Deep Learning specification Lecture Notes

Nolan

2023.4

Contents

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h{}^{[\text{number of layers } +1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

·$Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

·$y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

·$W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer

·$b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

·$\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^{m} \mid y^{(i)} - \hat{y}^{(i)} \mid$

2 Deep Learning representations

For representations:

· nodes represent inputs, activations or outputs

· edges represent weights or biases

Here are several examples of Standard deep learning representations

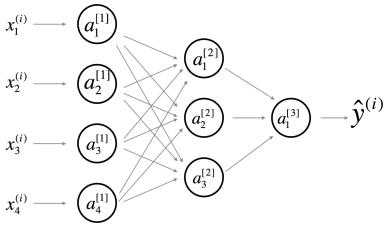

Figure 1: Comprehensive Network: representation commonly used for Neural Networks. For better aesthetic, we omitted the details on the parameters ($w_{ij}^{[l]}$ and $b_i^{[l]}$ etc...) that should appear on the edges

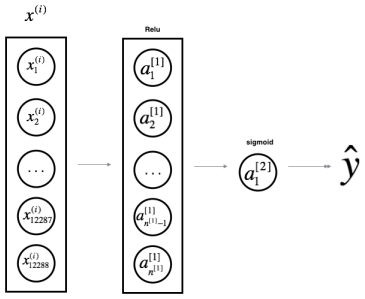

Figure 2: Simplified Network: a simpler representation of a two layer neural network, both are equivalent.

Chapter 1

Neural Networks and Deep Learning

1.1 Logistic Regression

Logistic regression is an algorithm for binary classification.

Definition 1.1.1 Logistic Regression

$$\textbf{Given x, predict } \hat{y} = P(y = 1|x)$$

$$\textbf{Where :}$$

$$x \in \mathbb{R}, \hat{y} = \sigma(w^T x + b)$$

$$\textbf{sigmoid function} : \sigma(z) = \frac{1}{1 + e^{(} - z)}$$

Note:-

Use **column** vectors instead of row vectors to represent X.

$$\mathbf{X} \in \mathbb{R}^{m \times n_x}, y \in \mathbb{R}^{1 \times m}$$

m is the number of samples.n_x is the number of features
And use w, b as parameters instead of using $\theta \in \mathbb{R}^{n_x+1}$ and adding a one vector in $\mathbf{X}$.

$$w \in \mathbb{R}^{n_x \times 1}, b \in \mathbb{R}^1$$

Logistic Regression is actually a simple neural network.

1.2 Cost function

To train a logistic model or any model, we need to define a **Loss function** and a **Cost function**.
Given$(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$, we want:

$$\hat{y}^{(i)} \approx y^{(i)}$$

Definition 1.2.1 Loss Function

Square Error:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

Cross-entropy Error:

$$L(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log (1 - \hat{y}))$$

Note:-

Using Square error loss function for logistic regression will lead to multi local optima in parameter space. Cross-entropy error will lead to a convex problem, which only have a global optima.

Loss function is for a single sample, for the entire data set, we have define a cost function.

Definition 1.2.2 Cost Function for logistic classification

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^i \log(\hat{y}^i) + (1-y^i)\log(1-\hat{y}^i)]$$

Note:-

Loss function is for a single sample, and cost function is the cost of model's parameters. So during the training of the model, we are going to find the optimal parameters to minimize the cost function$J(w,b)$.

1.3 Gradient Descent

A way to minimize the cost function, $J(w,b)$.

Definition 1.3.1 Gradient Descent

Start at a random point, walk to the direction where the function decrease the most (steepest). And Repeat until converge.

$$w := w - \alpha\frac{\partial J(w,b)}{\partial w}$$

$$b := b - \alpha\frac{\partial J(w,b)}{\partial b}$$

α, the learning rate

1.4 Computation Graph

To compute $J = 3(a + bc)$, we can first calculate $b * c$, write as u, the use $u + a$ to calculate J.

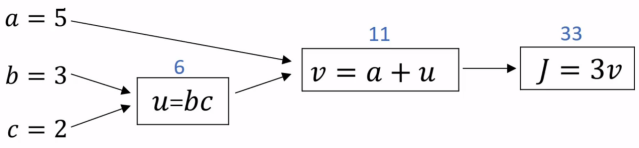

Figure 1.1: computation graph of J = 3(a + bc)

This is forward computation(forward propagation). We can use backward computation to calculate derivatives(back propagation).By chain rule:

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial a}$$

So we first calculate $\frac{\partial J}{\partial v}$, then calculate $\frac{\partial v}{\partial a}$, which is the reverse order of forward computation.

Note:-

For symbol simplification in code, Andrew uses 'dvar' as the partial derivative of final variable to median variable 'var'.
For example, dv represents $\frac{\partial J}{\partial v}$, da represents $\frac{\partial J}{\partial a}$

1.5 Logistic Regression Gradient Descent

Now we use computation graph in computing derivatives of logistic cost function.

1.5.1 Computation graph for Loss function(single sample)

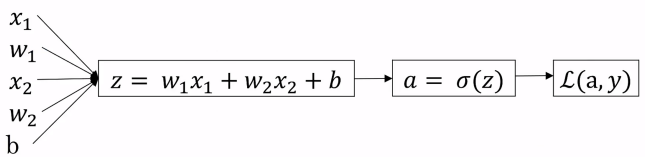

Figure 1.2: computation graph of cost function of Logistic regression

$$\frac{\partial L}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a} \qquad \frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = a - y$$

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_j} = x_j(a-y), \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b} = a - y$$

Then we can update the parameters using gradient descent:

$$w_1 := w_1 - \alpha\frac{\partial L}{\partial w_1}, w_2 := w_2 - \alpha\frac{\partial L}{\partial w_2}, b := b - \alpha\frac{\partial L}{\partial b}$$

Note:-

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z))$$

1.5.2 Gradient Descent for m samples

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}), a^i = \hat{y}^i$$

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L(a^i)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} dw_j^i = \frac{1}{m}\sum_{i=1}^{m} x_j^i(a^i - y^i)$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L(a^i)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} db^i = \frac{1}{m}\sum_{i=1}^{m}(a^i - y^i)$$

1.6 Vectorization

Instead using for loop in our coded for calculating cost function, using vectorization can really speed up Matrix calculations.

Note:-

Neural Network Programming Guideline:

when possible, avoid explicit for-loops.

1.6.1 Vectorization In Logistic Regression

First we need to represent out data, parameters in matrix form.

$$\mathbf{X} = \begin{bmatrix} | & & | \\ x^1 & \dots & x^m \\ | & & | \end{bmatrix} \mathbf{y} = \begin{bmatrix} y^1 & \dots & y^m \end{bmatrix} \mathbf{w} = \begin{bmatrix} w_1 \\ \dots \\ w_{n_x} \end{bmatrix} \mathbf{b} = \begin{bmatrix} b & \dots & b \end{bmatrix}$$

where:

$$\mathbf{X} \in \mathbb{R}^{n_x \times m}, \mathbf{y} \in \mathbb{R}^{1 \times m}, \mathbf{w} \in \mathbb{R}^{n_x \times 1}, \mathbf{b} \in \mathbb{R}^{1 \times m}, n_x : number\ of\ features$$

Then we calculate the sigmoid function and the cost function.

$$\mathbf{A} = \sigma(\mathbf{w}^T \mathbf{X} + \mathbf{b}) = \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}^1 + b) & \dots & \sigma(\mathbf{w}^T \mathbf{x}^m + b) \end{bmatrix}$$

Note:-

Code in python would be : A = sigmoid(Z) = sigmoid(np.dot(w.T, x) + b)
Here sigmoid is usr-defined function and b is actually a scalar instead of a vector, numpy's broadcasting can turn scalar or vectors into high dimensional vectors if possible.

1.6.2 Vectorizing Logistic Regression's Gradient Computation

Recall in Section 1.5.1, we have:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}), a^i = \hat{y}^i$$

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L(a^i)}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} dw_j^i = \frac{1}{m} \sum_{i=1}^{m} x_j^i (a^i - y^i)$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L(a^i)}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} db^i = \frac{1}{m} \sum_{i=1}^{m} (a^i - y^i)$$

So:

$$db = \frac{1}{m} np.sum(dz) = \frac{1}{m} np.sum(A - y)$$

$$dw = \frac{1}{m} np.dot(\mathbf{X}, dz.T) = \frac{1}{m} \begin{bmatrix} | & & | \\ x^1 & \dots & x^m \\ | & & | \end{bmatrix} \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}^1 + b) - y^1 \\ . \\ . \\ . \\ \sigma(\mathbf{w}^T \mathbf{x}^m + b) - y^m \end{bmatrix}$$

$$= \frac{1}{m} \begin{bmatrix} \sum_{i=1}^{m} x_1^i dz^i \\ . \\ . \\ . \\ \sum_{i=1}^{m} x_{n_x}^i dz^i \end{bmatrix} = \frac{1}{m} \begin{bmatrix} \sum_{i=1}^{m} x_1^i [\sigma(\mathbf{w}^T \mathbf{X} + \mathbf{b}) - y^i] \\ . \\ . \\ . \\ \sum_{i=1}^{m} x_{n_x}^i [\sigma(\mathbf{w}^T \mathbf{X} + \mathbf{b}) - y^i] \end{bmatrix}$$

Note:-

Here db, dw means $\frac{\partial J(\mathbf{w},b)}{\partial b}, \frac{\partial J(\mathbf{w},b)}{\partial \mathbf{w}}$ instead of partial derivatives of loss function.

Let's see how to implements them in python using numpy.

```
import numpy as np
# Backpropagation
Z = np.dot(w.T, X) + b
A = sigmoid(Z)
dz = A - y
dw = 1/m * np.dot(X, dz.T)
db = 1/m * np.sum(dz)
# Gradient Descent
w = w - alpha * dw          # alpha is the learning rate
b = b - alpha * db
```

Note:-

The code above only updates the parameters once, to reach the optima we need to update many times until converge.

Definition 1.6.1 Broadcasting in Numpy

In numpy, when you are trying to add or (-, / , *) a (m, n) shpae matrix with a (m, 1) shape matrix, numpy would automatically expand (m, 1) matrix into (m, n) matrix where each column of the new matrix is the original (m, 1) matrix. This also applys to (m, n) with (1, n).
For more about broadcasting, please refer to numpy's official documentation.

1.6.3 Note in numpy vectors

Because of the powerful but also strange broadcasting in numpy, we may make hard-to-find bugs in our code. Here is some notes.

Note:-

1. Don't use one dimensional array. (those .shpae would return (int,)), use np.random.randn(1, 5) instead of np.random.randn(5)
2. use assert(a.shape == (1, 5))
3. when you get a rank one array(one dimensional), reshape it using .reshape(1, 5)

1.7 W2 Homework

Two notebooks: W2 Homework and Answers, Github Repo

- Python basics with numpy
- Logistic Regression as Neural Network

1.8 Neural Network Representation

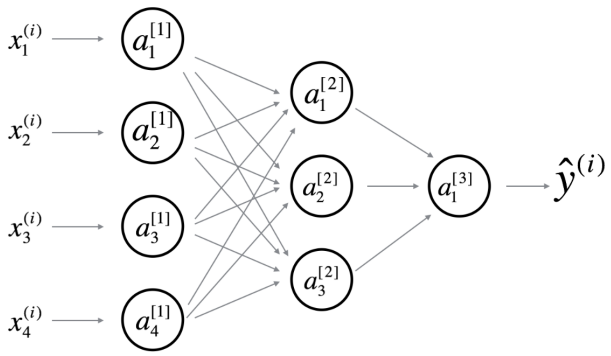

Figure 1.3: One hidden layer Neural Network

Neural Networks can be combinations of many sigmoid functions. The figure above shows how a one hidden layer neural network looks like. Each node is calculated as sigmoid(w.T@x + b), where x is left layer.

