

How to improve the model

More data may be required

Data needs to have more diversity

Algorithm needs longer training

More hidden layers or hidden units are required

Add Regularization

Change the Neural network architecture like activation function etc.

There are many other considerations you can think of..

Distribution of data for improving the accuracy of the model

Training set

Which you run your learning algorithm on.

Dev (development) set

Which you use to tune parameters, select features, and make other decisions regarding the learning algorithm. Sometimes also called the **hold-out cross validation set** .

Test set

which you use to evaluate the performance of the algorithm, but not to make any decisions regarding what learning algorithm or parameters to use.

Partition your data in different categories

Training Set	Dev Test or Hold Out Cross Validation Set	Test Set
--------------	---	----------

Traditional Style partitioning 70/30 or 60/20/20

But in the era of Deep Learning may even go down to 99 0.5 0.5

If the data size is 1,00,0000 then 5000 5000 data size will still be there in dev and test sets

Importance of Choosing dev and test sets wisely

The purpose of the dev and test sets are to direct your team toward the most important changes to make to the machine learning system

Very important that dev and test set reflect data you expect to get in the future and want to do well on.

Bad distribution will severely restrict analysis to guess that why test data is not giving good results

Importance of Dev Set

- If your team improves the classifier's accuracy from 95.0% to 95.1%, you might not be able to detect that 0.1% improvement from playing with the app.
- Having a dev set and metric allows you to very quickly detect which ideas are successfully giving you small (or large) improvements, and therefore lets you quickly decide what ideas to keep refining, and which ones to discard.

When Metric Evaluation may fail

- Suppose that for your cat application, your metric is classification accuracy. This metric currently ranks classifier A as superior to classifier B.
- But suppose you try out both algorithms, and find classifier A is allowing occasional pornographic images to slip through.
- Even though classifier A is more accurate, the bad impression left by the occasional pornographic image means its performance is unacceptable.
- Change the metric to heavily penalize letting through pornographic images.

Data Distribution Mismatch

It is naturally good to have the data in all the sets from the same distribution.

For example Housing data coming from Mumbai and we are trying to find the house prices in Chandigarh.

Else wasting a lot of time in improving the performance of dev set and then finding out that it is not working well for the test set.

Sometime we have only two partitioning of the data in that case they are called Train/dev or train/test set.

Using a single Evaluation Metric

You should be clear about what you are trying to achieve and what you are trying to tune

Classifier	Precision	Recall
A	95	90
B	98	85

Precision – of examples recognized as true how many are true

Recall – of total true examples how many have been correctly extracted

Remember: We are calculating these figures from the dev set

Using a single Evaluation Metric

You should be clear about what you are trying to achieve and what you are trying to tune

Classifier	Precision	Recall	F1 Score
A	95	90	92.4
B	98	85	91

$$\text{F1 Score} = \text{Harmonic mean of Precision and Recall} \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

Optimize one parameter and satisfy others

Classifier	Accuracy	Running Time	Safety	False Positive
A	90	20ms	No			
B	92	80ms	Yes			
C	96	2000ms	Yes			

Maximize ????? Subject to ????? And ????? And...

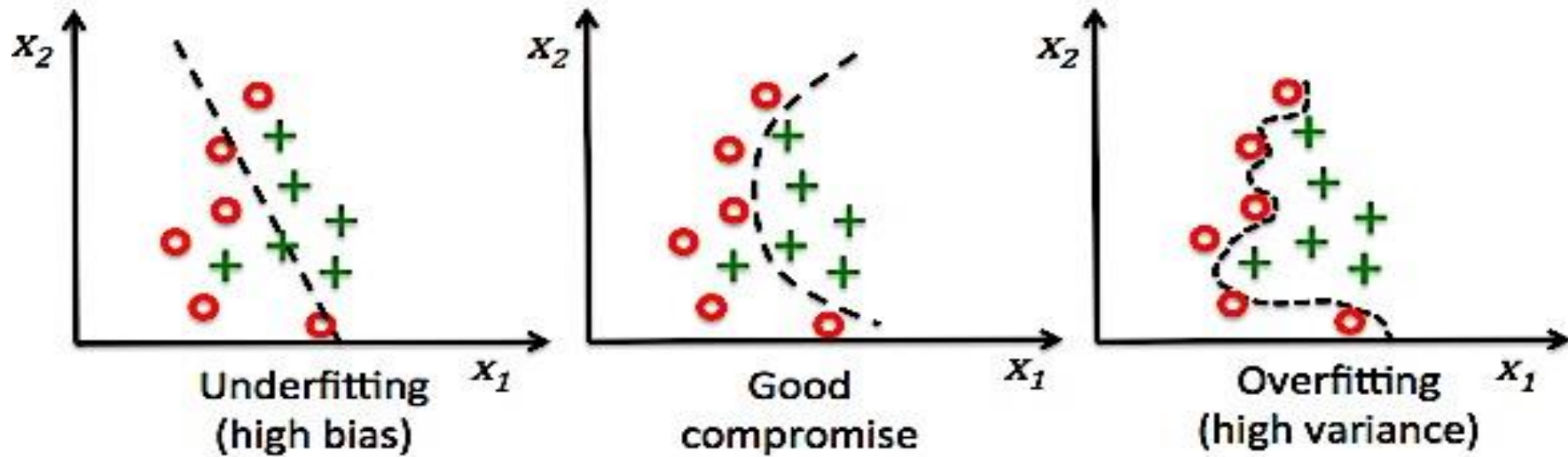
So few of them can be satisficing metric.

e.g if we say that running time needs to be minimum 100ms that running time is satisficing metric and accuracy can be optimizing metric

False Positive and False Negative

- Amazon Echo listening for “Alexa”; Apple Siri listening for “Hey Siri”; Android listening for “Okay Google”.
- False positive rate—the frequency with which the system wakes up even when no one said the wakeword—as well as the false negative rate—how often it fails to wake up when someone says the wakeword.
- One goal is to minimize the false negative rate (optimizing metric), subject to there being no more than one false positive every 24 hours of operation (satisficing metric).

Bias/Variance



Bias/Variance

The algorithm's error rate on the training set is algorithm's **bias** .

How much worse the algorithm does on the dev (or test) set than the training set is algorithm's **variance** .

Bias/Variance

Train Set Error	1	12	8	1
Dev Set Error	9	13	16	1.5
	High Variance	High Bias	High Bias, High Variance	Low Bias Low Variance
	Overfitting	Underfitting	Underfitting	Good fit

Multi dimensional system can have high bias in some areas and high variance in some other areas of the system, resulting in High Bias and High Variance issue

High Bias

Increase the model size (such as number of neurons/layers)

It allows to fit the training set better. If you find that this increases variance, then use regularization, which will usually eliminate the increase in variance.

Modify input features based on insights from error analysis

Create additional features that help the algorithm eliminate a particular category of errors. These new features could help with both bias and variance.

Reduce or eliminate regularization (L2, L1 regularization, dropout)

reduces avoidable bias, but increase variance.

Modify model architecture (such as neural network architecture) so that it is more suitable for your problem

This can affect both bias and variance.

High Variance

Add more training data

Simplest and most reliable way to address variance, so long as you have access to significantly more data and enough computational power to process the data.

