DEEP LEARNING AND NEURAL NETWORK SPECIALIZATION (COURSERA)

COURSE 2

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

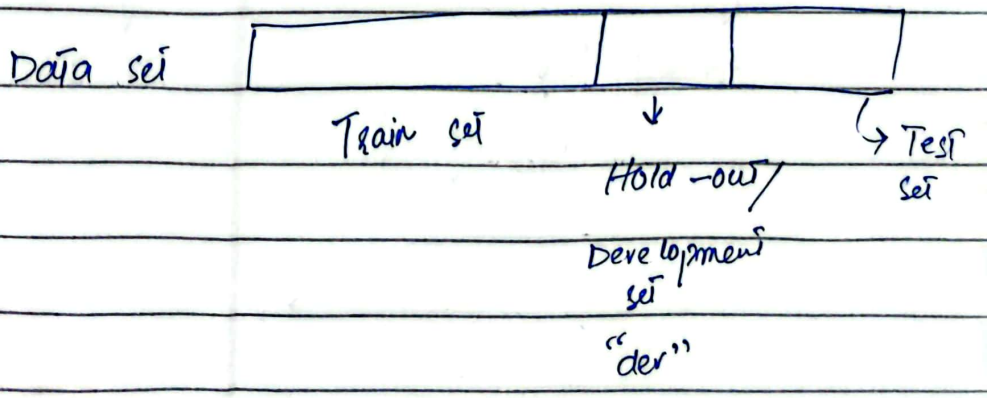
WEEK 1

- Setting up your Machine Learning Application:

1) Train/Dev/Test sets

Applied ML is a highly iterative process. We have to consider following

#layers? #hidden units?
#learning rates? #activation functions?



Dev set: It is usually for knowing which algorithm is best to use for these sets. Gradually these sets are decreasing.

Traditional ratio is 60/20/20
(for small dataset)

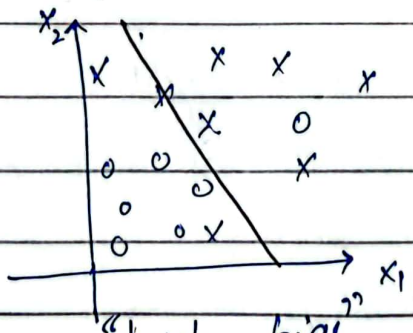
- Mismatched train/test distribution

Picture from web pages for train and camera for test set
So, we have to have same distribution.

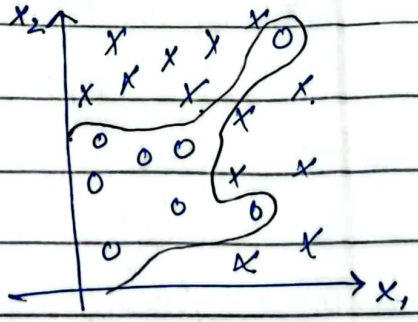
Note:

"Not having a test set might be okay."

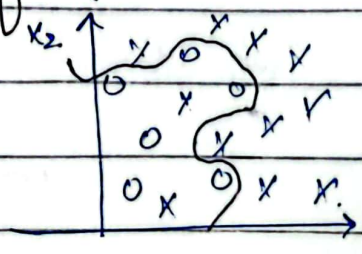
2) Bias / Variance:



"high bias"
underfitting



"high variance"
overfitting



"just right"

Cat Classification: $y=1$ $y=0$

Cat Dog

Train set error : 1%
Dev set error : 11%

You are doing very well on the training set and relatively poorly on development set. This looks like you might have overfit the training set. This has high variance.

Train set error:	15%	15%	0.5%
Dev set error:	16%	30%	1%
	high bias & high variance.	high bias & low variance	low bias & low variance

Because training error is so much then it means it is underfitting the data. So, it has high bias.

This all when bayes error or optimal error is $\approx 0\%$

3) Basic Recipe for Machine Learning:

high bias? (Training data performance) → ?
- Bigger Network: more hidden layers
- Train longer.
- NN architecture search

↓

high variance? (Dev set performance) →
- More data
- Regularization.

Bias Variance Tradeoff:

In the pre deep learning era, when bias increases variance goes down and when variance increases bias goes down.

But now we can focus on one without hurting other much. Regularization is a very useful technique for reducing variance. It might increase bias a little bit.