Note:-

By convention, the first layer, which is also called input layer, is not counted into total layers, the last layer, which is the output layer, is counted. So the figure above represents a one-hidden layer or two-layer neural network. Input layer can be denoted as : $x_i = a_i^{[0]}$

Because of now existing many nodes, parameters are now matrix instead of vectors, for a single sample, the ith layer vector's parameters are:

$$\mathbf{W}^{[i]} = \begin{bmatrix} -w_1^{[i]T}- \\ \cdot \\ \cdot \\ -w_n^{[i]T}- \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1^{[i]} \\ \cdot \\ \cdot \\ b_n^{[i]} \end{bmatrix}$$

where n is the number of nodes in ith layer, x is the left layer's vector.

Note:-

If the amount of (i-1)th layer's nodes is m, ith layer's is n, then $W^{[i]}$'s shape is (n, m)
Key point: one column of $\sigma(\mathbf{W@X} + \mathbf{b})$ actually represents one layer calculate from one sample of $\mathbf{X}$, so the number of columns of $\sigma(\mathbf{W@X} + \mathbf{b})$ is the number of samples of $\mathbf{X}$

So, for the two-layer neural network example, forward propagation works like this:

$$\mathbf{X} = \mathbf{A}^{[0]} \text{ (input layer)} \qquad \mathbf{Y} = \mathbf{A}^{[2]} \text{ (output layer)}$$

$$\mathbf{A}^{[1]} = \sigma(\mathbf{W}^{[1]}\mathbf{A}^{[0]} + \mathbf{b}^{[1]})$$

$$\mathbf{A}^{[2]} = \sigma(\mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{b}^{[2]})$$

1.9 Activation Function

We have being using sigmoid function as our activation function, but there other activation functions like tanh function($tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$), tanh function is actually a shift version of sigmoid function. tanh almost always works better than sigmoid function for centering around zero(-1 ¡ tanh(z) ¡ 1 for any z). So centering data then using tanh function as activation function makes learning easier.

Another problem of sigmoid function is that when z is very large or very small, the gradient would be very small, which slows learning.

Definition 1.9.1 ReLU

ReLU function(Rectified linear unit function) is more preferred activation function in deep learning.

$$ReLU(z) = \max(0, z)$$

The derivative of ReLU is 1 in (0, ∞), 0 in (−∞, 0), it have no derivative at 0 but due to computer's discrete nature, its derivative at 0 is 1

Note:-

Rule of Thume for choosing Activation function:
If the output layer's values have be within (0, 1), use sigmoid function for output layer and use ReLU for other layer's activation function.
There are also another version of ReLU called Leaky ReLU($\max(0.01z, z)$, 0.01 is not fixed.), whose derivatives are positive in (−∞, 0), the advantage of ReLU and Leaky ReLU is that in most area of z we can get a non-zero derivative, which makes learning faster.

Note:-

Why do we need non-liner activation functions?
If we remove the activation functions, no matter how many layers we add, we end up getting a linear combination of linear functions, which is just a linear function. So non-linear activation function gives neural network the ability to learn any continuous functions.

Definition 1.9.2 Derivatives of Activation functions

Sigmoid:

$$a' = \frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

tanh:

$$a' = \frac{dtanh(z)}{dz} = 1 - (tanh(z))^2$$

ReLU

$$a' = \frac{dReLU}{dz} = 0 \text{ if } z < 0, 1 \text{ if } z \geqslant 0$$

Leaky ReLU

$$a' = \frac{dLeaky_ReLU}{dz} = 0.01 \text{ if } z < 0, 1 \text{ if } z \geqslant 0$$

1.10 Gradient Descent for Neural Networks

Definition 1.10.1 Parameters in a one-hidden layer NN

$$\mathbf{W}^{[1]} \in \mathbb{R}^{n^{[1]} \times n^{[0]}}, \mathbf{W}^{[2]} \in \mathbb{R}^{n^{[2]} \times n^{[1]}}, \mathbf{b}^{[1]} \in \mathbb{R}^{n^{[1]} \times 1}, \mathbf{b}^{[2]} \in \mathbb{R}^{1 \times 1}$$

where:

$$n^{[0]} = n_x, n^{[1]} = \text{ number of nodes in first layer }, n^{[2]} = 1 \text{ (output layer has one nodes)}$$

Here we are also using broadcasting so **b** is not in matrix form.

Note:-

General form of parameters:

$$\mathbf{W}^{[i]} = \begin{bmatrix} -w_1^{[i]T} - \\ \cdot \\ \cdot \\ -w_{n^{[i]}}^{[i]T} - \end{bmatrix}, \mathbf{W}^{[i]} \in \mathbb{R}^{n^{[i]} \times n^{[i-1]}}, \mathbf{b}^{[i]} = \begin{bmatrix} b_1^{[i]} \\ \cdot \\ \cdot \\ b_{n^{[i]}}^{[i]} \end{bmatrix}, \mathbf{b}^{[i]} \in \mathbb{R}^{n^{[i]} \times 1}$$

1.11 Forward propagation and Backward propagation(Optional)

For interested readers, please refer: Backward propagation Intuition

1.12 Random Initialization

Definition 1.12.1 Symmetry-breaking problem

We use zeros to initialize our logistic regression's parameters since it won't cause any problems, but we can not use zeros for **W** in Neural Networks, which would lead to symmetric nodes who will implement same functions, and they will stay equal during iteration. So as you can see, not just zeros can would cause this problem, but also equal $w_j^{[i]}$s. This kind of problem is called **symmetry breaking problem**

```
# Using samll initial values of w, b to aovid
# slow learning with tanh, sigmoid as activation function
W = np.random.randn((n_1, n_2)) * 0.01
b = np.zeros((?, ?))
```

0.01 in the above code is actually depending on the depth of your neural networks.

1.13 W3 Homework

One Jupyter Notebook W3 Homework and answers, Github Repo

- Planar data classification with one hidden layer

1.14 Deep Neural Networks

Definition 1.14.1 Deep NN

Neural Network with many hidden layers

Note:-

Forward propagation and backpropagation in Deep Neural Networks

1.14.1 forward propagation

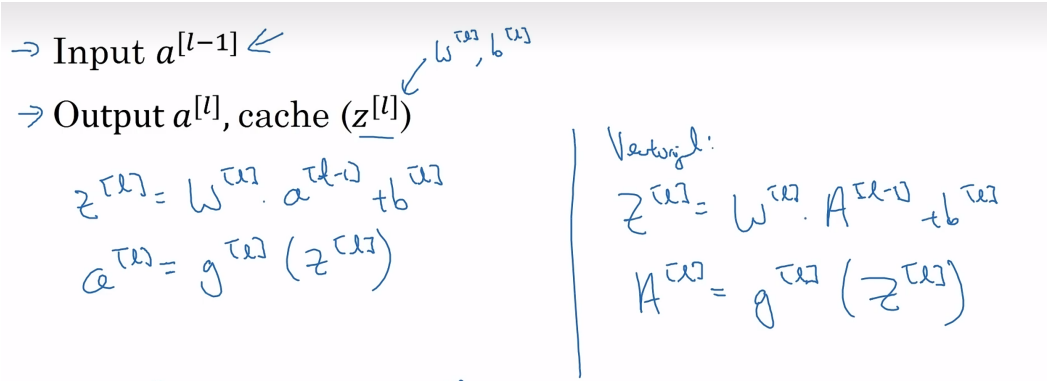

Figure 1.4: forward propagation

1.14.2 Backward propagation

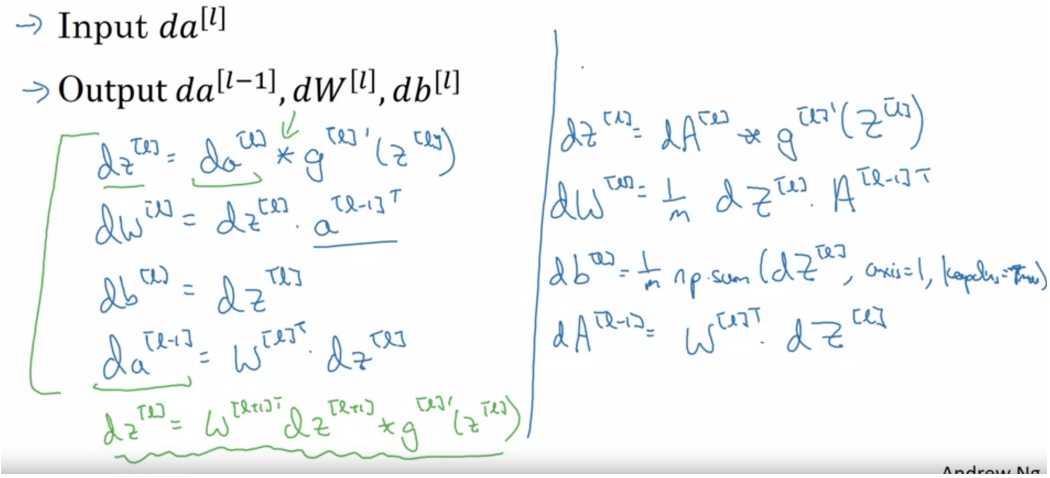

Figure 1.5: Backward propagation

1.15 Hyperparameters Vs Parameters

Definition 1.15.1 Hyperparameters

Parameters that are human chosen instead of function parameters. Learning Rate, α
# of iterations
# of hidden layers, L
# of hidden units, $n^{[i]}$
Choice of activation functions
momentum
mini-batch size
various forms of regularization parameters
.....

Note:-

Deep learning is a very empirical process. Selecting the best hyper parameters using cross-validation or the movement of cost functions during training.

1.16 W4 Homework

Two jupyter notebooks W4 HW and Solutions, Github Repo

- Building your Deep Neural Network - Step by Step
- Deep Neural Network Application - Image Classification

1.17 Outline of Constructing a Neural Network

The figure below shows the forward propagation and back propagation formulas of a L layers deep neural networks using sigmoid as activation function and logistic error as loss function.

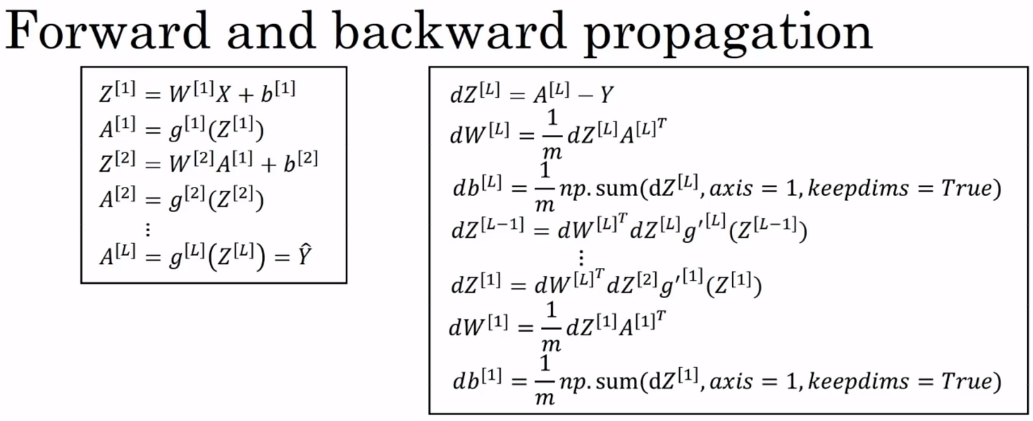

Figure 1.6: forward and backward propagation

To build your neural network, you'll need first implement several "helper functions." Each small helper function will have detailed instructions to walk you through the necessary steps. Here's an outline of the steps in constructing a deep neural network:

