Deep Learning Notes

Yiqiao YIN
Statistics Department
Columbia University
Notes in LaTeX

February 5, 2018

Abstract

This is the lecture notes from a five-course certificate in deep learning developed by Andrew Ng, professor in Stanford University. After first attempt in Machine Learning taught by Andrew Ng, I felt the necessity and passion to advance in this field. I have decided to pursue higher level courses. The materials of this notes are provided from the five-class sequence by Coursera website. [1]

[1] I want to specially thank Professor Andrew Ng for his teachings. This knowledge provide crucial guidance for the formulation of my own company, Yin's Capital, LLC..

This note is dedicated to Professor Andrew Ng and all my friends.

Contents

1 NEURAL NETWORKS AND DEEP LEARNING

Go back to Table of Contents. Please click TOC

1.1 Welcome

The courses are in this following sequence (a specialization): 1) Neural Networks and Deep Learning, 2) Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization, 3) Structuring your Machine Learning project, 4) Convolutional Neural Networks (CNN), 5) Natural Language Processing: Building sequence models (RNN, LSTM)

1.2 Intro to Deep Learning

Suppose the following data set: we have a collection of observations in housing prices. We have, for each observation, the size of houses and the price of houses. We have a simple neural network: Size $\rightarrow \odot \rightarrow$ Price, which we can write $x \rightarrow \odot y$. This linear regression model is called Rectified Linear Unit (ReLU). A larger neural network is formed by taking a group of single neurons and stacking them together. A larger data set can have the following structure: size and # bedrooms imply family size, zip code implies walk-ability, zip code and wealth imply school qualify. Then family size, walk-ability, and school quality would imply price (or determine price). This will give us a net-shape structure, with the first layer size, # bedrooms, zip code, and wealth, while the second layer family size (or instead, the first node of the first layer), the second node, and the third node. The third layer will give us a price.

There are supervised learning and unsupervised learning. For supervised learning, we have a given output (y). Such applications have successful results in real estate, online, advertising, photo tagging, speech recognition, machine translation, autonomous driving. The goal would just to be trying to fit neural networks to these data sets. In real estate, it is standard neural networks that work out well. For photo tagging, it is CNN that works well. For sequence data such as speech recognition and language translation, RNN and improved RNN work out well. Autonomous would work better with CNN or custom CNN (hybrid CNN).

There could also be the difference of structured and unstructured data. For data set such as predicting housing price given size, # of bedrooms, etc. variables, these data sets are structured data because they are well defined in each covariate. Unstructured data could be audio, images, or texts. These are unstructured data sets. In fact, human brains have evolved to be good at interpreting unstructured data. Thanks to neural network, we are able to attempt the field of unstructured data with machines.

Concepts of deep learning have been around for about a hundred years. It only had been taking off for the past few years. To answer this question, we consider a graph that takes the following information. The x-axis is mount of data and the y-axis is the performance, we would see a curve griding up and pretty much go horizontally with large data. That is, facing large data (especially modern days in digital world), traditional learning algorithms have been slowly performing worse and worse as we collect more and more data. This is not the case for neural network. Neural network is able to train a huge data set and is able to perform well especially with large-scale data sets. We are going to use m to denote size of data sets. For small training sets, large neural network is still able to perform better off. The scale, data, computation, and algorithms, are driving deep learning progress. Using sigmoid functions, the progress may be extremely slow when it hits the part with small slopes. ReLU functions do

not suffer this problem and can be trained faster, yet ReLU can easily converge to one because of its linearity and training may not hit optimal point yet. Gradient for ReLU is much less likely to shrink to zero. Just by switching to sigmoid functions from ReLU functions, it allows to train neural network at a much larger scale. Idea is written down in codes. Codes can be trained and tested in experiment. The results of experiments provide us ideas.

Here we provide an outline of this Course: 1) Introduction, 2) Basics of Neural Network programming (forward propagation and backward propagation), 3) One hidden layer Neural Networks, 4) Deep Neural Networks.

1.3 Logistic Regression as Neural Network

1.3.1 Binary Classification

Logistic regression is a binary classification model. For example, inputing a cat image, the output label would be 1 == cat if it is a cast, or 0 == not cat if it is not a cat. For these pictures, we can take a pixel-format image as one observation. Set the size as 64 by 64 and in three colors. We could have a total of $n_x = 64 \times 64 \times 3 = 12288$, which is the number of pixels.

A single training example is represented as (x, y) while $x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$. Then we have m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$. Say we have m training samples, then we have $m_{\text{test}} = \#$ test examples. Thus, we can put together a matrix

$$\underbrace{\begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}}_{n_x \text{ rows }, m \text{ columns}}$$

while we have $\mathbf{X} \in \mathbb{R}^{n_x \times m}$. For computation to be easier, we want to stack output together as well. Set

$$Y = [y^{(1)}, y^{(2)}, ..., y^{(m)}]$$

while $Y \in \mathbb{R}^{1 \times m}$.

1.3.2 Logistic Regression

Given x, we want to output $\hat{y} = P(y = 1|x)$ the likelihood $y = 1$ (yes, it is a cat picture) given x. The parameters $\omega \in \mathbb{R}^{n_x}, b \in \mathbb{R}$. The output we would use sigmoid function, $\hat{y} = \sigma(\omega^T x + b)$. Let $z = \omega^T x + b$ and denote $\sigma(z) = \frac{1}{1+\exp(-z)}$. If z is large, $\sigma(z) \approx 1/1 = 1$. If z is small or negative, then $\sigma(z) = 1/(1 + \infty) \approx 0$. We want to keep ω and b separate, which is different in some conventions.

1.3.3 Logistic Regression Cost Function

Recall output estimators $\hat{y} = \sigma(\omega^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$. Given training set $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$, we want the prediction $\hat{y}^{(i)} \approx y^{(i)}$. Define the Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$, a description of how "good" the estimator $\hat{y}$ is. This is a convex function that we can use later on to find the optimal point. Then $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$. If $y = 1$, then $\mathcal{L}(\hat{y}, y) = -\log \hat{y} + 0 = -\log \hat{y}$. That is, we want $\log \hat{y}$ to be large, which means $\hat{y}$ needs to be large. If $y = 0$, then $\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$. That is, we want $\log 1 - \hat{y}$ to be large, which means $\hat{y}$ to be small. There are many more options for the loss function, but we will justify this point later.

Define cost function (how well the model is doing on entire training set) to be

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

1.3.4 Gradient Descent

Recall the estimator $\hat{y} = \sigma(\omega^T x+b)$, and sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$. Also recall the Loss function: $J(\omega, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})$. The loss function estimates how well the estimator $\hat{y}^{(i)}$ matches or predicts $y^{(i)}$ by controlling parameters ω, b. The gradient descent algorithm is on $J(\omega, b)$, which is a convex bowl-shape curve. There could very well by multiple local "bottoms", i.e. multiple local optimum. Randomly initialize a point, the gradient descent will take small steps to hit a global optimum or close to global optimum.

Let us image a one dimensional plot with $\omega \in \mathcal{W}$ and the loss function $J(\omega)$, a convex function. The algorithm goes

Algorithm 1.3.1. Repeat {

$$\omega := \omega - \alpha\frac{dJ(\omega)}{d\omega}$$

while α is the learning rate that controls the size of the step of the gradient descent, the derivative term is the change of the steps we would use $d\omega$ to represent this term; that is, we define $\omega := \omega - \alpha d\omega$.

}

We could have initial point starting from the right side of the local minimum. We first calculate the derivative which is just the slope of the $J(\omega)$ at that point. However, if ω starts at the left side of the local minimum, the derivative term $dJ(\omega)/d\omega < 0$; that is, the subtraction of derivative would move the point ω to the positive side, the right side point at the local minimum.

From calculus, we can compute $\frac{dJ(\omega)}{d\omega}$ which represents the slope of the function and allows us to to know which direction to go. In logistic regression, the cost function $J(\omega, b)$ is shown and we want to update

$$\omega := \omega - \alpha\frac{dJ(\omega, b)}{d\omega}$$

$$b := b - \alpha\frac{dJ(\omega, b)}{db}$$

Sometimes we also write $\frac{\partial J(\omega,b)}{\partial\omega}$ as equal as the derivatives above as well, which notates "partial derivatives" in calculus. They are referring to the same notion. Lower case "d" and partial derivative symbol ∂ are usages of whether it is in calculus or not. In coding, we can simply notate "dw" versus "db".

1.3.5 Derivatives

This small section let us dive in to talk about a few points in calculus. We can consider a function $f(a) = 3a$. If $a = 2$, then we have $f(a) = 6$. If $a = 2.001$, then $f(a) = 6.003$, which is the change of f as a changes 0.001.Then we are going to say the slope (or derivative) of $f(a)$ at $a = 2$ is 3. To see why this is true, we take a look at another point $a = 5$ with the value of function $f(a) = 15$. If we move a 0.001 which means we set $a = 5.001$ then the value of the function would be $f(a) = 15.003$ which is the same increment changes from before when we move a from 2 to 2.001. We conclude

$$\frac{df(a)}{a} = 3 = \frac{d}{da}f(a)$$

for the derivative of the function $f(a)$.

Next, we consider $f(a) = a^2$. Starting from $a = 2$, we have the value of the function $f(a) = 4$. If we move a to $a = 2.001$, we have $f(a) \approx 4.004$. Then the slope (or derivative) of $f(a)$ at $a = 2$ is 4. That is,

$$\frac{d}{da}f(a) = 4$$

However, since $f(a)$ is higher power this time it may not have the same slope. Consider $a = 5$ with value $f(a) = 25$. We have value $f(a) \approx 25.010$ when $a = 5.001$. Then we have

$$\frac{d}{da}f(a) = 10 \text{ when } a = 5$$

which is larger than before.

Moreover, we have $f(a) = \log_e(a) = \ln(a)$, then $\frac{d}{da}f(a) = \frac{1}{a}$.

1.3.6 Computation Graph

Let us use a simple example as a start. Consider a function

$$J(a, b, c) = 3(a + bc)$$

while let us write $u = bc$, $v = a + u$, and $J = 3v$. If we let $a = 5$, $b = 3$, and $c = 2$, we will have $u = bc = 6$, $v = a + u = 11$ and finally we have $J = 3v = 33$. For the derivatives, we would have to go backward.

Let us say we want the derivative of J, i.e. $\frac{d}{dv} = ?$. Consider $J = 3v$ while $v = 11$. Then we can compute $\frac{d}{dv} = 3$. From same procedure, we can compute $\frac{dJ}{da} = 3$. Then $\frac{dv}{da} = 1$, then by chain rule, $\frac{dJ}{dv}\frac{dv}{da} = 3 \times 1$.

It is the same thing to implement Gradient Descent for Logistic Regression. Let us say we have the following setup

$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

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

Now we are trying to solve z to get a and thus to reduce loss $\mathcal{L}$. In coding, we would have

$$\text{da} = \frac{d\mathcal{L}(a, y)}{da} = -\frac{y}{a} + \frac{1 - y}{1 - a}$$

and also

$$\text{dz} = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a, y)}{dz} = 1 - y = \frac{d\mathcal{L}}{da} \cdot \underbrace{\frac{da}{dz}}_{a(1-a) \leftrightarrow -\frac{y}{a} + \frac{1-y}{1-a}}$$

Now let us consider m examples for

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

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

with one training example to be $(x^{(i)}, y^{(i)})$. Then the derivative w.r.t. w_1 would be

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

Algorithm 1.3.2. Let us initialize $J = 0$, $dw_1 = 0$, $dw_2 = 0$, $db = 0$, then
for $i = 1$ to m:

$$z^{(i)} w^T x^{(i)} + b, a^{(i)} = \sigma(z^{(i)})$$

$$Jt = -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}, dw_1 t = x_1^{(i)}, dw_2 t = x_2^{(i)} dz^{(i)}, dbt = dz^{(i)}$$

Then

$$J/ = m$$

$$dw_1 m; dw_2/ = m; db/ = m$$

Then

$$dw_1 = \frac{\partial J}{\partial w_1}$$

and implement

$$w_1 := w_1 - \alpha dw_1, w_2 := w_2 - \alpha dw_2, b := b - \alpha db$$

1.4 Vectorization

The art of getting rid of "for" loop is called vectorization. The reason is to have efficient algorithm with minimum time wasted for computation.

In logistic regression, we need $z = w^T x + b$ with $w \in \mathbb{R}^{n_x}$ and $x \in \mathbb{R}^{n_x}$. Hence, for non-vector: $z = 0$

for i in range $(n - x)$:

$$zt = w[i] \times x[i]$$

and finally $zt = b$ While vectorized computation takes the following steps

$$z = \underbrace{\text{np.dot}(w, x)}_{w^T x} + b$$

which is a faster code. Python Jupiter scripts is actually using CPU. Both GPU and CPU are SIMO single distinction, which is remarkably good at computing.

That is, whenever possible, try to avoid explicit for-loops. Define $\mu = Av$, $\mu_i = \sum_j A_{ij} v_j$, and $\mu = \text{np.zeros}((n, 1))$
for i... $\leftarrow$
for j... $\leftarrow$

$$\mu[i] += A[i][j] \times v[j]$$

and, however, $\mu = \text{np.dot}(A, v)$. Say you need to apply the exponential operation on every element of a matrix/vector.

$$\nu = \begin{bmatrix} \nu_1 \\ \vdots \\ \nu_n \end{bmatrix}, \mu = \begin{bmatrix} e^{\nu_1} \\ e^{\nu_2} \\ \vdots \\ e^{\nu_n} \end{bmatrix}$$

$$\mu = \text{np.zeros}((n, 1))$$

for i in range(n):

$$\mu[i] = \text{math.exp}(\nu[i])$$

Thus, the next question is how to implicitly use vectorization of entire training set without using for loop. To achieve this vectorization in logistic regression, recall the following set up:

$$z^{(1)} = w^T x^{(1)} + b, z^{(3)} = w^T x^{(2)} + b, z^{(3)} = w^T x^{(3)} + b$$

while

$$a^{(1)} = \sigma(z^{(1)}, a^{(2)} = \sigma(z^{(2)}), a^{(3)} = \sigma(z^{(3)})$$

$$\mathbf{X} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix} \text{ while } \mathbb{R}^{n_x \times m}$$

and we stack them together

$$[z^{(1)} \ z^{(2)} \dots z^{(m)}] = w^T X + \underbrace{[b \dots b]}_{\text{all } 1 \times m} = w^T x^{(1)} + \dots$$

and we use $z = np.dot(w, T, X) + b$ in python and this becomes $\sigma(z)$.

There is this technique called broadcasting in programming and let us illustrate this with the following example. Consider an example matrix from carbs, Proteins, Fats in 100g of different foods (see python example).