— Regularizing your Neural Network

1) Regularization:

Forbenius norm formula

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

Note:

The rows "i" of the matrix should be the number of neurons in the current layer $n^{[l]}$; whereas the columns "j" of the weight matrix should equal the number of neurons in the previous layer $n^{[l-1]}$

— To prevent overfitting.

Logistic Regression

Regularization parameter

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

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

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$ ← L2 Regularization

Because you are using Euclidean norm

Why do we just w not b?

Because w has a lot of parameters as n_x while b is only a single parameter

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

w will be sparse which means there will be a lot of zeros.

↓

L1 Regularization

Note: L1 Regularization can not be used for suppressing the model.

$\lambda \rightarrow$ is the hyper parameter which we may have to tune

Above will be L2 regularization for Logistic Regression and now for Neural Network:

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

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} \left(w_{ij}^{[l]}\right)^2$$

$w: (n^{[l]}, n^{[l-1]})$

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

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

Note:

Sometime L2 regularization is called as weight decay.

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

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

2) Why Regularization Reduces Overfitting?

3) Dropout Regularization:

Other technique from L2 regularization is dropout Regularization.

- We do dropout the nodes.

Implementing dropout ("Inversed dropout")

Illustrate with layer $l=3$

```
d3 = np.random.rand(a3.shape[0],
a3.shape[1]) < keep_prob
```

keep_prob = 0.8

It means there is 20% chance of eliminating hidden units.

a3 = np.multiply(a3, d3)

We will multiply both so that there will zeros so that matrix will be reduced.

a3 /= (0.8) or keep_prob

What is happening here.

Let's say we have 50 units then that means 20% will be zeroed out which is approximately that means.

$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$

↳ reduced by 20%

So we will divide it by 0.8 so that value of z can be remain same as by division, value will increase.

Making predictions at test time

No drop out.

④ Understanding Dropout:

Dropout prevents overfitting.

One big drawback that there will be no exact cost function.

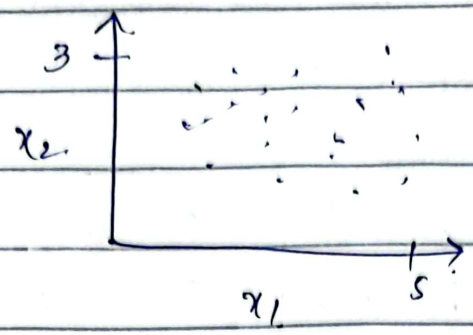
⑤ Other Regularization Methods.

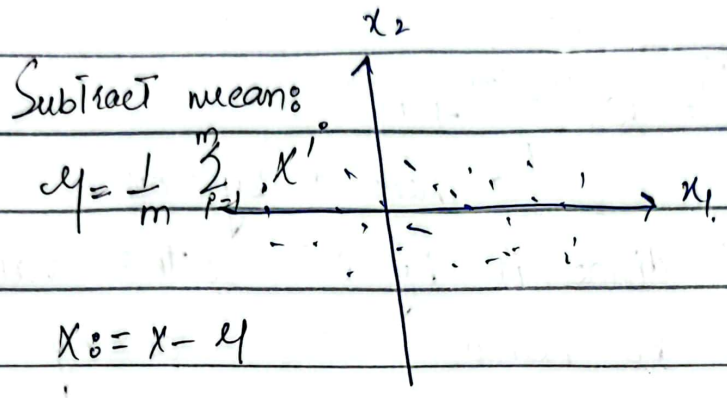
- Data augmentation.
- (make additional fake data)
- Early Stopping

It is beneficial because we don't have to use lambda values more then.

-Setting Up your Optimization Problem:

1) Normalizing Inputs





Subtract mean:

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

$$x := x - \mu$$

Normalize variance

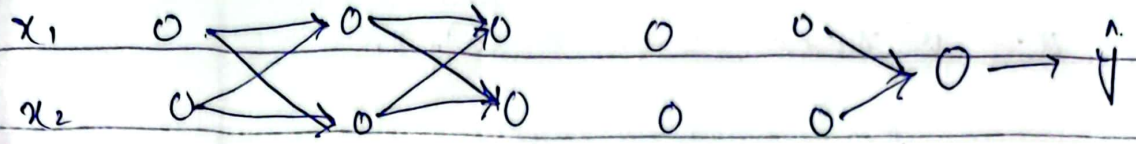
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$$

$$x \mathrel{/}= \sigma$$

Why normalize inputs?