Note:-

- Initialize the parameters for a two-layer network and for an L-layer neural network
- Implement the forward propagation module (shown in purple in the figure below)
  - Complete the LINEAR part of a layer's forward propagation step (resulting in $Z^{[l]}$).
  - The ACTIVATION function is provided for you (relu/sigmoid)
  - Combine the previous two steps into a new [LINEAR→ACTIVATION] forward function.
  - Stack the [LINEAR→RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR→SIGMOID] at the end (for the final layer L). This gives you a new L_model_forward function.
- Compute the loss
- Implement the backward propagation module (denoted in red in the figure below)
  - Complete the LINEAR part of a layer's backward propagation step
  - The gradient of the ACTIVATE function is provided for you(relu_backward/sigmoid_backward)
  - Combine the previous two steps into a new [LINEAR→ACTIVATION] backward function
  - Stack [LINEAR→RELU] backward L-1 times and add [LINEAR→SIGMOID] backward in a new L_model_backward function
- Finally, update the parameters

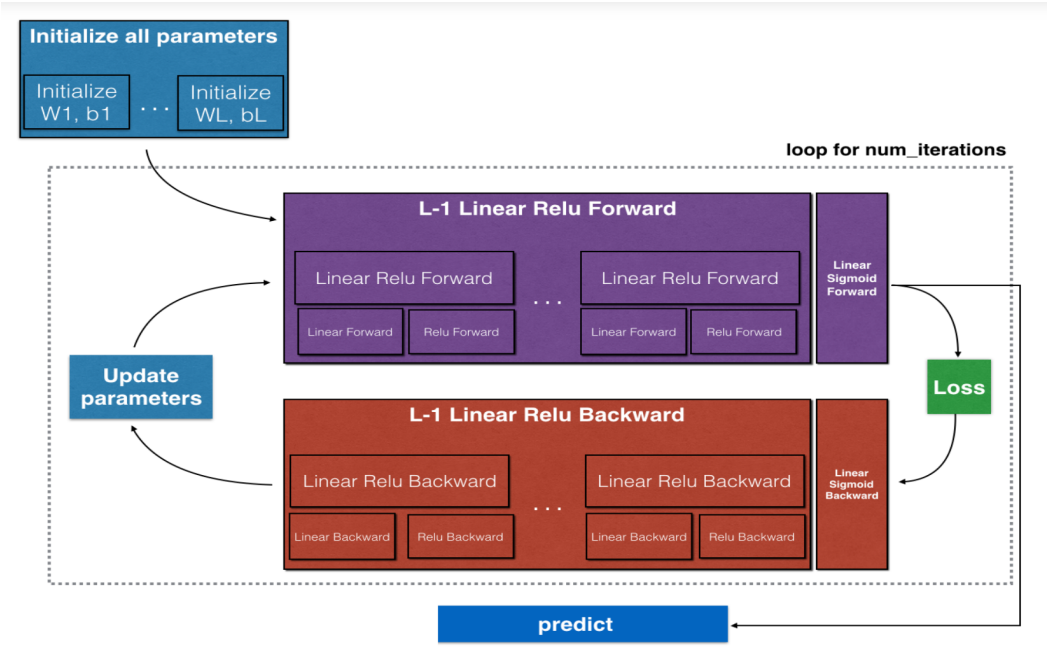

Figure 1.7: Outline of Deep Learning

Chapter 2

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

2.1 Train/Dev/Test sets

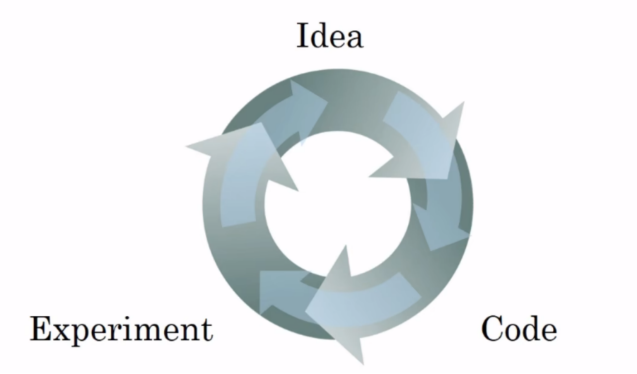

Figure 2.1: Applied DL is a highly iterative process

Definition 2.1.1 Data Set splits

Training set : for training your model
Development Set(Cross-Validation Set) : for getting an unbiased estimates of the performance of different models. **Test Set** : for testing the accuracy of your final model.

Note:-

(70, 30) for train, test split and (60, 20, 20) for train, dev, test split is usually chosen for small data set. For large data set, the size of dev and test is as small as possible as it can correctly choose the best model.

Note:-

Rule of thumb for data set split:
Make sure your train, dev, test set come from the same distribution

2.2 Bias / Variance

Definition 2.2.1 Bias and Variance

Bias:

the difference between the average prediction of our model and the correct value which we are trying to predict

Variance

the variability of model prediction for a given data point or a value which tells us spread of our data

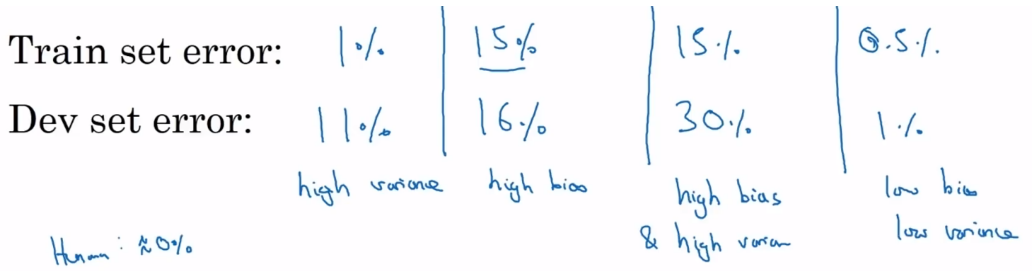

Figure 2.2: Example of bias and variance

Note:-

The judging criterion of above figure is based on human error (optimal error)

Note:-

Bias-Variance Trade-off

In the earlier era of machine learning, there used to be a lot of discussion on what is called the bias variance trade-off. And the reason for that was that, back in the pre-deep learning era, we didn't have many tools, we didn't have as many tools that just reduce bias, or that just reduce variance without hurting the other one. But in the modern deep learning, big data era, so long as you can **keep training a bigger network, and so long as you can keep getting more data**, which isn't always the case for either of these, but if that's the case, then getting a bigger network almost always just reduces your bias, without necessarily hurting your variance, so long as you regularize appropriately. And getting more data, pretty much always reduces your variance and doesn't hurt your bias much.

2.3 Regularization

If you suspect your neural network is over fitting your data, that is, you have a high variance problem, one of the first things you should try is probably regularization. The other way to address high variance is to get more training data that's also quite reliable. But you can't always get more training data, or it could be expensive to get more data. But adding regularization will often help to prevent overfitting, or to reduce variance in your network.

2.3.1 L1, L2 Regularization

Definition 2.3.1 L1 and L2 regularization in Logistic regression

L1 :

$$\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w_i| = \frac{\lambda}{2m}||\mathbf{w}||_1$$

L2:

$$\frac{\lambda}{2m}\sum_{i=1}^{n_x}w_i^2 = \frac{\lambda}{2m}||\mathbf{w}||_2^2$$

m, the number of samples. λ, regularization parameter. n_x, the dimension of $\mathbf{w}$.

To implements regularization in neural networks, we just need do a little a bit change to the above term.

Definition 2.3.2 L1, L2 regularization in NN

L1 :

$$\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}|\mathbf{W}_{i,j}^{[l]}|$$

L2:

$$\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(\mathbf{W}_{i,j}^{[l]})^2 = \frac{\lambda}{2m}\sum_{l=1}^{L}||\mathbf{W}||_F^{[l]}$$

where $||\mathbf{W}||_F^{[l]} = \sqrt{\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(\mathbf{W}_{i,j}^{[l]})^2}$ is called **Frobenius norm**

Note:-

How does Regularization reduce Over-fitting ?

2.3.2 Dropout Regularization

Definition 2.3.3 Dropout Regularization

During training, some number of layer outputs are randomly ignored or "dropped out." This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "view" of the configured layer.

Implements (Inverted dropout)

Dropout is implemented during training.

```
keep_pro = 0.8  # keep probability, hyperparameter
drop_out = np.random.randn(A.shape[0], A.shape[]1) < keep_pro
A = A * drop_out
A = A / keep_pro    # keep the mean of A unchanged
```

Note:-

The keeping probabilities for different layers are different. We usually don't dropout input layer. For layer with many connections, we may should use a lower keep_pro.

Note:-

Why does dropout works ?

2.3.3 Other Regularization methods in NN

Data augmentation(training Image editing) and Early stopping (Stop iteration when dev error reach minima). For complete intro : other regularization methods, Andrew

2.4 Normalize Input

Normalizing input can accelerate the training of neural networks.

Definition 2.4.1 Normalizing Input

Steps:

- Subtract mean :

$$x^i := x^i + \frac{1}{m}\sum_{i=1}^{m} x^i$$

- Divide variance :

$$x^i := x^i / (\frac{1}{m}\sum_{i=1}^{m} x^i **2)$$

(**Note**: The x^i in the second step is the de-meaned x^i, and x^i is vector or one sample)

Note:-

Use Same μ and Same σ^2 to scale your training set and testing set !

2.5 Vanishing and exploding of Gradient

Definition 2.5.1 Vanishing and Exploding of Gradient

When a neural network have many layers, if the parameters are smaller than 1, then the $A^{[l]}$ and dW, db will decrease exponentially, which leads to really slow learning and even stack underflow. This is called **Vanishing gradient**. Exploding gradient is just the opposite (with w greater than 1) .

Definition 2.5.2 Solution to Vanishing and Exploding of Gradient

For sigmoid activation function:

$$\text{To make } Var(w) := \frac{1}{n^{[l-1]}}, \text{ Set} \mathbf{W}^{[l]} = np.random.randn(shape) * np.sqrt(\frac{2}{n^{[l-1]}})$$

(If using ReLU as activation function.)

Other variance:

$$\sqrt{\frac{1}{n^{[l-1]}}} \text{ for } tanh(z) \text{ activation function (Xavier Initialization)}$$

$$\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$$

2.6 Numerical Approximation of Gradients and Gradient checking

For complete introduction :

- Numerical Approximation of Gradients
- Gradient checking, Notes on Gradient checking

2.7 Mini-batch Gradient descent

For large data sets, training all data set(Batch gradient descent) takes a lot of time even using vectorization. By splitting data sets into different mini-batches, we can speed up our training.

Definition 2.7.1 Mini-batch Gradient descent

Splits the data sets $\mathbf{X}$ $\mathbf{Y}$ into different mini-batches ($\mathbf{X}^1$, $\mathbf{X}^2$, $\mathbf{X}^3$...) , ($\mathbf{Y}^1$, $\mathbf{Y}^2$, $\mathbf{Y}^3$...). And use the following mini-batches gradient descent algorithm to train our model:

```
        for i in range(num_of_mini_batches):
            # forward propagation
            # Calculates the cost
            # backward propagation
            # Updates parameters
```

Note:-

- One pass through of entire data set is called one epoch, so the algorithm above is doing one epoch.
- Batch gradient descent updates parameters per epoch, but mini-batch updates multiple times per epoch.

2.7.1 Choosing the size of mini-batch

Definition 2.7.2 Size of mini-batch

For mini-batch size = m :

Batch gradient descent

For mini-batch size = 1 :

Stochastic Gradient descent

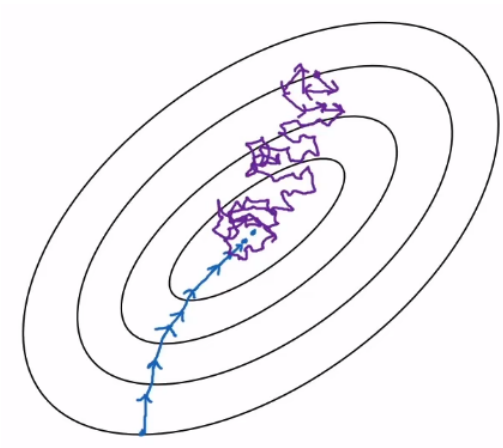

Figure 2.3: BGD and SGD steps on contours

Note:-

- SGD would never reach the true minima
- We choose some size between 1 and m for below considerations:
  - BGD takes too long to update parameters
  - SGD loses the speed-up opportunity using vetorization, which also making training slow.