1.4.1 Explanation of Logistic Cost Function

Let us take the following space to explain a little about Logistic regression cost function. Consider $y = 1$, $p(y|x) = \hat{y}$, if $y = 0$, $p(y|x) = 1 - \hat{y}$ and these two statements give us $p(y|x)$. Then we have

$$p(y|x) = \underbrace{\hat{y}^y}_{\text{end up with } \hat{y}} \underbrace{(1 - \hat{y})^{(1-y)}}_{(1-\hat{y})^0}$$

then if $y = 1$, then we have $\hat{y}$, and if $y = 0$, we would have $1 - \hat{y}$. we can also take log on both sides

$$\log p(y|x) = \log \hat{y}^y (1 - \hat{y})^{(1-y)} = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

which is the lost function on a single function.What about m examples? We have the set up

$$p(\text{labels in training set}) = \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$$

and

$$\log p(\text{labels in training set}) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)})$$

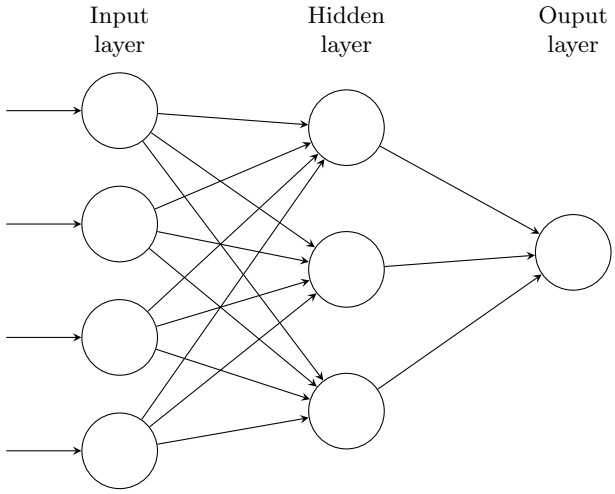
Then we compute

$$\begin{aligned} \log p(\text{labels in training set}) &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\ \Rightarrow \text{Cost: } J(w,b) &= \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \end{aligned}$$

By minimizing the cost function, we are carrying out maximization problem for this logistic regression function. This will give us a sense of why we should use the cost function in the algorithm.

1.5 Shallow Neural Network

We observe samples which become the first layer, the input layer. The second layer (the first layer of the neural network) would be the hidden layer. In the end there is an output layer that is observed in training set and is what we want to predict.



The input layer is denoted by $a^{[0]} = x$. The hidden layer is denoted by $a^{[1]} = [a_1^{[1]}, a_2^{[2]}, ..., a_4^{[4]}]$, that is,

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[2]} \\ \vdots \\ a_4^{[4]} \end{bmatrix}, \hat{y} = a$$

Let us say we observe x_i for $i \in \{1,2,3\}$. For one node in a hidden layer, we compute $z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}$ and then $a_1^{[1]} = \sigma(z_1^{[1]})$, which finishes up the first node of the first hidden layer. We repeat the same $z_2^{[1]} = w_2^{[1]T}x + b_2^{[1]}$ and $a_2^{[1]} = \sigma(z_2^{[2]})$. We can go ahead and write out

$$z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}, a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T}x + b_2^{[1]}, a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T}x + b_3^{[1]}, a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T}x + b_4^{[1]}, a_4^{[1]} = \sigma(z_4^{[1]})$$

We count, for one hidden layer, neural network to be a 2-layer neural-networks. We usually do not count the input layer. From this structure, we are essentially operating among matrices, that is,

$$\underbrace{\underbrace{\begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ \vdots \\ w_4^{[1T]} \end{bmatrix}_{\text{size=(4,3)}}}_{\text{four rows, each row three weights}}}_{\text{which we call } z^{[1]}\text{, first hidden layer}} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_4^{[4]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ \vdots \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix}$$

After we have the set up, we notice that the use of sigmoid function can be switched. Sigmoid function $a = \frac{1}{1+e^{-z}}$ can be other non-linear function $g(z^i)$. It turns out there could also be $a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ which is just a shifted sigmoid function. Let

$$g(z^{[i]}) = \tanh(z^{[i]})$$

works almost always better. In practice, programmers mostly likely would use tanh functions since it is strictly superior than sigmoid function. One could also use ReLU (recitified linear unit). This function has slope at 0.0000000000... near zero. To correct this, there is also leaky ReLU that could be used potentially.

1.5.1 Activation Function

Consider the following:

$$z^{[1]} = W^{[1]}x + b^{[1]}, a^{[1]} = \underbrace{g^{[1]}(z^{[1]})}_{\text{let us write } z^{[1]}}$$

$$z^{[2]} = W^{[2]}x + b^{[2]}, a^{[2]} = \underbrace{g^{[2]}(z^{[2]})}_{\text{let us write } z^{[2]}}$$

then we have

$$a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]}$$

and we plug $a^{[1]}$ in the following equation

$$a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

Then we have

$$\begin{aligned} a^{[2]} &= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} \\ &= (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]}) \end{aligned}$$

If you are using linear activation function than neural network would just output a linear function. This is only because if the model is trying to predict housing price which is considered continuous output. In this case, we should use ReLU or tanh and set the final neuron to be linear instead of non-linear.

1.5.2 Derivative of Activation Functions

Consider a sigmoid activation function $g(z) = \frac{1}{1+e^{-z}}$. Then the slope of the function is $\frac{d}{dz}g(z)$ from our knowledge in calculus and this is the slope of $g(z)$ at z. Then

$$\begin{aligned} \frac{d}{dz}g(z) &= \text{slope of } g(x) \text{ at } z \\ &= \frac{1}{1+e^{-z}}\big(1 - \frac{1}{1+e^{-z}}\big) \\ &= g(z)(1-g(z)) \\ &= a(1-a), \text{ then } g'(z) = a(1-a) \end{aligned}$$

Notice that $z = 10 \to g(z) \approx 1$, and $g'(z) \approx 1(1-1) \approx 0$. $z = -10 \to g(z) \approx 0$, and $g'(z) \approx 0(1-0) \approx 0$, $z = 0 \to g(z) = 1/2$ and $g'(z) = 1/2(1-1/2) = 1/4$.

Now let us look tanh function. Define $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Then $g'(z) = \frac{d}{dz}g(z) =$ slope of $g(z)$ at $a = 1 - (\tanh(z))^2$. That is, $z = 10, \tanh(z) \approx 1, g'(z) \approx 0$; $z = -10, \tanh(z) \approx -1, g'9z) \approx 0$; $z = 0, \tanh(z) = 0, g'(z) = 1$.

Finally, we can discuss a little about ReLU and Leaky ReLU as activation functions. For ReLU, $g(z) = \max(0, z)$, then

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{N/A} & \text{if } z = 0, \text{ this is undefined} \end{cases}$$

For advanced readers, $g'(z)$ is called a sub gradient of the activation function $g(z)$ which why gradient descent still works. One can simply consider that at this point $z \to 0$ and get infinitely closer so in practice the algorithm still runs.

For Leaky ReLU, denote $g(z) = \max(0.01z, z)$ and

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

technically z is not defined at 0 but one can still consider it as a number very close to 0.

1.5.3 Gradient Descent for Neural Networks

Consider a neural network with a single hidden layer. There are the following parameters $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ and so for $n_x = n^{[0]}, n^{[1]}, n^{[2]}1$. Then the matrix $w^{[1]}$ would be $(n^{[1]}, n^{[0]}), (n^{[1]}, 1), (n^{[1]}, n^{[1]}), (n^{[1]}, 1)$. Then the cost function would be

$$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\underbrace{\hat{y}}_{a^{[2]}}, y)$$

Algorithm 1.5.1. For gradient descent, we repeat
Compute predicts $(\hat{y}^{(i)}, i-1, ..., m)$
$dw^{[1]} = \frac{\partial J}{\partial w^i}, db^{[1]} = \frac{\partial J}{\partial b^{[1]}}; ...$
$w^{[1]} := w^{[1]} - \alpha dw^{[1]}$
$b^{[1]} := b^{[1]} - \alpha db^{[1]}$
...

Algorithm 1.5.2. For forward propagation,

$$z^{[1]} = w^{[1]}x + b^{[1]},$$

$$A^{[1]} = g^{[1]}(z^{[1]}),$$

$$z^{[2]} = w^{[2]}A^{[1]} + b^{[2]},$$

$$A^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]}).$$

For back propagation,

$$dz^{[2]} = A^{[2]} - Y \text{ for } Y = [y^{(1)}L, y^{(2)}, ..., y^{(m)}],$$

and

$$dw^{[2]} = \frac{1}{[m]} dz^{[2]} A^{[1]T},$$

$$db^{[2]} = \frac{1}{m}\text{np.sum}(dz^{[2]}, \underbrace{\text{axis} = 1, \text{ keepdims} = \text{True}}_{\text{to keep dimensions fixed } (n^{[2]},1)})$$

then

$$dz^{[1]} = \underbrace{w^{[2]T}dz^{[2]}}_{(n^{[1]},m)} \underbrace{*}_{\text{element-wise product}} \underbrace{g^{[1]'}(z^{[1]})}_{(n^{[1]},m)}$$

$$dw^{[1]} = \frac{1}{m} dz^{[1]x^T}$$

$$db^{[1]} = \frac{1}{m}\text{np.sum}(dz^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

For forward pass, given $\{x, w, b\}$, we can feed them into $z = w^T x + b$, then we feed z into $a = \sigma(z)$ which finally is used to compute $\mathcal{L}(a, y)$. For backward pass, we compute da and then we compute dz to finally obtain $\{x, w, b\}$. Then the loss

$$\mathcal{L}(a, y) = -y \log a - (1 - y) \log(1 - a)$$

then

$$\frac{d}{da}\mathcal{L}(a, y) = da = -\frac{y}{a} + \frac{1 - y}{1 - a}$$

and

$$dz = a - y = da \cdot g'(z) = g(z) = \sigma(z)$$

Notice that this is $\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{da}{dz} = \frac{d}{dz} g(z) = g'(z)$.

Now we can take a look at two-layer neural network which is just single layer neural network added one more hidden layer before output.

$$\left[\begin{array}{c} W^{[2]} \\ b^{[2]} \\ \left[\begin{array}{c} x \\ W^{[1]} \\ b^{[1]} \end{array}\right] \rightarrow z^{[1]} = W^{[1]}x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]}) \end{array}\right] \rightarrow z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$\rightarrow a^{[2]} = \sigma(z^{[2]}) \rightarrow \mathcal{L}(a^{[2]}, y)$$

Then backward propagation would start from the last hidden layer, which in this case is the second layer, to compute $da^{[2]}$, $dz^{[2]}$, $dW^{[2]}$, $db^{[2]}$, $da^{[1]}$, $dz^{[1]}$, and finally $\{x, W^{[1]}, b^{[1]}\}$.

One note deserves attention is the initialization of random weights. Suppose we set up

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \text{ then } a_1^{[1]} = a_2^{[1]}$$

From this, when computing backward propagation, then $dz_1^{[1]} = dz_2^{[1]}$, which means then the output weights are identical. Then the hidden units are completely symmetric. By proof by induction, units are computed the same since

$$\begin{bmatrix} u & v \\ u & v \end{bmatrix}, W^{[1]} = W^{[1]} - \alpha dW$$

then there is no point computing more than one hidden units (hidden neurons). A better way is to randomly initialize the parameters:

$$W^{[1]} = \text{np.random.randn}((2,2)) * \underbrace{\underbrace{0.01}_{\text{choose something small}}}_{\text{to avoid fat part of sigmoid}}$$

$$b^{[1]} = \text{np.zero}((2,1))$$

and $W^{[2]} = \text{np.random.randn}((1,2)) * 0.01$ and so on.

1.6 Deep Neural Network

Let us take all the ideas about parameter initialization, vectorization, propagation, etc. and put all of them together. We have seen logistic regression and neural network with 1 hidden layer. As a contrast to shallow neural network which is just a simple neural network, deep neural network usually consists of more than three hidden layers. Consider $z^{[1]} = W^{[1]}x + b^{[1]}$ and $a^{[1]} = g^{[1]}(z^{[1]})$ for the first layer. Then $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ and $a^{[2]} = g^{[2]}(z^{[2]})$ for the second layer. This can keep going to say the fourth layer, which consists of $z^{[4]} = W^{[4]}g^{[3]} + b^{[4]}$ and $a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$. Note that x in $z^{[1]}$ is also the first layer (observations) which is $a^{[0]}$. Thus,

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

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

That is, vertically, we have

$$z^{[1]} = W^{[1]}X + b^{[1]} \text{ while } X = A^{[0]}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

with all the training examples the column vectors. Then, finally, we have predictions

$$\hat{Y} = g(Z^{[4]}) = A^{[4]}$$

which we can convert to a binary form. In this case, a "for loop" is required in the algorithm and it is okay to have one loop to do that.

Consider the following example, there is a neural network with five hidden layers and we have the following steps to compute forward propagation

$$z^{[1]} = W^{[1]}X + b$$

which gives us the first layer with, for example, say three hidden units. For the five hidden layers, we have $n^{[1]} = 3$, $n^{[2]} = 5$, $n^{[3]} = 4$, $n^{[4]} = 2$, and $n^{[5]} = 1$. Let us say there are two explanatory variables so we have $n^{[0]} = 2$. Then we have $z^{[1]}$ has shape $(n^{[1]}, 1) = (3, 1)$. We also have $W^{[1]}$ with shape $(n^{[1]}, 2) = (3, 2)$ while x with shape $(n^{[0]}, 1) = (2, 1)$. That is, for $W^{[1]}$ has shape $(n^{[1]}, n^{[0]})$. In general, we have dimensions of $W^{[l]}$ to be $(n^{[l]}, n^{[l-1]})$. Then we can compute $z^{[2]} = w^{[2]} \cdot a^{[1]}$.

One reason that deep neural networks work well is the following. There are functions you can compute with a "small" l-layer deep neural network that shallower networks require exponentially more hidden units to compute.

Now we can take a look at forward and backward functions. For large l, we have $W^{[l]}$ and $b^{[l]}$. Then for forward propagation, input $a^{[l-1]}$ and we want output $a^{[l]}$. In this case, we have

$$z^{[l]} := W^{[l]}a^{[l-1]} + b^{[l]}, \text{ cache } z^{[l]}$$
$$a^{[l]} := g^{[l]}(z^{[l]})$$

For backward propagation, we want to input $\text{da}^{[l]}$, which we cache it, and output $\text{da}^{[l-1]}$. That is, the basic deep neural network will be as follows:

$$\text{Forward propagation: } a^{[0]} \to \underbrace{W^{[1]}, b^{[1]}}_{\substack{\text{cache } z^{[1]} \\ \downarrow}} \overset{a^{[1]}}{\to} \underbrace{W^{[2]}, b^{[2]}}_{z^{[2]}} \overset{a^{[2]}}{\to} \cdots \to \underbrace{W^{[l]}, b^{[l]}}_{z^{[l]}} \overset{a^{[l]}}{\to} \hat{y}$$

$$\text{Backward propagation: } \underbrace{a^{[0]}}_{\text{no good}} \leftarrow \overbrace{\underbrace{W^{[1]}, b^{[1]}}_{\text{dW}^{[1]},\text{db}^{[1]}}}^{\text{cache } z^{[1]}} \overset{a^{[1]}}{\leftarrow} \overbrace{\underbrace{W^{[2]}, b^{[2]}}_{\text{dW}^{[2]},\text{db}^{[2]}}}^{z^{[2]}} \overset{a^{[2]}}{\leftarrow} \cdots \leftarrow \overbrace{\underbrace{W^{[l]}, b^{[l]}}_{\text{dW}^{[l]},\text{db}^{[l]}}}^{z^{[l]}} \overset{a^{[l]}}{\leftarrow} \text{da}^{[l]}$$

Now we can discuss a few things about parameters and hyperparameters. Parameters in the model are $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ...$ However, there are hyperparameters as well, leaning rate α, number of iterations n, number of hidden layers l, number of hidden units $n^{[1]}, n^{[2]}, ...$, choice of activation function. Later on we will discuss other hyperparameters such as momentum, mini-batch size, regularizations, and so on. In some sense, the application of deep learning is a very empirical process. We test our ideas with parameters, we code the idea and conduct experiments. Perhaps we observe some cost function via number of iterations that we like or we do not like. After certain number of trials, we select the final parameters to try out hyperparameters. This might be an unsatisfying part of deep learning, but it is also necessary for a lot of programmers especially good programmers to experience this progress.

Any relations between deep learning and human brain? Not a whole lot. The only analogy is that the construction of artificial nodes in the layers of the neural network can be (or look like on paper) the actual neurons in human brain. Besides that there is no real reason of where the name comes from.