② Exploding/Vanishing Gradients

Sometimes slope can be very high or very very small.



$w^{[1]}$ $w^{[2]}$ $w^{[3]}$ ... $w^{[l]}$

$g(z) = z$ $b^{[l]} = 0$

$y = w^{[l]} w^{[l-1]} \dots w^{[1]} x$

$z^1 = w^{[1]} x$

$a^1 = g(z^1) = z^1$

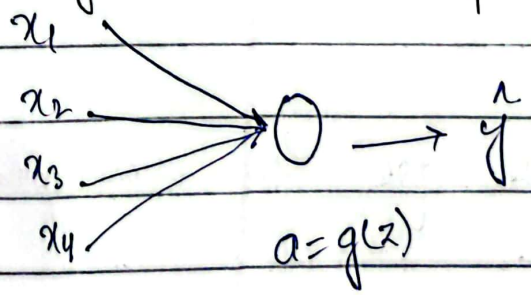
$a^2 = g(z^2) = g(w^2 a^1)$

$w^1 = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ $y = w^l \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{l-1}$

③ Weight Initialization for Deep Networks

The problem from Exploding/Vanishing gradient is that when there is large neural network then the weights can have very large slope or very small. Now the solution for that is by initializing weights

Single neuron example



$z = w_1 x_1 + w_2 x_2 + \cdots w_n x_n$

large $n \rightarrow$ small w_i

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

This is for Relu activation function

For tanh.

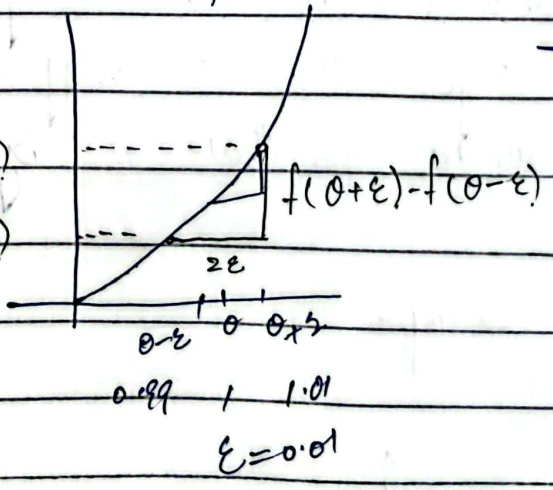
$$\tanh \sqrt{\frac{1}{n^{l-1}}}$$

For Xavier activation

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

④ Numerical Approximation of Gradients

$f(\theta) = \theta^3$



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

≈ 3.0001

$g(\theta) = 3\theta^2 = 3$

We do use this for less error. production

⑤ Gradient Checking:

help find bugs in the back prop.

Take $w^1, b^1, \dots w^l b^l$ and reshape into a big vector. θ

Same for $dw^1, db^1 \dots db^l$ and reshape into $d\theta$

Grad check

$$J(\theta) = J(\theta_1, \theta_2, \theta_3 \dots)$$

for each i

$$d\theta_{approx}^{[i]} = \frac{J(\theta_1 + \theta_2 \dots + \theta_i + \varepsilon) - J(\theta_1 + \theta_2 + \dots + \theta_i - \varepsilon)}{2\varepsilon}$$

$$\simeq d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

$$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \simeq 10^{-7}$$

Then that means the difference is correct or u can say there is small value and implementation is correct

⑥ Gradient Checking Implementation Notes.

- Don't use in training - only to debug
- If algorithm fails grad check

look at components to try to identify bug.

- Remember Regularization.
- Doesn't work with drop out.
  (For this we use keep-prob is 1.0).

- Optimization Algorithms

① Minibatch gradient Descent

$X = [x^1\ x^2\ x^3\ \dots\ x^m]$ $(n \times m)$

$Y = [y^1\ y^2\ \dots\ y^m]$ $(1 \times m)$

let's say we have $m = 500,00,000$

then we will do because it will take a lot of times.

To do well against this, we will divide the data in the mini-section of 1000.