Add regularization (L2, L1 regularization, dropout)

This technique reduces variance but increases bias.

Add early stopping (stop gradient descent early, based on dev set error)

Reduces variance but increases bias.

Modify model architecture (such as neural network architecture) so that it is more suitable for your problem

This affects both bias and variance.

Often compare with human level performance

Image
recognition,
spam
classification.

Ease of obtaining data from human labelers

Error analysis can draw on human intuition.

Use human-level performance to estimate the optimal error rate and also set a “desired error rate

Tasks Where we don't compare with human level performance

Picking a book to recommend to you;

It is harder to obtain labels

picking an ad to show a user on a website;

Human intuition is harder to count on

predicting stock market.

It is hard to know what the optimal error rate and reasonable desired error rate is

Classification example for animals

Type	Scenario 1	Scenario 2	
Humans (Bayes error)	1	7.5	
Training error	8	8	
Dev error	10	10	
	Focus on Bias	Focus on Variance	
Avoidable Bias	7	0.5	

New scenario

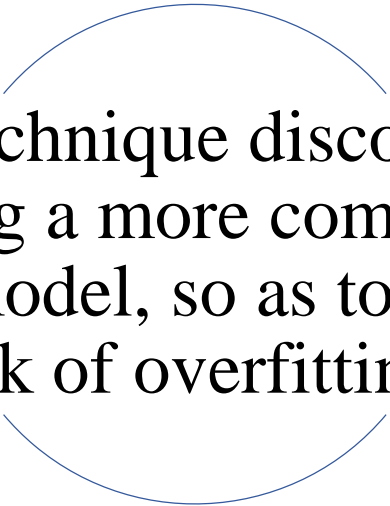
Type	Scenario 1	Scenerio2	
Human (Bayes) error	1	1	1
	0.7	0.7	0.7
	0.5	0.5	0.5
Training error	5	1	0.7
Dev error	6	5	0.8
	Bias issue	Variance Issue	Difficult

Two fundamental
Assumptions:

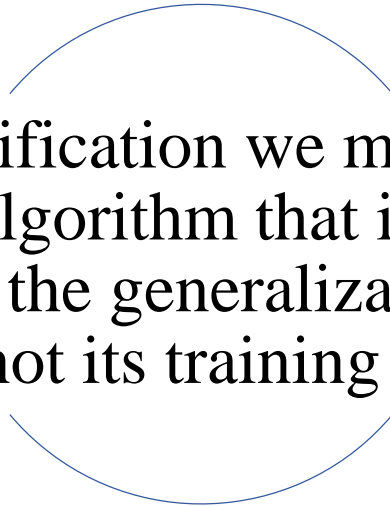
You can fit the training set well

Training set performance should Generalize to dev/test set.

Regularization



This technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting.



Any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error

Regularization for a Neural Network

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

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

Frobenius Norm

Weight decay

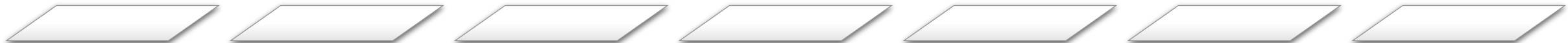
$$w^{[l]} = \left(1 - \frac{\alpha \lambda}{m}\right) w^{[l]} - \alpha dw^{[l]}$$

L2 Regularization. Intuition

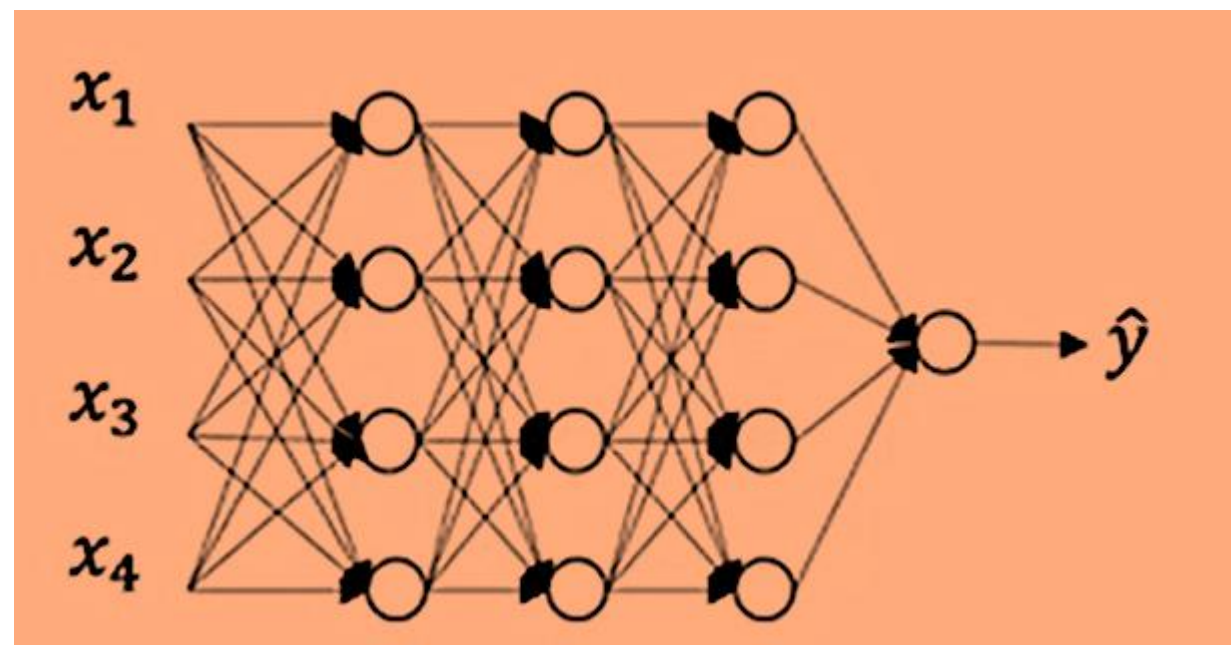
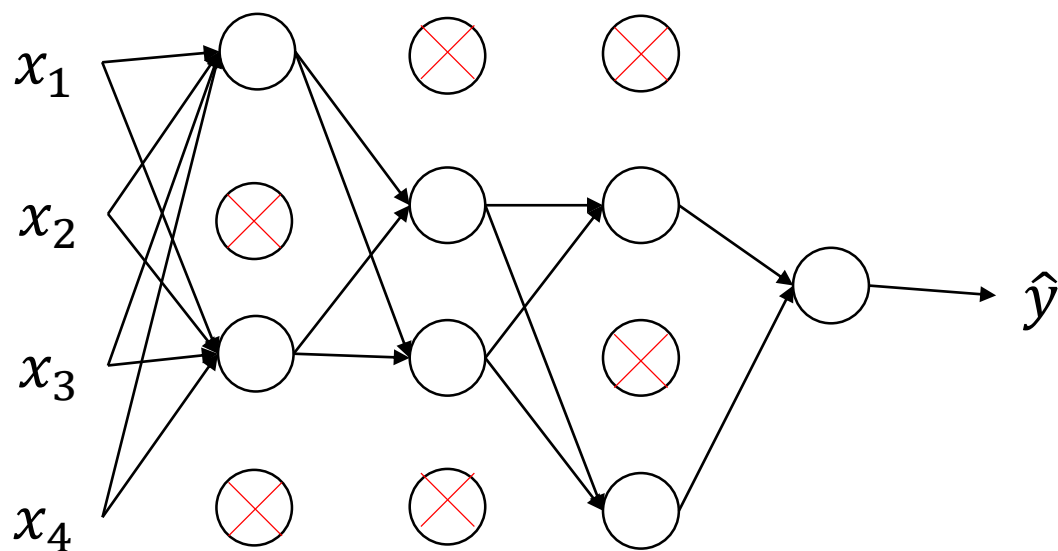
Mostly we use L2 Regularization instead of L1. It has Direct effect on the weights as we end up in the reduction of weights by a factor of $(1 - \alpha \lambda / m)$



This is also referred to as weight decay. The larger values of lambda may lead to weights going close to zero and that may then go towards underfitting. The value of λ needs to be balanced. Similarly the small values of λ may not have any significant impact on the weights. λ , becomes another hyperparameter to handle.