2 DEEP NEURAL NETWORKS: HYPER-PARAMETER TUNING, REGULARIZATION AND OPTIMIZATION

Go back to Table of Contents. Please click TOC

This is the practical aspects of deep learning. We will start from the practical aspects first such as hyperparameters, random problems, and so on. When training neural network, one needs to make a lot of decisions such as 1) number of layers, 2) number of hidden units, 3) learning rates, 4) activation functions, and so on. One would always start with an idea (or a particular data set). Then we code the idea and try some quick results. Based on the outcome, we need to change our ideas to code and such cycle goes on. Structured data ranges from advertisers to any search bar results from website online.

A conventional method is to have training set, development set, as well as a testing set. We use training set to train our machines and we test different parameters on development set. In the very end, once we are happy with all the parameters we would finally touch test set for final testing accuracy.

2.1 Bias-Variance Trade-off

An important essence is to understand bias and variance in machine learning. This is the famous bias-variance trade-off. Let us consider three examples in two-dimensional plot. The first one is under-fitting, imaging a straight line trying to classify a bunch of dots. The second one is "just right", imaging a curve cutting through a bunch of dots. The third one is over-fitting, imaging a high-dimensional line wiggling through the dots to try to find the path to fully classify the dots. To understand this, we look at two key numbers which are training set error and development set error. Suppose we have 1% for training and 11% for development set. This might be high variance. Now suppose we have different results 15% for training and 16% for development set error. This does not look like it will be good models for predictions since human error is approximately 0% at telling cats and dogs. Then we say this is high bias. Now suppose we have 15% training set error and we have 30% development set error. This model is probably worse off than previous two and we say this is high bias and high variance. Now in end suppose we have 0.5% training set error and 1% development set error. We say this model has ow bias and low variance.

To illustrate this point strongly, we can look at the math behind this idea. We consider the decomposition for squared error proceeds as follows. For notational convenience, abbreviate $f = f(x)$ and $\hat{f} = \hat{f}(x)$. First, recall that, by definition, for any random variable X, we have

$$\text{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

and by rearranging equation, we get

$$\mathbb{E}[X^2] = \text{Var}[X] + \mathbb{E}[X]^2$$

Since f is deterministic, then $\mathbb{E}[f] = f$. This, given $y = f + \epsilon$ and $\mathbb{E}[\epsilon] = 0$, implies $\mathbb{E}[y] = \mathbb{E}[f + \epsilon] = \mathbb{E}[f] = f$. Also, since $\text{Var}[\epsilon] = \sigma^2$, then

$$\text{Var}[y] = \mathbb{E}[(y - \mathbb{E}[y])^2] = \mathbb{E}[(y - f)^2] = \mathbb{E}(f + \epsilon - f)^2] = \mathbb{E}[\epsilon^2] = \text{Var}[\epsilon] + \mathbb{E}[\epsilon]^2 = \sigma^2$$

Thus, since ϵ and $\hat{f}$ are independent, we can write

$$\begin{aligned}
\mathbb{E}[(y-\hat{f})^2] &= \mathbb{E}[y^2+\hat{f}^2-2y\hat{f}] \\
&= \mathbb{E}[y^2]+\mathbb{E}[\hat{y}]^2-\mathbb{E}[2y\hat{f}] \\
&= \text{Var}[y]+\mathbb{E}[y]^2+\text{Var}[\hat{f}]+\mathbb{E}[\hat{f}]^2-2f\mathbb{E}[\hat{f}] \\
&= \text{Var}[y]+\text{Var}[\hat{f}]+(f^2-2f\mathbb{E}[\hat{f}]+\mathbb{E}[\hat{f}]^2) \\
&= \text{Var}[y]+\text{Var}[\hat{f}]+(f-\mathbb{E}[\hat{f}])^2 \\
&= \sigma^2+\text{Var}[\hat{f}]+\text{Bias}[\hat{f}]^2
\end{aligned}$$

2.2 Regularization

In practice, we first ask does the algorithm, from training data performance, have high bias? Then to test this, we can try bigger network, longer network, and so on. Next, we ask ourselves does the model, from development set performance, have high variance? We try different sets of parameters such as more data or regularization.

What if we cannot get more data, we need to do a regularization. Consider cost function

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}||w||_2^2$$

while

$$||w||_2^2=\sum_{j=1}^{n_x}w_j^2=w^Tw \text{ is } L_2 \text{ regularization}$$

or we could have

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}||w||_2^2+\underbrace{\frac{\lambda}{2m}b^2}_{\text{usually omit}}$$

or instead we add

$$\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w|=\frac{\lambda}{2m}||w||_1, \text{ which is } L_1 \text{ regularization}$$

Note that λ is the regularization parameter. One can characterize the value and see which one does the best.

In neural network, we consider

$$J(w^{[1]},b^{[1]},...,w^{[l]},b^{[l]})=\frac{1}{m}\sum_{i=1}^{n}\mathcal{L}(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}\sum_{l=1}^{L}||w^{[l]}||^2$$

while

$$||w^{[l]}||^2=\sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2 \text{ while } w:(n^{[l]},n^{[l-1]})$$

In propagation, as shown in previous section, we have $dw^{[l]}$ and $w^{[l]}:=w^{[l]}-\alpha dw^{[l]}$. Now since have the λ term, we would have

$$\text{dw}^{[l]}=(\text{from backprop})+\frac{\lambda}{m}$$

$$w^{[l]}:=w^{[l]}-\alpha\text{dw}^{[l]}$$

then instead we have

$$w^{[l]}:=w^{[l]}-\alpha[(\text{from backprop})+\frac{\lambda}{m}w^{[l]}]=w^{[l]}-\frac{\alpha\lambda}{m}w^{[l]}-\alpha(\text{from backprop})$$

which is why L_2 norm propagation is called "weight decay" because of this $(1 - \frac{\alpha\lambda}{m} w^{[l]}$ term.

Why does regularization help reducing over-fitting? Let us take a look at the following. Recall the three examples we discussed above with high bias, "just right", and high variance. Recall

$$J(w^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(h^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||w^{[l]}||_F^2$$

One piece of intuition is if we set λ to be sufficiently large and set $w^{[l]} \approx 0$ then we may be able to single out the impact of a lot of the hidden units. In other words, for one path in the neural network would have very small effect, and in fact it is almost like a logistic regression unit.

2.2.1 Softmax Function

Another point is to consider tanh function, $g(z) = \tanh(z)$. Then the intuition is if one use larger regularization terms, the penalized $w^{[l]}$ would be relatively small, and hence $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ would approximately to linear. For implementation, we take our definition of the cost function J and we actually modify by adding an extra term. Then one of the steps is actually to plot the cost function and we expect to see it decrease monotonically.

In addition to L_2 regularization, another very powerful regularization techniques is called "dropout." Let us see how that works. Say you train a neural network like the one on the left and there is over-fitting. Here is what you do with dropout. Consider a copy of the neural network. With dropout, what we are going to do is go through each of the layers of the network and set some probability of eliminating a node in neural network. Say that for each of these layers, we are going to, for each node, toss a coin and have a 0.5 chance of keeping each node and 0.5 chance of removing each node. So, after the coin tosses, maybe we will decide to eliminate those nodes, then what you do is actually remove all the outgoing things from that no as well. So you end up with a much smaller, really much diminished network. And then you do back propagation training. There is one example on this much diminished network. And then on different examples, you would toss a set of coins again and keep a different set of nodes and then dropout or eliminate different than nodes. And so for each training example, you would train it using one of these neural based networks. So, maybe it seems like a slightly crazy technique. They just go around coding those are random, but this actually works.

Drop out does this seemingly crazy thing of randomly knocking out units on your network. Why does it work so well with a regularizer? Previously, we discussed this intuition that drop-out randomly knocks out units in your network. It is as if on every iteration you are working with a smaller neural network, and so using a smaller neural network seems like it should have a regularizing effect. Here is a second intuition which is, let us look at it from the perspective of a single unit.

Let us say the first one in a layer, i.e. some hidden neuron. Now, for this unit to do his job as for inputs and it needs to generate some meaningful output. Now with drop out, the inputs can get randomly eliminated. Sometimes those two units will get eliminated, sometimes a different unit will get eliminated. So, what this means is that this unit, which is pointed out, it cannot rely on any one feature because any one feature could go away at random or any one of its own inputs could go away at random. Some particular would be reluctant to put all of its bets on, say, just this input, right? The weights, we are reluctant to put too much weight on any one input because it can go away. This unit will be more motivated to spread out this way and

give you a little bit of weight to each of the four inputs to this unit. By spreading all the weights, this will tend to have an effect of shrinking the squared norm of the weights. Hence, similar to what we saw with L_2 regularization, the effect of implementing drop out is that it shrinks the weights and does some of those outer regularization that helps prevent over-fitting. But it turns out that drop out can formally be shown to be an adaptive form without a regularization. But L_2 penalty on different weights are different, depending on the size of the activations being multiplied that way.

To summarize, it is possible to show that drop out has a similar effect to L_2 regularization. Only to L_2 regularization applied to different ways can be a little bit different and even more adaptive to the scale of different inputs.

In addition to L_2 regularization and drop out regularization there are few other techniques to reducing over-fitting in the neural network. Consider a cat classifier. If you are over fitting getting more training data can help, but getting more training data can be expensive. What one can do is augment your training set by taking image and flipping it horizontally and adding that also with your training set. By flipping the images horizontally, you could double the size of your training set. Because the training set is now a bit redundant this is not as good as if you had collected an additional set of brand new independent examples. Other than that, one can also take a random crop of the image as a new observation.

2.3 Set Up Optimization Problem

When training a neural network, one of the techniques that will speed up your training is if you normalize your inputs. Consider a training set with two input features. The input features X are two dimensional. Normalizing your inputs corresponds to two steps. The first is to subtract out of to zero out the mean. So you set $\mu = 1$ over m sum over i of X_i. This is a vector and X gets set as $X - \mu$ for every training example. This means that we move the training set until it has 0 mean. That is,

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{[i]}$$

$$X := x - \mu$$

Then the second step is to normalize the variances. That is,

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

After this work, we will have moved any scattered plots on the Euclidean graph to center 0.

Why normalize? Recall $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$. If unnormalized, we could have a cost function that is like a bowl-shape surface but pointing at any directions, not necessarily pointing up. However, if normalized, we have a relatively more spherical surface pointing straight up. This transformation will allow us to have a more efficient computing process.

One of the problems of training neural network, especially very deep neural networks, is data vanishing and exploding gradients. That means when one trains a very deep network, the derivations or slopes can sometimes get either very big or very small. This makes the training of neural network difficult. Consider parameters $w^{[1]}, ..., w^{[l]}$, and activation function $g(z) = z$ with bias $b^{[l]} = 0$. We can show

$$y = w^{[l]} w^{[l-1]} \dots w^{[2]} \underbrace{w^{[1]} x}_{z^{[1]} = w^{[1]} x}$$

while $a^{[1]}g(z^{[1]}) = z^{[1]}$ and next $a^{[2]} = g(z^{[2]}) = g(w^{[2]}a^{[2]})$.

We discussed potential problems such as vanishing and exploding gradients. A partial solution to this can be better or more careful choice of the random initialization for your neural network. Let us consider an example of initializing the ways for a single neuron. Consider i from 1 to 4, then we have X_1, X_2, X_3, X_4 and a single neuron $a = g(z)$ while $z = w^{[1]}X_1 + \cdots + w^{[n]}x_n + b$. For large n, we would have small w_i, then $\text{var}(w_i) = \frac{1}{n}$. An approach is to set np.random.rand(shape) $\star$ np.sqrt() in python to set standard normal weights to create similar variance. We can also use Xavier initialization

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

or

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

When you implement back propagation you will find that there is a test called gradient checking that can really help the implementation of back-propagation to be correct. Consider function $f(\theta)$ and say it is defined as $f(\theta) = \theta^3$. For θ, we can take an interval $(-\epsilon, \epsilon)$ for $\epsilon > 0$. It turns out the height of $\theta + \epsilon$ and 2ϵ will give us better gradient. For $\theta + \epsilon$ we have $f(\theta + \epsilon)$. Another point is $\theta - \epsilon$ and $f(\theta - \epsilon$. Then

$$\begin{aligned} (1/2)(f(\theta+\epsilon) - f(\theta-\epsilon) &\approx g(\theta) \\ \text{Or } (1/2(0.01))((1.01)^3 - (0.99)^3) &= 3.0001 \end{aligned}$$

and this can be checked that approximate error will be 0.0001, which gives us a better idea of how to check the gradient and slope of the function. For mathematical expression, we generally write

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon} \text{ while } \epsilon \text{ is from notation } O(\epsilon^2)$$

To do so, take $W^{[1]}, b^{[1]}, ..., W^{[l]}, b^{[l]}$ and reshape into a big vector θ. Then take $eW^{[1]}, db^{[1]}, ...$ and reshape into a big vector $d\theta$. The question would be is this the gradient of cost function J? Recall $J(\theta) = J(\theta_l)$. To implement grad.check, we apply

1. for each i:
Apply $d\theta[i] = \frac{J(\theta_1,...,\theta_i+\epsilon,...) - J(\theta_1,...,\theta_i-\epsilon,...)}{2\epsilon} \approx d\theta[i] = \partial J/\partial\theta_i$
And output $d\theta$, and check $d\theta_{\approx} \approx d\theta$?
2. Check

$$\frac{||d\theta_{\approx} - d\theta||_2}{||d\theta_{\approx}||_2 + ||d\theta||_2}$$

and typically we use $\epsilon = 10^{-7}$, which should be the ratio from above equation. If it is a lot bigger than that, say bigger than 10^{-5}, then perhaps we need to do something about it. This way we can create a signal for us to go back to check our algorithm and debug if necessary.

One would notice the following, to implement gradient checking, do not use in training (only to debug). If algorithm fails gradient checking, look at components to try identify buy. One would always remember regularization terms. One would also realize that gradient checking does not work with dropout. One could always run at random initialization and train a network for a while; and later again run gradient check.

2.4 Optimization Algorithms

We need to optimize our algorithms to let the code run efficiently. This is where we can discuss batch vs. mini-batch gradient descent. Vectorization allows you to efficient

compute on m examples

$$X = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$$
$$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$$

However, what about $m = 5,000,000$. Let us say you divide your training set into baby training sets with each of them only 1,000 observations. Then we take the next 1,000 observations as a second mini-batch, and so on. That is, we have

$$X = [\underbrace{x^{(1)}, x^{(2)}, ..., x^{(1000)}}_{x^{[1]}} \dots, ..., \underbrace{..., x^{(m)}}_{x^{(5000)}}]$$

$$Y = [\underbrace{y^{(1)}, y^{(2)}, ..., y^{(1000)}}_{y^{[1]}} \dots, ..., \underbrace{..., y^{(m)}}_{y^{(5000)}}]$$

In algorithm running, we do
for $t = 1, ..., 5000$, 1 step of grad descent using $x^{(t)}, y^{(t)}$
Forward propagation on $x^{(t)}$, and

$$z^{[1]} = w^{[1]}x^{[t]} + b^{[l]}$$

$$A^{[1]} = g^{(1)}(Z^{[1]})$$

...

$$A^{[l]} = g^{[l]}(z^{[l]})$$

which will process 1,000 examples at the same time
Compute cost

$$J = \frac{1}{1000}\sum_{i=1}^{l}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2\times 1000}\sum_{l}; ||W^{[l]}||_i^2$$

Backward propagation to compute gradients cost

$$J^{(t)} \text{ using } x^{(t)}, y^{(t)}$$

Update $W^{[l]} := W^{[l]} - \alpha dW^{[l]}, b^{[l]} := b^{[l]} - \alpha db^{[l]}$ while "one epoch" means pass through training set with 5,000 gradient descent steps. However, in batch gradient descent, a single pass through the training allows you to take only one gradient descent step. For mini-batch gradient descent, every iteration the cost function may not necessarily decrease. The overall trend will going down but it will be more noisy than batch gradient descent. For size of mini-batch to be m, this will be batch gradient descent. For size of batch to be 1, this will be stochastic gradient descent, it takes a long time and will never converge. For stochastic gradient descent, you lose the speed-up from vectorization and the process is time consuming. In practice, we choose something in between. That is, we conduct mini-batch gradient descent greater than size 1 and less than size m. In other words, if size of training set is less than 2000, it is fine using batch gradient descent. Otherwise, it is recommended to use mini-batch size. Usually the size of mini-batch can be 64, 128, 256, 512, and etc. It is rare to see size of mini-batch 1024, yet it is still possible to occur. One needs to make sure mini-batch, $x^{(t)}, y^{(t)}$ fit in CPU/GP memory. If not, the performance may be worse off.