$X = \{ \underbrace{x^1\ x^2\ \dots\ x^{1000}}_{X^{[1]}} \mid \underbrace{x^{1000}\ \dots\ x^{2000}}_{X^{[2]}} \mid \dots \mid \underbrace{x^m}_{X^{[5000]}} \}$

same for Y

for $t = 1 \dots 5000$ ← 1 step of gradient descent using X^t, Y^t

Forward prop on $X^{[t]}$ for 1000

$$Z^1 = W^{(1)} X^{[t]} + b^{[t]}$$
$$A^1 = g^{(1)} Z^{(1)}$$
$$\vdots$$
$$A^l = g^l Z^l$$

} vectorize implementation

Compute cost

$$J^{(t)} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^i, y^i) + \frac{\lambda}{2 \cdot 1000} \sum \|w^l\|_F^2$$

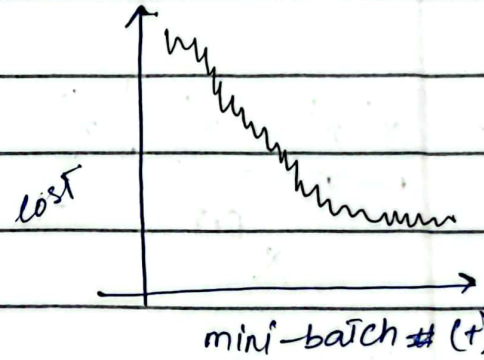
Backprop to compute gradient w.r.t. $J^{(t)}$ (using $X^{\{t\}}, Y^{\{t\}}$)

$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}, \quad b^{l} = b^{l} - \alpha\, db^{l}$$

(2) Understanding Mini-batch Gradient Descent:

Batch Gradient Descent

Mini-batch gradient Descent

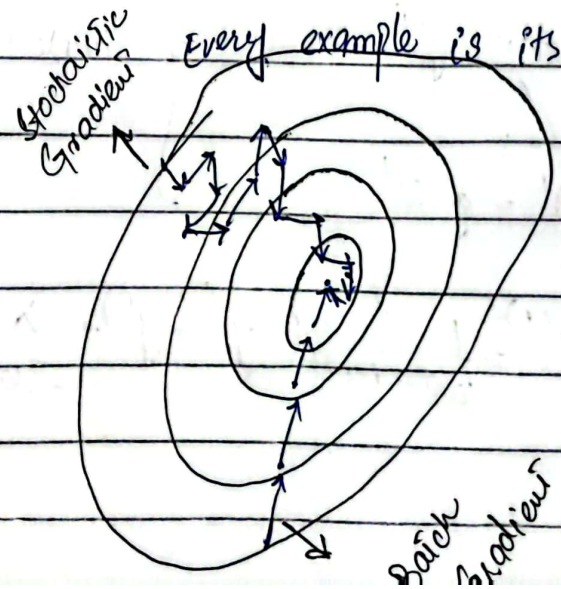


If mini-batch size=m : Batch gradient descent

$(X^{t}, Y^{t}) = (X, Y)$

" " " size=1 : Stochastic gradient descent

Every example is its own mini-batch.



In-between

Too long per iteration (Batch Gradient Descent)

Stochastic Gradient Descent (Loose speedup from vectorization)

↓

Fastest learning

- Vectorization
- make progress without needing to wait.

Choosing your mini-batch size:

- If small training set: use batch gradient descent
- Typical mini-batch sizes:

  64, 128, 256, 512.

  2^6, 2^7, 2^8, 2^9.

- Make sure mini-batch fit in CPU/GPU memory.

③ Exponentially Weighted Averages:

Because data is a bit noisy when plotted. So make it look cooler. we will do

$V_0 = 0$

$V_1 = 0.9V_0 + 0.1\theta_1$

$V_2 = 0.9V_1 + 0.1\theta_2$

⋮

$V_t = 0.9V_{t-1} + 0.1\theta_t$

So, this will be called as exponentially weighted average.

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

As the values of β increases, then averages also takes place which has slow effect

④ Understanding Exponentially weighted averages:

- Key component for several optimized algorithms that you used to train your neural networks.
- It takes very little memory.
- This is known as bias correction.