2.7.2 Guidelines for choosing mini-batch size

- Using BGD for small data set (< 2000).
- Typical mini-batch size : 64, 128, 256, 512 (better using the power of 2 for computer structure reason).
- Make sure mini-batch fits in CPU GPU memory.

2.8 Exponentially weighted averages

Assuming we have a time-series data :

$$V_1, \quad V_2, \quad V3....$$

To smooth the data, we calculates the Exponentially weighted averages of above data :

$$V'_0 = 0, \quad V'_{i+1} = \beta V'_i + (1-\beta)V_{i+1}$$

V'_i are approximately averaging over $\approx \frac{1}{1-\beta}$ days before data. (for example, $\beta = 0.9$ approximately averaging last 10 days' data.) . Then reason goes like this:

$$V'_{i+1} = \beta V'_i \ + \ (\ 1-\beta\ )V_{i+1}$$

$$= \beta(\ \beta V'_{i-1} + (\ 1-\beta\ )V_i\ ) + (1-\beta\ )V_{i+1}$$

$$= (\ 1-\beta\ )V_{i+1} + \beta(\ 1-\beta\ )V_i + \beta^2(\ 1-\beta\ )V_{i-1} + \beta^3(\ 1-\beta\ )V_{i-2}.....$$

And we know that :

$$(\ 1\ -\ \epsilon\ )^\epsilon \approx \frac{1}{e} \quad \text{when } \epsilon \text{ is small}$$

So $\beta^{\frac{1}{1-\beta}} \approx \frac{1}{e}$, which is approximately 0.3, so $(\frac{1}{1-\beta})$th day's data has a weight about $0.3 * 1-\beta$, which is low enough.

2.9 Bias correction

As explained in previous section, V' is used to calculate the exponentially weighted average of V. But it needs some correction for beginning terms.

$V'_1 = (\ 1\ -\ \beta\ )V_1$, which is much smaller than V_1

Definition 2.9.1 Bias correction for exponential weighted average

$$V'_0 = 0, \quad V'_i = \frac{1}{1-\beta^i}(V'_{i+1} = \beta V'_i \ + \ (\ 1-\beta\ )V_{i+1})$$

2.10 Gradient descent with momentum

Definition 2.10.1 Gradient descent with momentum

Compute an exponentially weighted average of gradients and use that gradient to update weights.

```
        # Momentum:
        # on iteration t:
            # compute dw, db on mini-batch:
            Vdw = beta * Vdw + (1 - beta) * dw
            Vdb = beta * Vdb + (1 - beta) * db

            # update weights with Vdw, Vdb
            w = w - alpha * Vdw
            b = b - alpaha * Vdb
```

Note:-

$\beta = 0.9$ works pretty good in practice, and don't need bias-correction since it only takes 10 iteration to "warm up" EWA

Sometimes people omit $(1-\beta)$ in literature, it has same effect as times Vdw or Vdb $\frac{1}{1-\beta}$, so the beta in the" $Vdw = beta * Vdw + dw$ is actually $\frac{\beta}{1-\beta}$ ", or learning rate is changed to $\frac{1}{1-\beta}\alpha$.

2.11 RMSprop (root-mean-square-prop)

```
# RMSprop:
# on iteration t:
    # compute dw, db on mini-batch:
    Sdw = beta * Sdw + (1 - beta) * np.power(dw, 2)  # elements sqaring
    Sdb = beta * Sdb + (1 - beta) * np.power(db, 2)

    # update weights with Vdw, Vdb
    w = w - alpha * dw / np.sqrt(Sdw)
    b = b - alpha * db / np.sqrt(Sdb)
```

With RMSporp, we can use bigger learning rate without worrying divergence.

Note:-

To avoid dividing zero in updating, we usually add a small ϵ in the denominator when updating (10^{-8} is ok)

2.12 Adam optimization algorithm

Adam(Adaptive moment estimation) optimization algorithm is just the combination of momentum and RMSprop (Adam **needs bias correction** though).

```
# Adam optimization algorithm:
# on iteration t:
    # compute dw, db on mini-batch:
    Vdw = beta_1 * Vdw + (1 - beta_1) * dw  # Monmentum
    Vdb = beta_1 * Vdb + (1 - beta_1) * db
    Sdw = beta_2 * Sdw + (1 - beta_2) * np.power(dw, 2)  # RMSprop
    Sdb = beta_2 * Sdb + (1 - beta_2) * np.power(db, 2)

    # Bias correction
    Vdw_corct, Vdb_corct = Vdw / (1 - beta_1 ** t), Vdb / (1 - beta_1 ** t)
    Sdw_corct, Sdb_corct = Sdw / (1 - beta_2 ** t), Sdb / (1 - beta_2 ** t)

    # update weights with Vdw, Vdb
    w = w - alpha * Vdw_corct / (np.sqrt(Sdw_corct) + epsilon )
    b = b - alpha * Vdb_corct / (np.sqrt(Sdb_corct) + epsilon )
```

Hyperparameters choice:

- α , needs to be tune
- β_1, 0.9
- β_2, 0.999
- ϵ, 10^{-8}

2.13 Learning Rate decay

When approaching the optimal, using a smaller learning rate help reach the optimal.

```
# After each epoch :
alpha = 1 / (1 + decay_rate * num_of_epoch)
```

Above code means we scale down learning rate once an epoch.

Note:-

Other learning rate decay:

$$\alpha := 0.95^{n_{epoch}}\alpha$$

$$\alpha := \frac{k}{\sqrt{n_{epoch}}}\alpha \quad or \quad = \frac{k}{\sqrt{t}}\alpha$$

or discrete staircase(constant for a while then decrease then stay then ... like stairs), or we can manually decay instead of using a formula

2.14 the problem of local optima

A lot of people are worrying about getting into local optima when using gradient descent, but it actually pretty hard to reach local optima in very high dimensions, since which requires convex or concave in every dimension. The true problem is plateaus, which is a region where the derivative is close to zero for a long time.

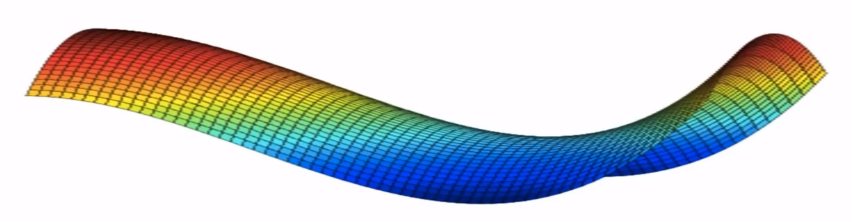

Figure 2.4: The problem of plateaus

2.15 Tuning process

Order of importance of hyperparameters:

- α, learning rate
- β in momentum(0.9), number of hidden units, mini-batch size
- number of layers, learning rate decay
- don't tune Adam parameters ($\beta_1, \beta_2, \epsilon$)

Note:-

- **Try random values, don't use a grid**
- Coarse of fine (random sample first, then sample randomly in the small region where you get good performance)

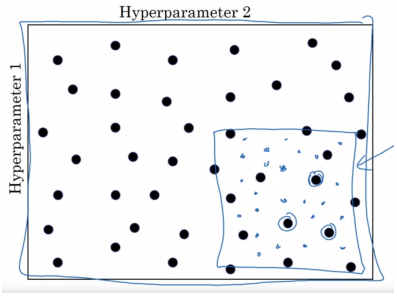

Figure 2.5: Coarse of fine

2.16 Appropriate scale for hyerparameters

Take learning rate as an example:
Assuming α is in the range of (0.0001, 1), if we sample α uniformly at random, 99% times we sample a value in (0.1, 1), so we may need a log scale. (lower bond below is the the power part of lower bound in $10^?$ form, we assume 1 is the upper bound here)

$$\alpha = 10^{**}(\text{lower\_bound} * \text{np.random.rand(1)})$$

Note:-

np.random.rand(k) generates a length k array from a uniform distribution in [0, 1]

For exponentially weighted average parameter β, we first calculate the range of $1-\beta$, then use above log scale to sample.

2.17 Panda Vs Caviar

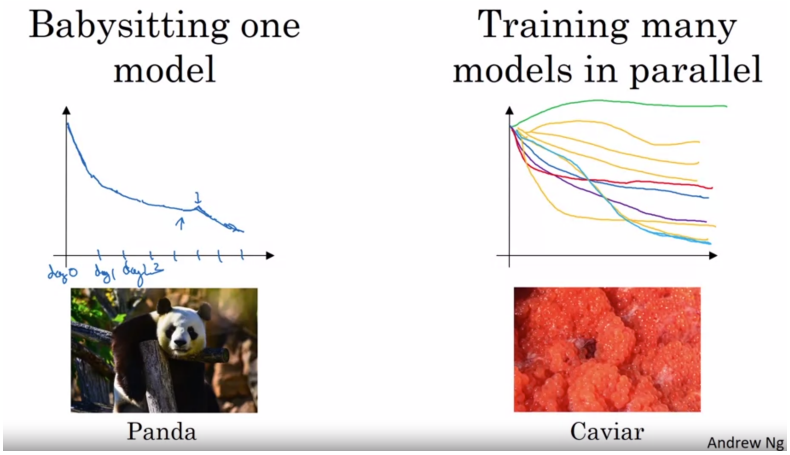

Figure 2.6: Panda and Caviar, by Andrew

2.18 Batch Normalization

We normalized inputs to make training faster before, but we can also normalize the hidden layers instead of only normalizing one input layer. This is what batch normalization does. There is some debate about norm $a^{[l]}$ or $z^{[l]}$ (before activation or after activation), Andrew recommends to norm $z^{[l]}$.

Definition 2.18.1 Batch Norm in a single layer

For nodes in a specific layer l :

$$z^{(1)}, \quad z^{(2)}, \quad ..., \quad z^{(n_l)}$$

$$\mu = \frac{1}{m}\sum z^{(i)}, \quad {}^2 = \frac{1}{m}\sum (z^{(i)} - \mu)^2$$

$$z^i_{norm} = \frac{z^i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \qquad \tilde{z}^{(i)} = \gamma z^i_{norm} + \beta$$

where γ, β are learnable parameters of the model (updates $d, d\gamma$ in training)

Choosing $\tilde{z}$ instead of using z_{norm} is for not constraining the distribution of z into a standard normal distribution but let the model decides.

Note:-

- For dnn, we can just repeat the above process. But since we are subtracting the mean of z when using BN, we won't need parameter b anymore since it will always be subtracted.
- When using mini-batch gradient descent, we will subtract the mean of the given min-batch, won't use same mean of variance
- BN can also works with momentum and RMSprop algorithms.

Why choosing BN :

- First, it normalize the data so can speed up the training.
- BN can avoid "covariance shift". When the distribution of our training set changes, we would have to retrain our model even the mapping of input to output didn't change.(If we train a cat classification model on black cat, we have to retrain one if we want to classify other color cats.)
- When updating parameters, every layer's dw, db(say in layer l) are influenced by the changing of A^{l-1} which is caused by the updating of dw, db in layer (l - 1). So BN can reduce this effect, making each layer more independent and making learning easier.
- BN has a slightly regularization effect(when using mini-batch):
  - Each mini-batch is scaled by the mean/variance computed just on that mini-batch.
  - This adds some noise to the values of z within the mini-batch. So similar to dropout, it adds some noise to each hidden layer's activation.

Note:-

Don't use BN as a way of regularization but a way of normalizing activation to speed learning.