Now we can discuss exponentially weighted averages, which takes the following form

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

then in other words, we have, for example, let $t = 100$, then

$$\begin{aligned} v_{100} &= 0.1\theta_{100} + 0.9v_{99} \\ &= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98}) \\ &= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9v_{97})) \end{aligned}$$

where v_{97} can then be written with more summation terms and this is something that exponentially decay in the end. In general, we observe that $0.9^{10} \approx e^{-1}$. Moreover, take $\epsilon = 0.9$, then $(1-\epsilon)^{1/\epsilon} \approx 1/e$, and $0.98^{10} \approx 1/e$.

To implement exponentially weighted averages, consider

$$\begin{aligned} v_0 &= 0 \\ v_1 &= \beta v_0 + (1-\beta)\theta_1 \\ v_2 &= \beta v_1 + (1-\beta)\theta_2 \\ v_3 &= \beta v_3 + (1-\beta)\theta_3 \\ &\dots \end{aligned}$$

With this idea in mind, we can write

$$\begin{aligned} V_\theta &:= 0 \\ V_\theta &:= \beta v_0 + (1-\beta)\theta_1 \\ V_\theta &:= \beta v_1 + (1-\beta)\theta_2 \\ V_\theta &:= \beta v_3 + (1-\beta)\theta_3 \\ &\dots \end{aligned}$$

This, however, brings up concerns for long run. That is, for large t, we may get very different moving averages. To correct this problem, there is step called bias correction. That is, we take

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

and we can modify this formula by

$$v_t = \frac{\beta}{(1-\beta^t)} v_{t-1} + \frac{(1-\beta)}{(1-\beta^t)}\theta_t$$

which becomes the weighted average of exponentially averages.

Using this notion as building blocks, we can establish momentum in gradient descent. This is important because in some occasions the gradient descent, while taking baby steps towards the local minimum, can oscillate quite a lot to a point where it can actually diverge. We do not want this to happen. We implement the following algorithm with the notion of exponentially weighted averages.

Algorithm 2.4.1. On iteration t: (assume initialization $v_{dW} = 0$, $v_{db} = 0$)
Compute dW, db, on the current mini-batch
$v_{dW} = \beta v_{dW} + (1-\beta)dW$
$v_{db} = \beta v_{db} + (1-\beta)db$
$W = W - av_{dW}, b = b - \alpha v_{db}$
Hyper-parameters: α, β, and choose $\beta = 0.9$.

There is another algorithm called Root Mean Square Propagation (RMS-Prop) that can also speed up gradient descent. In other words, on iteration t, the algorithm computes dW, db on count mini-batch. Then $s_{dW} = \beta s_{dW} + (1-\beta)dW^2$. Notice that W^2 is an element-wise multiplication. Next, $s_{db} = \beta s_{db} + (1-\beta)db^2$. Then we update $w := w - \alpha\frac{dw}{\sqrt{s_{dw}}}$, and $b := b - \alpha\frac{db}{\sqrt{s_{db}}}$. That is, we want to slow down the oscillations but the b direction we are going fast. This way we will be able to make the algorithm more efficient. In most cases, we actually combine RMS-Prop with momentum. To prevent the fraction blow up, we sometimes also add an ϵ to ensure that the bottom of the fraction does not approximate to zero. The implementation of RMS-Prop and momentum will become Adam optimization algorithm.

Algorithm 2.4.2. Set $V_{dW} = 0$, $S_{dW} = 0$, $V_{db} = 0$, $S_{db} = 0$:
On each iteration t:
Compute dW, db using mini-batch
$v_{dW} = \beta_1 V_{dW} + (1-\beta_1)dW, V_{db} = \beta_1 V_{db} + (1-\beta_1)db \leftarrow$ "momentum"β_1
$S_{dW} = \beta_2 V_{dW} + (1-\beta_1)dW^2, S_{db} = \beta_2 V_{db} + (1-\beta_2)db \leftarrow$ "RMS-prop"β_2
$V_{dW}^{\text{Corrected}} = V_{dW}/(1-\beta_1^t), V_{db}^{\text{Corrected}} = V_{db}/(1-\beta_1^t)$
$S_{dW}^{\text{corrected}} = S_{dW}/(1-\beta_2^t), S_{db}^{\text{Corrected}} = S_{db}/(1-\beta_2^t)$
while update $W := W - \alpha \frac{v_{dW}^{\text{Corrected}}}{\sqrt{S_{db}^{\text{Corrrected}}}+\epsilon}, b := b - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{Corrected}}}+\epsilon}$

In this sense, the hyper-parameters choices are: α needs to be tuned, $\beta_1 := 0.9$ for dW, $\beta_2 := 0.999$ for dW^2, and $\epsilon := 10^{-8}$. The term "Adam" is short for "adaption moment estimation".

Consider a mini-batch gradient descent, let us say as we iterate the path oscillates around the local minimum and is going around it. One way is that we can slowly decrease α, which is the learning rate, to make the path smaller and smaller to be closer to the local minimum. To do that, we conduct the following steps. Recall one epoch is one pass through dart. Set up

$$\alpha = \frac{1}{1 + \text{decay rate} \times \text{epoch}}$$

and this is to say that we can get a path for epoch. For epoch to be one, set $\alpha = 0.1$. For epoch to be two, set $\alpha = 0.67$, For epoch to be three, set $\alpha = 0.5$ and so on. This way, we have a descending α as epoch increases. Other than this formula of decay, we can also consider exponential decay. Set $\alpha = 0.95^{\text{epoch-run}}\alpha_0$ which is exponentially decay. Or we set $\alpha = \frac{k}{\sqrt{\text{epoch-run}}\alpha_0}$ or $\frac{k}{\sqrt{t}}\alpha_0$, which looks like designated staircase. Or we can simply do manual decay, which is the slowest way but occasionally works.

2.5 Hyperparameter Tuning

We have now seen by now that changing neural nets can involve setting a lot of different hyperparameters. How do we go about finding a good setting for these hyperparameters? One painful thing about training deepneess is the sheer number of hyperparameters in the model, ranging from the learning rate α which is the momentum term, or the hyperparameters for the Adam optimization algorithm which are β_1, β_2, and ϵ. Maybe we have to to pick the number of layers, the number of hidden units, or the size of mini-batch array. In most situations, α, the learning rate is the most important hyperparameter to tune. Other than α, a few other hyperparameters would be the next. For example, it could be the momentum term, say 0.9 is a good default value. One could also tune the mini-batch size to make sure that the algorithm is running efficiently. Sometimes one could also play around with the hidden units.

Traditionally, people have been working with a grid, such as a lattice with two hyperparameters. However, this will be computationally costly for large values of hyperparameters. A better approach can be using random sampling to form a random plot in the grid instead of using lattice. It turned out sampling at random does not imply sampling uniformly at random over the range of values. Instead it is important to pick the appropriate scale on which to explore the hyperparameters. Suppose one desires to choose a value for hidden units ranging in $n^{[l]} = 50, ..., 100$. Suppose the number of layers L would take values from 2 to 4. The number of layers can be uniformly set randomly, but the number of hidden units one should just pick a few ranges and start the trial. Consider one desires to search for learning rate α, ranging from 0.0001 to 1. Also consider sampling uniformly at random. This will be computationally costly.

Instead, we can use a log scale to search 0.0001, 0.001, 0.01, 0.1, and 1. Now we have more resources dedicated to search for these values. In python, we run

$$r = -4 * \text{np.random.rand}() \leftarrow r \in [-4, 0]$$

$$\alpha = 10^r \leftarrow 10^{-4}, \dots, 10^0$$

and once we find two values that we desire we can then sample uniformly between these two values.

Facing all of these hyperparameters, what are some tuning process in practice? One can always start with just babysitting one model. One train and observe the path of error rate. As path moves on, one learns about the effect parameters have on the training error. With human supervising, this is very costly yet can be done more carefully. However, one can always train many models in parallel. Same time, one can train multiple models and observe multiple error rate. The human supervising process is called "panda" approach. The approach one trains multiple models and the name is "caviar" approach. The names come from the difference where pandas nurture one kid with great care while fish lay lots of eggs and hope for at least one would work.

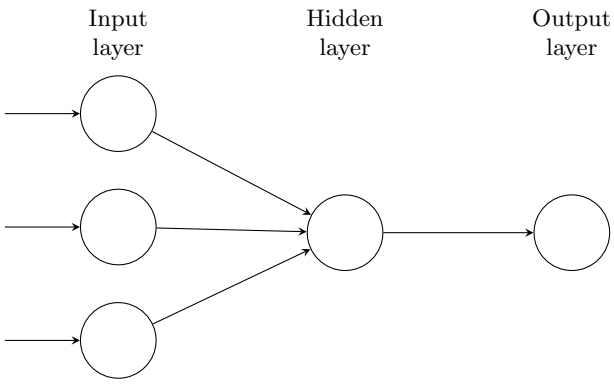
2.6 Batch Normalization

In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization, created by two researches, Sergey Loffe and Christian Szegedy. Batch normalization makes your hyperparameter search problem much easier, makes your neural network much more robust. The choice of hyperparameters is a much bigger range of hyperparameters that work well, and will also enable you to much more easily train even every deep networks.

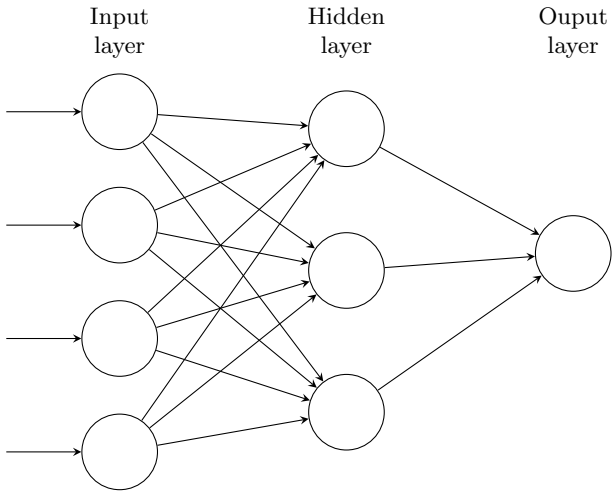
Consider training a logistic regression as the following. We can really normalize the data set before we feed into the model. Take

$$\mu = \frac{1}{m} \sum_i x^{(i)}, X = X - \mu$$

$$\sigma^2 = \frac{1}{m} \sum_i x^{(i)}, X = X / \sigma^2$$



However, what if we have a more complex model as the following. Opposed as in logistic shown above, this example, the question is can we normalize $a^{[2]}$ so we can train $w^{[3]}, b^{[3]}$ faster. Technically, we will normalize $z^{[2]}$. There are debates before activation or after. In practice, normalize $z^{[2]}$ is done very often.



Given some intermediate values in neural net, $z^{(1)}, ..., z^{(m)}$ while we notate them $z^{[l](i)}$. One can compute mean

$$\mu = \frac{1}{n}\sum_i z^{(i)} \text{ and } \sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)-\mu}}{\sqrt{\sigma^2 + \epsilon}}, \text{ and } \tilde{z}^{(i)} = \gamma z^{(i)}_{\text{nrom}} + \beta$$

while γ and β are learnable parameters of the model. Notice that the effect of parameters allow us to set mean. In fact, $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$ will exactly invert the z-norm equation. In other words, these equations are essentially just computing the identify function. This way we can assure that the hidden units are all standardized.

How does this fit in training of deep neural network? Consider a hidden units constructed by z and a (being activation). Ordinarily, we would fit z_1 into the activation function to compute a_1, which takes the following form

$$x \to z^{[1]} \text{ by } x^{[1]}, b^{[1]}$$

but now we can compute

$$x \overset{x^{[1]}, b^{[1]}}{\to} z^{[1]} \overset{\text{Batch norm}}{\to} \tilde{z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{z^{[1]}}) \to z^{[2]} \dots$$

In this way, the parameters are $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ..., W^{[l]}, b^{[l]}$. Additionally, we have $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, ..., \beta^{[}l], \gamma^{[l]}$. After this update, we can then use gradient descent and so on. The same idea also for mini-batch gradient descent. One only needs to notice that the μ and σ^2 are computed on the mini-batch the subset of training instead of the entire training set. Under this construction, every time a new batch is selected we would compute new μ and σ^2. Notice that for mini-batch we are supposed to have $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$ but we can update it as $\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{\text{norm}} + \beta^{[l]}$, which is different than before.

Consider a shallow neural network testing cat or non-cat. Suppose we trained with black cats. However, for color cats later on, it may not do well. This is not surprising since the distribution may from the same one but they could change their location or cluster of locations. This will create a covariant shift in the construction of the hidden layers since these hidden units are changing all the time. What batch norm does is that it reduces the amount that the distribution of these hidden unit values shifts around.

In other words, in batch norm these values will be more stable than before. Batch norm has a second effect as regularization. Each mini-batch is scaled by the mean/variance computed on just that mini-batch. This adds some noise to the values $z^{[l]}$ within that mini-batch. Similar to dropout, it adds some noise to each hidden layer's activations. This has a slight regularization effect. By using large mini-batch size, one can also reduces the noises in the data set during normalizing.

Batch norm processes data one mini batch at a time, but the test time you might need to process the examples one at a time. Recall that during training, the equations you would use to implement batch norm are the following

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$

To make it concrete, consider layer l. There are mini-batches $X^{\{1\}}, X^{\{2\}}, ...$ and we have $\mu^{\{1\}[l]}, \mu^{\{2\}[2]}$, and so on. Just like how we use exponentially weighted average to compute the mean of θ_1 and so on, we use the same notion and that means the exponentially calculated weighted average would become the mean of z's hidden layer and, similarly, you use exponentially weighted average to keep track of these values of σ^2 that you see on the first mini batch in that layer.

2.7 Multi-class Classification

We have discussed the classification with examples using binary classification, where the labels take two possible values 0 or 1. What if we have multiple possible classes? There is a generalization of logistic regression called Softmax regression. Consider example where output 4 values. In the $t = e^{z^{[l]}}$ and this case t is a vector with size (4,1). Then we have

$$a^{[l]} = \frac{\exp(z^{[l]})}{\sum t_i}$$

Now suppose, as an example, $z^{[l]} = [5; 2; -1; 3]$ and

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}, \sum_{j=1}^{4} t_j = 176.3$$

and also

$$a^{[l]} = \frac{t}{176.3} = \begin{bmatrix} e^5/176.3 = 0.842 \\ e^2/176.3 = 0.042 \\ e^{-1}/176.3 = 0.002 \\ e^3/176.3 = 0.114 \end{bmatrix}$$

That is, what Softmax is doing from $z^{[l]}$ to $a^{[l]}$ is to use exponential function to get temporary variable t and then normalizing with the summation of t. The unusual thing about this particular activation function is that the process takes a vector and needs to take the summation to normalize each values in the vector. In great details, let us consider

$$z^{[l]} = [5; 2; -1; 3]; t = [e^5; e^2; e^{-1}; e^3]$$

and activation function

$$a^{[l]} = g^{[l]}(z^{[l]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \underbrace{\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}}_{\text{softmax}}$$

The final decision takes the highest value as if the four outputs are assigned the values as percentages. The take-away is that the softmax function is just a generalization of logistic regression to more than two classes.