Dropout regularization



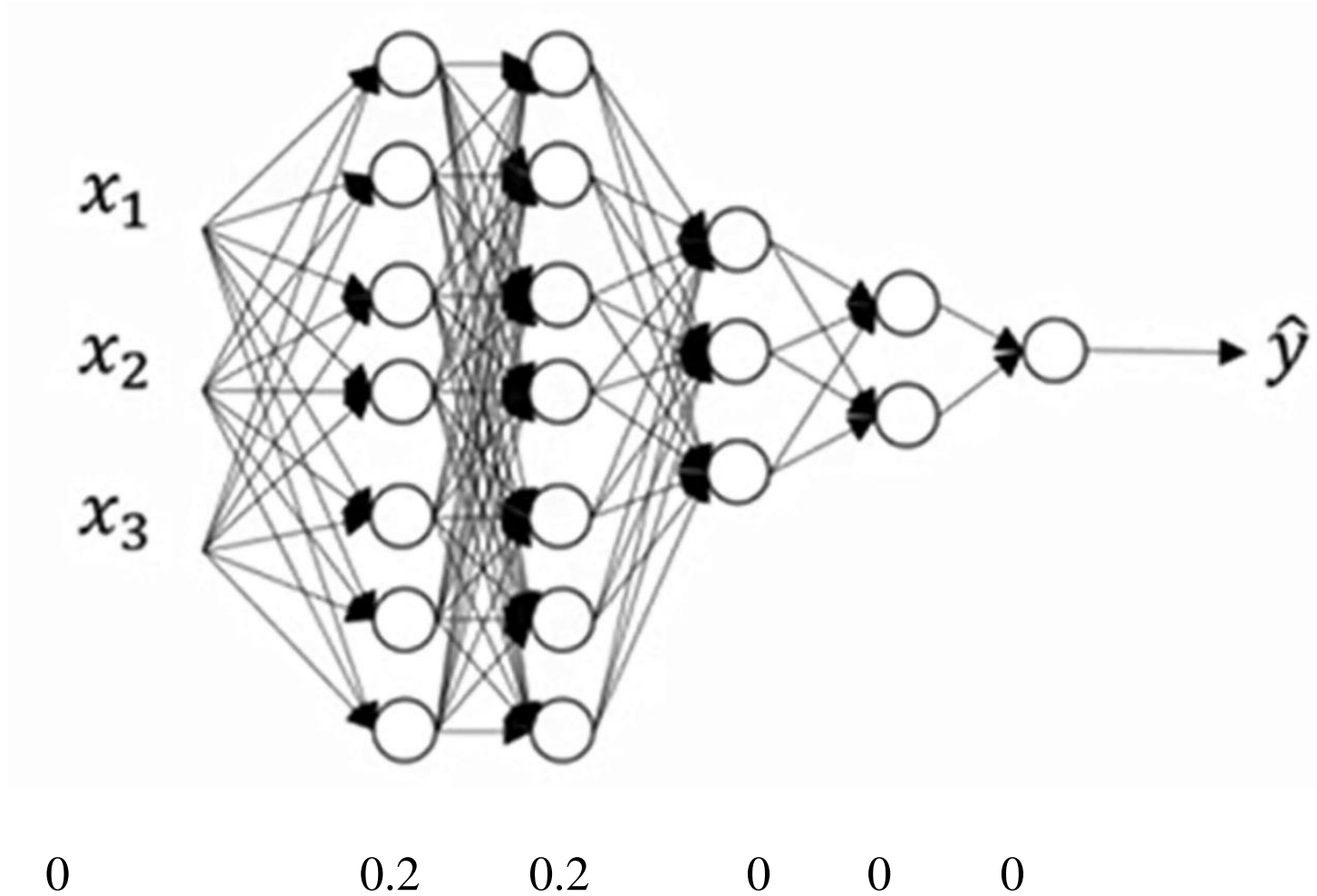
Drop out regularization: Prevents Overfitting

This technique has also become popular recently. We drop out some of the hidden units for specific training examples. Different hidden units may go off for different examples. In different iterations of the optimization the different units may be dropped randomly.

The drop outs can also be different for different layers. So, we can select specific layers which have higher number of units and may be contributing more towards overfitting; thus suitable for higher dropout rates.

