should be matrices which can be multiplied together to form a new object. The pattern of this chain suggests us to cut at four different points $a^{[\ell]}$, $z^{[\ell]}$, $w^{[\ell]}$ and $b^{[\ell]}$.
For $a^{[l]}$, we have

$$\begin{aligned}\frac{\partial L^{(d)}}{\partial a_j^{[\ell]}} &= \frac{\partial L^{(d)}}{\partial a_{i_L}^{[L]}}\frac{\partial a_{i_L}^{[L]}}{\partial z_{i_L}^{[L]}}\frac{\partial z_{i_L}^{[L]}}{\partial a_{i_{L-1}}^{[L-1]}}\frac{\partial a_{i_{L-1}}^{[L-1]}}{\partial z_{i_{L-1}}^{[L-1]}}\frac{\partial z_{i_{L-1}}^{[L-1]}}{\partial a_{i_{L-2}}^{[L-2]}}\cdots\cdots\frac{\partial z_{i_{\ell+2}}^{[\ell+2]}}{\partial a_{i_{\ell+1}}^{[\ell+1]}}\frac{\partial a_{i_{\ell+1}}^{[\ell+1]}}{\partial z_{i_{\ell+1}}^{[\ell+1]}}\frac{\partial z_{i_{\ell+1}}^{[\ell+1]}}{\partial a_j^{[\ell]}} \\ &= \frac{\partial L^{(d)}}{\partial z_{i_{\ell+1}}^{[\ell+1]}}\frac{\partial z_{i_{\ell+1}}^{[\ell+1]}}{\partial a_j^{[\ell]}} \\ &= \frac{\partial L^{(d)}}{\partial z_{i_{\ell+1}}^{[\ell+1]}}w_{i_{\ell+1},j}^{[\ell+1]}\end{aligned}$$

which is a row vector produced by multiplication of a row vector $\frac{\partial L}{\partial z}$ and a matrix w.
And for $z^{[\ell]}$, we have

$$\frac{\partial L^{(d)}}{\partial z_{i_\ell}^{[\ell]}} = \frac{\partial L^{(d)}}{\partial a_{i_\ell}^{[\ell]}}\frac{\partial a_{i_\ell}^{[\ell]}}{\partial z_{i_\ell}^{[\ell]}} = \frac{\partial L^{(d)}}{\partial a_{i_\ell}^{[\ell]}}\cdot g'^{[\ell]}(z_{i_\ell}^{[\ell]})$$

Clearly this is element-wise multiplication for two matrix $\frac{\partial L^{(d)}}{\partial a^{[\ell]}}$ and $g^{[\ell]}(z^{[\ell]})$
And for $w^{[l]}$,

$$\frac{\partial L^{(d)}}{\partial w_{i,j}^{[\ell]}} = \frac{\partial L^{(d)}}{\partial z_{i_\ell}^{[\ell]}}\frac{\partial z_{i_\ell}^{[\ell]}}{\partial w_{i,j}^{[\ell]}} = \frac{\partial L^{(d)}}{\partial z_{i_\ell}^{[\ell]}}\delta_{i,i_\ell}\cdot a_j^{[\ell-1]} = \frac{\partial L^{(d)}}{\partial z_i^{[\ell]}}\cdot a_j^{[\ell-1]}$$

This is just a multiplication of a column vector and a row vector and thus produce a rectangular matrix.

Lastly, for $b^{[\ell]}$ we obtain

$$\frac{\partial L^{(d)}}{\partial b_j^{[\ell]}} = \frac{\partial L^{(d)}}{\partial z_{i_\ell}^{[\ell]}} \frac{\partial z_{i_\ell}^{[\ell]}}{\partial b_j^{[\ell]}} = \frac{\partial L^{(d)}}{\partial z_{i_\ell}^{[\ell]}} \delta_{i_\ell,j}$$

which just induces a vector $\frac{\partial L}{\partial z^{[\ell]}}$.

Thus, we have taken down the full chain and finding out its structure built by four computing blocks. Now we have all pieces to implement the GD algorithm to all kind of special cases.

1.2.2 Let matrix take cares of data-set

We need to get the whole Cost function's derivatives, i.e. eq.(?). We can make use of matrix to represent $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ in a compact way.

1.2.3 One layer NN - Logistic Regression

We consider the simplest case, one layer NN whose output data is a real number, that is $n_L = 1$ and thus $a^{[L]} = a^{[1]}$, and $z^{[L]} = z^{[1]}$ are 1×1 matrix(scalar). In this case, the derivative have its simplest version

$$\begin{aligned}\frac{\partial L^{(d)}}{\partial w_{i,j}^{[\ell]}} = \frac{\partial L^{(d)}}{\partial w_j^{[1]}} &= \frac{\partial L^{(d)}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial w_j^{[1]}} \\ &= \frac{\partial L}{\partial a^{[1](d)}} \cdot g'(z^{[1](d)}) \cdot x_j^{(d)}\end{aligned}$$

As we pointed above, we restore the data index (d) back to data variable x,z and a in the second line. Before making the computing procedure concrete, we can check the matrix shape which we encounter so far.

Thus we have the derivative of the Cost function

$$\left(\frac{\partial J}{\partial w}\right)_j = \frac{1}{m} \sum_{d=1}^{m} \frac{\partial L}{\partial a^{[1](d)}} \cdot g'(z^{[1](d)}) \cdot x_j^{(d)}$$

by which we can deduce the updating parameters from the simplest version of eq.(?)

$$w_j^{[1]} \Rightarrow w_j^{[1]} - \alpha \cdot \left(\frac{\partial J}{\partial w_j^{[1]}}\right)$$

$$b^{[1]} \Rightarrow b^{[1]} - \alpha \cdot \left(\frac{\partial J}{\partial b^{[1]}}\right)$$

To implement gradient descent algorithm, we switch to computer thinking and define some variables.

To discuss further special case, in practice we assign g and L Procedure of building an one layer NN algorithm can be listed as the following:

1. Initialize W and b to zero matrices
2. From X, compute Z=WX+b
3. Obtain Cost function $J = \sum L$ (Forward propagation)
4. Obtain gradients $\frac{\partial J}{\partial W}$ $\frac{\partial J}{\partial b}$
5. Update W and b then compute J for the updated W,b.
6. Do 5. maybe thousand times and see if J decreasing toward zero.