How to train a neural network with softmax function? Consider loss function as the following

$$y = [0;1;0;0] \text{ to be cat}$$
$$a^{[l]} = \hat{y} = [0.3;0.2;0.1;0.4]$$

Then the loss would be

$$\mathcal{L}(\hat{y}, y) = \underbrace{-\sum_{j=1}^{4} y_j \log \hat{y}_j}_{\text{left with } -y_2 \log \hat{y}_2 = log \hat{y}_2}$$

the underbrace in the first term would be a form of maximum likelihood function. Then we implement

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

Consider response Y and estimator $\hat{Y}$, respectively, to be

$$\begin{aligned} Y &= [y^{(1)}, y^{(2)}, ..., y^{(m)}] \\ &= \begin{bmatrix} 0 & 0 & 1... \\ 1 & 0 & 0... \\ 0 & 1 & 0... \\ 0 & 0 & 0... \end{bmatrix} \end{aligned}$$

and

$$\begin{aligned} \hat{Y} &= [\hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(m)}] \\ &= \begin{bmatrix} 0.3 & ... \\ 0.2 & ... \\ 0.1 & ... \\ 0.4 & ... \end{bmatrix} \end{aligned}$$

How do we implement gradient descent? Consider the last layer $z^{[l]} \to a^{[l]} = \hat{y} \to \mathcal{L}(\hat{y}, y)$. This is due to backpropagation: $dz^{[l]} = \hat{y} - y$, which is a (4,1) table and dz is $\frac{\partial J}{\partial z^{[l]}}$.

Some famous deep learning frameworks are the following: caffe/caffe2, CNTK, DL4J, Keras, Lasagne, mxnet, PaddlePaddle, TensorFlow, Theano, and Torch. The criteria recommended is the following: (1) ease of programming (development and deployment), (2) running speed, and (3) truly open (open source with governance). Let us consider tensorflow as a deep learning framework example. Consider

$$J(w) = w^2 - 10w + 2s$$

as a simple version of cost function whereas in practice we will be face $J(w, b)$.

3 STRUCTURING MACHINE LEARNING PROJECT

Go back to Table of Contents. Please click TOC

This course is about structure machine learning projects, that is, about machine learning strategy. This course is intend to train users to learn about how to quickly and efficiently get machine learning systems working. Consider an example to classify cats and say we have a machine at 90%. Also say one user or programmer has a bunch of ideas such as 1) collect more data, 2) collect more diverse training set, 3) train algorithm longer with gradient descent, 4) try Adam instead of gradient descent, 5) try bigger network, 6) try smaller network, 7) try dropout, and so on. This course will also discuss a few secrets about machine learning progress.

3.1 Orthogonalization

One interest thing is that some experienced machine learning programmers are very clear about what to tune in order to try to achieve one effect. This is a process called orthogonalization. An intuitive example is to consider an old-school TV with a lot of knobs while each knob to have its own function. Maybe there is one for horizontal position and another for width and so on. Another intuitive example can be driving a car. There are steering, accelerating, and breaking.

With intuitive understanding, one needs to have a chain of assumptions. One should at least watch that the model fits training set well on cost function. One shall then hope this model will fit developed set well on cost function, and henceforth test set well too. Then you have some evidence to hope that this model performs well in real world.

3.2 Set Up

First thing is to pick a single number evaluation metric. One can look at F_1 score which takes the formula: $2/(1/p + 1/r)$, which is also called "harmonic mean" while p means precision and r means recall. If, however, the data set is fitted by a few models and the output is to test each country in a few countries how accurate the model is at classifying cat or not. In this case, it might make more sense to set up an average accuracy to create one single number of goal to follow.

It is not always easy to combine all the things one care about into a single row number evaluation metric. Consider classifier A, B, and C with different accuracies and different running time. Then perhaps it makes sense to combine cost = accuracy - 0.5 $\times$ running time. In this case, we can consider accuracy an optimizing metric while running time is a satisfying metric. This would be a reasonable to put trade-off together that is more acceptable to the programmer. With this setup, it would be more easier to pick optimal classifier.

Another important step in machine learning before one gets started is to set up development set and test sets. In earlier ages, we used to sample size of 100, 1000, or even 10,000. For such sizes, the split of seven-to-three and six-two-two seem reasonable. For data set with, nowadays, a million sample sizes, it is okay to take a much larger fraction into the training such as nine-one split.

3.3 Compare to Human Level

A lot more machine learning teams have been talking about comparing the machine learning systems to human level performance. Why is this? There are two main

reasons. First, it is because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance. Second, it turns out that the workflow of designing and building a machine learning system is much more efficient when you are trying to do something that humans can also do.

Consider x-axis as a time and y-axis as accuracy. Also consider an arbitrary dotted line to be human accuracy. It turns out that machine learning accuracy will increase drastically before reaching human accuracy and only slowly going up or even plateau after slightly surpassing human accuracy. Theoretically, there is another dotted line called the Bayes optimal error, which is the best possible error rate (some arbitrary level above human level. Then the ascending machine learning accuracy can only go so far before it plateau in front of Bayes' optimal accuracy. What does all of this mean?

Consider a medical image classification example with the following assumptions: 1) typical human performs 3% error, 2) typical doctor performs 1% error, 3) experienced doctor 0.7% error, and 4) a team of experienced doctors perform 0.5% error. We can reasonably to assume that the last performance will be a good estimate of Bayes' error, and we can say that Bayes's error is less or equal to 0.5%.

There are two fundamental assumptions of supervised learning. 1) You can fit the training set pretty well. 2) The training set performance generalizes pretty well to the development or test set. One can look at the difference between human-level error and training error, aka the avoidable bias; and also one can look at the difference between training error and development error, aka the variance. To get rid of avoidable bias, one can 1) train bigger model, train longer/better optimization algorithms, using momentum, RMSprop, or Adam, and NN architecture/hyperparameters search, RNN, or CNN. To get rid of variance, one can 1) obtain more data, 2) regularization, using L_2, dropout, data augmentation, or 3) NN architecture/hyperparameter search.

4 CONVOLUTIONAL NEURAL NETWORKS (CNN)

Go back to Table of Contents. Please click TOC

4.1 Convolutional Neural Networks

This section is about convolutional networks. Computer vision is one of the areas that has been advancing rapidly, thanks to deep learning. Deep learning computer vision is now helping self-driving cars to figure out where are the other cars and the pedestrains around us so as to avoid them.

4.1.1 Filter or Kernel

One challenge is to detect the edges, horizontal and vertical edges. We create this notion of a filter (aka, a kernel). Consider an image as the following

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix}$$

and we take a filter (aka, a kernel, in mathematics) which takes the following form,

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

and we apply "convolution" operation, which is the following

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

and the first element would be computed as

$$3 \times 1 + 1 \times 1 + 2 \times 1 + 0 + 0 + 0 + 1 \times -1 + 8 \times -1 + 2 \times -1 = -5$$

Shifting the 3 by 3 matrix, which is the filter (aka, kernel), one column to the right, and we can apply the same operation. For a 6 by 6 matrix, we can shift right 4 times and shift down 4 times. In the end, we get

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}$$

One can use tensorflow function, tf.nn.conv2d, or to use python language, conv_forward, as function to implement. Notice that the filter above is for vertical edge detection and the horizontal edge detection, by the same notion, would be the following

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \text{ and } \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \text{ for vertical and horizontal, respectively}$$

In computer vision literature, there is a lot of debates about what filter to use. A famous one is called Sobel filter, which takes the following form

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

and the advantage of Sobel filter is to put a little bit more weight to the central row. Another famous filter is called Sharr filter, which is

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

Hand-picking these parameters in the filter can be problematic and can be very costly to do so. Why not let the computer to learn these values? That is, we can treat the nine numbers of this filter matrix as parameters, which you can then learn this filter automatically and create a filter detection by computer. Other than vertical and horizontal edges, such computer-generated filter can learn information from any angles and will be a lot more robust than hand-picking values.

4.1.2 Padding

In order to build deep neural networks, one modification to the basic convolutional operation that one needs is padding. Consider earlier example,

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}$$

that is, we take a 6 by 6 matrix and apply a 3 by 3 filter, and we end up with a 4 by 4 matrix. It ends up with a 4 by 4 filter because we can only shift a 3 by 3 matrix one row up/down or one column left/right once and this will sum up 4 times. That is, there are 4 by 4 possible positions for a 3 by 3 matrix to appear in a 6 by 6 matrix. In formula, given an $n \times n$ matrix and an $f \times f$ matrix, we can generate a matrix by applying up/down left/right rule and this will matrix will be $n-f+1$ by $n-f+1$. The first problem is that we are always shrinking the size of the matrix as we move along to apply this computation. In practice, we can only shrink this so big that the image will get really small. Another problem is that the pixels are not used evenly. This means that the pixel on the corner of the image and the pixel in the center of the image are used once and many times, respectively, which is very uneven. This way we will throw away a lot of information. A solution, which is padding, is to pad the image with an extra layer (or additional few layers) of outside edges with zero values with p padding amount. In this case, we have output matrix $n+2p-f+1$ by $n+2p-f+1$, which is a

lot bigger. Taking $p = 1$, we would end up with a 6 by 6 size matrix. We say that the convolution is "valid" meaning that we start with $n \times n = 6 \times 6$ and apply convolution operation on $f \times f = 3 \times 3$ to obtain 4×4 matrix. We say that the convolution gives "same" matrix, meaning that we apply padding and we would end up with matrix that is the size as the input matrix. We can simply solve

$$\begin{aligned} n + 2p - f + 1 &= n \\ 2p - f + 1 &= 0 \\ 2p &= f - 1 \\ p &= (f-1)/2 \end{aligned}$$

notice that by this construction we always need f to be odd. Because if f is even, we would have $(f-1)/2$ to be odd which will cause asymmetric information creation. Moreover, sometimes, especially for odd sized matrix, we would notate the value (or position) in the very center of the filter to be a center position.

4.1.3 Strided Convolutions

Strided convolutions is another piece of the basic building block of convolutions as used in convolutional neural networks.

$$\begin{bmatrix} 2 & 3 & 7 & 4 & 6 & 2 & 9 \\ 6 & 6 & 9 & 8 & 7 & 4 & 3 \\ 3 & 4 & 8 & 3 & 8 & 9 & 7 \\ 7 & 8 & 3 & 6 & 6 & 3 & 4 \\ 4 & 2 & 1 & 8 & 3 & 4 & 6 \\ 3 & 2 & 4 & 1 & 9 & 8 & 3 \\ 0 & 1 & 3 & 9 & 2 & 1 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 91 & 100 & 83 \\ 69 & 91 & 127 \\ 44 & 72 & 74 \end{bmatrix}$$

Using a stride to be two steps, this means that after we finish computation of the first element in the output matrix we would shift two steps to the right instead of just one. When we finish the first row, we would do the same applying stride to be two steps. This means we would shift two rows downward instead of just one. This way, we start with $n \times n$ and apply $f \times f$ filter with padding p and stride s. In formula, we have $(n+2p-f)/s+1$ by $(n+2p-f)/s+1$ output matrix. Then in the above example we have $(7+0-3)/2+1 = 4/2+1 = 3$. However, if occasionally the formula computes a non-integer, we have the notation $\lfloor z \rfloor$ to denote the lower bound by the integer. Thus we have the size for strided convolutions to be

$$\lfloor \frac{n+2p-f}{s} + 1 \rfloor \text{ by } \lfloor \frac{n+2p-f}{s} + 1 \rfloor$$

In conventional math sense, there is a slight difference between between cross-correlation and convolution. Consider the above example and we want to do a convolution via conventional mathematical sense. We would update the filter matrix and flip it upside down as well as left side right. That means,

$$\begin{bmatrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 & 2 & 5 \\ 9 & 0 & 4 \\ -1 & 1 & 3 \end{bmatrix}$$

then we apply the same calculation to the target matrix. In conventional literature, people address this operation as convolution. In signal processing, people also conduct $(A \star B) \star C = A \star (B \star C)$, but we do not complicate our work by this.

4.2 Convolution Over Volume

Now let us consider a volume which is a colorful image. Images in color are usually sized in RGB instead of greyscale. Then there are three sizes: height, width, and number of channels. Suppose target matrix is sized 6 by 6 by 3 and a filter is sized 3 by 3 by 3. We conduct the operations applying the filter to the target matrix one step by one step. This is to say, we want to detect, using the same filter, the information in each channel, i.e. each color layer in the images.

On top this, we have edge detector, which we set middle column of filter to be zeros. We also have options to filter a particular channel we want, simply setting the rest of the channels zeros. We can also detect pictures with different orientation instead of just vertical edges. We can detect horizontal edges. This is to say, we take the same matrix operations and obtain multiple final matrices to stack them together.

In math, let us say we have $n \times n \times n$ matrices to convolve with $f \times f \times n$. We will get

$$\lfloor n - f + 1 \rfloor \times \lfloor n - f + 1 \times n_c' \rfloor$$

assuming using stride $s = 1$.

As a summary, we write the following notation. If layer l is a convolution layer, we have $f^{[l]}$ to be filter size, $p^{[l]}$ is padding, and $s^{[l]}$ is stride. We have input matrix $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$, where H, W, and C denotes, "height", "width", and "convolution", respectively. The output matrix would be $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ where the size is given

$$n^{[l]} = \lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor \text{ for } n_H, n_W, \text{ respectively}$$

4.3 Pooling Layers

Other than convlutional layers, ConvNets often use pooling layers to reduce the size of the representation to speed up the computation as well as to make some of the features that detects signals more robust.

Suppose you have a 4×4 input, and you want to apply a type of pooling called max pooling. The output of this particular implementation of max pooling will be a two by two output. Take the 4×4 input and break it into different regions and let us color the four regions as follows.

$$\begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix} \longrightarrow \begin{bmatrix} 9 & 6 \\ 6 & 3 \end{bmatrix}$$

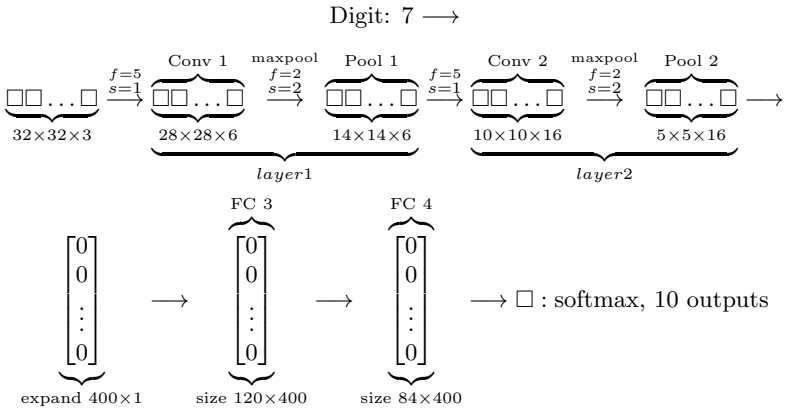
In the output, which is 2×2, each of the outputs will just be the max from the corresponding reshaded region. The upper left, the max of these four numbers, would be nine. On upper right, the max of the blue numbers is three. To compute each of the numbers on the right, we take the max over a 2×2 regions. This is as if you apply a filter size of two because you are taking a 2×2 regions and you are taking a stride of two. These are actually the hyperparameters of max pooling because we start from this filter size.