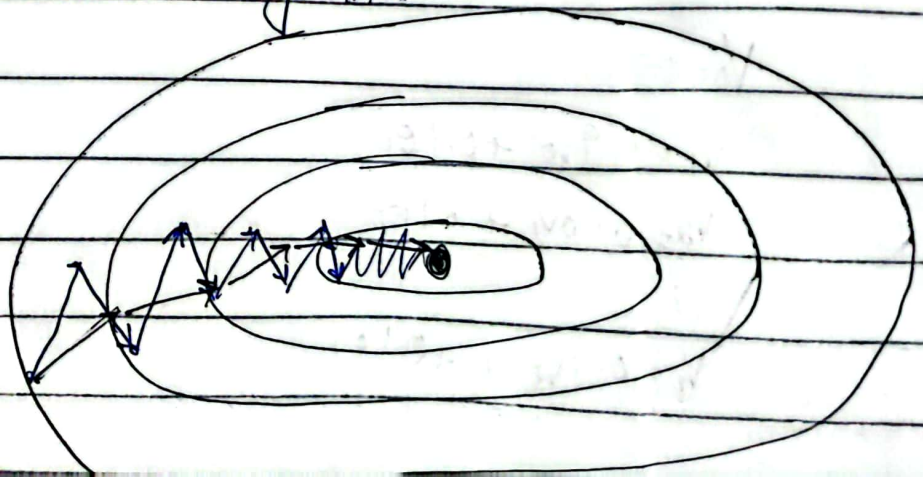
⑤ Bias Correction in exponentially Weighted Average:

When we initialize with zero then it starts with very low. So, to resolve that

$$\frac{V_t}{1-\beta^t}$$

⑥ Gradient Descent with Momentum:

- always work faster than gradient descent algorithm



This will become up and down oscillation with ~~slow~~ smaller learning rate.

We want faster learning, so with

Momentum:

On iteration t:

compute dw, db on current mini-batch

$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$

"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

$V_{db} = \beta V_{db} + (1-\beta)\, db$

$w = w - \alpha V_{dw}$, $b = b - \alpha V_{db}$

Hyper parameters : α, β $\beta = 0.9$

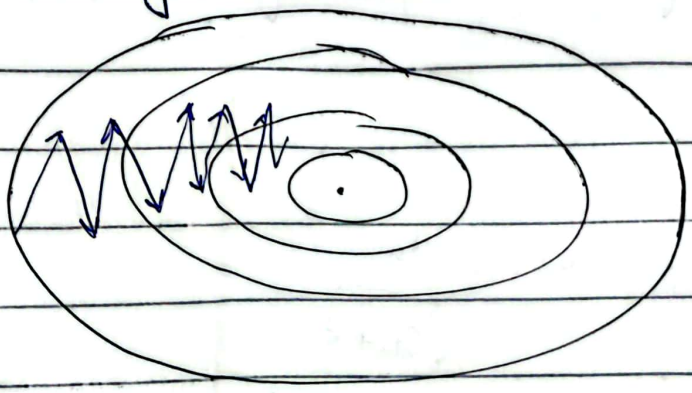
Sometimes, you will see

$V_{dw} = \beta V_{dw} + dw$

Because, it does not matter for small values —?

⑦ RMSprop

- algorithm



We want to speed up in horizontal with smaller vertical.

compute dw, db on current mini-batch

$$S_{dw} = \beta S_{dw} + (1-\beta) dw^2 \leftarrow \text{element wise}$$

↳ small

$$S_{db} = \beta S_{db} + (1-\beta) db^2$$

↳ large

$$w := \alpha \frac{dw}{\sqrt{S_{dw}}} \qquad b := \alpha \frac{db}{\sqrt{S_{db}}}$$

Because S_{dw} is smaller so. by dividing it will smaller then w will be larger.

and same for b but its value will be smaller so give us fast. Now you can use larger α rate.

RMSprop → Root Mean Square prop

we are going to use β_2 for RMSprop

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$$

⑧ Adam Optimizat Algorithm

$V_{dw}=0$, $S_{dw}=0$, $V_{db}=0$, $S_{db}=0$

On iteration t:

Compute dw, db using current batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw \quad , V_{db} = \beta_1 V_{db} + (1-\beta_1) db$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2 \quad , S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

After Bias correction

$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t) \quad , \quad V_{db}^{corr} = V_{db} / (1-\beta_1^t)$$

$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t) \quad , \quad S_{db}^{corr} = S_{db} / (1-\beta_2^t)$$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corr}} + \varepsilon}$$

$$b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corr}} + \varepsilon}$$

α: needs to tune

β_1: 0.9 (dw)

β_2: 0.9999 (dw^2)

ε: 10^{-8} (By Adam's Paper)