1.2.4 Two layers NN

We consider slightly complex NN, the two-layer one, in which we still have the output y to be scalar. Thus the derivative of L for this case is

$$\begin{aligned}\frac{\partial L^{(d)}}{\partial w_{i,j}^{[2]}} &= \frac{\partial L^{(d)}}{\partial w_j^{[2]}} \\ &= \frac{\partial L^{(d)}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial w_j^{[2]}} \\ &= \frac{\partial L}{\partial a^{[2](d)}} \cdot g'(z^{[2](d)}) \cdot a_j^{[1](d)}\end{aligned}$$

$$\begin{aligned}\frac{\partial L^{(d)}}{\partial w_{i,j}^{[1]}} &= \frac{\partial L^{(d)}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a_{i_1}^{[1]}} \frac{\partial a_{i_1}^{[1]}}{\partial z_{i_1}^{[1]}} \frac{\partial z_{i_1}^{[1]}}{\partial w_{i,j}^{[1]}} \\ &= \frac{\partial L}{\partial a^{[2](d)}} \cdot g'^{[2]}(z^{[2](d)}) \cdot w_{i_1}^{[2]} \cdot g'^{[1]}(z_{i_1}^{[1](d)}) \cdot \delta_{i,i_1} \cdot x_j^{(d)}\end{aligned}$$

1.2.5 Deep NN

Having introduced all basic elements of N-layer NN, we can implement in practice how to make deep learning work.

Compared to the one and two layer NN whose recipe can be composed of 5 or 6 steps, that in deep learning correspond to 5 or 6 blocks to build the whole

procedure. We list as follows:

- 1. Initialize parameters W1,b1 ..... WL,bL
- 2. Linear+Relu forward propagation for 1 to L-1 layers then Linear+Sigmoid for L layer, i.e. [LINEAR + RELU] $\times$(L-1) $\Rightarrow$ LINEAR + SIGMOID. [5]
- 3. Compute the Loss and Cost function.
- 4. Linear+ Sigmoid backward propagation for L-th layer then Linear+Relu backward propagation for L-1 to 1 layers.
- 5. Update parameters W's and b's
- 6. Do the above for iterations

[5]To be explicitly, Z1=W1X+b1 , A1= relu(Z1) ; Z2=W2A1+b2, A2=relu(Z2)... ; ZL=WLAL-1 +bL , AL=sigmoid(ZL)

For the step1 , we create an initialize function **Initialize-parameter-deep(layer-dims)** whose input is a list contain dimensions [n0, n1, ..., nL-1] of each layer and output is a dictionary $\{"W1", "W2", ..., "WL", "b1", "b2", ..., "bL"\}$ stored all initialized parameters.

For step2 , we create a one-layer forward propagation function **Linear-Forward(A, W, b)** whose input is parameters W and neurons A from previous layer. It output Z of this layer and a tuple (A,W,b) stored the values the notation shows .

We then extend the function in step2 to a function **Linear-Activation-Forward(A-prev, W, b, activation)** which has an additional input activation. The output is neuron A in current layer and a tuple (A-prev, Z, A, W, b) of current layer

Then we create a full forward propagation function **L-Model-Forward(X, parameters)**, whose input is X and the parameter initial function defined in step1. Output is final neuron AL and a list [A-prev, Z, A, W, b] for each layer.

For step3, Loss and Cost function can be derived with aid of some calculus.

For step4, we do backward propagation similarly to forward case. First create a function **Linear-Backward(dZ, cache)** , whose input is dZ which can be pre-computed before this step and cache which forward propagation gives a help. This function's output dW db and dA-prev of current layer can be derived by cache which stored (A-prev, W, b) Then we create a extended version **Linear-Activation-Backward(dZ, cache, activation)** which outputs dA-prev, dW and db. Last one in this step is to create a full backward propagation function **L-Model-Backward(AL,Y,caches)** which takes the Y and AL so it can start by the initial point dAL of backward prop.. While doing the full backward prop., which is composed of first sigmoid one followed by L-1 relu ones, we create a dictionary grads to store dA, dW and db of each layer. The output is just this dictionary { "dA1",...,"dAL-1" , "dW1",..., "dWL" , "db1",...,"dbL"} .

For step5, updating W and b term-wise with help of the dictionary in step 4.

For step6, we iterate our computation above and check if Cost function decreases.

1.2.6 Some convenient tricks

We've completed the picture of GD algorithm for deep learning. In the process we described above, it's found to be good to have some helper functions. The first helper function is **Initialize-parameter-deep(layer-dims)** , and we will also encounter many situations that other helper functions will facilitate our computation process to be more compact. For example,

1.3 Improve the Gradient Descents

GD based on simple mathematical idea abeit already powerful, faces several barriers as the capabilities Deep NN requires more than conventional ML. In this section we will describe the material more computational way rather than mathematical way which we've done most of the important part previously.

To be brief, this rely on the various way of updating our parameters W and b. We'll see how we update W,b every subset of dataset rather than the whole dataset(Mini-Batch GD), how to update W,b in a modified way in which W and b is updated by amount V_{d_w} and V_{d_b} which is kind of an average over several dW and db in previous iterations(GD with Momentum), and finally considering an additional updating factor called RMSprop that the updating quantity is further replaced by $\frac{V_{d_W}^{corrected}}{\sqrt{S_{dW}^{corrected}}+\epsilon}$ and $\frac{V_{d_b}^{corrected}}{\sqrt{S_{db}^{corrected}}+\epsilon}$(Adam) where those V's we'll introduce in detail later.

1.3.1 Mini-Batch GD

1.4 Regularizations

Very often we meet the high variance situation(overfitting), and we find some ways to make our model simpler to sacrifice some training accuracy to seek a better test accuracy. Such simplification is called regularizations. In this section, we discuss dropout , L2 and some similar to L2.

1.5 Normalization and Batch Normalization

In ML or DL, features of datas can be diverse, while one ranges from -1 to 1 and others from 5000 to 100000. Such mismatch of values can cause our Cost function to form a elongate shape making GD stuck and easily getting our optimization very slow. Aside by normalizing the features of input X, neurons at hidden layers are also need to be normalized, which usually makes most versions of GD better.

Introducing normalization to neurons of [l]-th layer, sometimes called Batch Norm(BN), we just transform $z^{[l]}$ to $\tilde{z}^{[l]}$ by a transformation N($\beta^{[l]}$, $\gamma^{[l]}$) before the activation function $g^{[l]}$ acting on $z^{[l]}$. Thus, the process at ℓ-th layer would be like

$$\begin{aligned} a^{[\ell-1]} &\Longrightarrow z^{[\ell]} = W^{[\ell]}a^{[\ell-1]} + b^{[\ell]} \\ &\Longrightarrow \tilde{z}^{[l]} = N(z^{[\ell]}) \\ &\Longrightarrow a^{[\ell]} = g^{[\ell]}(\tilde{z}^{[l]}) \Longrightarrow z^{[\ell+1]} = W^{[\ell+1]}a^{[\ell]} + b^{[\ell+1]} \end{aligned}$$

It turns out that Batch Normalization is more sophisticated than Normalization in two aspect. Normalization on input X is eyed on optimization efficiency, and BN can stabilize the learning process by making neurons values in each layer being distributed in very stable ranges.[6] As a result, BN speeds up the learning.