For some of the layers drop-out can be 0, that means no dropout

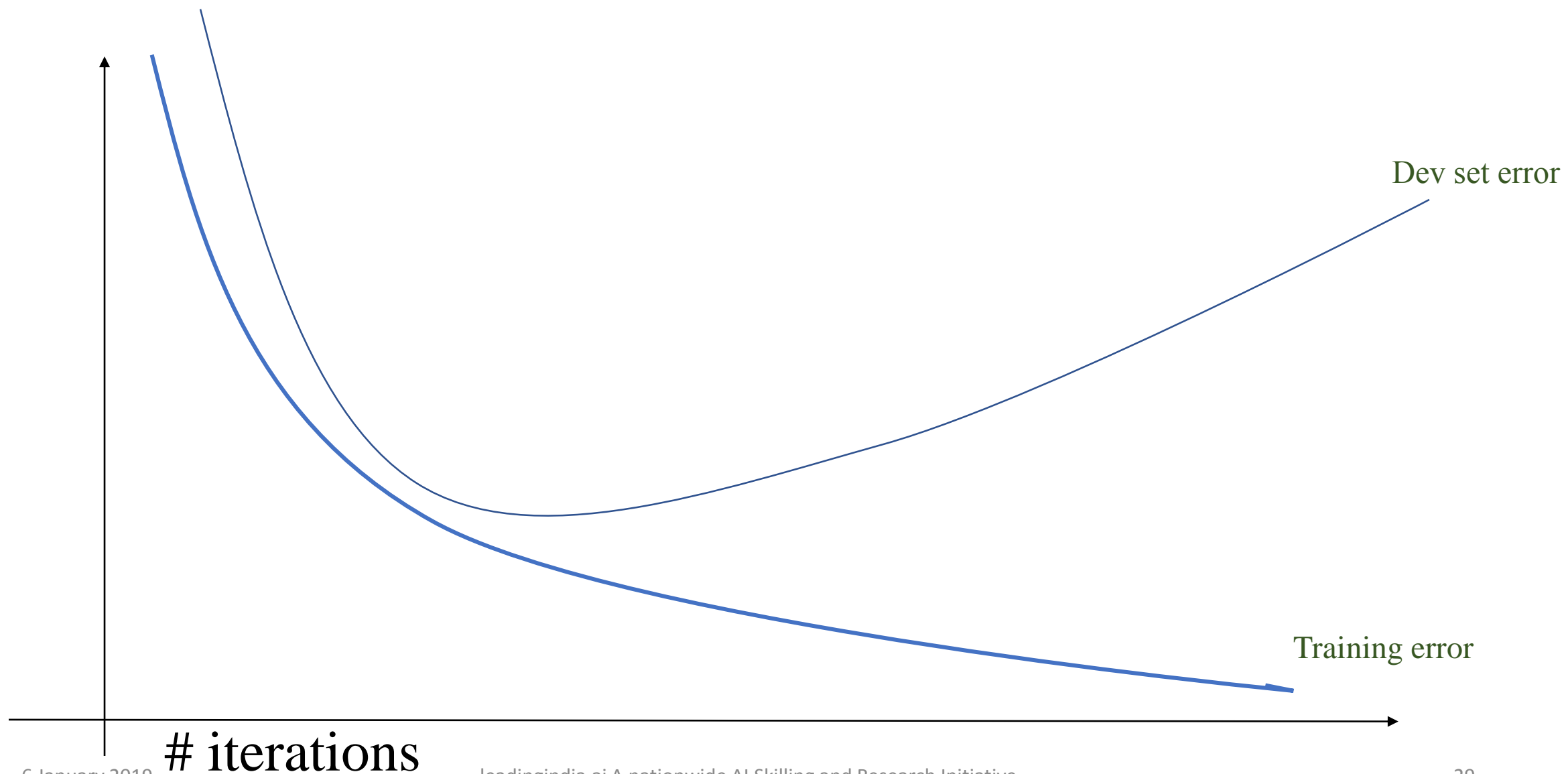
Layer wise drop out



Drop out

- Drop out also help in spreading out the weights at all layers as the system will be reluctant to put more weight on some specific node. So it help in shrinking weights and has an adaptive effect on the weights.
- Dropout has a similar effect as L2 regularization for overfitting.
- We don't use dropout for test examples
- We also need to bump up the values at the output of each layer corresponding to the dropout

Early stopping



Early Stopping

Sometime dev set error goes down and then it start going up. So you may decide to stop where the curve has started taking a different turn.

By stopping halfway we also reduce number of iterations to train and the computation time.

Early stopping does not go fine with orthogonalization because it contradicts with our original objective of optimizing (w, b) to the minimum possible cost function.

We are stopping the process of optimization in between to take care of the overfitting which is a different objective then optimization.

Data Augmentation

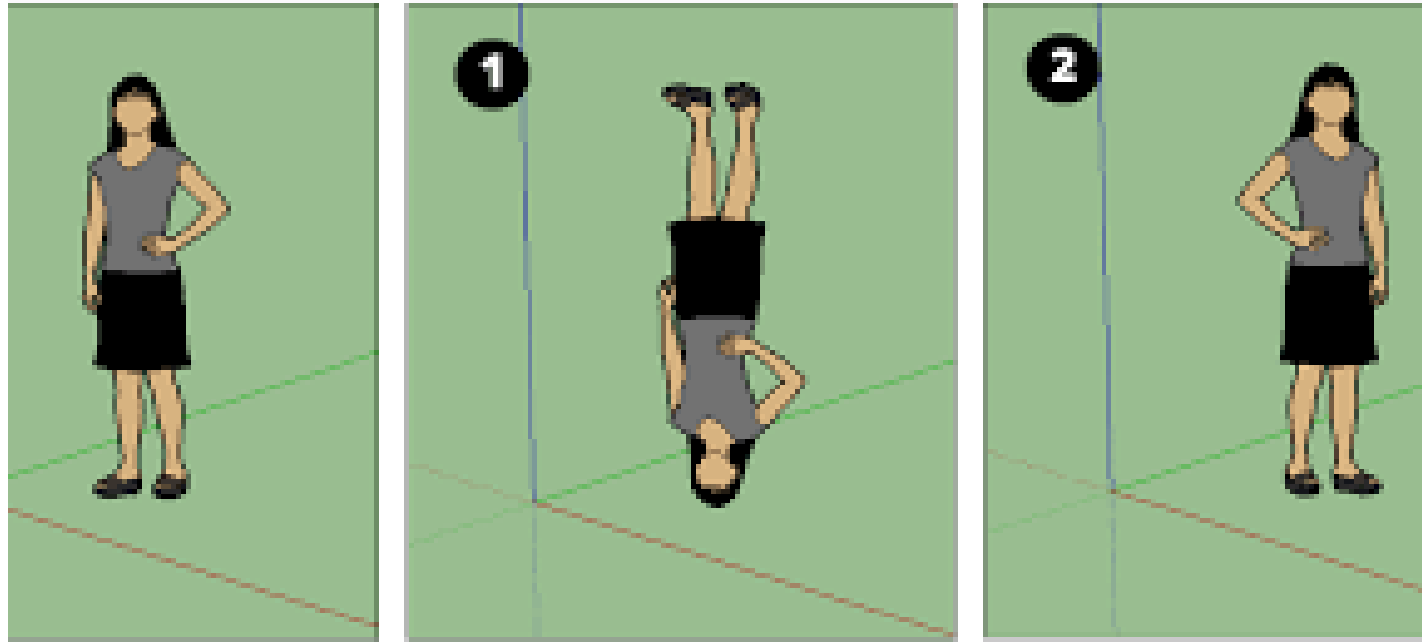
More training data is one more solution for overfitting.

As getting additional data may be expensive and may not be possible

Flipping of all the images can be one of the ways to increase your data.

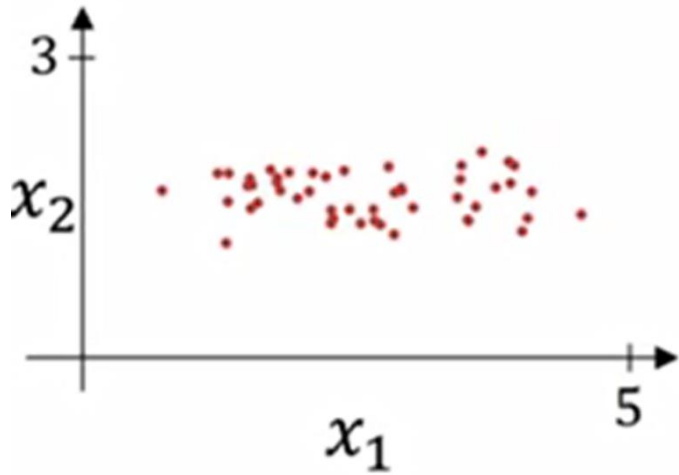
Randomly zooming in and zooming out can be another way

Distorting some of the images based on your application may be another way to increase your data.