This can be thought of as some kind of features. The activations in some layer of the neural network, a large number, means that it may be detected a particular feature. That is, the upper left-hand quadrant has this particular feature. It maybe a vertical edge. Clearly that feature exists in the upper left-hand quadrant where as this feature, maybe it is not a cat detector. Where as this feature, it does not really exist in the upper right-hand quadrant. It then remains preserved in the output of max pooling.

What the max operates to do is really , if these features detected anywhere in this filter then keep a high number. But if this feature is not detected, maybe this feature does not exist in the upper right-hand quadrant at all. Then the max of all those numbers is still itself quite small. There is not any theoretical reason why this approach works well. It is just a fact that this approach happen to work well in some data set sets.

4.4 CNN Example

Now let us take a look at an example. You have an input as an image which is $32 \times 32 \times 3$, so it is an RGB image and perhaps you are trying to do handwritten digit recognition. You have a number "7" in a 32×32 RGB initiate trying to recognize which one of the 10 digits from zero to nine is this. Let us conduct the following



and as one can imagine there are two types of conventions to count the number of layers. One can count conv layer and pooling layer together as one convolution layer. Alternatively, one can count convolution layer and pooling layer separately as two layers. In the following discussion, we will count them as one layer.

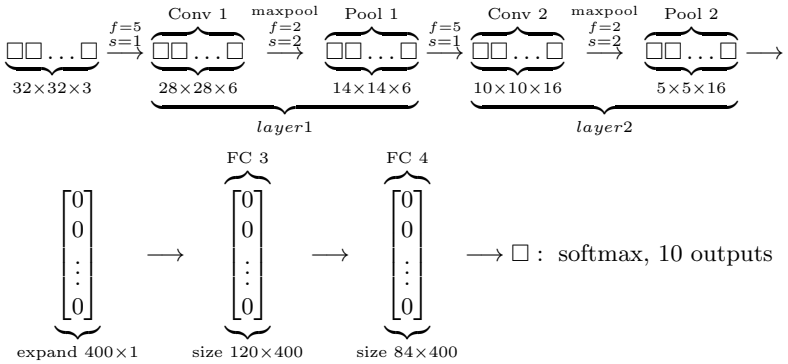
The deeper you go into neural network, the number of heights and width will drop. A combination of a few convolution layers and neural network layers can be conventional model to construct CNN. The example above is a LeNet5. Let us notate all the sizes down into a table below:

–	Activation Shape	Activation Size	# parameters
Input:	(32,32,3)	3072	0
CONV1 (f=5,s=1)	(28,28,8)	6272	208
POOL1	(14,14,8)	1568	0
CONV2 (f=5,s=1)	(10,10,16)	1600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48001
FC4	(84,1)	84	10081
Softmax	(10,1)	10	841

4.5 Why Convolution

To wrap this section, let us discuss why convolutions are so useful when they are included in the neural network construction. There are two main advantages of convolutional layers over just using fully connected layers: (1) parameter sharing, and (2) sparsity of connections. Let us illustrate this with an example. Recall the previous example

Digit: 7 ⟶



From the sizes above, we have $32 \times 32 \times 3$ dimensional image. Using a $5 \times 5 \times 6$ filter, we have a $28 \times 28 \times 6$ dimensional output. That is, 32 by 32 by 3 is 3,072 while 28 by 28 by 6 is 4,704. If we were to connect every one of these neurons together, then the weight matrix, the number of parameters in a weight matrix would be about 14 million. This is a lot of parameters to train. Though today we have large computing powers to train more than 14 million, we should keep in mind that this is just one small image. If we were to look at 1000 by 1000 image, then the display matrix will become invisibly large.

Parameter sharing is a feature detector (such as a vertical edge detector) that is useful in one part of the image is probability useful in another part of the image. Sparsity of connections means that, in each layer, each output value depends only a small number of inputs.

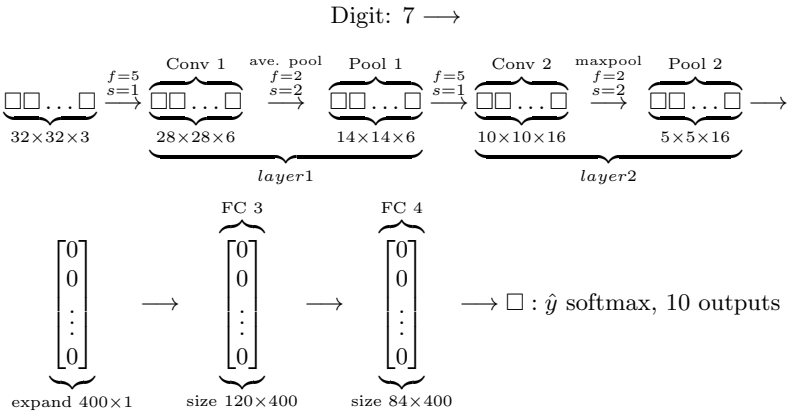
Consider a training set $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$. In this case, the explanatory variables can be a bunch of pixels and the response variables can be k classes. The convolutional layers and the neural network hidden layers will have a variety of parameters. With the parameters chosen, we have cost function

$$\text{Cost: } J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

To train this neural network, we can use gradient descent, gradient descent with momentum, or RMS-prop, Adam, and etc..

4.6 LeNet5

First published in LeCun et al., 1998 [9], LetNet5 is structured as the following:

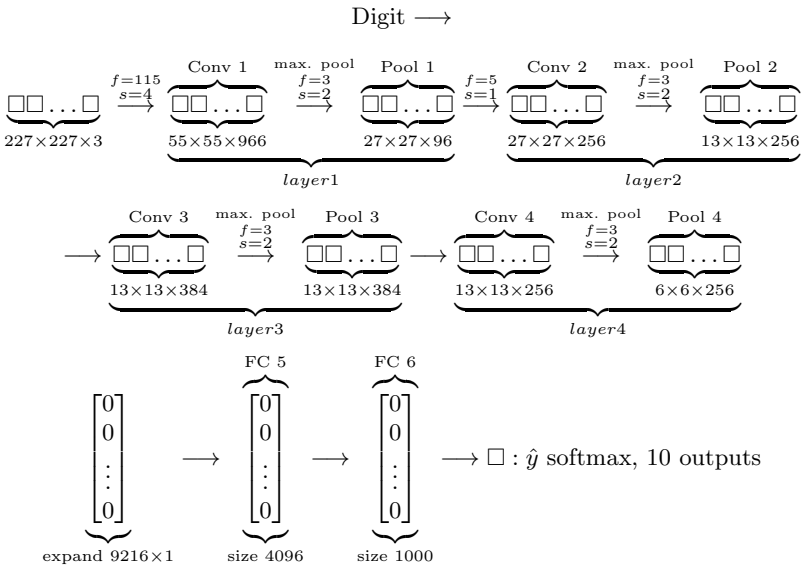


Based on such structure, this neural network was small by modern standards only with 60,000 parameters. Today people often use neural networks with anywhere from 10 million to 100 million parameters and it is no unusual to see networks that literally about a thousand times bigger than this network. One trait we observe is that as you go deeper in a network from left to right, the height and width tend to go down.

From the original paper [9], people use sigmoid/tanh and ReLU mainly.

4.7 AlexNet

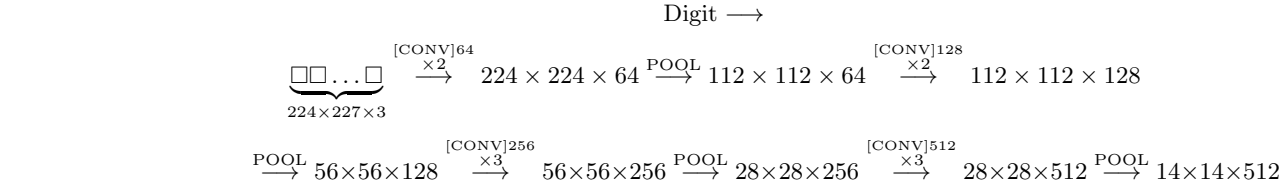
The second example of a neural network to be presented is AlexNet, named after Alex Krizhevsky, who was the first author of the paper describing this work [8]. The other author's were Ilya Sutskever and Geoffrey Hinton. The AlexNet input starts with 227 by 227 by 3 images. The paper actually refers to 224 by 224 by 3 images. However, the instructor just use 224 by 224 by 3 as size. AlexNet also uses a large stride of four, the dimensions shrinks to 55 by 55, that is doing down by a factor of 4 with max pooling of a 3 by 3 filter. That way, we have $f = 3$ and $s = 4$.



This structure actually has a lot of similarities with LeNet5. However, this is a much larger net at around 60 million parameters while LeNet5 only has about 14 million parameters. A better way is to use ReLU and of course this is open for tuning. Parallel training can be conducted using two GPUs together.

4.8 VGG-16

A work done by Simonyan & Zissermann 2015 used a filter of 3 by 3 and stride of 1. They also used max pooling with 2 by 2 and stride of 2. The structure looks like the following



$$\overset{[\text{CONV}]512}{\underset{}{\overset{\times 3}{\longrightarrow}}} 14 \times 14 \times 512 \overset{\text{POOL}}{\longrightarrow} 7 \times 7 \times 512 \longrightarrow \text{FC } 4096 \longrightarrow \text{FC } 4096 \longrightarrow \text{Softmax } 1000$$

while

$$[\text{CONV } 64] \times 2 := \underbrace{\overbrace{\underbrace{\square\square\ldots\square}_{55\times55\times966}}^{\text{Conv 1}} \overset{\substack{\text{max. pool}\\ f=3\\ s=2}}{\longrightarrow} \overbrace{\underbrace{\square\square\ldots\square}_{27\times27\times96}}^{\text{Pool 1}}}_{layer1}, \text{ and etc..}$$

This is a very large network with 138 million parameters. This network has 16 layers while sometimes there are people even talking about using VGG-19 which has 19 layers under above structure. However, VGG-16 does almost as well as VGG-19.

4.9 ResNet

Deep neural networks are difficult to train because of vanishing and exploding gradient types of problems. Here let us discuss skipping connections which allows you to take the activation from one layer and suddenly feed it to another layer even much deeper in the neural network. Using this as a tool, we are able to build networks that enable us to train very deep structures. Sometimes we can train even over 100 layers. ResNets [5] are built out of something called a residual block. Let us consider the following example.

$$a^{[l]} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \overset{a^{[l+1]}}{\longrightarrow} \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} a^{[l+2]}$$

where the step from $a^{[l]}$ to $a^{[l+1]}$ is a linear relationship governed by $z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$ and the application of ReLU to step $a^{[l+1]}$ is governed by $a^{[l+1]} = g(z^{[l+1]})$. This gives us $a^{[l+1]}$. To move from $a^{[l+1]}$ to $a^{[l+1]}$, we apply the same procedure but change our index to $l+1$ and $l+2$. That is, we have $z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$ while $g^{[l+2]} = g(z^{[l+2]})$. That is, we have the following theoretical map:

$$a^{[l]} \longrightarrow \text{Linear: } z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \longrightarrow \text{ReLU: } a^{[l+1]} = g(z^{[l+1]})$$
$$\longrightarrow \text{Linear: } z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]} \longrightarrow \text{ReLU: } g^{[l+2]} = g(z^{[l+2]}) \longrightarrow a^{[l+2]}$$

For ResNets, we are going to make a change to the above map:

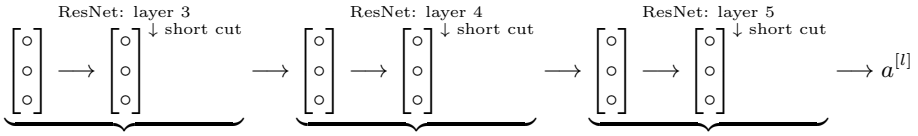
$$a^{[l]} \longrightarrow \text{Linear: } z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \longrightarrow \text{ReLU: } a^{[l+1]} = g(z^{[l+1]})$$
$$\longrightarrow \text{Linear: } z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]} \longrightarrow \text{Short cut: add and store} a^{[l]}$$
$$\longrightarrow \text{ReLU: } [\underbrace{g^{[l+2]} = g(z^{[l+2]})}_{\text{instead of it}}] \text{ We use: } a^{[l+2]} = g(z^{[l+2]} + a^{[l]} \longrightarrow a^{[l+2]}$$

From this construction, we can look at the graphical demonstration, presented in the beginning of this section with more layers added to it:

$$X \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} a^{[l]}$$

which is a conventional neural network with which we can add the constructed short cuts as designed above. This gives us

$$X \longrightarrow \underbrace{\begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix}^{\downarrow \text{ short cut}}}_{\text{ResNet: layer 1}} \longrightarrow \underbrace{\begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \longrightarrow \begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix}^{\downarrow\text{short cut}}}_{\text{ResNet: layer 2}} \longrightarrow$$



which shows five residual blocks stacked together which constructs a residual network. The training for conventional procedure will give us a descending then ascending training set error (going down first but then going back up) as we increase number of hidden layers while the theoretical training error will only get better not worse. However, the usage of residual building blocks in a network, a ResNet, will not give us such problem. We observe, even over a thousand layers, descending training set error.

Why do residual networks work well? Consider the following example

$$X \to \text{Big NN} \to a^{[l]}$$

$$X \to \text{Big NN} \to \begin{bmatrix}\circ\\ \circ\\ \circ\end{bmatrix} \to \begin{bmatrix}\circ\\ \circ\\ \circ\end{bmatrix} \overset{\downarrow}{\to} a^{[l+2]}$$

For the sake of argument, say we are using ReLU activation functions. That is, we have $a \geq 0$ with possible exception of input observations. Then $a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$. If you are using L_2 regularization as weight decay, this will tend to shrink the value of $W^{[l+2]}$. However, if we choose $W^{[l+2]} = 0$ and $b^{[l+2]} = 0$, then we have $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$. What this means is that the identity function is easy for residual block to learn. It is easy to get $a^{[l+2]}$ equal to $a^{[l]}$.

4.10 Networks with One by One Convolution

In terms of designing content architectures, one of the ideas that really helps is using a one by one convolution [10]. How is that helpful? For one by one filter, we can put in number two there. If you take the six by six image, and convolve it with this one by one filter, you would end up just taking the image and multiplying it by two. That is, a one by one filter does not seem particularly useful. However, let us look at a 6 by 6 by 32 filter instead of 1 by 1. The story will be very different. The convolution will take element-wise product between 32 numbers on the left and 32 numbers in the filter. Then we apply a ReLU non-linearity to it after that step. One can refer to the following graphics for illustration:

$$\text{Image} \to \begin{bmatrix}1 & 2 & 3 & \dots\\ \vdots & & & \end{bmatrix} \star \text{filter: } [2] \to \begin{bmatrix}2 & 4 & 6 & \dots\\ \vdots & & & \end{bmatrix} \text{... not helpful}$$

However, if one choose a 1 by 1 by 32 sized filter, one would obtain

$$\text{Image} \to \begin{bmatrix}\circ & \circ\\ \circ & \circ\end{bmatrix} \star \text{filter: } [\circ][\circ]...[\circ]_{1\times 1\times 32} \to \begin{bmatrix}\circ & \circ\\ \circ & \circ\end{bmatrix}\begin{bmatrix}\circ & \circ\\ \circ & \circ\end{bmatrix} ... \begin{bmatrix}\circ & \circ\\ \circ & \circ\end{bmatrix}_{6\times 6\times 32} \text{ very helpful}$$

4.11 Inception Network

When designing a layer for a convolution network, one might have to pick different and a various of a parameters. One can choose 1 by 3 filter, or 3 by 3, etc. One can also choose pooling layer. The notion of inception network says why not do them all [14]. Let us say one has inputted a 28 by 28 by 192 dimensional volume. The inception network would, instead choosing what filter size to use, choose a group of filters together. One can use a 1 by 1 convolution and that will output 28 by 28 by

something. One can, in addition, add 28 by 28 by 64 output and he/she would only need to put the volume next to the above one together as one convolution layer. One can keep going to select any sizes of filter and any number of filters he/she wants to construct the layer of convNet.

$$\text{Image: } \rightarrow \begin{bmatrix} \circ & \circ \\ \circ & \circ \end{bmatrix} \rightarrow \begin{array}{lccc} (1): & 1\times 1 & \rightarrow & 28\times 28\times 64 \\ (2): & 3\times 3 & \rightarrow & 28\times 28\times 128 \\ (3): & 5\times 5 & \rightarrow & 28\times 28\times 32 \\ (4): & \text{Max-pool} & \rightarrow & 28\times 28\times 32 \end{array} \rightarrow \begin{array}{c} \square\square\ldots\square \\ \square\square\ldots\square \\ \square\square\ldots\square \\ \square\square\ldots\square \end{array} \begin{array}{c} 28\times 28\times 64 \\ 28\times 28\times 128 \\ 28\times 28\times 32 \\ 28\times 28\times 32 \end{array}$$

$$\longrightarrow \underbrace{\square\square\ldots\square}_{28\times 28\times 256 \text{ sized convNet, conv layer 1}}$$

There is a problem with the construction of inception network, which is computational cost. What exactly would be the cost? By the above example, there are 5×5 same filters chosen for the first convolution layer with each of them 5×5192. This is a total of $28\times 28\times 32\times 5\times 5\times 192\approx 192$ million parameters, which is a pretty expensive computation for just one layer.