BN process data one mini-batch at a time. When making predictions, we may want to predict a sample at a time, to do so, we use exponentially weight average of every mini-batch's μ and to make predictions. For example, in mini-batches X^1, X^2, X^3....., we can calculates μ in different mini-batches but in same layer l : $\mu^{1[l]}, \mu^{2[l]}, \mu^{3[l]}$, then we calculates the exponentially weighted average of the above sequence. Same for . Then we calculates z_norm and $\tilde{z}$ using above μ, and the γ β learned from our model.(The prediction is actually pretty robust to different ways of calculating μ and , so not very important)

2.19 Softmax regression

The discussion so far is about binary classification, softmax is a generalization of logistic regression that can make prediction of many classes.

Definition 2.19.1 Softmax

Softmax is used in the output layer of the net. Say there are C classes, and the label of different classes are (0, 1, 2...C-1). To implement softmax in the output layer, fist calculate $Z^{[last]}$ as usual, then the activation function is :

$$T = e^{Z^{[last]}},\ T \in \mathbb{R}^{C \times m} \quad \text{(or } \mathbb{R} \times 1 \text{ for one sample)element-wise operation}$$

$$A^{[last]} = \frac{e^{Z^{[last]}}}{\sum_{j=1}^{C} T_i} \quad \sum \text{ in denominator is summing the rows of T}$$

As the above formula shows, the column sum of the output layer equals to one.
The name softmax comes from generating a vector of probabilities instead of a vector with one element equals one, others equal zero(hardmax)

Definition 2.19.2 Training a softmax regression

Loss function(y below is one sample and is of shape $C \times 1$):

$$L(\hat{y}, y) = -\sum_{j=i}^{C} y_j \log \hat{y}_j = -\log \hat{y}_k \quad \text{where k is the row index of 1 in y}$$

Cost function:

$$J(W^{[1]}, b^1, W^{[2]}...) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}$$

Above is for forward propagation, to implement backward propagation, the important part is to calculate dz:

$$dz^{[last]} = \frac{\partial J}{\partial z^{[last]}} = \hat{y} - y$$

2.20 Deep learning frameworks

Criteria for choosing DL frameworks:

- Easy of programming
- Running speed
- Truly open (open source of good governance)

Pytorch

2.21 Homework

- Week 5
  - Initialization
  - Regularization
  - Gradient Checking
- Week 6 : Optimization Methods
- Week 7 : Tensorflow Introduction

Chapter 3

Structuring Machine Learning Projects

3.1 Orthogonalization

Chain of assumptions(or steps) in Machine learning:

1. Fit training set well on cost function
2. Fit dev set well on cost function
3. Fit test set well on cost function (maybe try bigger dev set if not good)
4. Performs well on real world. (change dev set or cost function)

Orthogonalization means focus on one thing at a time, that is why Andrew doesn't suggest using early stopping, which is trying to fitting well on training set and fitting well on dev set at the same time.

3.2 Single number evaluation metric

When doing machine learning works, we may usually write some code then test its performance then improve it. Having a single number evaluation metric makes decision-making and iterating faster. For example in cat classification, if we use two evaluation metric say **Precision** (the percentage of true cat in the cat predictions) and **Recall** (the percentage of cat that is classified of all cats), it is hard to pick one model if two models do well in different evaluation. Using a single evaluation like F1 score ($\frac{2}{\frac{1}{P}+\frac{1}{R}}$) that combines these two evaluation maybe a better choice.
We can also try different ways of combining multiple evaluation metrics, like linear combination, choose one as optimization others as some thresholds(satisfying some condition)...

3.3 Train/dev/test set distributions

One words to conclude:

Random Shuffle your data then split them into train dev, test set.

Sometimes, we should choose a dev set and test set to reflect data you expect to get in future and consider important to do well on.

Note:-

Size of test set Set your test set to be big enough to give high confidence in the overall performance of your model

Human-level-performance is a good measure of the performance of the network and a good proxy for **Bayes error**(Best error we can achieve). As a proxy for Bayes error, human-level-performance can provide us a standard for judging whether the model has a big bias or variance.

Chapter 4

Convolutional Neural Network

4.1 Convolution, Edge detection, Padding and Strided convolution

Definition 4.1.1 Convolution

Say we have a $k \times k$ matrix X and $a \times a$ matrix F, the convolution of two matrix, denoted by X * F, is a $(k-a+1) \times (k-a+1)$ matrix whose i,j element is the sum of element product of X[i : i+a, j : j+a] and F

Definition 4.1.2 Edge detection

The following matrix can be used to do vertical edge detection:

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

The transpose of above matrix can detect horizontal edge. There are other edge detection matrix or we can set the filter as parameter so we can use back propagation to find one:

$$\text{Sobel filter: } \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{Scharr filter: } \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

There are some problems with convolution:

- First, the size of the input image changed.
- Second, pixels at the edge have less weight in the output matrix.

Definition 4.1.3 Padding

Padding expand the matrix to be convoluted so that the size of the output picture keeps original dimension. For Matrix X of shape (n, n) and filer of size (f, f), to keep the dimension unchanged, we pad the image with $\frac{f-1}{2}$ 1×1 borders (n+2k-f+1 = n, k=(f-1)/2).
Padding can be divided into two category:

- **Valid** : No padding
- **Same** : Padding as above so the size unchanged.

Convolution above moves the filter matrix by one step to produce one element of the output matrix, we can move multiple steps.

Definition 4.1.4 Strided convolution

Given a matrix E of size $n \times n$, a filter matrix of size $f \times f$, padding p and stride step s, the output matrix is of size $floor(\frac{n+2p-f}{s} + 1) \times floor(\frac{n+2p-f}{s} + 1)$

Note:-

The convolution above is actually different from convolution defined in math, which defines above as cross-correlation. But by convention in deep learning, we define the one without skipping the filter matrix as Convolution.

4.2 Convolution in volumes

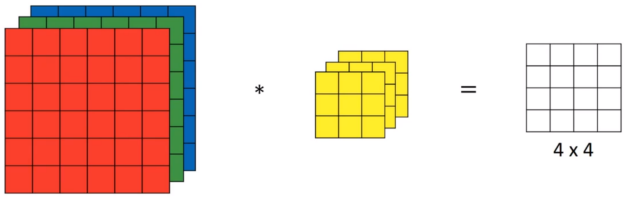

Figure 4.1: Convolution on RGB image

Convolution on 3-D images like RGB images need a 3-D filter whose third dimension or channel in RGB is same as the image or input. Then do as before on the three layers of the image and sum the result to get the output 2-D matrix.
And to apply **multiple filter** on the image, we can stack the output of each filter in the third dimension so we can get a 3-D output.

4.3 One layer of a CNN

One layer of CNN is just the same as one normal layer but with many filters as **W**.

Definition 4.3.1 Parameters in one layer CNN

$$f^{[l]} = \text{filer size}, \quad p^{[l]} = \text{padding}, \quad s^{l} = \text{stride}, \quad n_c^{[l]} = \text{number of filters}$$

$$\text{Input} : n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$$

$$\text{Output} : n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]} \text{ where } n_H^{[l]} = floot(\frac{n_H^{[l]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1)$$

$$\text{Each filer is of size} : = f^{[l]} \times f^{[l]} \times n_c^{[l-1]},$$

$$\text{Activation} : a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]} \text{ for one sample and } A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]} \text{ for m samples}$$

$$\text{Weights} : f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$$

$$\text{bias} : 1 \times 1 \times 1 \times n_c^{[l]}$$

4.4 Layers in a CNN : Conv, Pool, FC

Types of layers in a CNN:

- Conv : Convolution layer as shown in the last section
- Pool : Max pooling and average pooling as shown below.(Usually No padding on this layer)
- FC : Fully connected layer, just a standard neural network.

Definition 4.4.1 Max Pooling

Two hyperparameters :

$$\text{Filter size : } f, \text{ Stride size : } s$$

What max pooling does is use the max value of the region filter as output value instead summing all values.
Even max pooling have hyperparameters, but it has no parameters to learn .
For high dimension image like 3-D, pooling does not change the third dimension like convolution filter.

There is **Average pooling** which does exactly what its name says as max pooling. We can call a pooling layer one layer or a ConV and a Pool layer as one layer, we use latter here. Below shows an example of a convolution neural network for one sample (task : 1-10 digits Recognition. number in [] is the shape or parameter of the layer, number in () is the size of the output):

$$input(32,32,3) \rightarrow ConV1[f=5,s=1,n_{\text{filter}}=8](28,28,8) \rightarrow Pool1[f=2,s=2](14,14,8)$$
$$\rightarrow ConV2[f=5,s=1,n_{\text{filter}}=16](10,10,16) \rightarrow Pool2[f=2,s=2](5,5,16)$$
$$\rightarrow flatten(400,1) \rightarrow FC3[400,120](120,1) \rightarrow FC4[120,84](84,1)$$
$$\rightarrow Softmax[84,10](10,1)$$

4.5 Why convolution

: **Parameter sharing** and **Sparsity of connections**:

- Parameter sharing: A feature detector that is useful in one part of the image is probably useful in another part of the image.
- Sparsity of connections : In each layer, each output value depends on only a small number of inputs.

4.6 Classical networks , ResNet and Inception

Outline:

- Classic networks:
  - LeNet-5
  - AlexNet
  - VGG
- ResNet
- Incetion

4.6.1 LeNet-5

LeNet-5 is used for digit recognition and is trained on grayscale image (one channel).

Definition 4.6.1 LeNet-5

LeNet-5's simplified structure looks like this ($32 \times 32 \times 1$ image as an input example):

$$input(32,32,1) \to ConV1[f=5,s=1,n_{\text{filter}}=6](28,28,6) \to Pool1[f=2,s=2](14,14,6)$$
$$\to ConV2[f=5,s=1,n_{\text{filter}}=16](10,10,16) \to Pool2[f=2,s=2](5,5,16)$$
$$\to flatten(400,1) \to FC3[400,120](120,1) \to FC4[120,84](84,1)$$
$$\to Softmax[84,10](10,1)$$

(Back in 1998, people use average pooling, we now better choose Max pooling)

Today's CNNs have more parameters or layers and use padding a lot. But the patterns that one Pooling layer after one ConV layer then repeat then FC layer and decreasing width, height and increasing channels is always used.

