Week 3 : Hyperparameter Tuning, Batch Normalization & ~~Multiclass~~ Programming Frameworks

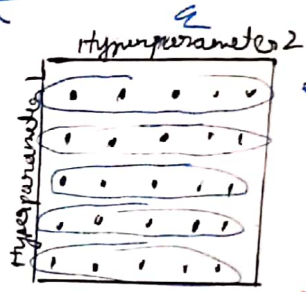
Hyperparameter Tuning

Tuning Process:

- We need to choose α, β, $(\beta_1, \beta_2, \varepsilon)$ - adam, # layers, # hidden units, learning rate decay, mini-batch size

● Most important
● second important
● Third important

- People used to use a grid, but this is inefficient



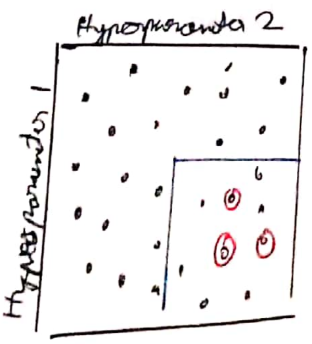
← almost same values

matters / doesn't matter

If hyperparameter 1 is α and hyperparameter 2 is ε we would try 25 different ~~values of~~ examples, out of which only 5 values of α (even though this is more important). Hence every row will have almost same answer since ε doesn't matter much

Instead using random values is better since that way we ~~choose~~ try 25 values of α.

- Another method is coarse to fine,

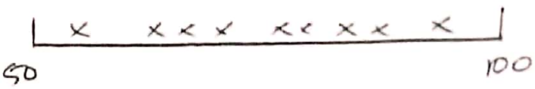


If we are getting good results in a particular area, we can focus our search there.

Using an appropriate scale to pick hyperparameter:

- Suppose we want to pick values for

$$n^{[l]} = 50, \dots, 100 \quad - \text{ between 50 and 100}$$



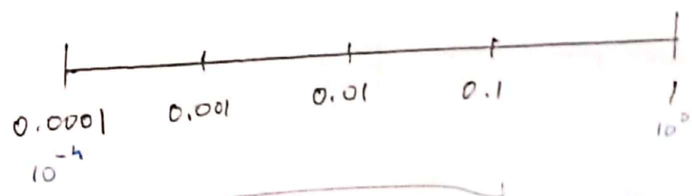
Then we can randomly pick between the range 50 to 100

- However consider finding values for an exponential hyperparameter like α:

$$\alpha = 0.0001, \dots, 1$$

If we use the previous method, then 90% of the values will be between 0.1 - 1

$\therefore$ We use log



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

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

- For exponentially weighted averages:

$$\beta = 0.9 \dots 0.9999$$

So we take $1-\beta = 0.1 \dots 0.0001$

$$\therefore r \in [-4, -1]$$

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

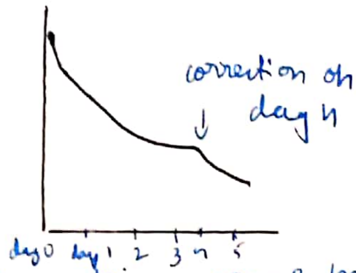
$$\boxed{\beta = 1 - 10^{r}}$$

Hyperparameters tuning in practice:

Pandas VS Caviar

Panda

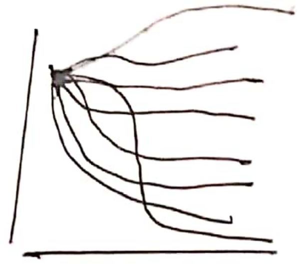
- Pandas ~~only~~ have one (or two) babies and give them full attention



- Babysitting one model (setting a hyperparameter value. If it doesn't work change during training)
- Used when dataset is very large

Caviar

- Caviar lay 1000s of eggs and ~~let~~ see which survive

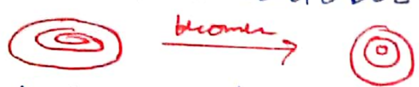


- Training many models in parallel
- Used when good CPU/GPU is available for multiprocessing

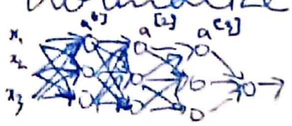
Batch Normalization

Normalizing activations in a network

- We previously saw that normalising the input ~~values~~ feature x helps gradient descent learn faster since the contour becomes more uniform →



- However in a NN, we have other layers, and we can normalize them too!