How do we reduce this cost? A natural way is to implement one by one convolution. That is, we will start with inputting observations and apply a conv layer of 1×1, 16, and $1\times 1\times 192$, which will give us $28\times 28\times 16$ before we apply inception network. In this case, the cost of the first convolution layer would be $28\times 28\times 16$ which would be around 2.4 million parameters. The second convolution layer would be $28\times 28\times 32\times 5\times 5\times 16$ which would be 10 million parameters. This is a total of 12 million parameters, a tenth of the previous previous procedure.

4.12 A Few Advices

This section we discuss a few advices for doing research and constructing networks.

First of all, one should consider doing is to search for open source implementation. We can go much faster to use previous scholars' work and codes.

Continuing from the first point, it may be more efficient to take a previous work and start building on top of that. This is especially true for tuning and parameters choices. For example, one wants to build a cat classification. One may have a small training set. What one can do is to download not just the codes as well as the weights for the already trained data sets, which usually have a lot more observations and well-tuned results.

4.13 Detection Algorithm

This section we learn about object detection. This is one of the areas of computer vision that is really exploding currently.

4.13.1 Object Localization

The problem we learn to build in the network is classification with localization. This means that not only do we want to label this as some object but the algorithm also needs to put a bounding box or draw a bounding box around the position in the image.

We are already familiar with image classification problem, in which we input a picture to ConvNet with multiple layers and the results are several classes. The standard classification pipeline are pedestrian, car, motorcycle, or background.

What about bounding box? We need to have a bounding frame to define bounding box. That is, we need a coordinate in the picture. Standard notation writes (0,0) as the upper-left corner and (1,1) to be the lower-right corner. Then we have (b_x, b_y, b_n, b_w) as the bounding box. The numeric output of this vector may look like $(0.5, 0.7, 0.3, 0.4)$.

In this case, the loss function would be

$$\mathcal{L}(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \cdots + (\hat{y}_j - y_j)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 1 \end{cases}$$

which describes loss function depending on the output of y_1. That is, if there is target object detected, then we want to output the coordinate. If there is not any detected, then we simply care about the neural network training.

4.13.2 Landmark Detection

Neural network can output just the important coordinates, called landmarks, in the picture. For example, we want the results to be a car or someone's right eye. That is, we only want the coordinates of the targeted object or objects. That is, consider the following illustration:

$$\text{Image} \longrightarrow \text{ConvNet} \longrightarrow \underbrace{\begin{bmatrix} \circ \\ \circ \\ \circ \end{bmatrix} \text{Output: face,} l_{1x}, l_{1y}, ..., l_{64x}, l_{64y}}_{\text{total of 129 outputs}}$$

To do this, programmer (or someone) would have to go through all the landmarks in the algorithm in terms of coding purpose (this is still a supervised learning project).

4.13.3 Object Detection

Here we learn object detection by learning sliding windows detection algorithm.

Starting with an arbitrary window size, we input the sub-picture into already-trained ConvNet to detect objects. Then we shift the detection window right with one step and feed the observations into ConvNet. We need to go through every region so the stride would need to be small for the detection algorithm to run smoothly. One can even change (this can be tuned) the size of the detection window so that we can detection different sizes of the picture. However, the sliding windows object detection algorithm using a ConvNet is heavy computationally costly.

4.13.4 Convolutional Implementation

We learned that the sliding window object detection algorithm is very costly to be done. Fortunately, there is a solution to implement that algorithm convolutionally. That is, instead of choosing a smaller picture and feed that picture into ConvNet, we convolutionally run algorithm to output what we want and finish every output each convolution sub-picture within the ConvNet. This way we do not need to do run convolution step twice for all the strides.

4.13.5 Bounding Box Predictions

We have discussed how to use a convolutional implementation of sliding windows, which is more computationally efficient. However, it still has a problem of not quite outputting the most accurate bounding boxes. Let us see how we can get the bounding box predictions to be more accurate.

The problem is that sliding windows may or may not exactly crop the exact image just right. For an example, say we want to detect a car in the picture. The sliding windows (since we initially set the sizes and we did not know how big the car is) may crop half of the car. In some cases, the car may look like a rectangle instead of square.

To solve this problem, we can use YOLO algorithm (YOLO stands for "you only look once") [11]. For example, let us say we have a picture with two cars in it. Let us use a grid for 3 by 3. For each grid cell, we want to label $y = [p_c; b_x; b_y; b_h; b_w; c_1; c_2; c_3]$. First we run algorithm through the first grid and we would output no cars. We keep going until we hit a grid that crops a car or part of a car. Once a car is detected we set the grid to the car. This will give us a target output of $3 \times 3 \times 8$ tube. Now we have a neural network such as

$$\underbrace{\text{Image}}_{\text{size: } 100\times100\times3} \longrightarrow \text{CONV} \longrightarrow \text{MAXPOOL} \longrightarrow \cdots \longrightarrow \underbrace{\square\square\ldots\square}_{\text{size: } 3\times3\times8}$$

Another advantage is that this is a convolution implementation so this algorithm actually runs very fast.

4.13.6 Intersection Over Union

How do we tell if the object detection algorithm is working well? Here we discuss a notion called "intersection over union" . In object detection task, we expect to localize the object as well. The intersection over union function (IOU) computes the intersection over union of these two bounding boxes. The union of these two bounding boxes is the pixels that belong to the first union or the second unions. We include all of these pixels together as a set to be the union. Intersection would be the pixels in both pixels. Taking the ratio of these two sets, we are able to judge whether the prediction of bounded boxes is satisfied. Conventionally, we use 0.5 as cut-off to measure the accuracy of whether such algorithm detects localization correctly. That being said, 0.5 is just chosen arbitrarily. Sometimes people use 0.6 or 0.7 for higher accuracy, but rarely drops below 0.5.

4.13.7 Non-max Suppression

Non-max suppression needs us to put a grid on top of the picture. Since we are running image classification on each grid cell and since we are using a finer grid, it is possible that we will end up with multiple detection. Non-max suppression will clean up these detections according to the probabilities associated with these detections. The algorithm will take the largest one and hide that detection temporarily. The rest of the detections with high IOU will then get suppressed. We apply the same process for the rest of the detections. After this is done, we get rid of all the non-max detection and release the one with the highest probability. The threshold we can use to discard detections with $p_c \leq 0.6$. While there are any remaining boxes: (1) pick the box with the largest p_c output that as a prediction. and (2) discard any remaining box with IOU ≥ 0.5 with the box output in the previous step.

4.13.8 Anchor Boxes

One of the problems with object detection is that each of the grid cells can detect only one object. What if a grid cell wants to detect multiple objects? One of the ideas is to use anchor boxes . An anchor box is a pre-defined shape so that we are able to associate the box with a particular prediction. In general, we may be using multiple anchor boxes. For example, let us say out output has classes human and car. Anchor box for human may be a vertical rectangle while that for car may be a horizontal rectangle. Thus, we may label

$$y = [\underbrace{p_c; b_x; b_y; b_h; b_w; c_1; c_2; c_3}_{\text{anchor box 1}}; \underbrace{p_c; b_x; b_y; b_h; b_w; c_1; c_2; c_3}_{\text{anchor box 2}}]$$

Previously, each object in training image is assigned to grid cell that contains that object's midpoint. Now with two anchor boxes, each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with the highest IOU.

4.13.9 YOLO Algorithm

Suppose we want to train algorithm to detect 1 == pedestrian, 2 == car, and 3 == motorcycle. We define the following label

$$y = [\underbrace{p_c; b_x; b_y; b_h; b_w; c_1; c_2; c_3}_{\text{anchor box 1}}; \underbrace{p_c; b_x; b_y; b_h; b_w; c_1; c_2; c_3}_{\text{anchor box 2}}]$$

while we can set size of y to be $3 \times 32 \times 8 = 3 \times 3 \times 16$. That means, for grid that there is nothing in it, we have output of

$$y = [\underbrace{0; ?; ?; ?; ?; ?; ?; ?; ?}_{\text{anchor box 1}}; \underbrace{0; ?; ?; ?; ?; ?; ?; ?; ?}_{\text{anchor box 2}}]$$

while for anchor box with objects in it, say a car (anchor box 2), the output would be

$$y = [\underbrace{0; ?; ?; ?; ?; ?; ?; ?; ?}_{\text{anchor box 1}}; \underbrace{p_c; b_x; b_y; b_h; b_w; 0; 1; 0}_{\text{anchor box 2}}]$$

4.14 Face Recognition

Face recognition problems commonly fall into two categories: (1) Face Verification - is this the claimed person? or (2) Face Recognition - who is this person?

In face verification, we are given two images and we need to train a machine to tell if they are of the same person. The easy way is to compare the two images pixel-by-pixel. If the distance between the raw images are less than a chosen threshold, it may be the same person. Of course such algorithm performs very poorly since the pixel values change dramatically due to variations in lighting, orientation of the person's face, even minor changes in head position, and so on.

A solution is to use FaceNet model that takes a lot of data and long time to train. Such architecture is proposed following the inception model by [14].

4.14.1 Triplet Loss

For an image x, we denote its encoding $f(x)$, where f is the function computed by the neural network.

$$x = \text{image} \longrightarrow \text{Inception model} \longrightarrow f(x) = \begin{pmatrix} 0.931 \\ 0.433 \\ 0.331 \\ \vdots \\ 0.942 \\ 0.158 \\ 0.039 \end{pmatrix}$$

Training will use triplets of images (A, P, N) while A is an "Anchor" image – a picture of a person, P is a "Positive" image – a picture of the same person as the Anchor image, and N is a "Negative" image – a picture of a different person than the Anchor image.

These triplets are picked from our training dataset. We will write $(A^{(i)}, P^{(i)}, N^{(i)})$ to denote the i-th training example. We need to make sure that an image $A^{(i)}$ of an individual is closer to the Positive $P^{(i)}$ than to the Negative image $N^{(i)}$) by at least a margin α:

$$||f(A^{(i)}) - f(P^{(i)})||_2^2 + \alpha < ||f(A^{(i)}) - f(N^{(i)})||_2^2$$

This becomes an optimization problem and we need to minimize the following "triplet cost":

$$\mathcal{J} = \sum_{i=1}^{m} [\underbrace{||f(A^{(i)}) - f(P^{(i)})||_2^2}_{(1)} - \underbrace{||f(A^{(i)}) - f(N^{(i)})||_2^2}_{(2)} + \alpha]_+$$

To implement the triplet loss as defined by the above formula. Here are 4 steps:

1. Compute the distance between the encodings of "anchor" and "positive": $|| f(A^{(i)}) - f(P^{(i)}) ||_2^2$

2. Compute the distance between the encodings of "anchor" and "negative": $|| f(A^{(i)}) - f(N^{(i)}) ||_2^2$

3. Compute the formula per training example: $|| f(A^{(i)}) - f(P^{(i)}) | - || f(A^{(i)}) - f(N^{(i)}) ||_2^2 + \alpha$

4. Compute the full formula by taking the max with zero and summing over the training examples:

$$\mathcal{J} = \sum_{i=1}^{m} [|| f(A^{(i)}) - f(P^{(i)}) ||_2^2 - || f(A^{(i)}) - f(N^{(i)}) ||_2^2 + \alpha]$$

Useful functions: 'tf.reduce_sum()', 'tf.square()', 'tf.subtract()', 'tf.add()', 'tf.maximum()'. For steps 1 and 2, you will need to sum over the entries of $|| f(A^{(i)}) - f(P^{(i)}) ||_2^2$ and $|| f(A^{(i)}) - f(N^{(i)}) ||_2^2$ while for step 4 you will need to sum over the training examples.

4.15 Neural Style Transfer

Neural Style Transfer (NST) is one of the most fun techniques in deep learning. As seen before, we can take a content image and a style image to merge two images together to become a generated image. Namely, we have $C + S = G$.

Neural Style Transfer (NST) uses a previously trained convolutional network and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning . Following Gatys et al. (2015) [4], the algorithm take VGG model, speicially VGG-10, a 19-layer version of the VGG network and start building on top of that.

4.15.1 Cost Function

As we saw in lecture, the earlier (shallower) layers of a ConvNet tend to detect lower-level features such as edges and simple textures, and the later (deeper) layers tend to detect higher-level features such as more complex textures as well as object classes.

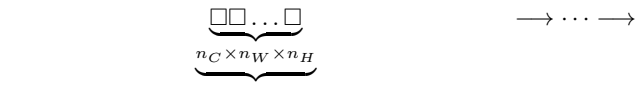
We would like the "generated" image G to have similar content as the input image C. Suppose you have chosen some layer's activations to represent the content of an image. In practice, you'll get the most visually pleasing results if you choose a layer in the middle of the network–neither too shallow nor too deep. (After you have finished this exercise, feel free to come back and experiment with using different layers, to see how the results vary.)

Suppose you have picked one particular hidden layer to use. Now, set the image C as the input to the pre-trained VGG network, and run forward propagation. Let $a^{(C)}$ be the hidden layer activations in the layer you had chosen. (In lecture, we had

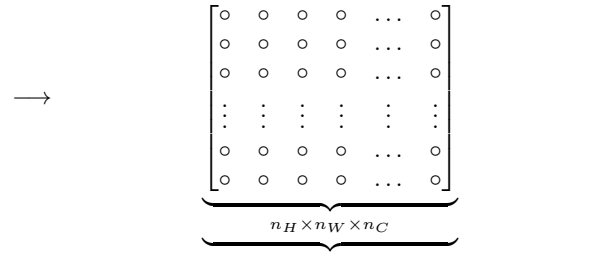
written this as $a^{[l](C)}$, but here we will drop the superscript $[l]$ to simplify the notation.) This will be a $n_H \times n_W \times n_C$ tensor. Repeat this process with the image G: Set G as the input, and run forward propagation. Let $a^{(G)}$ be the corresponding hidden layer activation. We will define as the content cost function as:

$$J_{\text{content}}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

Here, n_H, n_W and n_C are the height, width and number of channels of the hidden layer you have chosen, and appear in a normalization term in the cost. For clarity, note that $a^{(C)}$ and $a^{(G)}$ are the volumes corresponding to a hidden layer's activations. In order to compute the cost $J_{\text{content}}(C, G)$, it might also be convenient to unroll these 3D volumes into a 2D matrix, as shown below. (Technically this unrolling step isn't needed to compute J_{content}, but it will be good practice for when you do need to carry out a similar operation later for computing the style constant J_{style}.)