4.6.2 AlexNet

Definition 4.6.2 AlexNet

AlexNet's structure looks like this ($227 \times 227 \times 3$ image as an input example):

$$input(227,227,3) \to ConV1[f=11,s=4,n_{\text{filter}}=96](55,55,96) \to Pool1[f=3,s=2](27,27,96)$$
$$\to ConV2(\textbf{Same Padding})[f=5,s=1,n_{\text{filter}}=256](27,27,256) \to Pool2[f=3,s=2](13,13,256)$$
$$\to ConV3(\textbf{Same Padding})[f=3,s=1,n_{\text{filter}}=384](13,13,384)$$
$$\to ConV4(\textbf{Same Padding})[f=3,s=1,n_{\text{filter}}=384](13,13,384)$$
$$\to ConV5(\textbf{Same Padding})[f=3,s=1,n_{\text{filter}}=256](13,13,256) \to Pool3[f=3,s=2](6,6,256)$$
$$\to flatten(9216,1) \to FC3[9216,4096](4096,1) \to FC4[4096,4096](4096,1)$$
$$\to Softmax[4096,1000](1000,1)$$

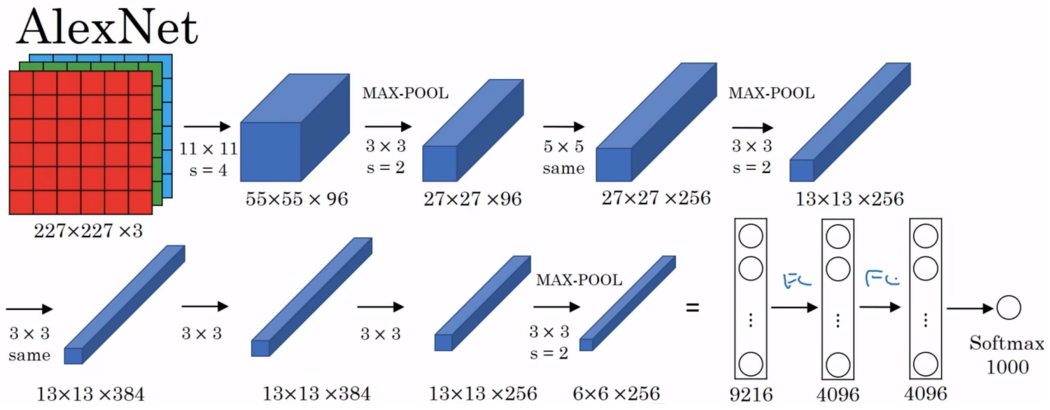

Figure 4.2: AlexNet

4.6.3 VGG-16

VGG simplifies AlexNet by using uniform shape ConV and Max-Pooling:

$$ConV = 3 \times 3 \text{ filter}, s = 1, same \text{ Max-Pool} = 2 \times 2, s = 2$$

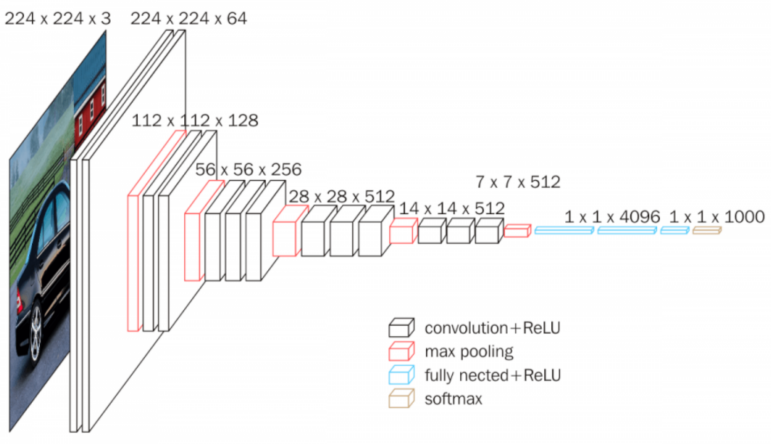

Figure 4.3: VGG-16

4.6.4 Residual Networks

ResNet : Deep Residual Learning for Image Recognition

Definition 4.6.3 Skip connection

For a normal (or plain in the above paper) two layer network, it have only one path like below:

$$a^{[l]} \rightarrow Linear \rightarrow ReLU \rightarrow a^{[l+1]} \rightarrow Linear \rightarrow ReLU \rightarrow a^{[l+2]}$$

But skip connection can connect $a^{[l]}$ directly with $a^{[l+2]}$ without going through $a^{[l+1]}$:

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

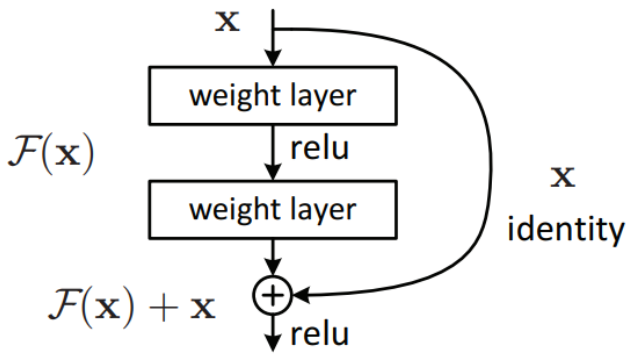

Figure 4.4: Skip connection (or short connection) from ResNet paper figure 2

By stacking many blocks like above together, we get the ResNet.

4.7 Network in networks and One by one Convolution

For one channel image, one by one filter just multiple the image by a constant, but for a image with many channels, a one by one filter is doing something like a fully connected neural network the each element in the image.
This is also called network in network, which can used to reduce the size of channels as compared to Pooling layers for reducing height and width. (Network In Network, Min Lin, Qiang Chen, Shuicheng Yan)

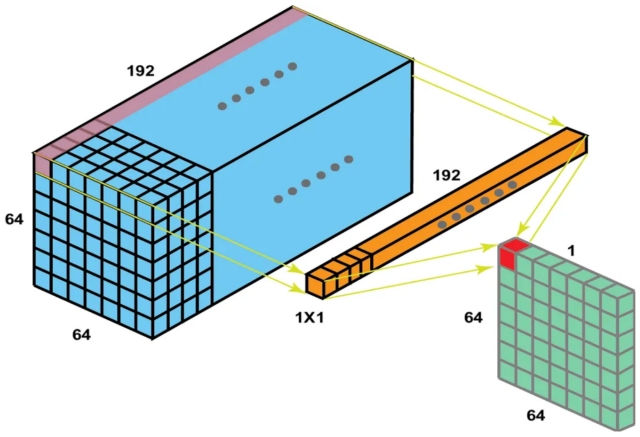

Figure 4.5: One by one filter

4.8 Inception Network

For all we have learned, we might not know what layer to choose or what size filter to use, Inception solves this problem by using them all and stack their result on the third dimension (or channel dimension). To keep height and width same, inception uses same padding (and stride=1) for Max-Pool, which is not unusual in other architectures.

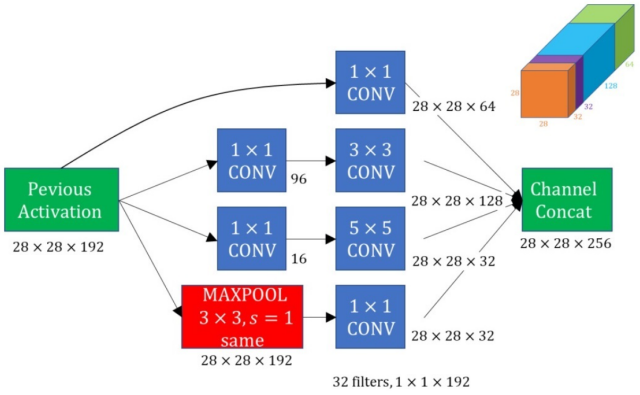

Figure 4.6: An example of Inception module

The problem of inception is **high Computation cost**. Just take one 5×5 convolution filter of above as example, for each value in the output block we need $28 \times 28 \times 192$ times calculation, for calculate the entire block once we need to calculate $28 \times 28 \times 32 \times 28 \times 28 \times 192 \approx 3.77 \times 10^9$ times. To improve this problem, we use a **Bottleneck layer**, which is a one-by-one convolution filter that reduce the amounts of channels.

By combining above module and add Max-Pool in between for reducing height-width and use some FC layers, a softmax layer at the end of last fewer inception modules, we get the inception network or google net since it's developed by google.

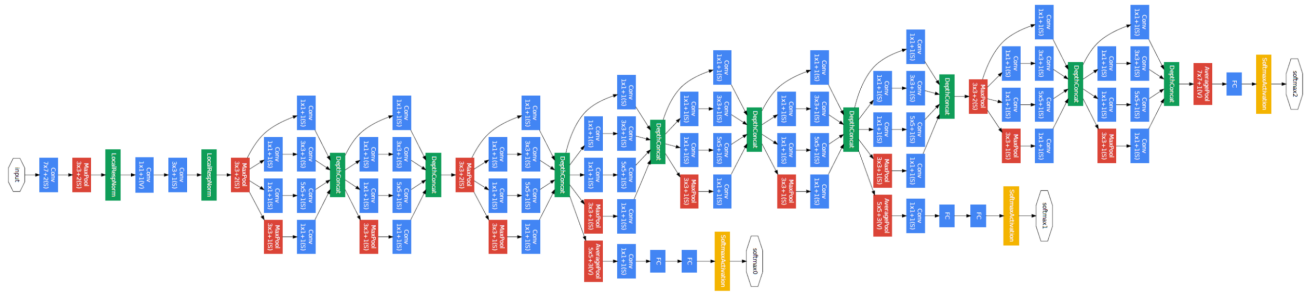

Figure 4.7: GoogleNet

4.9 Transfer learning, Data augmentation

Transfer learning:

- If we only a small amount of data, we can just remove the pre-trained model's original softmax layer and add our own softmax and frozen layers' parameters before the softmax.
- For having large amount of data, we can choose to frozen smaller amount of parameters of the pre-trained model or directly train all parameters, which called fine-tuning.

Data augmentation(often used in CV):

- Mirroring
- Random Cropping
- or Rotation, Shearing, Local warping ...
- Color shifting.

Chapter 5

Sequence Models

Notation:
Here we use $x^{[i]<t>}$ to represent the t th element of i th sample of x. The t th element may be an element in a time series or a word in words. For example, we have one temperature series : $[20, 21, 18, 28]$, the t th element would the t th day's temperature. If we have a sequence of words : Harry is my friend. The t th element would be the t th word. The length of x is donoted as T_x, and the length may not be same among samples so $T_x^{(i)}$ also have a superscript to represent the i th sample's length.

5.0.1 Data representation

Definition 5.0.1 Representing words in NLP

To represent words in model, we need first define a dictionary:

[a, ..., back, ... zoom...]

Then we use the index of the word to represent the word.
Sometimes we maybe using a small dictionary which can't represent all words in our training data set at which we should include the most frequently appeared words.
After setting the dictionary, we use one hot representation(making every word a vector of len(dictionary) whose k th element is 1 and all others are 0 (k is the index of the word in the dictionary)). So x_t is a $len(dictionary) \times 1$ vector, one sample x is a matrix.

5.1 Recurrent Neural Networks

Why not using a standard deep net?

- Because of the length of the input and output may change, standard deep net is not a good choice.
- Don't share features learned across different positions of text.

Definition 5.1.1 Forwardpropagation in RNN

$$a^{<0>} = \overrightarrow{0}$$
$$a^1 = g_1(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$$
$$y^{<1>} = g_2(W_{ya}a^{<1>} + b_y)$$
$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t-1>} + b_a)$$
$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

(W_{aa}, W_{ya}, W_{ax} is same over t !)

g_1 and g_2 are different activation functions for the a and y. tanh is often used for g_1 and the choice of g_2 depends on the task of the model(what output y we want).
For notation convenience, we stack W_{aa} and W_{ax} horizontally and stack $a^{<t-1>}$ $x^{<t>}$ vertically:

$$W_a = \begin{bmatrix} W_{aa} & W_{ax} \end{bmatrix}, \quad \begin{bmatrix} a^{<t-1>}, x^{<t>} \end{bmatrix} = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$$

So the equation above becomes:

$$a^{<t>} = g_1(W_a \begin{bmatrix} a^{<t-1>}, x^{<t>} \end{bmatrix} + b_a)$$

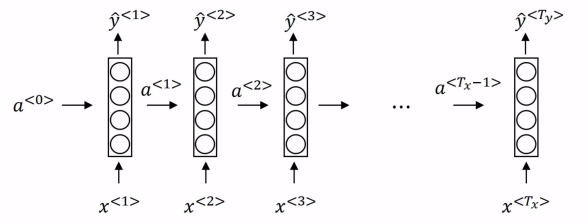

Figure 5.1: Forward propagation in RNN (T_x equals T_y)

Back propagation through time

5.2 Different Architectures of RNN

We talked about a T_x equals T_y RNN last section, which is also called many-to-many architecture. But most of time the data in NLP, speech recognition or translation have different T_x and T_y, so we need more architecture of RNN to handle these problems. Below are some of these architectures:

- **One-to-one** and **Attention architecture**.
- **Many-to-many** : Figure 5.1 (T_x or T_y nodes since T_x equals T_y)
- **Many-to-one** : Only produce y in the last node (T_x nodes).
- **One-to-many** : Using x in the first node then keep produce y without adding x (T_y nodes).
- **Many-to-many but with different length** : Don't produce y until all x comes in ($T_x + T_y$ nodes, and the T_x part is called **encoder**, T_y part is called **decoder**).