The second aspect is somehow tricky, that BN on mini batch will introduce regularization effect. Since BN transforms neurons in a layer by means and variance

[6]Normalization reshapes the feature of input X zero mean and one variance, and BN does the same with exceptional freedom governed by $\beta^{[\ell]}$ and $\gamma^{[\ell]}$.

of "this mini batch", which differs from those of the whole data set, it adds some noise to it. It's similar to dropout regularization which adds noise by introducing zero/one with some probability to a layer's neurons. The larger mini batch size we use, the less regularization effect it will induce since the mean and variance approach those of the whole data set.

In this process, original (without BN) parameters $W^{[\ell]}$ and $b^{[\ell]}$ can be replaced by $W^{[\ell]}$, $\beta^{[\ell]}$ and $\gamma^{[\ell]}$ as can be shown by the definition of what BN done to this layer.

It's obvious that Deep NN with BN can be combined with all the improvement techniques we discussed in this section.

1.6 Hyper-parameters

So far, we shall reach a useful concept called hyperparameters, in contrast to parameters W and b. We've encountered many of them, and we can organize them in a systematic way to gain a broad view of controlling deep NN.

Learning rate α, momentum β_1 β_2 , size of mini batch i, BN parameters $\beta^{[l]}$ $\gamma^{[l]}$

Chapter 2

Machine Learning Strategy

To find the bottleneck of our ML performance, we always face many strategies to be determined. For instance, we may think of collecting more data or more diverse data, making training iteration longer, trying another GD algorithm such as Adam etc, trying deeper NN, or trying regularizations such as L2 or dropout. Also changing activation function and number of hidden units is possible.

It's extremely important to build a systematic strategy to decide which of the above are worth pursuing or discarding. And yet, this subject is always changing as DL evolving, since new algorithms are created like every month although most of them are not widely purposed but specialized for some sort of problems. So it should be keep in mind that strategies to deal with ML problems are like framework of ML packages which are always evolving.

2.1 Orthogonalization

One category of ML strategies is finding a way to tune parameters(maybe one or multiple ones) to achieve a specific adjustment of learning without affecting others too much. Just like when driving a car, we're not expecting an effect on directions while we brake, which is obviously designed for deceleration.

In ML, we generally expect a not bad fitting on training set first, in many applications, it means comparably to human level performance. Then goes to dev set, test set and finally the performance in real world. For training, dev and test set, we can measure the performance by some metrics.

Single number evaluation metric

We often need to define metrics to measure which classifier is better than others on hand. **Precision and recall** are common used evaluation metrics. However, they are not single number and sometimes hard to use them to pick out the best model on hand. Hence, a single number is defined from them called **F1 score** by doing harmonic average of precision and recall.

$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

There are also some circumstances that multiple metrics appear to be important and we have to decide which one is **optimizing metric** and the other one is **satisficing metric**.

Human level error

It's a basic problem to figure out our model is under fitting or over fitting. In other word, it's bias or variance. Recall that high-bias mean high training error, and high variance means large gap between dev error to training error. It's the bias that we can discuss more.

We can define a new term **avoidable bias** which means the gap between training error and human-level error. With this in mind, sometimes high bias doesn't refer to bad performance on training set because human-level error is about the same so that avoidable error is quite small and thus training performance cannot be better.

At some task, training error surpasses human-level error,

2.1.1 Transfer learning

It's a useful strategy to learn from task A to task B. What it means is to train our NN by a larger dataset for A and then train the following layers by dataset of B(and maybe mixed with that of A). Another version of transfer learning we can learn from data of multiple A's say $A_1, A_2, ..., A_n$, and then train the rest layer by data of B. Those transfer learning technique works when

- 1. All A's and B share the same lower level features.

- 2. All A's datasets are about the same size.
- 3. NN should be larger.

2.1.2 End to end deep learning

End to end deep learning is the emergent paradigm of machine learning. It was typical to extract features by human intelligence. However, with the birth of big data, the intermediate engineering is omitted today. Nevertheless, some intermediate human engineering steps are still needed when we have relatively small amount of data.

The cons of end to end DN is the needing of huge amount of data.

Chapter 3

Convolutional Neural Network

One problem we face in image classification is the large input X. Todays photos is far more pixels than 64x64. Although 64x64 already induce a large feature of input X whose shape is [12288,1] vector, Deep NN suffice to classify them. Today's photo easily surpass 1024x1024 pixels, which induce a [3millions,1] vector X, and this will make first layer $W^{[\ell]}$, if we have 1000 units in first layer, consisting of 3 billion parameters(1M x 1000). It's hard to avoid overfitting with such huge number of parameters when we feed our image datas. To deal with this difficulty, we introduce the convolutional operation.

3.1 Convolutional operation

Edge detection is a starting point to this concept, which we collect different kind of edges in a photo and treat them as new set of features of this photo. Mathematically, this can be done by convolutional operation. For example, we have an image with raw pixel(grayed) matrix 6x6. The convolutional operation introduce a filter(sometimes called kernel) matrix , say 3x3 , and convolutional operate on the 6x6 data matrix to produce a 4x4 edge matrix. [1] [2]

[1] In python, convol operation is implemented by conv-forward. In Tensorflow, it's tf.nn.conv2d. In Keras, it's Conv2D

[2] The operation we use in ML is actually called **Cross-correlation operation** in math, however in ML community it's just called convolution operation which is slightly different in math.

For vertical edge detection, filter matrix has the form

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

For simplicity, we try an image matrix as follows

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix}$$

which explicitly shows an vertical edge in the middle of the image. The result of this convolutional operation gives the following matrix as we can easily check

$$\begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

We see that filter matrix indeed extract an edge from image matrix by convolutional operation. However, the edge in the edge matrix seems to be thicker than it should be in the raw image matrix, whose effect will be negligible while out raw pixel matrix is huge as today's photo is.

It can be shown the negative of the same edge matrix can be resulted from raw image matrix with 10 and zero changing sides. Thus, these two types of vertical edges can be presented clearly by those matrix.

Similarly, horizontal filter matrix is of this form

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

to show this works, the image matrix we consider it operates on is

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix}$$

which leads to

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 30 & 10 & -10 & 30 \\ 30 & 10 & -10 & 30 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

thus, it captures the horizontal edge of the image.[3]There are other version of filter matrix for adjusting the ones in the filter matrix we've introduced.

In pratical DL, we can set the 3x3 filter matrix by parameter W and let the DL learn those w_1 to w_9 parameters.

3.1.1 Padding

In general, we convol an (n,n) matrix by a (f,f) filter matrix and get a (n-f+1,n-f+1) matrix smaller than the raw pixel matrix. What if the matrix getting too small after many convol. operation? And we also observe the convol. operation make more use of the raw pixel in the middle of the image than those on the corner. This means we get more information of this image from its center and waste much information from its corner. How to solve these shrinking matrix and the throwing away of corner information problems?