Convolution layer: each color represents the act. of a filter



Unrolled Version: each row represents the activations of a filter

4.16 Style Matrix

The style matrix is also called a "Gram matrix" . In linear algebra, the Gram matrix G is a set of vectors $(v_1, ..., v_n)$ is the matrix of dot products, whose entries are $G_{ij} v_i^T v_j =$ np.dot(v_i, v_j). In other words, G_{ij} compares how similar v_i is to v_j: if they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large.

Note that there is an unfortunate collision in the variable names used here. We are following common terminology used in the literature, but G is used to denote the Style matrix (or Gram matrix) as well as to denote the generated image G. We will try to make sure which G we are referring to is always clear from the context.

The result is a matrix of dimension (n_C, n_C) where n_C is the number of filters. The value G_{ij} measures how similar the activations of filter i are to the activations of filter j.

One important part of the gram matrix is that the diagonal elements such as G_{ii} also measures how active filter i is. For example, suppose filter i is detecting vertical textures in the image. Then G_{ii} measures how common vertical textures are in the image as a whole: If G_{ii} is large, this means that the image has a lot of vertical texture.

By capturing the prevalence of different types of features (G_{ii}), as well as how much different features occur together (G_{ij}), the Style matrix G measures the style of an image.

5 NATURAL LANGUAGE PROCESSING

Go back to Table of Contents. Please click TOC

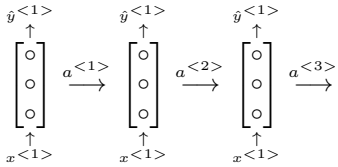
This is the fifth course on deep learning. In this course, we discuss about sequence models, one of the most exciting areas in deep learning. Models like recurrent neural networks or RNNs have transformed speech recognition, natural language processing and other areas. Moreover, in this course, you learn how to build these models for yourself. Let us start by looking at a few examples of where sequence models can be useful. In speech recognition you are given an input audio clip X and asked to map it to a text transcript Y. Both the input and the output here are sequence data, because X is an audio clip and so that plays out over time and Y, the output, is a sequence of words. So sequence models such as a recurrent neural networks and other variations, you will learn about in a little bit have been very useful for speech recognition. Music generation is another example of a problem with sequence data. In this case, only the output Y is a sequence, the input can be the empty set, or it can be a single integer, maybe referring to the genre of music you want to generate or maybe the first few notes of the piece of music you want. But here X can be nothing or maybe just an integer and output Y is a sequence. Some other examples can be DNA sequence data, machine translation, video activity recognition, and so on.

Let us discuss some notations. Let X be "Harry Potter and Hermionie Granger invented a spell". The Y can be (110110000). Each word from X can be $x^{<1>}$, ..., $x^{<n>}$, etc. Say Harry can be $x^{<1>}$ in the X. We use T_X and T_Y to denote the length of the vectors. In this case, $T_X = 9$. Then let us check the dictionary. We observe the following index order, (a, aaron, ..., and, ..., harry, ..., potter, ..., zulu), which gives us indexed order for each word, i.e. (1, 2, ..., 367, ..., 4075, ..., 6830, ..., 10,000).

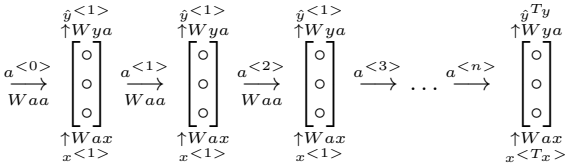
A natural way is to throw these variables into neural network and observe the outcome. It turns out this performs poorly. There are the following reasons: (1) input and output have different lengths; (2) a naive neural network does not share features learned across different positions of text.

5.1 Recurrent Neural Network

Let us say we are reading a sentence. We read the sentence word by word. that is, the store information of the first word we read and this is taken into consideration when we read the second word. In graphical analysis, we present the following flow chart

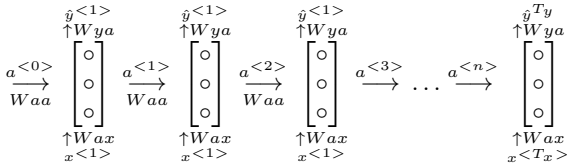


each step we also compute the weights that govern the predictions, which allow us to implement the above flow chart as the following



5.2 Different Types of RNN

The input X and Y can be of many types and they do not have to be of the same lengths. Let us recall the structure of recurrent neural network, which is a many-to-many architecture



Alternatively, we can have a data set that is a sentiment analysis. The data set records people's preferences about movies. Another example can be that we are trying to recognize music segments. The vanilla RNN simply consists of a architecture such that the size (i.e. length) of the input and output variables are the same. However, depending on different sizes, we have ourselves one-to-many, many-to-one, and many-to-many types of RNN.

5.3 Language Model

Formal languages, like programming languages, can be fully specified. All the reserved words can be defined and the valid ways that they can be used can be precisely defined. We cannot do this with natural language. Natural languages are not designed; they emerge, and therefore there is no formal specification. There may be formal rules for parts of the language, and heuristics, but natural language that does not confirm is often used. Natural languages involve vast numbers of terms that can be used in ways that introduce all kinds of ambiguities, yet can still be understood by other humans.

Further, languages change, word usages change: it is a moving target. Nevertheless, linguists try to specify the language with formal grammars and structures. It can be done, but it is very difficult and the results can be fragile. An alternative approach to specifying the model of the language is to learn it from examples.

Recently, the use of neural networks in the development of language models has become very popular, to the point that it may now be the preferred approach. The use of neural networks in language modeling is often called Neural Language Modeling, or NLM for short. Neural network approaches are achieving better results than classical methods both on standalone language models and when models are incorporated into larger models on challenging tasks like speech recognition and machine translation.

Let us consider an example. For each RNN project, we want to first identify a corpus of text as the training set. One can take the text, "cats average 15 hours of sleep a day". We can also add "EOS", meaning end of sentence, to indicate to us that the sentence ends.

5.4 Gated Recurrent Unit (GRU)

Recall that $a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ while $g(\cdot)$ is a tanh function. The earlier scholarly papers such as [2] and [3] provided thorough ideas in this realm. They raised the notion of a memory cell, namely C. At time t, the memory would have content $C^{<t>}$, which is equal to $a^{<t>}$. This will give us

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

which will be a candidate to replace $C^{<t>}$. We call the gate $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$. To compute the actual value of GRU, we use

$$C^{<t>} = \Gamma_u \star \tilde{C}^{<t>} + (1 - \Gamma_u) \star C^{<t-1>}$$

In other words, we have the following flow chart

$$C^{<t-1>} = a^{<t-1>} \longrightarrow \left[\begin{array}{cccc} \boxed{C^{<t>}} & \leftarrow & & \\ \uparrow & \tilde{C}^{<t>} & \uparrow & \Gamma_u \\ \boxed{\tanh} & & \boxed{\sigma} & \\ \uparrow & & \uparrow & \\ \boxed{X^{<t>}} & \rightarrow & & \end{array} \right] \longrightarrow C^{<t>} = a^{<t>}$$

which can be helpful for vanishing gradient problem.

5.5 Long Short Term Memory (LSTM)

Long short-term memory (LSTM) units (or blocks) are a building unit for layers of a recurrent neural network (RNN). A RNN composed of LSTM units is often called an LSTM network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feed-forward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.

The work for Long Short Term Memory (LSTM) is published first by [7]. Recall GRU takes the following path

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

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_u = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$
$$c^{<t>} = \Gamma_u \star \tilde{c}^{<t>} + (1 - \Gamma_u) \star c^{<t-1>}$$
$$a^{<t>} = c^{<t>}$$

while LSTM takes the following path

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

$$\text{update: } \Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$
$$\text{forget: } \Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$
$$\text{output: } \Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$
$$c^{<t>} = \Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$$

There are several architectures of LSTM units. A common architecture is composed of a memory cell, an input gate, an output gate and a forget gate.

An LSTM (memory) cell stores a value (or state), for either long or short time periods. This is achieved by using an identity (or no) activation function for the memory cell. In this way, when an LSTM network (that is an RNN composed of LSTM units) is trained with backpropagation through time, the gradient does not tend to vanish.

The LSTM gates compute an activation, often using the logistic function. Intuitively, the input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit.

There are connections into and out of these gates. A few connections are recurrent. The weights of these connections, which need to be learned during training, of an LSTM unit are used to direct the operation of the gates. Each of the gates has its own parameters, that is weights and biases, from possibly other units outside the LSTM unit.

5.6 Bidirectional RNNs

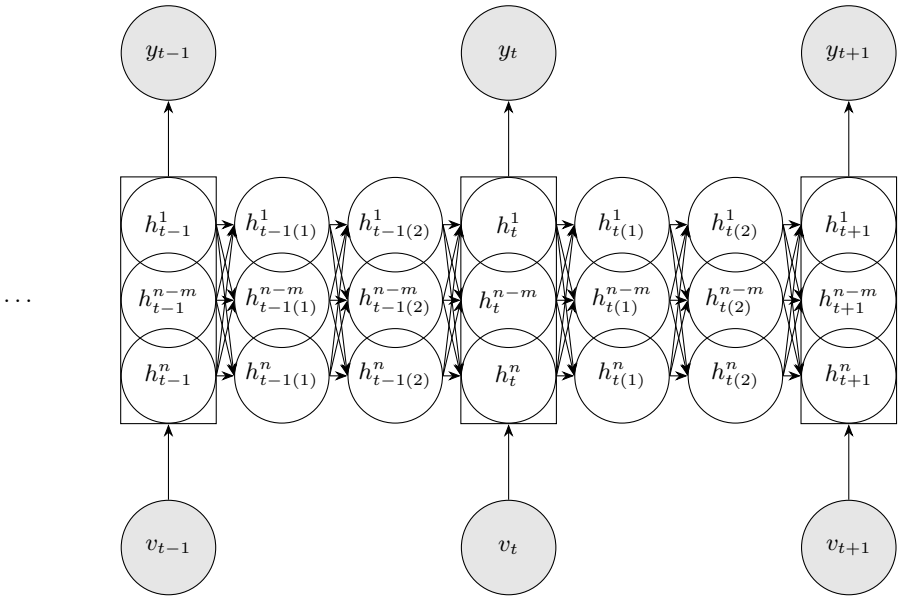
Bidirectional Recurrent Neural Networks (BRNN) were invented in 1997 by Schuster and Paliwal [12]. BRNNs were introduced to increase the amount of input information available to the network. For example, multilayer perceptron (MLPs) and time delay neural network (TDNNs) have limitations on the input data flexibility, as they require their input data to be fixed. Standard recurrent neural network (RNNs) also have restrictions as the future input information cannot be reached from the current state. On the contrary, BRNNs do not require their input data to be fixed. Moreover, their future input information is reachable from the current state. The basic idea of BRNNs is to connect two hidden layers of opposite directions to the same output. By this structure, the output layer can get information from past and future states.

BRNN are especially useful when the context of the input is needed. For example, in handwriting recognition, the performance can be enhanced by knowledge of the letters located before and after the current letter.

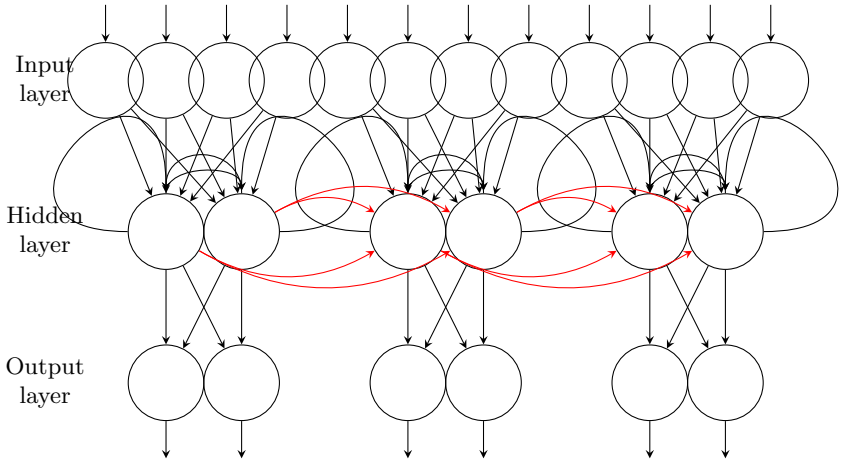
The principle of BRNN is to split the neurons of a regular RNN into two directions, one for positive time direction (forward states), and another for negative time direction (backward states). Those two states' output are not connected to inputs of the opposite direction states. The general structure of RNN and BRNN can be depicted in the right diagram. By using two time directions, input information from the past and future of the current time frame can be used unlike standard RNN which requires the delays for including future information.

5.7 Deep RNNs

A more sophisticated graphical illustration can be the following.



Alternatively, we can use the following graph.



5.8 Word Embeddings

The historical representation is to present words as an index from a dictionary. Given a 10,000 words dictionary v and a word "man" indexed to be the 5391th word, we simply code the word "man" as [0,0, ..., 1, ..., 0] with 1 at the 5391th place and 0s elsewhere. A problem is due to the randomness of dictionary and our day-to-day conversation. We can say "I want a glass of orange juice" or we can say "I want a glass of apple juice". However, the indexed order for "apple" and "juice" are nowhere close to each other. This causes a potential problem.

t-distributed stochastic neighbor embedding (t-SNE) is a machine learning algorithm for dimensionality reduction developed by Geoffrey Hinton and Laurens van der

Maaten [6], which can then be visualized in a scatter plot. Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points.

The t-SNE algorithm comprises two main stages. First, t-SNE constructs a probability distribution over pairs of high-dimensional objects in such a way that similar objects have a high probability of being picked, whilst dissimilar points have an extremely small probability of being picked. Second, t-SNE defines a similar probability distribution over the points in the low-dimensional map, and it minimizes the Kullback–Leibler divergence between the two distributions with respect to the locations of the points in the map. Note that whilst the original algorithm uses the Euclidean distance between objects as the base of its similarity metric, this should be changed as appropriate.

t-SNE has been used in a wide range of applications, including computer security research, music analysis, cancer research, bioinformatics[15], and biomedical signal processing. It is often used to visualize high-level representations learned by an artificial neural network.

Index

References

[1] Ng, Andrew, Coursera Deep Neural Network 5-Sequence Lecture.

[2] Cho et al, 2014, "On the Properties of Machine Learning Translation: Encoder-decode Approaches".

[3] Chung et al 2014, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling".

[4] Gatys et al., 2015, "A Neural Algorithm of Artistic Style".

[5] He et al., 2015, "Deep residual networks for image recognition".

[6] van der Maaten, L.J.P.; Hinton, G.E. (Nov 2008). "Visualizing High-Dimensional Data Using t-SNE" (PDF). Journal of Machine Learning Research. 9: 2579–2605.

[7] Hochreiter Schmidhuber 1997, "Long short-term Memory".

[8] Krizhevsky et al., 2012, "ImageNet classification with deep convolutional neural networks".

[9] LeCun et al., 1998, "Gradient-based learning applied to document recognition".

[10] Lin et al., 2013, "Network in Network".

[11] Redmon et al, 2015, "You Only Look Once: Unified, Real-Time Object Detection".

[12] Schuster, Mike, and Kuldip K. Paliwal, 1997, "Bidirectional recurrent neural network".

[13] Simonya, Zisserman, 2015. "Very deep convolutional networks for large-scale image recognition".

[14] Szegedy et al 2014, "Going deeper with Convolutions".

[15] Wallach, I.; Liliean, R. (2009). "The Protein-Small-Molecule Database, A Non-Redundant Structural Resource for the Analysis of Protein-Ligand Binding". Bioinformatics. 25 (5): 615–620.