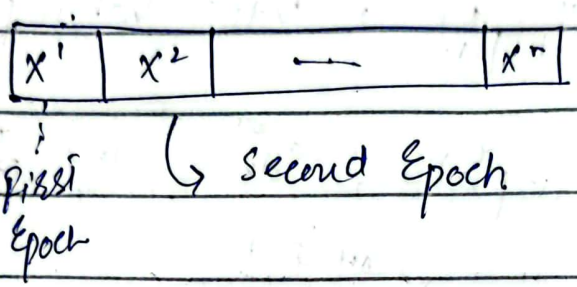
Adam: Adaptive moment estimation

⑨ Learning Rate Decay:

$$\alpha = \frac{1}{1 + \text{decay Rate} \times \text{epoch Number}} \alpha_0$$

We need to converge because if α is larger then it will be wandering, but if it is smaller then it will converge.

1 epoch = 1 pass through data



$\alpha_0 = 0.2$
decay rate = 1

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04

Other learning rate decay methods

$\alpha = 0.95^{\text{epoch num}} \times \alpha_0$ → exponential decay.

(10) The Problem of Local Optima.

Week 3

Hyperparameter Tuning, Batch Normalization And Programming Frameworks

- Hyperparameter Tuning

① Tuning Process

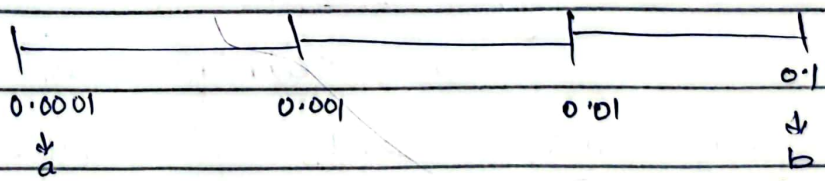
By default

$\beta \approx 0.9$

β_1	β_2	ε
0.4	0.999	10^{-8}

② Using Appropriate Scale to pick Hyperparameters.

We use random value from sampel for hyperparameter. So for uniform samplig



$r = -4 * np.random.rand()$ ← $r \in [-4, 0]$

$\alpha = 10^r$ $\quad$ $r \in [a, b]$

$\beta = 0.9 \ldots\ldots 0.999$

↓ 10 values $\quad$ ↓ 1000 values

$1-\beta = 0.1 \dots 0.001$

$\downarrow \quad \downarrow$

$10^{-1} \quad 10^{-3}$

$r \in [-3, -1]$

$\beta = 1 - 10^{r}$

③ Hyperparameters Tuning in Parractice: Pandas vs Caviar

- Retest your hyper-parameters once in a several months.
- Babysitting one model (One approach) → (Panda's Straiegy)
- Training many models in parralel (2nd approach) → (Caviar straiegy)

These 2 approaches depends on compuiaiin, power, we do have →

④ Normalizing Activations in a Network:

This technique can make very easier search for hyper-parameier but not for all neural network

$X = X/3$

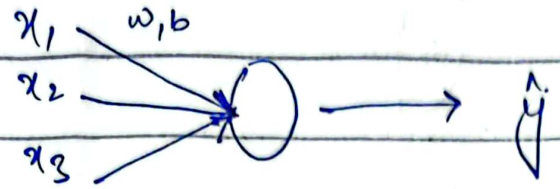
→ Normalizing inputs to speed up learning.

$\mu = \frac{1}{m} \sum x^{i}$

$X = X - \mu$

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

$$X = X/\sigma$$



When we have a deeper network Batch norm will normalize $z^{[2]}$ for $w^{[3]}$ $b^{[3]}$ or $a^{[3]}$.