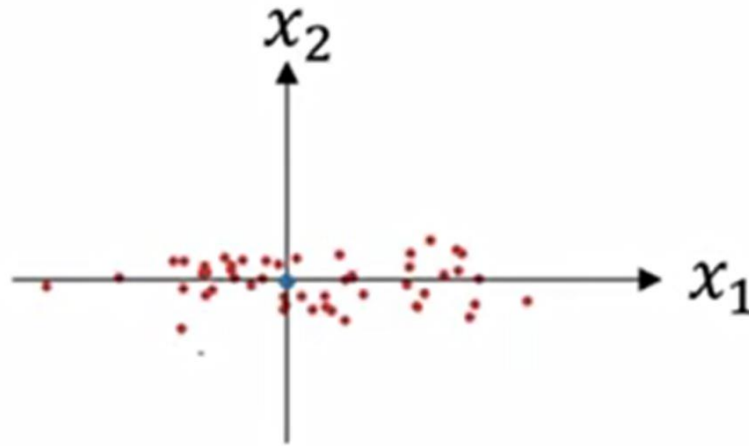


Data Augmentation

Normalizing Data Sets

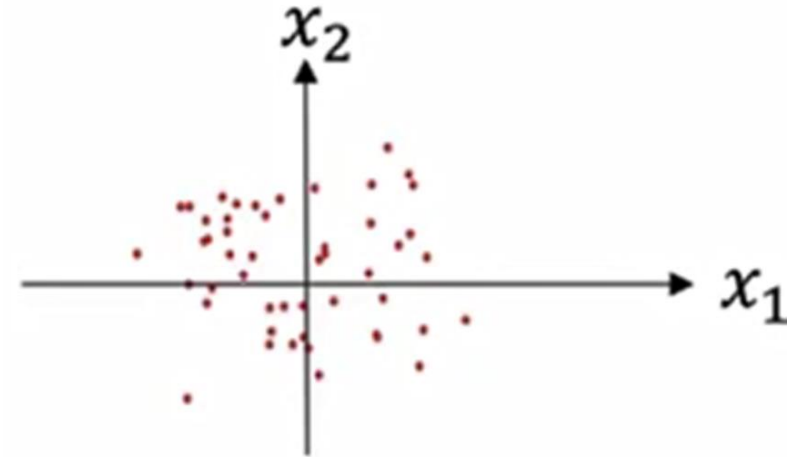


Original data



Subtract mean

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



Normalize Variance

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

Speed up the training/ Why normalization

Use same normalizer in the test set also, exactly in the same way as training set

If the features are on different scale 1, 1000 and 0,1 weights will end up taking very different values

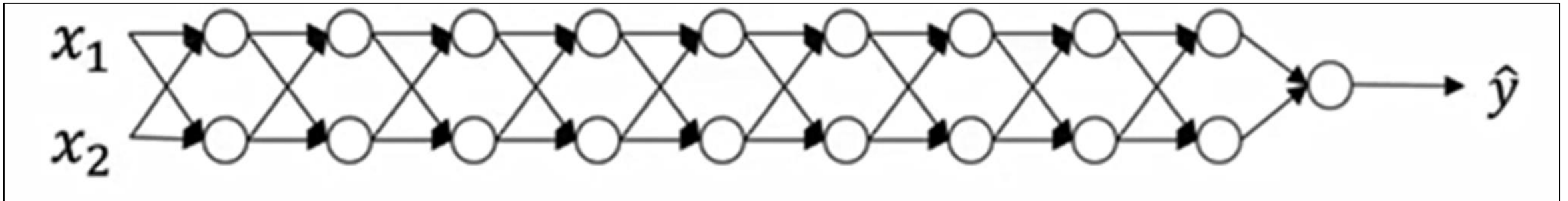
More steps may be needed to reach the optimal value and the learning can be slow.

Shape of the Normalized bowl will be more spherical and symmetrical making it easier to faster to optimize

Vanishing/exploding gradients

$g(z) = z$ # A linear function $b^{[l]}=0$

$$\hat{y} = w^{[l]}w^{[l-1]}w^{[l-2]} \dots w^{[3]}w^{[2]}w^{[1]} x$$



1.5	0
0	1.5

.5	0
0	.5

The matrix will be multiplied by 1-1 (as $w^{[l]}$ will be different dimension) leading to exploding and vanishing gradients

Exploding/vanishing gradients

Gradients/slope becoming too small or too large

So it is very important to see that how we initialize our weights

If the value of features are large then weights need to be very small

It has been proposed to have the variance between the weights to be $2/n$

Batch vs. mini-batch gradient descent

$$\begin{array}{c} X = [x^{[1]} \ x^{[2]} \ x^{[3]} \ \dots \ x^{[1000]} \mid x^{[1001]} \ \dots \ x^{[2000]} \mid \dots \mid \dots \ x^{[m]}] \\ \underbrace{\hspace{10em}} \quad \underbrace{\hspace{10em}} \quad \underbrace{\hspace{10em}} \\ (n_x, m) \qquad X^{\{1\}}(n_x, 1000) \quad X^{\{2\}}(n_x, 1000) \quad \dots \quad X^{\{5000\}}(n_x, 1000) \end{array}$$

M=5,000,000 5000 mini batches of 1000 each

$$\begin{array}{c} Y = [y^{[1]} \ y^{[2]} \ y^{[3]} \ \dots \ y^{[1000]} \mid y^{[1001]} \ \dots \ y^{[2000]} \mid \dots \mid \dots \ y^{[m]}] \\ \underbrace{\hspace{10em}} \quad \underbrace{\hspace{10em}} \quad \underbrace{\hspace{10em}} \\ (1, m) \qquad Y^{\{1\}}(1, 1000) \quad Y^{\{2\}}(1, 1000) \quad \dots \quad Y^{\{5000\}}(1, 1000) \end{array}$$

Mini batch gradient Descent

Size of the mini batch needs to be chosen carefully

If mini batch size = m then it is same as batch gradient descent

If mini batch size = 1 then all individual training examples are mini batch in themselves. The training becomes very slow and we can not use vectorization.