To soften this situation, we can do something like enlarging the image matrix from (n,n) to (n+2,n+2) before it is convol. operated. Such process is called

[3]The relatively dark value 10 in the middle is the effect of its mixing of positive edge and negative edge, which is negligible while our pixel number is getting large.

padding the image, and padding amount for this case is called p=1. Thus, in general we have (n+2p-f+1, n+2p-f+1) outcome edge matrix after one padding and convol. operation.

3.1.2 Valid and Same convolution

How much do we choose to pad, that is thus equivalently how much is p. There are terms called Valid convolution for p=0 and Same convolution for n+2p-f+1=n. The former is just no padding, and the later is keeping the matrix the same size after doing padding and convolution. For Same convolution, the odd size of filter matrix give the only possible chance for us to pad to keep the matrix size unchanged.

3.1.3 Strided convolution

In addition to traditional convolution operation, which according to eq(?) shrink the matrix from (n,n) to (n-f+1,n-f+1). We can also define another one called **strided convolution operation** with stride number s which taking (n,n) to $(\frac{n-f}{s}+1, \frac{n-f}{s}+1)$. Note that zero stride s=0 goes back to traditional convolution operation. Combining padding, we have (n,n) to $(\frac{n+2p-f}{s}+1, \frac{n+2p-f}{s}+1)$ for the padding and strided convol. operation.

If we encounter the above division is not integer , i.e. n+2p-f is not divisible by s, it's conventional to just round down $\frac{n+2p-f}{s}+1$.

Thus, in summary, we have the following matrix for the output of our full package of convolution operation

$$\left(\left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor, \left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor\right)$$

3.1.4 Convolutions over volume

Image has colors, we have to convolution those image datas over (n,n,3) "volumes" instead of just 2D images we've described so far. Hence we need our filter to upgrade to 3D as well. [4]

The filter matrix is now (f,f,3) volume. We see the output edge matrix is the same as that of 2D full convolution operation as noting that numbers of channel are the same for image and filter, thus each element of output matrix is produced the same way as in 2D but from 3 times more summation terms.

3.1.5 Multiple Filters

There is no reasons to stop ourselves from extracting vertical and horizontal or more other directions edges at the same time. What we do is just convol. image volume by filter1 and filter2 independently and stack the output matrix together to form, say (4,4,3) volume. To be clear, the third dimension of this volume is the number of filters we use.

[4]ML literatured call the third "color" dimension number of channel

3.2 Multiple-layer CNN

3.2.1 One-layer CNN

It's time to apply convolution operation to NN. Let's illustrate it by one layer NN. We first define our notation:

About Convolution Operation

f^l: filter size

$p^{[l]}$: amount of padding

$s^{[l]}$: amount of stride

$n_{[c]}^{[l]}$: number of filters

About Neurons

$(f^{[l]}, f^{[l]}, n_{[c]}^{[l-1]})$: dimensions of filters

$(n_h^{[l]}, n_w^{[l]}, n_{[c]}^{[l]})$: dimensions of $a^{[l]}$

Vectorizations

$(m, n_h^{[l]}, n_w^{[l]}, n_{[c]}^{[l]})$: dimensions of $A^{[l]} inonemini - batch$

About parameters

Number of W : $f^{[l]} \cdot f^{[l]} \cdot n_{[c]}^{[l-1]} \cdot n_{[c]}^{[l]}$

Number of b : $n_{[c]}^{[l]}$

With these notations, we know already from eq.(?) that

$$((n_h^{[l]}, n_w^{[l]}, n_{[c]}^{[l]})) = \left(\left\lfloor \frac{n^{[l]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor, \left\lfloor \frac{n^{[l]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor, n_{[c]}^{[l]} \right)$$

Assume the image data we input is x which is a (n_{px},n_{px},3) volume. In high analogy to NN, the role filters play in CNN is the same as that $W^{[1]}$ play in NN.

For that filters/$W^{[1]}$ "multiply" image volume/x to form (n,n,n_f) volume/$z^{[1]}$. Including bias b, we symbolize the one layer CNN as

$$a^{[1]} = g(conv \cdot x + b)$$

where $conv \cdot x$ is what we just described that n_f filters take x to n_f edge matrix and +b is just adding bias to each edge matrix. Then, stack these together to form a volume and apply a activation function g to get first layer neurons. In the analogy, number of filters we use determines the number of units in this layer.

One advantage can be pointed out already: As we've mentioned that the number of parameter W (via $W^{[1]}x$) is growing too large for large pixel image, however, we can see this problem disappears in CNN since the parameters in CNN are presented by filters which is always a small cubic box compared to image volume. The different way the parameters coupled to features of data is the key. The parameters(filters) in CNN "multiply" x convolutionally and parameters(weights) in NN "multiply" x directly.

3.2.2 Deep CNN

Let's set some concrete numbers for a 4-layer CNN:

$$\begin{aligned} n_h^{[0]} = n_w^{[0]} &= 39, \\ n_c^{[0]} &= 3, \\ f^{[1]} &= 3, \\ f^{[2]} &= 5, \\ f^{[3]} &= 5, \\ s^{[1]} &= 1, \\ s^{[2]} = s^{[3]} &= 2, \\ p^{[1]} = p^{[2]} = p^{[3]} &= 0, \\ n_c^{[1]} &= 10, \\ n_c^{[2]} &= 20, \\ n_c^{[3]} &= 40 \end{aligned}$$

It's a good exercise to illustrate these number by plotting a process of this CNN. By eq.(?), it leads to $n_h^{[1]} = n_w^{[1]} = 37$, the height and width of the "neuron box"

at first layer. This neuron box's number of channel[5] is of course $n_c^{[1]} = 10$. Second layer the neuron box shrink faster since the stride s=2 for this layer. It can be easily deduced $n_h^{[2]} = n_w^{[2]} = 17$ for its height and width, and the number of channel is given $n_c^{[2]} = 20$. For the third layer, we can deduce $n_h^{[2]} = n_w^{[2]} = 7$ and $n_c^{[3]} = 40$ is given. Finally, for the forth layer we can just flatten the (7,7,40) box $a^{[4]}$ to a vector and input a logistic or softmax function to finish our forward prop.

3.2.3 Three types of layer in CNN- Convolution, Pooling and Fully-connected

Pooling layer, whose filter's function is to pick the maximum value of the "pool region" it covers and map that value to the element of output matrix. In contrast to that of conv layer whose filter's function is to element-wise product with the pool region it covers and summing over elements.

A special fact to notice about this max pooling layer is the absence of parameters. There are only hyper-parameters filter-size f and stride s which we need to assign in the begininig. Thus, there is nothing to learn in the pooling layer.