- We normalise Z and not a

Let the intermediate values in NN be,

$z^{(1)}, \ldots, z^{(m)}$

↖ $z^{[l](i)}$

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

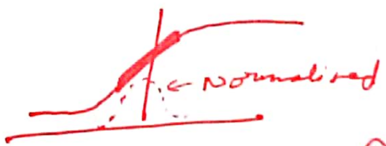
2. $\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$

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

4. $\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$ ← we pass this to activation function

Here γ and β are learnable parameters

Suppose we want to use sigmoid function that can't use normalized values



In such a case ~~we use $\tilde{z}^{[l]}$ instead of $z^{[l](i)}$~~

← it learns this

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

Then $\tilde{z}^{(i)} = \dfrac{z^{(i)} - \cancel{\mu}}{\cancel{\sqrt{\sigma^2 + \varepsilon}}} \times (\cancel{\sqrt{\sigma^2 + \varepsilon}}) + \cancel{\mu}$

$\therefore \tilde{z}^{(i)} = z^{(i)}$

Fitting Batch Norm into a NN

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

Hence in the middle, before passing z through the activation function, normalise it.

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots, w^{[L]}, b^{[L]}$

$\underset{(n^{[l]},1)}{\beta^{[1]}}, \underset{(n^{[l]},1)}{\gamma^{[1]}}, \beta^{[2]}, \gamma^{[2]}, \ldots, \beta^{[L]}, \gamma^{[L]}$

Usually all this is done by inbuilt code in tensorflow.

• working with mini-batches

$$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow g^{[1]}(\tilde{z}^{[1]}) = a^{[1]} \rightarrow \ldots$$

$$X^{\{2\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow \ldots$$

here we compute over the batch size and not over m

....

• b doesn't matter, you can set it to 0.
Since $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ this is automatically removed when we ~~mean~~ zero out the mean $z^{(i)} - \mu$

Implementing gradient descent:

for t = 1 .... num_mini-batches
  compute forward prop on $X^{\{t\}}$
    In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
  Use backprop to compute $dw^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$
  Update parameters
    $w^{[l]} := w^{[l]} - \alpha dw^{[l]}$
    $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$
    $\gamma^{[l]}$ - - - .

(works with momentum, RMSprop or Adam).

Why does Batch Norm work?

1. Like how normalising the input speeds up learning, normalising hidden features also speeds up learning

2. Covariate shift:
Lets say we train on black cat images but test on other colour cats, then it'll not recognise.

even though function is same it won't work because the distribution of x has changed.
But if we use Batch Norm, this doesn't happen because it standardizes the data at every layer (the mean and variance remain same)

Batch Norm also has a regularization effect (slight):

- Happens since each mini-batch is scaled by the mean/variance computed on just that mini batch.
- This adds some noise to $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- Don't use this for regularization - its effect is very slight. As the mini-batch size increases, this effect decreases.

Batch Norm at Test Time:

During testing we process an individual training example, and so we don't have a batch size. As a result, how do we find μ and σ^2?

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

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

What we do is, during training we use exponentially weighted average to get a μ and σ^2 till ~~a~~ how much ever examples have been trained

$$X^{\{1\}}, \quad X^{\{2\}}, \quad X^{\{3\}} \ldots$$

$$\uparrow \qquad \uparrow \qquad \uparrow$$

$$\mu^{\{1\}[l]} \quad \mu^{\{2\}[l]} \quad \mu^{\{3\}[l]}$$

This is similar to $\theta_1, \theta_2, \theta_3 \ldots$

1. $\mu_1 = \beta \mu_0 + (1-\beta)\theta_1$
2. $\mu_2 = \beta \mu_1 + (1-\beta)\theta_2 \quad \ldots$

If we ~~test~~ test now we take μ_2. So we keep a track of μ and σ^2 while training.