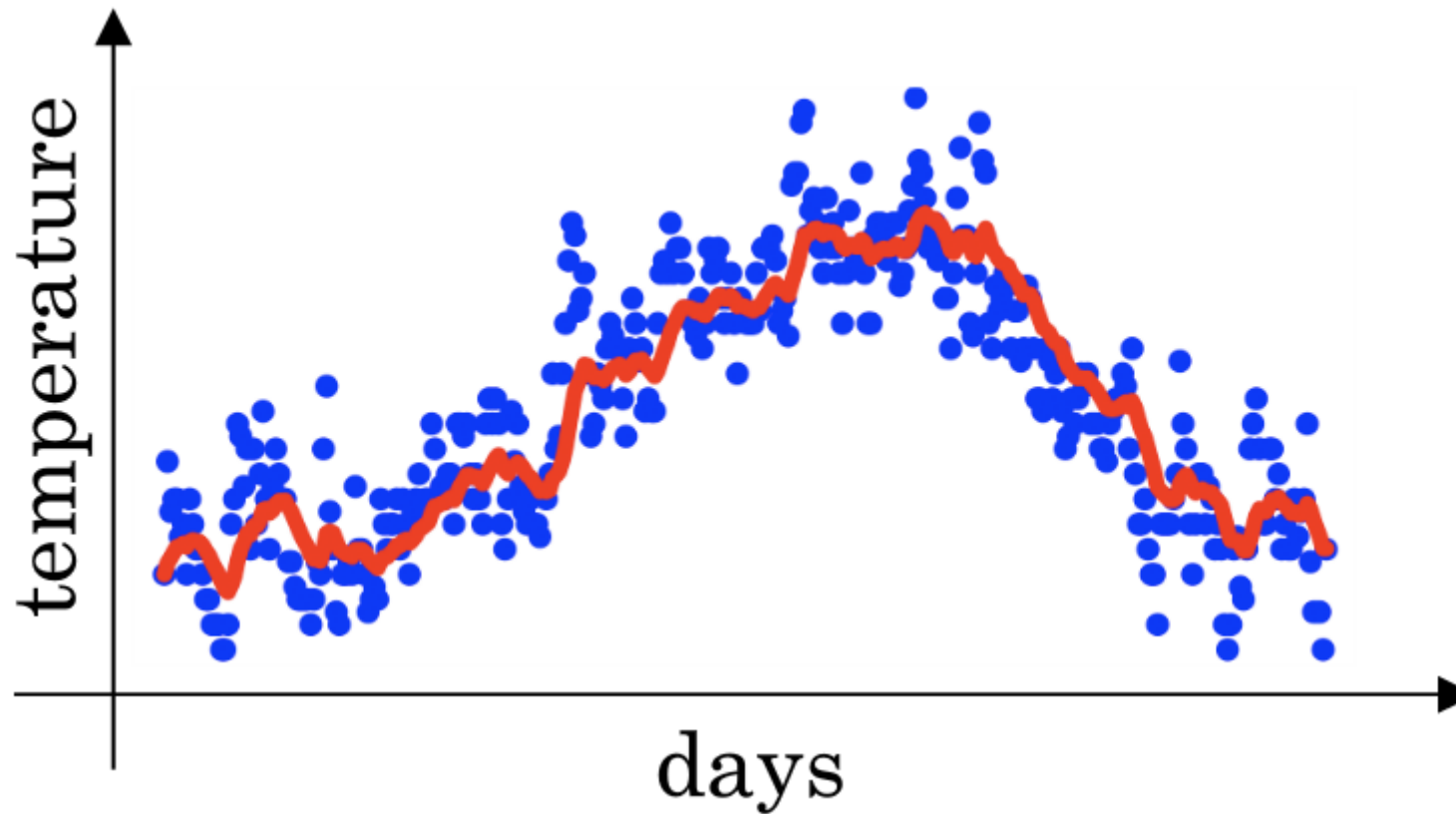
Mini batch GD helps us to make early progress without iterating through the entire data set and results in faster learning. It also takes advantage of vectorization.

So, the size need not be too big or too small. To take advantage of the CPU/GPU Memory the mini batch size may be in context of the available RAM.

Generally the minibatch size is taken as a power of 2. A mini batch of 512 or 1024 depending upon your application can be your starting point.

If the training data set is less than 5000 then there is no need of using mini-batches.

Exponentially weighted averages



Exponentially weighted averages

Let us understand with the example of every day Temperature for a one year period in Delhi

- $V_{100} = 0.9V_{99} + 0.1\theta_{100}$
- $V_{99} = 0.9V_{98} + 0.1\theta_{99}$
- $V_{98} = 0.9V_{97} + 0.1\theta_{98}$
- $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
- $\beta = 0.9$ V_t is approximately average over $1/(1-\beta)$ days eg $1/(1-0.9) = 10$ days

Exponentially weighted averages or moving averages

- $V_0 = 0$
- $V_1 = \beta V_0 + (1 - \beta)\theta_1$
- $V_2 = \beta V_1 + (1 - \beta)\theta_2$
-

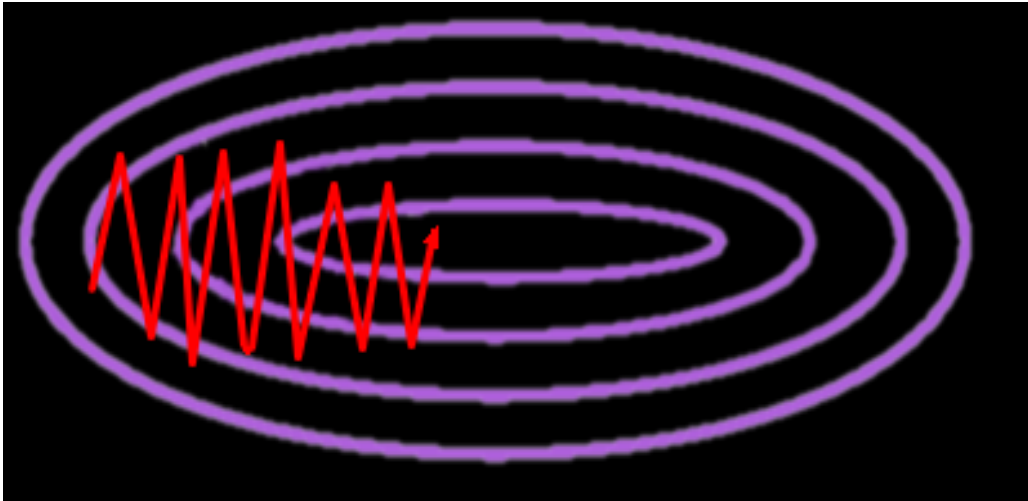
There is a problem of cold start-up in case the initial values are low. So we need to do some warming up.

To solve this we use Bias correction in Exponentially weighted averages $\frac{v_t}{1 - \beta^t}$

- $t=2, \frac{v_t}{1 - \beta^t} = \frac{\beta V_1 + (1 - \beta)\theta_2}{1 - \beta^t}$
- This is only used for initial few steps after that we don't have the issue of cold start up.

A moving average is commonly used with time series data to smooth out short-term fluctuations and highlight longer-term trends or cycles

Momentum



- On iteration t
- Compute dw , db on current mini-batch
- $V_{dw} = \beta V_{dw} + (1 - \beta) dw$
- $V_{db} = \beta V_{db} + (1 - \beta) db$
- $w = w - \alpha V_{dw}$
 $b = b - \alpha V_{db}$