Another kind of pooling is average pooling which is just replace max by average for the filter computation.

For the image box, we apply pooling layer with a filter box. A tricky difference to conv layer is, for the pooling layer we filter the image box independently for each channel thus output is a box, while conv layer's is a matrix because the computation of filtering for all channel at the same time to map to a real number.

Another type of layer, fully connected layer, is the same as NN's layer with a wieght w and bias b acting on the last layer of CNN which is vector from a flattened box of previous layer.

[5]It worth reminding of the analogy of the neuron number of channel in CNN to the number of neuron unit in NN

3.2.4 A digression on analogy of NN/CNN and mechanics/field theory

A casual and loose physics analogy of this is following: (Dynamical quantities v.s. indices) in field theory is kind of like (parameters v.s. neuron index) in NN.

In NN, neuron index directly represent parameters.(W_{ij} where i is current layer's neurons and j is previous layer neurons) the (dynamical quantities) , thus neuron index is just paramTeters. This is just like mechanics in physics where positions directly represent dynamical quantities. Moreover, in CNN, the status of neuron index is no longer directly representing parameters but only serving as index of parameters, parameters are represented by filters in CNN. This is like field theory in physics, which position is no longer representing dynamics but only serves as index of dynamics, and dynamics is represented by field in field theory.

3.2.5 Computer Vision- Object Detection

Computer Vision is a exploding area that CNN is a major tool or serve as a building block of it. In this area, we have the image classification, image localization, detection etc. as major tools from more basic to less. We've been familiar with image classification, so we can illustrate how image localization works.

Image Localization

Our task now is to let computer observe whether and where an object is in the image, say the objects are car, people, and bike. Image localization takes the image x as input and output a vector y which is defined by

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

where p_c is boolean value for existence of any object in the image , b's are the representation of the square in which this object is surrounded , and c's are representation of which object is in the image.

However, this method is for detecting one object in the image. The need is to detect multiple or multiple kinds of objects in the image, which the method is discussed in the following.

Sliding Windows Detection Algorithm

Object detection is based on the image classification model that is already trained. To be precise, assume we have a image containing two cars in it. (Fig(?)) We slice a piece in the up-left corner of it and feed into CNN to classify if it's a car or not and then right to it a second piece for the same process and so on until exhausting the whole image. This sliding little piece can be resize, typically enlarge, for the next iteration. Repeating many iterations with piece(or window) of different sizes scanning over the whole image, at some position that the window is currently located, the image classification model will tell us the small window image is a car image, thus a car is detected in the image in a specific square region.

Sliding Windows Detection Algorithm has a very expensive computational cost as we slide the windows of various sizes though the whole image. The solution is to implement this algorithm convolutionally, which convol. each sliding simultaneously at one single convol. operation, instead of independently computing each slide as described above.

YOLO Algorithm

Also, there is a location accuracy problem that it's often not able to precisely locate the window to cover the object. This can be solved by a way called YOLO(You Only Look Once) algorithm.[cite.Redmon et. al 2015] which makes use of image localization at each divided grid of the whole image. For this algorithm, if one grid catch the object it'll output the y that image localization does.

However, two common situations are that a object is declared by multiple grids and multiple objects are in the same grid. The solutions are , for the first one, it can be solved by Non-Max Suppression method, and for the second one we can

use Anchor Boxes method, which is kind of enlarging the output y to multiple size , for example two times for two anchor boxes , so as to encode multiple object in one grid. It worth mentioning the shortage of this method is that the bad handling of more than two objects in one grid or two anchor box having similar shape. However, as we divide the image into finer grids, say from 3 by 3 to 19 by 19 , such situation will happen rarely.

3.2.6 Computer Vision- Face Verification and Face Recognition

Face recognition is distinguished to the face verification for that in face verification, we input an image and the name of person in the image and let computer to claim it's that person's image or not. It's kind of one-to-one problem, and we may have the computer accuracy to 90+ percentage. On the other hand, face recognition deals with the problem that computer needs to recognize whether the input image is one of the K persons in database or not.

One Shot Learning Problem

If we have training set of only one image data for each of N persons, which is a common situation in real life, we meet the One Shot Learning Problem. For this problem, it's not feasible to train a classification model whose output is N value one for which is one of the N persons. Instead, we let the computer learn a "similarity" function d(image1,image2) which measure the difference between two images.

Siamese Network

To find the similar function d, we use a special CNN called Siamese Network. It take the input image to an output , say 128x1 vector, which we call $f(x^{(1)})$ as first person's image output. Also, $f(x^{(1)})$ is sometimes called the encoding of $x^{(1)}$. In parallel, the same CNN take the other one $x^{(2)}$ to $f(x^{(2)})$, and we can now define the similarity function as

$$d(x^{(1)}, x^{(2)}) = \left\| f(x^{(1)}) - f(x^{(2)}) \right\|$$

The parallel architecture of CNN is the Siamese Network, where every CNN is the same so the encoding is well-defined by the parameters of this CNN.

The leaning goal is to make sure the encoding of two image of the same person should be similar, i.e. d is small, while d of images from different persons should be large.

An important point to be made is that multiple images for some persons are always needed in training a face recognition system.

Triplet Loss Function

In ML, to learn a task, it's required to define a loss function which we can choose algorithm to optimize it. Here, the task is to verify whether the given two images represent the same person or not. Such task always consists of three types of images : Anchor image(A), positive image(P) and negative image(N). Anchor image is just someone's image we want to specify, and positive image stands for image of this person while negative one stands for different person's. We need our CNN model to learn to encode these three images as

$$d(A, P) < d(A, N)$$

and leads to

$$d(A, P) - d(A, N) < 0$$

to prevent a meaningless outcome of learning, we can set a value α called **margin** as follows

$$d(A, P) + \alpha < d(A, N)$$

say $\alpha = 0.2$, then if similar images A and P has d(A,P)=0.5, margin value forces d(A,N) be greater than 0.5+0.2=0.7.

With these quantities, we can define the Loss function

$$L(A, P, N) = \max\left(d(A, P) - d(A, N) + \alpha, 0\right)$$

If we have training set of 10k images of 1k persons, the Cost function is defined

$$J = \sum_{i=1}^{m} L(A^{(i)}, P^{(i)}, N^{(i)})$$

However, we face a problem of choosing A,P,N from training set at starting point. To solve this, we think about what's the result of choosing randomly for A,P and N. Is it ok for optimize the cost function, i.e. is there something to learn? If not, how to choose the A,P,N from training set. After all, we need to map our training set to a set of triplet A,P,N.

Having chosen APN from training set, we apply GD to learning the parameters so the correct encoding of images is learned. With the trained system, we can recognize an image by encoding this image by this system and getting to know if it's similar to some images in our training set.

