CSL446
Neural Network and Deep Learning
Module Gradient Descent

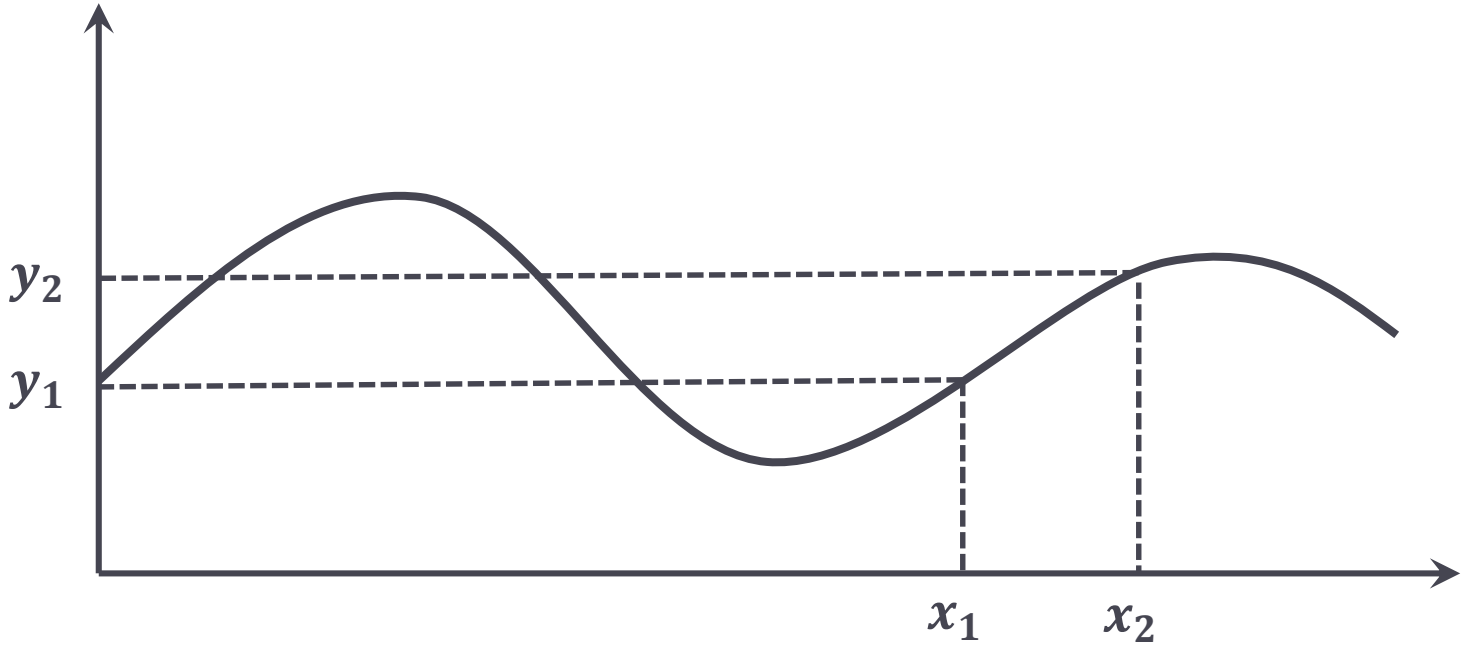
Dr. Snehal B Shinde

Computer Science and Engineering

Indian Institute of Information Technology, Nagpur.

Intuition about derivatives

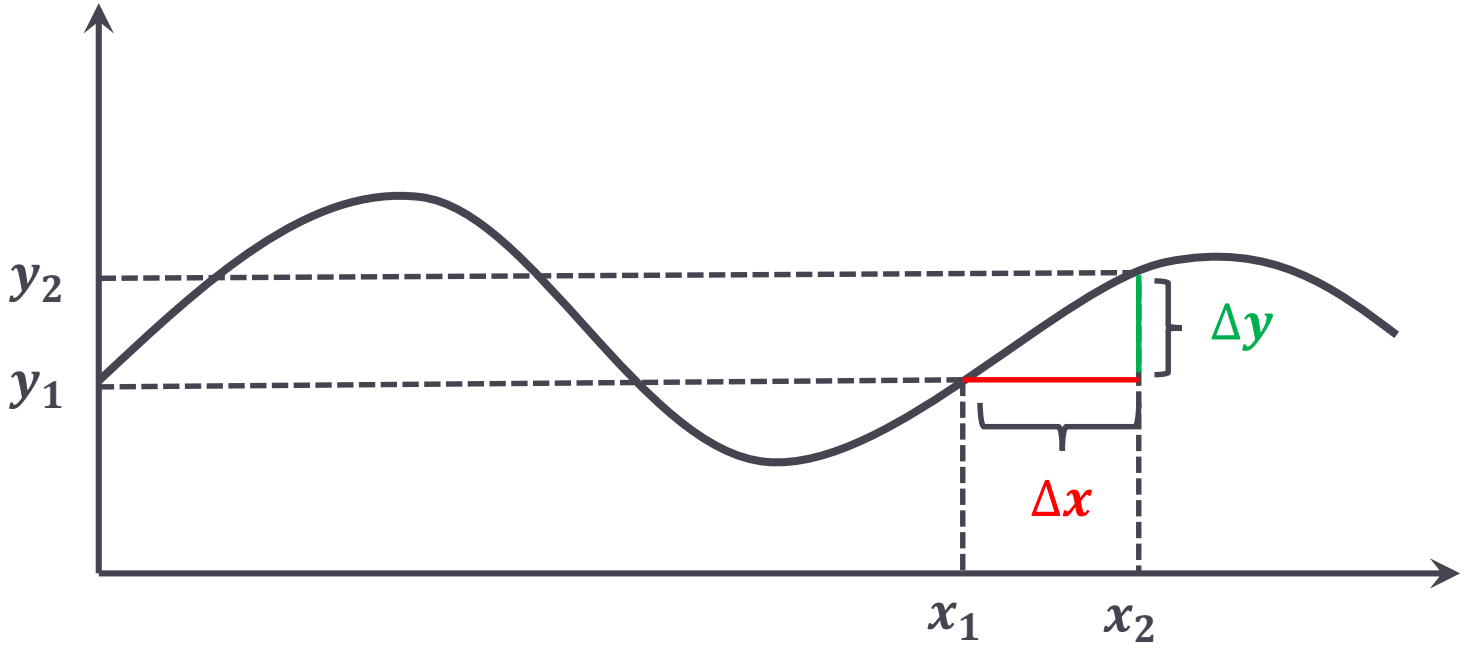
How much does y change as x changes?