Implementing Batch Norm

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

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

$$z^i_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

- Batch Normalization

$$\hat{z}^{(i)} = \gamma z^i_{norm} + \beta$$

↳ learnable parameters of model

If

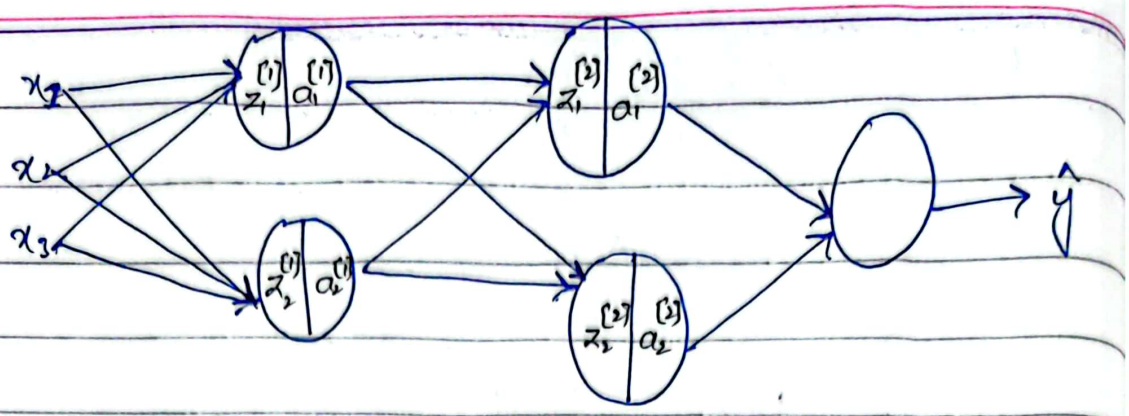
$$\gamma = \sqrt{\sigma^2 + \varepsilon}, \quad \beta = \mu$$

then

$$\hat{z}^i = z^{(i)}$$

How does it work from x's?

Fitting Batch Norm into a Neural Network!



$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm}]{\beta^{[1]}, \delta^{[1]}} \hat{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\hat{z}^{[1]})$

$\xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \delta^{[2]}} \hat{z}^{[2]} \rightarrow a^{[2]} = g^{[2]}(\hat{z}^{[2]})$

Parameters: $w^{[1]}, b^{[1]}, \ldots w^{[l]} b^{[l]}$ $\Big\}\ d\beta^{[l]}$

$\beta^{[1]}, \delta^{[1]}, \ldots \beta^{[l]} \delta^{[l]}$ $\Big|\ \beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$

↳ These are different from Adam's hyper parameters.

For programming we can use

tf.nn.batch-normalization

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

Batch Norm will remove $b^{[l]}$ by subtracting mean and dividing variance.

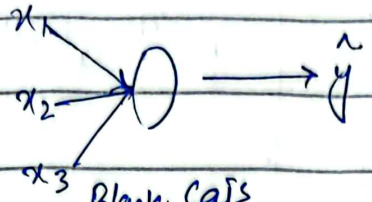
$$z^{[l]} = w^{[l]} a^{[l-1]}$$

$$\hat{z}^{[l]} = \delta^{[l]} z_{norm}^{[l]} + \beta^{[l]}$$

$z^{[l]}_{(n^{[l]}, 1)} \rightarrow n^{[l]}$ no. of hidden units.

$g^{[l]} = (n^{[l]}, 1)$ → Dimension.

③ Why does Batch Norm work?



Black cats

Cat
$y=1$

Non-cat
$y=0$

Colored cats

Cat
$y=1$

Not-cat
$y=0$

If the distribution of x changes then you have to retrain the model.

"Co-variate shift"

$X \rightarrow Y$

Suppose complex value for 3rd hidden layer

$w^{[3]}, b^{[3]}$ want to go to $\hat{y}$

Batch norm will reduce the variate distribution for the specific layer. It ensures the mean and variance. for the layers remains same.

④ Batch Norm at Test time:

Batch Norm process mini-batches and one mini-batch at a time while at test time it has to predict / process one example / emplete dataset at a time.

This is the equation at training time.

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

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

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

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

At test time, we examp is one

μ, σ^2: estimate using exponentially weighted average (across mini-batch)

$x^{\{1\}}, x^{\{2\}}, x^{\{3\}}, \ldots$

$\downarrow$

$\mu^{[1]\{1\}}, \mu^{[2]\{1\}} \rightarrow \mu$

σ^2

We will use the average across ~~th~~ 1s.