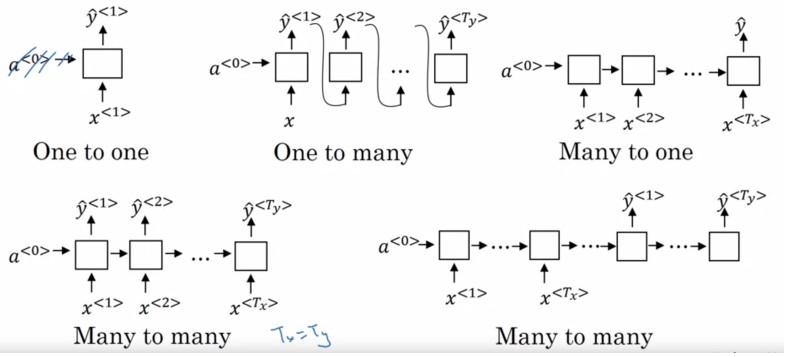

Figure 5.2: Different Architectures in RNN

5.3 Language model and Sequence Generation

Definition 5.3.1 Language model

A language model is a probability distribution over sequences of words.[1] Given any sequence of words of length m, a language model assigns a probability P(y_1, y_2, y_3...) to the whole sequence.
To train a language model, you first need a training set comprising of a large corpus of English text(or other language)
Corpus is an NLP terminology that means a large body or a large set of english text of english sentences.

To represent corpus in computer, we first need tokenize the sequences in the training set to get a vocabulary and map each word into one-hot vectors. Usually we add an extra token called a **EOS**(end of sentence) that helps capture the end of sequence. (For words not in the dictionary when dic size is small, we include token UNK-unknown words to represents those out of dic words).

After finishing tokenization step, we build RNN to model the chance of different sequences. The figure below shows the architecture of language model RNN, the activation function for y (g_2) is a softmax funcion for producing the probabilities.

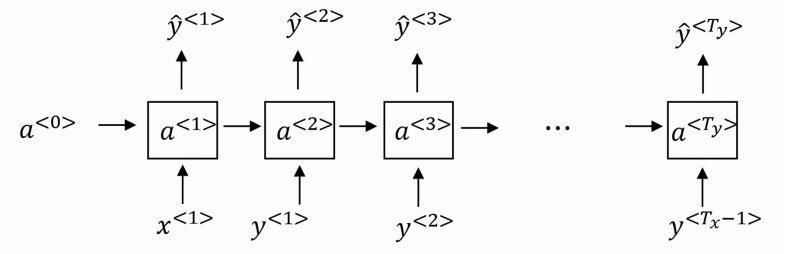

Figure 5.3: RNN for language model

Remember that g_2 is a softmax function so y_1 is calculating the probabilities of every word in the dictionary(plus EOS UNK), and because the input of y_2 is the y_1, so y_2 is the probabilities **conditioned on** y_1 of every word in the dic. And y_3 is the probabilities **conditioned on** y_1 **and** y_2 of every word in the dic. And so on. Then it's natural to derive the loss and cost function of language model:

$$L(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

$$J = \sum_t L(\hat{y}^{<t>}, y^{<t>})$$

5.4 Sampling novel sequences

To see the performance of the model, we use the model to sample a sequence since it produces a probability matrix. First randomly sample y_1 then use y_1 to get the probability distribution of y_2(hard encoding $y-1$ as input) ,then use y_1 and y_2 to calculate the probability distribution of y_3 and so on.

We can also build a character level vocabulary RNN using (a-z A-Z , . ? ! .....). The benefit of using this kind of vocabulary is that we don't have to worry about UNK, but also we end up with much long sequences models, which are much hard to train and may not be as good as character level RNN models at capturing the long range dependencies.

5.5 Vanishing Gradients with RNNs

RNN's main problem is vanishing gradients, which leads to weak ability to capture long term relation. Exploding gradient is not a very big problem compared to vanishing gradients since overflow values are easy to see (NaN). One solution to exploding is **gradient clipping** : If gradient vectors are bigger than some threshold, re-scale some vectors so that it is not too big. But vanishing is much harder to solve.

5.6 Gated Recurrent unit

Definition 5.6.1 GRU (Gated Recurrent unit)

To strengthen the long term memory of RNN, GRU (gated recurrent unit) includes a C or memory cell in the net:

$$c^{<t-1>} = a^{<t-1>}$$

$$\tilde{c}^{<t>} = tanh(W_c \left[\Gamma_r * c^{<t-1>}, x^{<t>}\right] + b_c)$$

$$\Gamma_u = \sigma(W_u \left[c^{<t-1>}, x^{<t>}\right] + b_u) \quad \text{Update gate}$$

$$\Gamma_r = \sigma(W_r \left[c^{<t-1>}, x^{<t>}\right] + b_r) \quad \text{Relevance gate}$$

$$a^{<t>} = c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

As above shows, $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$, Γ is called the gate, which controls whether keep former c or not.

And because $c^{<t>}$ comes from the combination of two parts, $c^{<t>}$ won't suffer gradient vanishing even Γ_u is really small. The reason that GRU takes this form comes from many researchers' hardworking and the form above is most robust and useful for many problems, you can also try different gate controls.

5.7 Long short term memory units

Definition 5.7.1 LSTM

The difference with LSTM and GRU is that in LSTM, $a^{<t>}$ won't equals to $c^{<t>}$, no relevance gate in $\tilde{c}^{<t>}$ and add a new forget Γ_f instead of using $1 - \Gamma_u$:

$$\tilde{c}^{<t>} = tanh(W_c \left[a^{<t-1>}, x^{<t>}\right] + b_c)$$

$$\Gamma_u = \sigma(W_u \left[a^{<t-1>}, x^{<t>}\right] + b_u) \quad \text{Update gate}$$

$$\Gamma_f = \sigma(W_f \left[a^{<t-1>}, x^{<t>}\right] + b_f) \quad \text{Forget gate}$$

$$\Gamma_o = \sigma(W_o \left[a^{<t-1>}, x^{<t>}\right] + b_o) \quad \text{Output gate}$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}, \quad a^{<t>} = \Gamma_o * tanh(c^{<t>})$$

There are also variants of LSTM, like peephole connection(stack c^{t-1} when calculating gates like GRU)

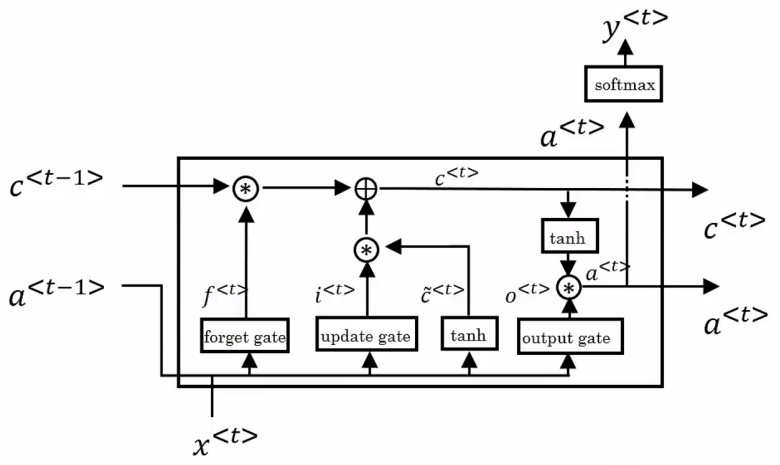

Figure 5.4: Long short term memory unit

5.8 Bidirectional RNN

Definition 5.8.1 BRNN

To take the information of both earlier and later sequences, BRNN is constructed by adding a backward recurrent layer whose forward propagation is calculated from left to right. And $y^{<t>}$ is calculated as :

$$\hat{y}^{<t>} = g(W_y)\left[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}\right] + b_y$$

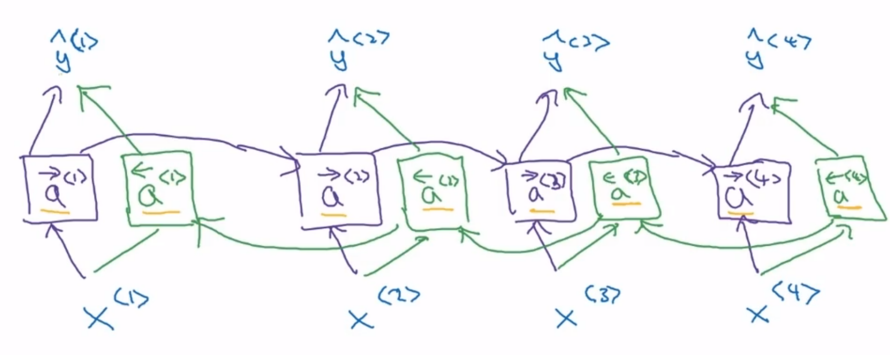

Figure 5.5: Bidirectional RNN

5.9 Deep RNNs

Definition 5.9.1 DRNN

We can also take $a^{<t>}$ as new $x^{<t>}$ to add more vertical layers. The resulting net is called deep RNN. Usually, because of the deep of horizontal layers, there won't be much vertical layers. But vertical layers can also be traditional networks that don't have horizontal connections.

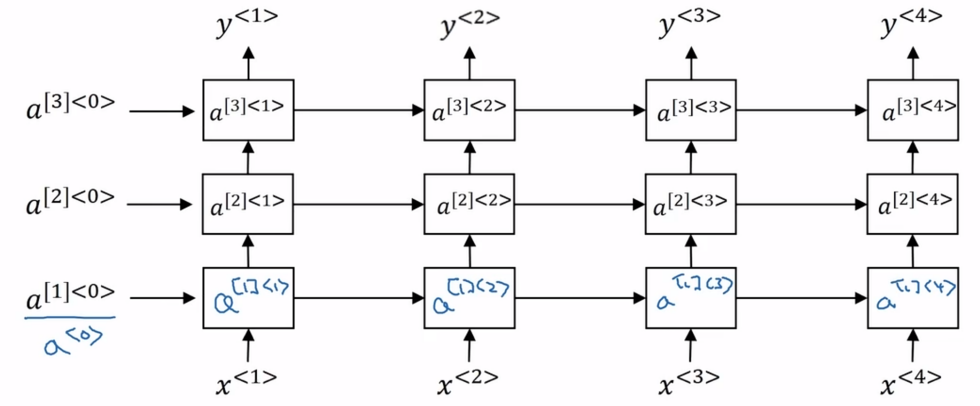

Figure 5.6: Deep RNN

5.10 Word Representation

So far we have learned one hot representation, which turn a word into a vector. For example word "jack" is in the 5432th place of the dictionary, then the one hot representation of jack is a column vector whose 5432th element is 1 and all other elements are 0 with dictionary's length. The main weakness of this representation is that words lost relations after one hot transformation - The dot or inner product of any pair of word is zero. So to solve this, we can use featurized representation in NLP : word embedding.

	Femininity	Youth	Royalty
Man	0	0	0
Woman	1	0	0
Boy	0	1	0
Girl	1	1	0
Prince	0	1	1
Princess	1	1	1
Queen	1	0	1
King	0	0	1
Monarch	0.5	0.5	1

Figure 5.7: Example of word embedding

Note:-

Transfer learning and Word embeddings

- Learn word embedding from large text corpus.(1-100B words)
- Transfer embedding to new task with smaller training set
- Optional : Continue to fine tune the word embedding with new data.

5.11 Properties of Word embedding

It turns out that word embedding can do analogies:

- Say we choose 300 words to do word embedding, and denote the resulting vector of woman, man, king, queen as $e_{woman}, e_{man}, e_{king}, e_{queen}$. So if $e_{woman} - e_{man} \approx e_{king} - e_{queen}$, algorithm can do the analogy that man-woman is approximately king-queen.