Face Verification as a Binary Classification

Alternatively, we can treat the face verification as a binary classification problem. In this approach, we also take image-1 and image-2 to Siamese Network and obtain their encoding $f(x^{(1)})$ and $f(x^{(2)})$, but this time we apply a sigmoid function to their element-wise difference with some weights(just like NN) to output $\hat{y}$

$$\hat{y} = \text{Sigmoid}(\sum_k \left|f(x^{(1)})_k - f(x^{(2)})_k\right|)$$

Note that $\sum_k \left|f(x^{(1)})_k - f(x^{(2)})_k\right|$ is just a special case of similarity function and can be replaced by other kind if we need.

3.2.7 What are layers of CNN computing?

It's hard to quantify every layer's learning activity, but it's useful to visualize qualitatively what they are learning from shallow ones to deeper ones. Let's focus on an unit in first layer. We try to observe what are the image patches that maximize this unit's activation. As we know what the filters do in CNN layers, we get a intuition that at first layer, this unit might learn very small patches of the input image such as all kinds of edges. Say first unit learn up-left to down-right inclined edge and second unit learn horizontal edge and third one learn vertical edge and so on... To see more sophisticated ways of visualize what layers are learning see[cite Zeiler and Fergus 2013 "Visualizing and understanding convolutional networks"]

Chapter 4

Recurrent Neural Networks(RNN)

RNN, an important example of sequence model, is used for speech recognition, natural language processing(NLP) and music generation etc.... Sequence models are dealing with problems that at least one of X and Y is sequence. For example,in speech recognition, we input audio clip $\mathbf{x}$ and wish to output a text $\mathbf{y}$. Such kind of data is called sequence data because $\mathbf{x}$ has time coordinate as a feature and $\mathbf{y}$ is a sequence of words. Other examples are shown in Fig(?).

4.1 RNN basics

We denote data $\mathbf{x}$ and $\mathbf{y}$ as

$$\mathbf{x} = (x^{\langle 1\rangle}, x^{\langle 2\rangle}, x^{\langle 3\rangle}, ..., x^{\langle t\rangle}, ..., x^{\langle T_x\rangle})$$
$$\mathbf{y} = (y^{\langle 1\rangle}, y^{\langle 2\rangle}, y^{\langle 3\rangle}, ..., y^{\langle t\rangle}, ..., y^{\langle T_y\rangle})$$

where we can see T_x is number of sequence of $\mathbf{x}$ and T_y is number of sequence of $\mathbf{y}$. Am important point to be made is that for different data $\mathbf{x}^{(i)}$ we can have different length of sequence, that is , the length is denoted by $T_x^{(i)}$, and the same for $T_y^{(i)}$.

NLP

For Natural Language Processing, we typically represent input $\mathbf{x}^{(i)}$ by a vocabulary, whose number of words V is ranging from 10,000 to 100,000 words that contained in it. With this vocabulary, each component(here text) of $\mathbf{x}^{(i)}$ is represented by an (V,1) vector. For each vector, its elements are zero except for the position where the text lies in the vocabulary, which we assign one to it. We call such a vector one-hot vector. Thus, the task for NLP is to learn a mapping from $\mathbf{x}$ to $\mathbf{y}$ by RNN architecture.

Length of input sequence T_x vs length of output sequence T_y

The architectures of RNN are slightly different for problems with same and different sequence length for input and output. This forms a family of RNNs(Reference: "Unreasonable effectiveness of Recurrent Neural Networks). They are many-to-many(Name verification, Machine translation), many-to-one(Sentimental classification) and one-to-many(Music generation) architectures.

Language model and sequence generation

Assume our training set is a large **corpus** of English text. Say, the following sentence:

Cats average 15 hours of sleep a day.

we first input $x^{<1>}$ =(maybe set to 0 or Cats) into first layer and output a predicted $\hat{y}^{<1>}$ which is a (V,1) vector. Then for second layer we input first word of this dataset $y^{<1>}$(or a.k.a. $x^{<2>}$) and output a predicted $\hat{y}^{<2>}$ and so on. Thus, this is a process that predict the next word given by some previous words. Each $\hat{y}^{<i>}$ is then operated by softmax to represent the probability distribution over N vocabularies.

4.1.1 Vanishing GD problem

Recall we have vanishing or exploding GD in deep NN, the same problem here, and for RNN which applies to sequence model of Language sampling or others it suffers some critical shortage of capturing the long correlation between former

and latter words of a sentence in common language use. The method addressing this problem is Gated Recurrent Unit(GRU).[Cho et. al. 2014. On the properties of neural machine translation: Encoder-decoder approaches][Chung et. al. 2014. Empirical evaluation of Gated Recurrent Neural Networks on Sequence Modeling]

Gated Recurrent Unit(GRU) and Long Short Term Memory(LSTM)

Additional quantity GRU and LSTM introduce are called memory cell(c) and adapted memory cell(c'). We denote them $c^{<t>}$ and $c'^{<t>}$. As Fig.? shown, they are defined by W_c, $a^{<t-1>}$, $x^{<t>}$ and gates Γ_u,Γ_r for GRU and Γ_u,Γ_f, Γ_o for LSTM where the first two gates in LSTM serve as the decision maker of choosing memory cell c or c' and are called **update gate** and **forget gate**.

4.1.2 Bidirectional RNN

Bidirectional RNN(BRNN) provides a more powerful model to language processing problem. In BRNN, for forward block in each unit, we add a backward block next to it. Thus, the original RNN is in some sense doubled in which the added second part of graph is computed in backward direction.[1]

4.1.3 Word representation

We previous introduce the vocabulary on which the text is modeled. It turns out that further represent words by additional features is quit efficient and useful for more advanced NLP tasks. In addition to one-hot vector, we represent words by **embedding**. With this manipulation, we put every one-hot word into a V_E space, where V_E is the dimension of the embedding space. For example, fruit, male and female are three common embedding feature of words. However, there may be large amount of uncommon features that cannot be termed in our daily language. This concept is quite similar to the encoding of image we previously mentioned in the image recognition task.

[1]The backward direction here is not backward propagation, but part of forward propagation.

Similarity function

With the embedding(or we can say encoding) of word representation, we can calculate some kind of "distance" between two words. Such distance function is called **similarity function**. One common used is **cosine similarity**, which is defined by

$$sim(u, v) = \frac{u^T v}{\|u\|\|v\|}$$

roughly saying the dot product of u and v.

Embedding matrix

Word embedding is just a kind of representation of our data, word here, like what we did for image recognition where we represent the image by some features. However, features are things for neural network to learn. The learned features are stored in a parameter matrix just like those in NN and CNN. Here the matrix is called embedding matrix.