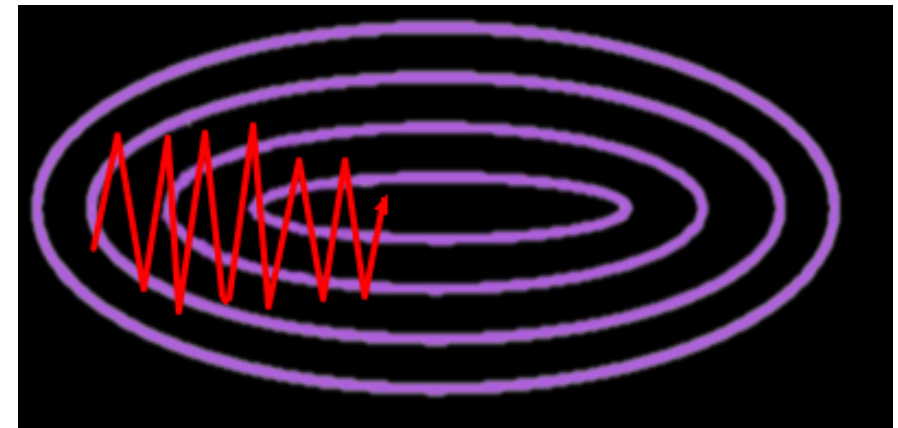
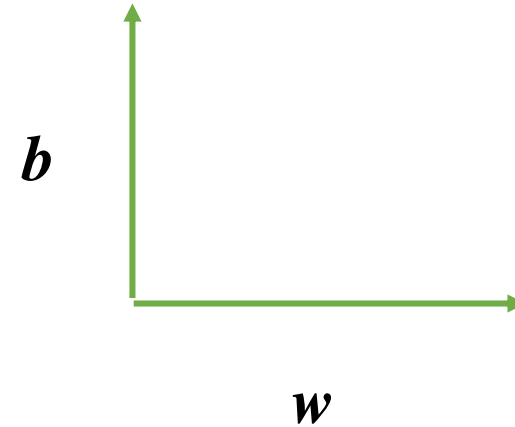
β is new hyperparameter and the recommended value is 0.9

Momentum

- It helps accelerate gradient vectors in the right directions, that helps in faster convergence.
- It will moderate the movements in the vertical direction and will help us move faster in the horizontal direction.
- Instead of calculating gradients independently we use exponentially weighted averages to calculate the gradient.

RMSProp (Root Mean Square Prop)

- On iteration t
- Compute dw , db on current mini-batch
- $S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) dw^2$
- $S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$
- $w = w - \alpha \frac{dw}{\sqrt{S_{dw}} + \epsilon}$
- $b = b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$
- $\epsilon = 10^{-8}$
- b is large and w is small



Adam optimization Algorithm

- On iteration t
- Compute dw, db on current mini-batch
- $V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dw$ $V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$
- $S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) dw^2$ $S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$

- $V_{dw}^{corrected} = \frac{v_{dw_t}}{1 - \beta_1}$ $V_{db}^{corrected} = \frac{v_{db_t}}{1 - \beta_1}$

- $S_{dw}^{corrected} = \frac{s_{dw_t}}{1 - \beta_2}$ $S_{db}^{corrected} = \frac{s_{db_t}}{1 - \beta_2}$

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

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

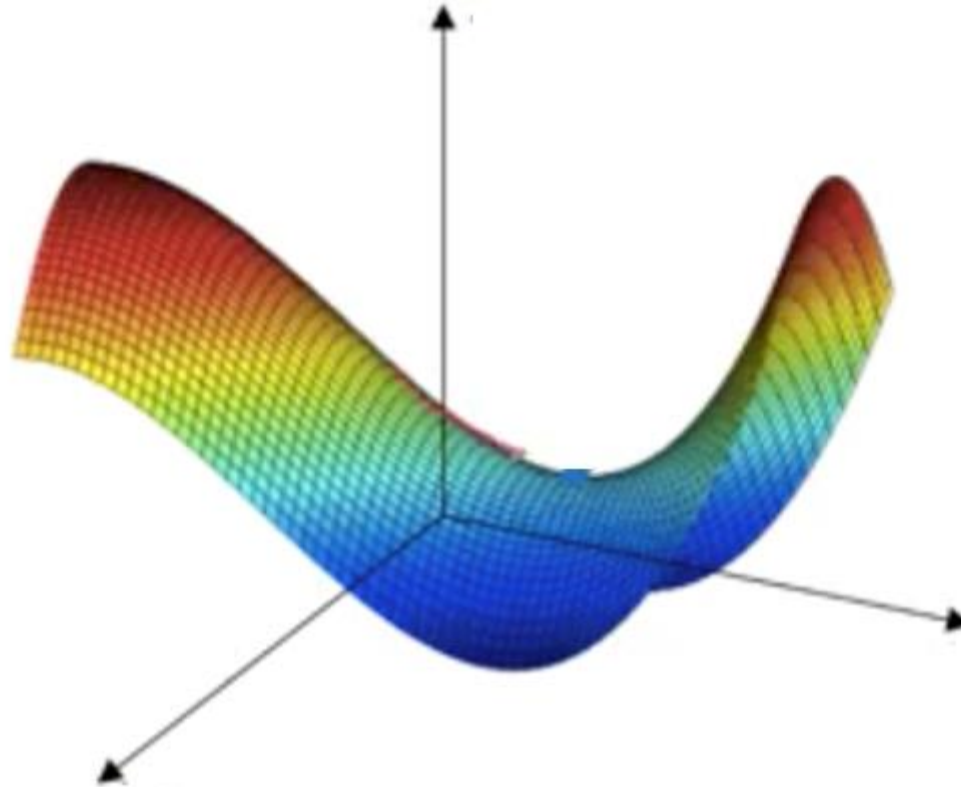
$\beta_1 = 0.9$ (dw) also referred to as moment 1
 $\beta_2 = 0.999$ (dw^2) also referred to as moment 2
 $\epsilon = 10^{-8}$

Adaptive Moment estimation

Adam Optimization Algorithm

- It is a combination of Momentum and RMS Prop Algorithm.
- It has proved good across different Neural network architectures to improve the learning speed
- It can handle sparse gradients in noisy problem
- Hyper parameters β_1 , β_2 and ϵ require little tuning and have intuitive interpretation

Problem of plateaus and saddle points



Plateaus can make learning slow
Unlikely to stuck in local minima

Learning rate decay

- 1 epoch = 1 pass through entire data

- $\alpha = \frac{1}{1 + \text{Decay rate} * \text{epoch_number}} \alpha_0$

- $\alpha_0 = 0.2$

- $\alpha = \frac{1}{1 + 1 * 1} 0.2 = 0.1$

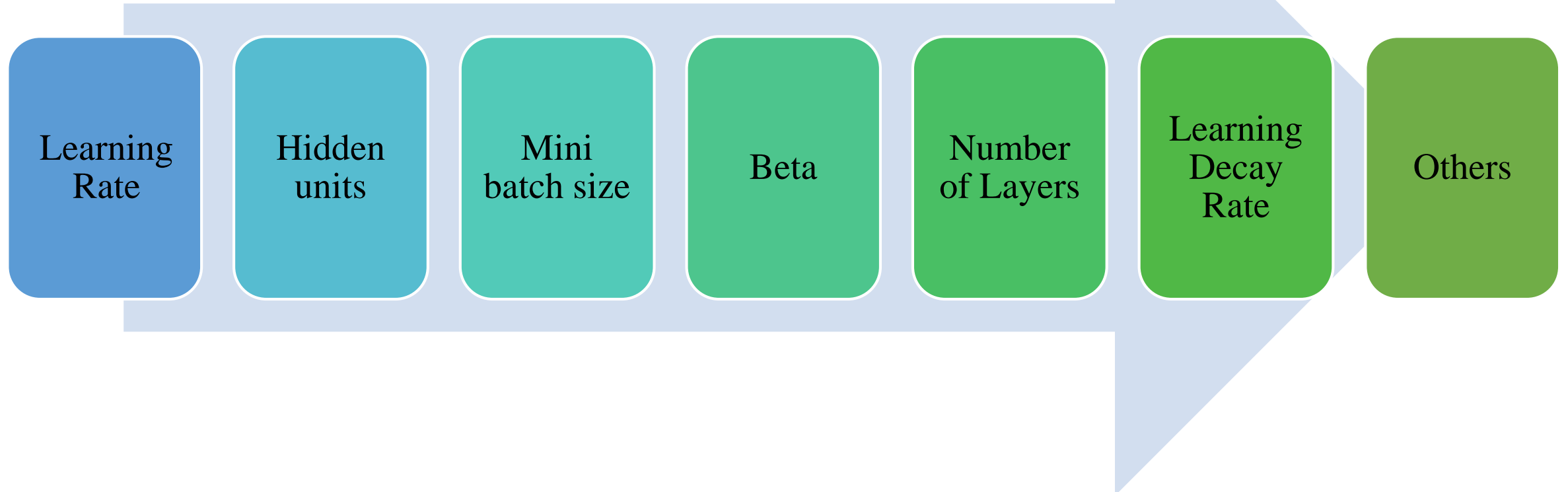
- $\alpha = \frac{1}{1 + 1 * 2} 0.2 = 0.67$