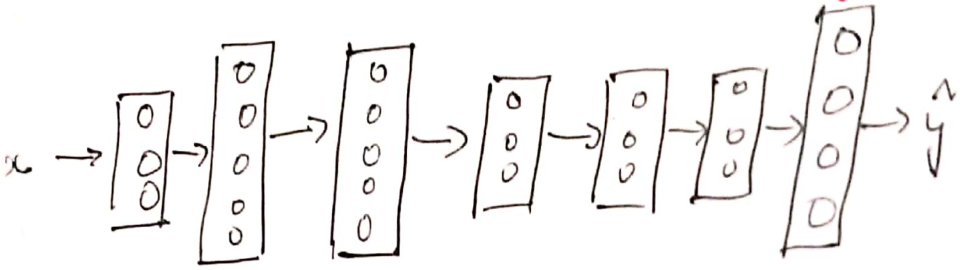
Multiclass classification - softmax

- Before we were doing binary classification (either 0 or 1), now we will use the softmax function for multiclass classification

no. of classes (Exa: cat, dog, baby chick, horse)
(cat 1, dog 2, baby chick 3, horse 4)

- For example, let $C = 4$

Layer L has 4 output units



$\rightarrow z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$

$\rightarrow$ Activation function

$\rightarrow t = e^{(z^{[L]})}$ ← element wise power

$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$ if $z = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$

$\rightarrow a^{[L]} = \dfrac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i}$ $\therefore a_i^{[L]} = \dfrac{t_i}{\sum_{i=1}^{4} t_i}$

Here $a^{[L]}$ will be (4, 1) dimension unlike ReLU or sigmoid where it is (1, 1)

Exa: $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$, $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$, $\sum_{i=1}^{4} t_i = 176.3$ → adding all up

$\therefore a^{[L]} = \dfrac{t}{176.3}$

$a^{[L]} = \begin{bmatrix} \frac{e^5}{176.3} \\ \frac{e^2}{176.2} \\ \frac{e^{-1}}{176.2} \\ \frac{e^3}{176.2} \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$
← 84.2%
← 4.2%
← 0.2%
← 11.4%

Training a softmax classifier:

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

Hardmax - it takes the highest element and makes it one while others as 0. $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

So softmax is a 'softer' version that maps to probabilities $\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$

-> Softmax regression generalises logistic regression to C classes.
i.e if $C=2$, softmax reduces to logistic regression $\left(a^{[L]} = \begin{bmatrix} 0.842 \leftarrow \text{logistic regression output} \\ \cancel{0.158} \leftarrow \text{ignore this} \end{bmatrix} \right.$

Loss function:

Suppose $y^{(1)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \leftarrow \text{cat} \qquad a^{[L](1)} = \hat{y}^{(1)} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ ← $\hat{y}_2$ needs to be big

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{4} y_i \log \hat{y}_i$$

← here since $y_1 = y_3 = y_4 = 0$, we get $-y_2 \log \hat{y}_2$

$\mathcal{L}(\hat{y}, y) = -\log \hat{y}_2$ since $y_2 = 1$

Hence if loss should be small $\hat{y}_2$ needs to be big

∴ Since $\hat{y}_2$ will be learnt to be large, it'll predict cat properly as it trains

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

$$= \begin{bmatrix} 0 & 1 & 0 & \\ 1 & 0 & 0 & \\ 0 & 0 & 0 & \cdots \\ 0 & 0 & 1 & \end{bmatrix} \quad (4,m)$$

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

$$= \begin{bmatrix} 0.3 & & \\ 0.2 & & \\ 0.1 & \cdots & \\ 0.4 & & \end{bmatrix} \quad (4,m)$$

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

Backprop,

$dz^{[L]} = \hat{y} - y$ $\therefore dz^{[L]}$ will be (4, 1)

But we only need to do forward prop properly, the framework (tenserflow) will figure out backprop.

Introduction to Programming Frameworks

Deep Learning Frameworks:

Rather than building everything from scratch, it is more practical and efficient to use libraries:

- Caffe / Caffe 2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- Paddle Paddle
- TenserFlow
- Theano
- Torch

Choosing a deep learning frameworks:

→ Ease of programming (development & deployment)

→ Running speed

→ Truly open (open source with good governance)

Tensorflow

Program to find w for $\underbrace{J(w)}_{\text{cost}} = w^2 - 10w + 25$

(Answer should be 5):

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-10], [25]])
w = tf.Variable([0], dtype = tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
for i in range(1000):
    session.run(train, feed_dict = {x: coefficients})
print(session.run(w))
```