The embedding matrix can be form by a $V_E \times V$ matrix where V_E is the dimension of embedding and V is number of words in our vocabulary. If we multiply this embedding matrix E by an one-hot vector o, say $E \cdot o_{9527}$ where 9527 is referred to word orange, the result is a $V_E \times 1$ vector representing the embedding of the word orange. We can denote this vector by e_{9527}

Thus, in general we have for the word w

$$e_w = E \cdot o_w$$

and our goal is to learn the parameters stored in E such that by the multiplication we know the embedding of some specific word is. We next describe some algorithm to learn the E.

4.1.4 Algorithms for learning E

In the history of learning embedding matrix, people tried many complex algorithms in the beginning. One of them is **neural language model** in which we feed the embedding vector to a neural network. Thus, the parameters are E and W,b.

Another simpler algorithm is the **skip-gram model**. In this model, we category our words into context word(c) and target word(t). Given a context word,

what's the target word at some specific position relative to the context word. For this setting, it suffices to learning the embedding of words. We can simplify the model to a single softmax unit.

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{V} e^{\theta_t^T e_c}}$$

we can see in the denominator we sum over the vocabulary which is typically slowing down the algorithm. The method to avoid this is using a tree softmax where we make use of a notification to remind the softmax to notice where in the tree the target word lying at. Once the target word is found somewhere in the tree, the summation over the whole vocabulary is not needed. This is called **hierarchical softmax classifier**. By the way, we note that this reduce the computational cost from V to logV.

However, the softmax still cost the computation too much. Thus we can refine our strategy of learning the embedding words. That we can focus on the sampling method. The way of sampling the context and target word will determine the speed of learning. Saying we randomly sample context word "c", then we can sample the target word "t" by some rule we give for example some relative position to the "c". However, sampling "c" is what we can smartly do. For example, we can apply some method to avoid sampling common words like "the", "a", "when" etc... too often.

Negative sampling

In the negative sampling method, we set the context word and target word as our input X and out put Y to be binary 1 and 0. For 1 it means context and target word are relevant which we call positive example, and 0 for opposite which we call negative example. The strategy Mikolov et al. provide is to choose first positive example and then choose k negative examples by randomly picking the target words with respect to the same context word. This leads to a model

$$p(y = 1|t, c) = sigmoid(e^{\theta_t^T e_c})$$

in contrast to the previous model which we have a summation over the vocabulary, in this model we only need to train k negative example and one positive example for each single context word.

Glove algorithm

There is one simpler algorithm within the word embedding problem called glove, aka. global vectors for word representation, algorithm[Pennington et al. 2014]. It makes explicit the appearance frequenct of target word t in the context of context word c by introducing a matrix X_{ij}. Then this model is to minimize the distance of t and c by taking difference of $\theta_i^T e_j$ and $\log(X_{ij})$ which the cost function here is

$$\sum_{i=1}^{V}\sum_{j=1}^{V} f(X_{ij})(\theta_i^T e_j - \log(X_{ij}))^2$$

where the weighting term f is defined by 1 for $X \neq 0$ and 0 for $X = 0$ is to avoid the singularity occurring at log(X) when $X_{ij} = 0$. Another function the f can work for is to balance the unimportantly frequent word and importantly but less frequent word by doing some clever arrangement.

4.1.5 Sentiment classification

An important application of word embedding is to classify sentiment. This problem can be built by input X as sentences and output Y as rating 0 to five stars.

4.2 Sequence to sequence model

In the previous section, we illustrate some model dealing with natural language processing. Here, we introduce a model called sequence to sequence model(SSM) whose inputs are sequence $x^{<1>}, x^{<2>}, ..., x^{<T_x>}$, and outputs are $y^{<1>}, y^{<2>}, ..., y^{<T_y>}$ where T_x and T_y are the numbers of sequence of input and output correspondingly. This model can deal with language translation problem.

First we setup a encoder RNN which can be GRU or LSTM, which output a sequence that fed into a language model we described in the previous section. Recall some basics of language model. In language model we have an initialized

input $a^{<0>}$ and $x^{<0>}$ fed into the model and output $y^{<1>}$ which is then input to the next unit. Thus, in principle we don't need some kind of input x to feed in this special type of RNN.

Here, in the sequence to sequence model, we have input $x^{<1>}, x^{<2>}, ..., x^{<T_x>}$ the output an a which is fed into the language model as the role of $a^{<0>}$ which we previously initialized to zeros. In this sense, SSM is to output a conditional probability $P(y^{<1>}, y^{<2>}, ..., y^{<T_y>}|x^{<1>}, x^{<2>}, ..., x^{<T_x>})$ in contrast to the previously mentioned language model whose output is $P(y^{<1>}, y^{<2>}, ..., y^{<T_y>})$. Finding the maximum of the conditional probability corresponds to finding the most probable sentence given a sentence to be translated. It suffices the task of machine translation. The algorithm to find the output y with the maximum probability is beam search. The beam search algorithm chosen here rather than greedy search is due to the fact that for translation task we expect to pick a better sentence by avoiding the way of searching word by word to maximize the probability. Moreover, we often let the algorithm do the approximately search because the amount of vocabulary V is large to search for each word in a sentence, say V^{T_x} possible choice.

4.2.1 Beam search

Assume our vocabulary is English one and its amount is V_E. If we want to translate a French sentence to English, beam search first pick an English word say $y^{<1>}$ and get the conditional probability of this word $P(y^{<1>}|x)$ where x is the whole French sentence to be translated. It's good to compare greedy search at this step. Greedy search will pick the maximum of $P(y^{<1>}|x)$ then move on to second step, while beam search is to pick, say B=3, choices of $y^{<1>}$ then move on. In second step, as in the language model, we obtain the conditional probability $P(y^{<2>}|y^{<1>})$ but here we have x as input, thus we'll have $P(y^{<2>}|x, y^{<1>})$. In this step, we browse over all V_E for $y^{<2>}$ and pick the top, say B=3, maximum $P(y^{<2>}|x, y^{<1>})$ then step into the next step. At this step, we have

$$P(y^{<1>}, y^{<2>}|x) = P(y^{<1>}|x) \cdot P(y^{<2>}|x, y^{<1>})$$

To step three, we do the similar thing in step two, obtain the top B maximum of $P(y^{<3>}|x, y^{<1>}, y^{<2>})$ and get

$$P(y^{<1>}, y^{<2>}, y^{<3>}|x) = P(y^{<1>}|x) \cdot P(y^{<2>}|x, y^{<1>}) \cdot P(y^{<3>}|x, y^{<1>}, y^{<2>})$$

and so on.

We can see the the beam width B determine a hyper-parameter of the SSM and B=1 is just the greedy search which means finding the most probable word at each step.

Length normalization

Beam search in SSM can be improved by a technique called **length normalization**. To introduce this concept, we first discuss a computational crisis to be avoid.