- $\alpha = \frac{1}{1 + 1 * 3} 0.2 = 0.5$

- $\alpha = \frac{1}{1 + 1 * 4} 0.2 = 0.4$

How to try Hyperparameters

First focus on Most important ones and then the lesser ones in the sequence



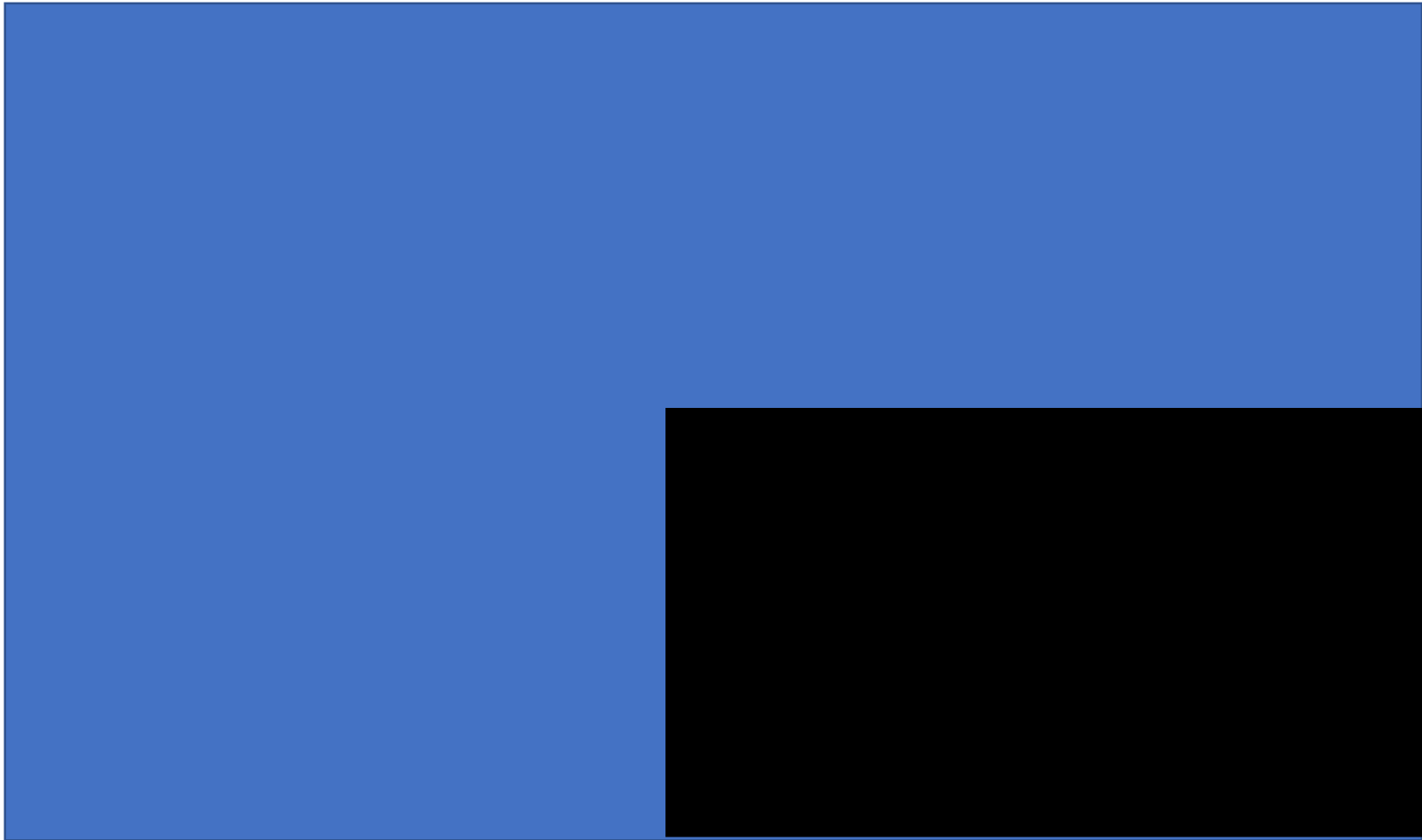
Random is better than a Grid

α

*	*	*	*	*	*	*
*	*	*	*	*	*	*
*	*	*	*	*	*	*
*	*	*	*	*	*	*
*	*	*	*	*	*	*
*	*	*	*	*	*	*
*	*	*	*	*	*	*
*	*	*	*	*	*	*

β

Coarser to finer



Picking hyperparameter at random and as per scale

- Is it ok to use the actual scale for all parameters or in some cases we require the log scale



Babysitting one model Vs Parallel model training

Panda or Caviar Approach

Depends upon the applications, resources and the time you have.

In babysitting we painstakingly try to observe and introduce mid-path corrections.

In Parallel model training we simultaneously try to train with different models

Batch Norm

It is an extension of normalizing inputs and applies to every layer of the neural network


Given some intermediate values in Neural Network

- $\mu = \frac{1}{m} \sum z^{(i)}$
- $\sigma^2 = \frac{1}{m} \sum (z - \mu)^2$
- $z_{norm}^i = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$
- $\tilde{z}^i = \gamma z_{norm}^i + \beta$ where γ and β are learnable parameters
- If $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$ then $\tilde{z}^i = z^{(i)}$


Applying Batch Norm

- $X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^i = a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$
- `tf.nn.batch-normalization`
- Each mini-batch is scaled by the mean/variance computed on just that mini-batch
- This adds some noise to the values of z within that minibatch. Similar to dropout it has some regularization effect, as it adds to hidden layers activations.


Why batch norm




Applying it on earlier layers helps in decoupling from the later layers.



That actually means that it provides a robustness to the changes in the covariance shift due to change in the input distribution.



So if there are frequent changes in the input examples than it will provide a cushion for the effect to taper off while going to later layers.



For test data we should prefer taking the exponentially weighted averages for μ and σ^2 of subsequent layers during training that will be better than the μ and σ^2 values of the training set itself.