To achieve above using an algorithm, we can first set the problem as find a word w that the pair king-w is similar to man-woman, then what we can do is the find the w in the feature space that w maximize the similarity$(e_w, e_{king} - e_{man} + e_{woman})$, here similarity function have many options, most used one is called cosine similarity:

$$sim(u, v) = \frac{u^T v}{||u||_2 ||v||_2}$$

Euclidian distance ($||u - v||^2$) is also an option but we need to take negative of it. And sometimes to visualize the feature space of word embedding, we use **t-SAE** to map, say, 300-D vector into a 2-D space to we can draw it on the paper.

5.12 Embedding Matrix

To use, say, 300 words to do word embedding representation of, say, 100000 words, we will end up getting a 300×100000 dimensional matrix called **E**. And to get the 6500th word - orange's representation e_6500, we can matrix multiple **E** with the one-hot vector of orange o_{6500} (Here the one-hot vector would be a 100000×1 vector where its 6500th value is 1 and others are 0).

Note:-

We do not use matrix multiplication in practice to find e_i since it is inefficient, just use some search or index method instead.

To learn a word embedding matrix, we can start from difficult algorithms since they are easier to understand. It turns out building a neural language model (given a sequence to predict the next word) is a reasonable way to learn an embedding matrix.

Definition 5.12.1 Neural language model

This comes from Bengio et al.,2003, A neural language model :
First we initialize $\mathbf{E}$, then use $\mathbf{E}@o_i$ to get e_i, then combine these vectors into one vector as the input of net. (Usually we use a historical window to choose the length of words before the target. If we use 4 words window and a 300 e, then the input nodes of the net would be 4×300). And the net has one hidden layer and a softmax output layer.

Above algorithm choose 4 words before as the **context** of the target words, we actually can also use words **after** the target to get embedding since we are trying to find the relation of words. And it turns out a text with only two words (one before the target and one after) is enough(**Skip-Gram model**)

Efficient Estimation of Word Representations in Vector Space

Definition 5.12.2 Word2Vec Skip-Gram model

Same as neural language model, we first build a NLM, but the the task now becomes given a one word context, randomly choose a word in a given length interval around the context word as the target word, then use these data to train a model predicting the probability of appearing target word given context word.

$$0_c \rightarrow E \rightarrow e_c \rightarrow Softmax \rightarrow \hat{y}$$

The main problem of skip-gram model is the computation speed, the softmax computation - $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$ is really time-consuing since need adding the length of dictionary(1000 here) times to get one probability. One way to solve this is using a **hierarchical softmax classifier**, which replace one softmax node into a binary tree(bisection). Another way is called **nectar sampling**, which will be introduced in the next section.

5.13 Negative sampling

Distributed Representations of Words and Phrases and their Compositionality

First we need to **generate the training set**:

- Pick a context word and randomly pick a target word around the context word as the feature(x) of first example, set output be 1
- For some number of times k, take the same context word and pick random word from the **dictionary** as new k rows where all y are 0.
- Repeat above.

Note:-

Mikolov and other authors recommend k = 5-20 for small data sets and 2-5 for bigger data sets. And use $P(w_i) = \frac{f(w_i)^{3/4}}{\sum_j f(w_j)^{3/4}}$ to random pick words from dictionary(f is the frequency of the word in language or training corpus).

Then we will train a **logistic classification** model:

$$P(y = 1|c, t) = \sigma(\theta_t^T e_c), \quad e_c = \mathbf{E}o_c$$

5.14 Glove word vectors

GloVe stands for global vectors for word representation. See GloVe, Wikipedia

5.15 Sentiment Classification

There are several ways to do sentiment classification

- Simple model: average the embedding representation(may comes from other large training set) of word in words the put it in a softmax. **Problem** : The oreder of word is not taken into consideration.
- RNN : Use many-to-one RNN to do classification.

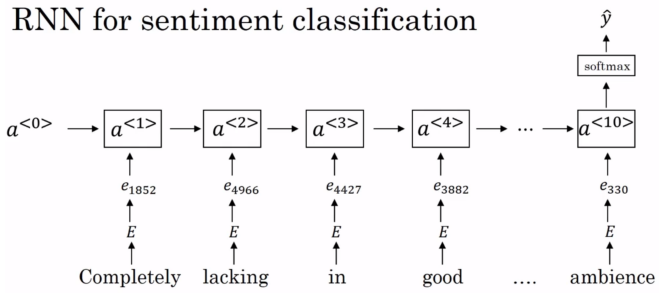

Figure 5.8: RNN for sentiment classification

5.16 Debiasing word embeddings

Word embeddings can reflect gender, ethnicity, sexual orientation bias of the text used to train the model. Steps for debiasing:

- Identify bias direction.
- Neutralize : For every word that is not definitional, project to get rid of bias.
- Equalize pairs.

Note:-

For complete intro : Andrew, Bilibili

5.17 Basic models

To do translation works in deep learning, we can use a **Seq2seq model**(Sequence to sequence model):

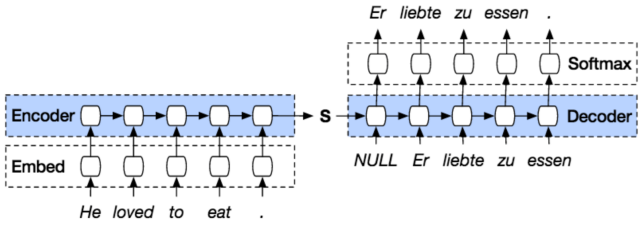

Figure 5.9: Seq2seq for Machine translation

Seq2seq machine translation model is actually a conditional language model, since the decoder part of the model can be regarded as a language model whose $a^{<0>}$ is the output of encoder instead of a zero vector.

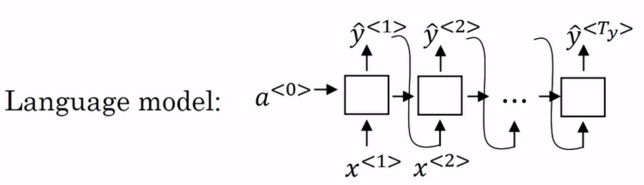

Figure 5.10: Language model

When doing translation, we don't want to sample from the model but choose the most likely output of the model. The most common algorithm for searching highest joint probability output is **beam search**. Greedy search usually does not work well.

5.18 Beam search

Beam search is actually a generalization of greedy search.

Definition 5.18.1 Beam search

Bead search has a parameter, say, k.(when k=1 we get greedy search).
To implement beam search in machine translation, consider figure 5.9 again, at the first element of the decoder part $y^{<1>}$, we pick the k candidates whose have highest probabilities in the softmax matrix, then take them as the input of $y^{<2>}$ respectively. Then we pick k candidates who have highest **joint probability** ($Yy^{<1>} \times y^{<2>}$) among k*len(dic) values. Then keep doing this.

Some refinements:

- Because the joint probability has many multiplications, there is a risk of underflow. So choosing log probability is preferred.
- Since probabilities are not greater than one, beam search tends to output short sentences. Normalizing the joint probability helps solve this problem(α below is between 0 and 1, 1 is full normalization, 0 is no-normalization, 0.7 is usually chosen).

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

Definition 5.18.2 Error analysis of beam search

To determine whether the RNN has error or beam search has error, we need first include a human translation as comparison(denote as y^*).

- If $P(y^*|x) > P(\hat{y}|x)$, then Beam search is at fault
- If $P(y^*|x) < P(\hat{y}|x)$, then RNN is at fault.

5.19 Attention model

Encode-decoder RNN has the problem that it has bad performance on long sentences, it's hard to "memorize" the hole sentence then translate it. Humans do reading and translation at the same time, and that's what attention model tries to imitate - look at **part of sentences** at a time.

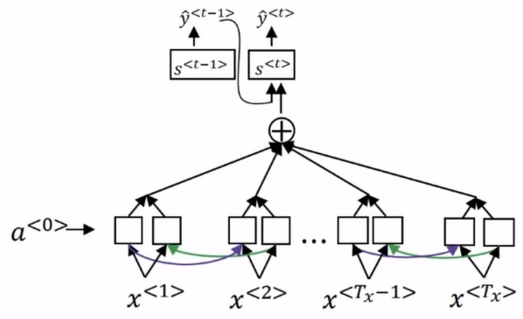

Figure 5.11: Attention model with a bidirectional layer as attention generator

Note:-

Andrew, Bilibili

5.20 Speech recognition and Trigger word detection

For complete intro :

Speech Recognition, Trigger word detection

5.21 Transformer Network

RNN, GRU and LSTM are all sequential models which require to compute all units before to continue the next one. Transformer architecture allows to run a lot more of computations for an entire sequence in parallel. (Vaswani et al. 2017, Attention is all you need)
Transformer Network Intuition: Attention + CNN : Self-Attention and Multi-Head Attention

5.22 Self, Multi-head Attention and Transformer net

Definition 5.22.1 Self attention

$$A(q, K, V) = \text{Attention based vector representation of a word}$$

For each word we calculate its attention based vector representation as follows:

$$A(q, K, V) = \sum_i \frac{exp(qk^{<i>})}{\sum_j exp(qk^{<j>})} v^{<i>}$$

For each word, it have q(query), k(key) and v(value), which are the input vectors to compute above transformer attention.

$$q^{<i>} = W^Q x^{<i>},\ k^{<i>} = W^K x^{<i>},\ v^{<i>} = W^V x^{<i>},$$

$$\textbf{Attention}(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Each time we calculate the self-attention is called a head, multi-head attention is calculating multiple times of self-attention.

Definition 5.22.2 Multi-head Attention

Instead of using q, k, v, we multiple q k v a weight so that they becomes:

$$W_1^Q q^{<i>},\ W_1^K k^{<i>},\ W_1^V v^{<i>}$$

So the first head attention becomes :

$$\textbf{Attention}(W_1^Q Q, W_1^K K, W_1^V V)$$

Then repeat above calculation with W_2^Q, W_2^K, W_2^V...

$$\textbf{MultiHead}(Q, K, V) = concat(head1, head2, ...)W_o$$

$$headi = \textbf{Attention}(W_i^Q Q, W_i^K K, W_i^V V)$$

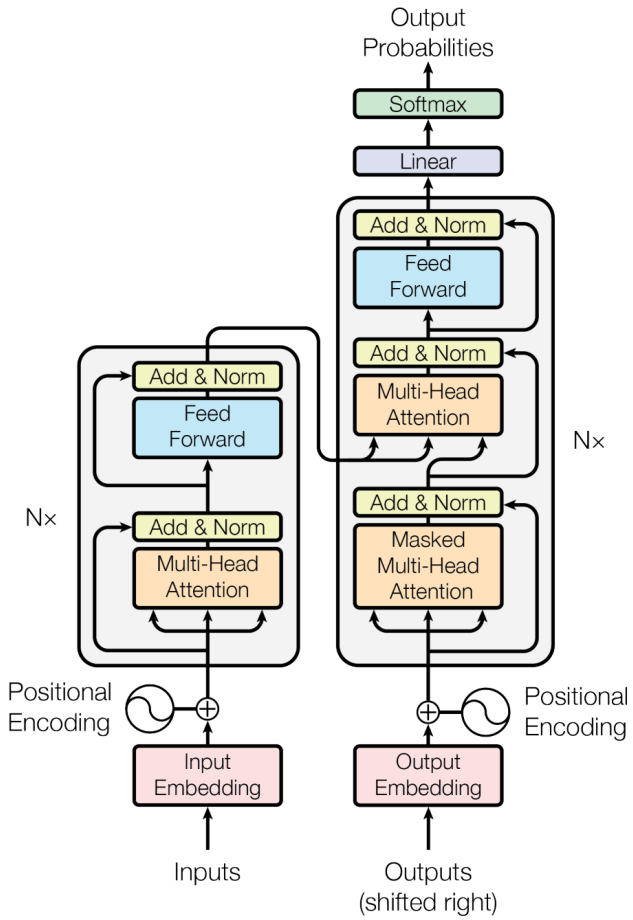

Figure 5.12: Transformer architecture, source: Attention is all you nee