Now we have the probability for us to optimize the translation.

$$P(y^{<1>}, y^{<2>}, ..., y^{<T_y>}|x) = P(y^{<1>}|x) \cdot P(y^{<2>}|x, y^{<1>}) ... P(y^{<T_y>}|x, y^{<1>}, y^{<2>}, ..., y^{<T_y-1>})$$

from this we can see this number can be easily too small for computer to represent because each P is often much less than one. To tame this, we can take log in both sides and get

$$\begin{aligned}
&log(P(y^{<1>}, y^{<2>}, ..., y^{<T_y>}|x)) \\
&= log(P(y^{<1>}|x)) + log(P(y^{<2>}|x, y^{<1>})) + ... + log(P(y^{<T_y>}|x, y^{<1>}, y^{<2>}, ..., y^{<T_y-1>})) \\
&= \sum_{t=1}^{T_y} log(P(y^{<T_y>}|x, y^{<1>}, y^{<2>}, ..., y^{<T_y-1>}))
\end{aligned}$$

which make the optimization more stable numerically.

Another effect of this probability formula is that the algorithm would prefer to translate in a shorter sentence than the best translation should be, because shorter sentence which induces less probability products poses a bigger total conditional probability. We can solve this by normalizing the probability

$$\frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} log(P(y^{<T_y>}|x, y^{<1>}, y^{<2>}, ..., y^{<T_y-1>}))$$

where hyper-parameter α is a constant between 0 and 1. This manipulation can reduce the penalty of choosing the longer sentence which can be tuning by α.

One more comment about beam search. How to choose the value of B? Intuitively, larger B can provide a better chance to pick up a better sentence within the computational ability.

In computer science, there are other search algorithm such as BFS and DFS that is exact algorithm. However, in the machine translation context, an approximate algorithm works efficiently.

4.2.2 error analysis on beam search

For an approximate search algorithm or so-called heuristic search algorithm such as beam search, there is space for error analysis to handle some sort of mistake beam algorithm might make. We can do some analysis on whether the defect of learning is coming from beam search or not.

In chapter 2, we mention the human performance. We denote the human translation as output y^* and machine translation output $\hat{y}$ where y^* is always better than $\hat{y}$ as we discussed in chapter 2.

Now we list the components of the recent task. We have

- RNN composed of encoding part and decoding part
- Beam search algorithm

Some comments can be made by comparing $P(\hat{y}|x)$ and $P(y^*|x)$.

- If $P(\hat{y}|x) < P(y^*|x)$, we can say it's beam search that fail to choose a more probable sentence.
- If $P(\hat{y}|x) \geq P(y^*|x)$ which violets the statistical foundation. We know this RNN model fail to give a correct probability P.

4.2.3 Bleu score

Machine translation, unlike many other tasks that outputs a single correct answer, can produce multiple equally correct answers. A technique to pick one sentence out of these multiple sentences is called Bleu(Bilingual Evaluation Understudy) score.[Papineni et al. 2002] The concept of it is to reference human performance

in dev set to pick the sentence which has more words in common with the human translation sentence. One thing to be careful about is that if the common word appears to be a highly repeated words, the credit of it should be limited because it got a high chance to be less important in a sentence. Other details can be looked up in the paper just listed.

4.2.4 Attention model

An important modification of SSM is called attention model. One thing to be emphasize is that attention concept also exist in other area of deep learning. What it does in this modified SSM is to capture or to memorize a piece of a long sentence at one time rather than the whole sentence for the input to the decoding part. [Bahdanau et al. 2014] Philosophically, this way of reading and translating sentence is much more the same as human does.

In attention model, we apply BRNN, composed of units forward $a^{<i>}$ and backward $a^{<i>}$, and consider additional parameters called attention parameter $\alpha^{<i,j>}$ which connect the BRNN to another RNN layer, composed of units $s^{<i>}$, which serves to generate the output $\hat{y^i}$.

The important concept is that the attention weights(parameters) $\alpha^{<i,j>}$ tells the "generator" units $s^{<i>}$ how many adjacent inputs $x^{<j>}$(thus $a^{<j>}$) it should pay attention when generating the i-th output $\hat{y^i}$.

In practice, GRU and LSTM are commonly used for the blocks of attention model. We would define the attention weight to be a softmax function which sum up to 1

$$\alpha^{<t,t'>} = \frac{\exp\left(e^{<t,t'>}\right)}{\sum_{T_x}^{t'=1} \exp(e^{<t,t'>})}$$

where $e^{<t,t'>}$ is unknown and is to be learned by this model.

4.3 Speech recognition

A modern application of SSM is speech recognition. For this task we have audio clip as input x and transcript as output y. Roughly speaking, audio clip is a physically data that stores the variation of air pressure with respect to time. In

practice, the audio clip is preprocessed and represented by frequency as a function of time and colored to represent loudness.

Datasets of audio clip for academic research are commonly thousands of hours length while in industry they are much more longer. The time step of input $x^{<i>}$ can be easily very long. For example if we feed a 10-second audio clip with features at 100 hertz, which means 100 samples per second. We'll have 1000 time step in our input. However, the output which is transcript is much shorter. To address this issue, we can apply a method so-called **Connectionist temporal classification(CTC cost)**. In this method we insert blanks and repeat characters into output y to fill the time step to match the length of input x.

4.4 Assignment - Neural Machine Translation

In this assignment, we build attention model to translate human readable date to machine readable date, e.g. "25th of June, 2009" to "2009-06-25". This model can be used to build machine translation, however, it needs large dataset and needs days of training, while training for readable date task is cheaper.

We are given m=10000 dataset of human readable dates as x and machine readable dates for y. We have a list of tuple (human readable date, machine readable date) and python dictionary $\text{human}_v ocab \text{and} \text{machine}_v ocab of(characterof humanreadabledate, inte$

Appendix :

TensorFlow and Keras

Basic structure of TensorFlow(TF)

In TF, it's basically variables, placeholder, algorithm and sessions(graph) that interplay. **Variables** are for parameters and cost W,b which to be optimized by **algorithm** that is run by **Sessions**. And **placeholders** serves as preserved space for dataset to be feeded by a comment "feed-dict" in sessions. The code in the following illustrates these [Fig. TF tuto]

Basic structure of Keras(K)

For Keras, some packages and functions within are useful for us

- layers :
  Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization, Flatten, Conv2D, AveragePooling2D, MaxPooling2D, Dropout, GlobalMaxPooling2D, GlobalAveragePooling2D, Lambda
- models :
  Model , load-model
- utils :
  layer-utils, data-utils: get-file, get-file, vis-utils, model-to-dot, plot-model, kt-utils

- applications:

  imagenet-utils preprocess-input

- backend :

- initializers :

  glorot-uniform

- preprocessing :

  image