So at test time

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \hat{z} = \gamma z_{norm} + \beta$$

But average σ^2, μ

- Multiclass Classification

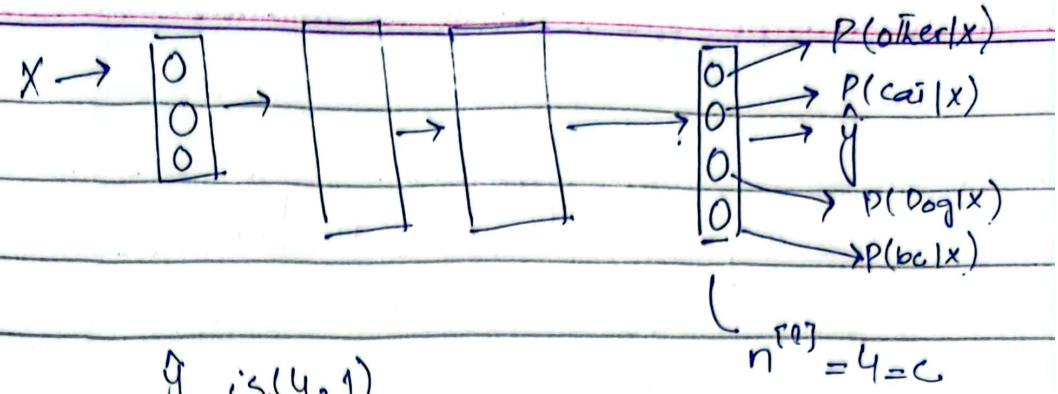
① Softmax Regression:

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i}$$

• Generalization of Logistic Regression

When there is more than two classes then.

C = # classes.



ŷ is (4, 1)

ŷ will give us the probabilities whose sum will be equal to 1.

We use softmax layer for the final layer to output these probabilities.

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

Activation function:

$$t = e^{(z^{[l]})} \quad (4,1)$$

Element wise exponentiation

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{i=1}^{4} t_i} \quad (4,1)$$

$$a_i^{[l]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

e.g.

$$z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

$$\sum_{i=1}^{4} t_i = 176.3$$

$\frac{e^5}{176.3}$ = P(otter|x) = 0.842

$\frac{e^2}{176.3}$ = P(cat|x) = 0.042

= 0.002

= 0.114

...

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

② Training a Softmax Classifier:

$C = 4$

$$z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

Softmax ↑ Hardmax

$$a^{[l]} = g^{[l]}(z^{[l]}) = \begin{bmatrix} e^5/e^5+e^2+e^{-1}+e^3 \\ = \\ = \\ = \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Softmax regression generalizes Logistic regression to C classes.

Eg. $C = 2$, softmax becomes logistic regression.

Loss function

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \text{ cat} \quad (y_2 = 1, \; y_1 = y_3 = y_4 = 0) \qquad a^{[l]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1} y_j \log \hat{y}_j$$

$$-y_2 \log \hat{y}_2 = -\log \hat{y}_2$$

make $\hat{y}_2$ large to make loss small

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

$Y = [y^1, y^2 \dots y^m]$ $\quad \hat{Y} = [\hat{y}^1, \hat{y}^2 \dots \hat{y}^m]$

$= \begin{bmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$ $\quad \hat{Y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \dots$

$(4, m)$

Gradient descent with softmax.

Back prop:

$$dz^{[l]} = \hat{y} - y$$

$\hookrightarrow \frac{\partial J}{\partial z^{[l]}}$

— Introduction to Programming FrameWorks

① Deep Learning Frameworks:

→ Caffe/ Caffe2
→ CNTK
→ DL4J
→ Keras
→ Lasagne
→ mxnet
→ PaddlePaddle
→ TensorFlow
→ Theano
→ Torch

② TensorFlow:

$J(w) = w^2 - 10w + 25$

cost

we have this function and how we will define it Tensorflow to minimize

To startup Tensorflow

import numpy as np

import tensorflow as tf

① - w = tf.Variable(0, dtype = tf.float32)

② - optimizer = tf.keras.optimizers.Adam(0.1)

Define train_step():

④ ← with tf.GradientTape() as tape:

To compute back prop ③ - cost = w**2 - 10*w + 25

⑥ - trainable_variables = [w]

⑦ - grads = tape.gradient(cost, trainable_variables)

⑧ - optimizer.apply_gradients(zip(grads, trainable_variables))

⑨ - print(w)

- w is a parameter we want to know so we declare it as a variable.

When there is a dataset then cost function

①, x = np.array([1.0, -10.0, 25.0], dtype = np.float32)

②, def training(x, w, optimizer):

def cost_fn():

return x[0]*w**2 + x[1]*w + x[2]

for i in range(1000):

optimizer.minimize(cost_fn, [w])

return w

w = training(x, w, optimizer)