$$\frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

Intuition about derivatives

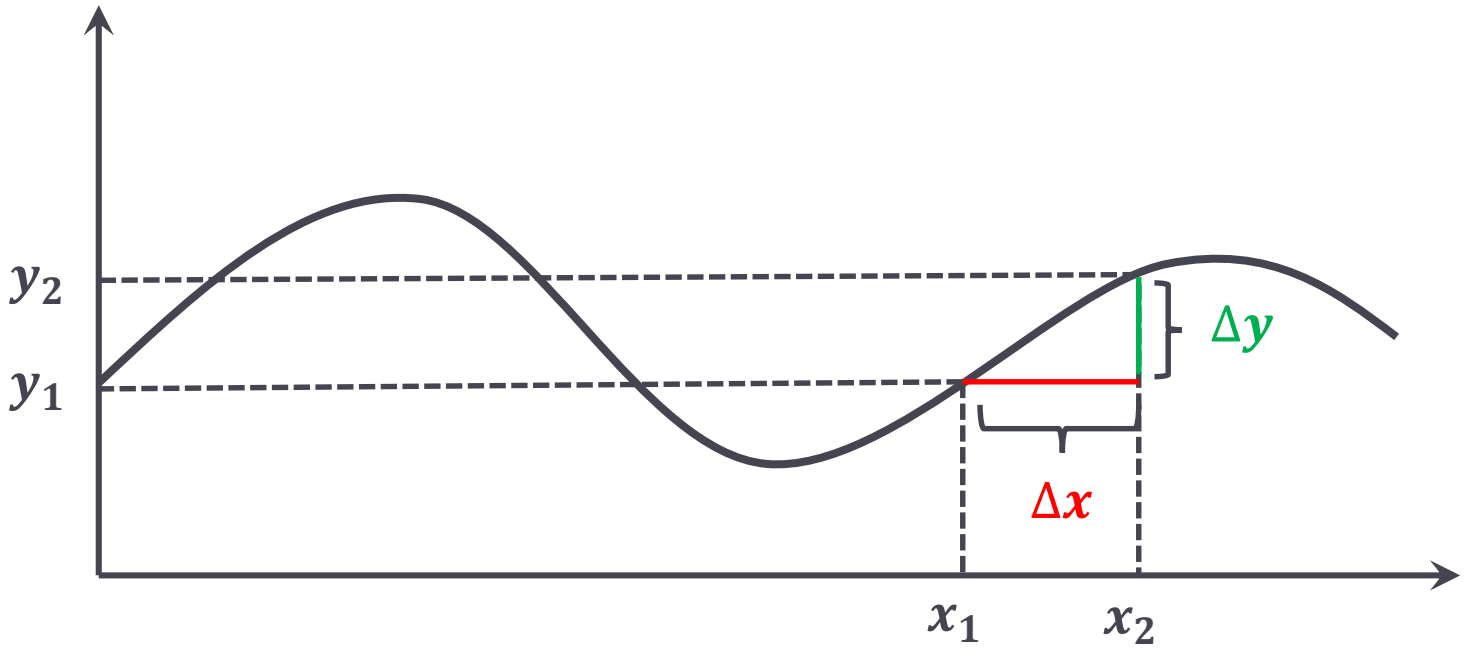
How much does y change as x changes?

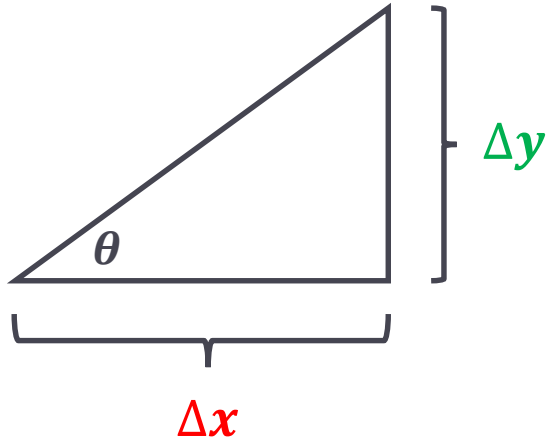


$$\frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

Differentiation

Slope of tangent of f(x) at x



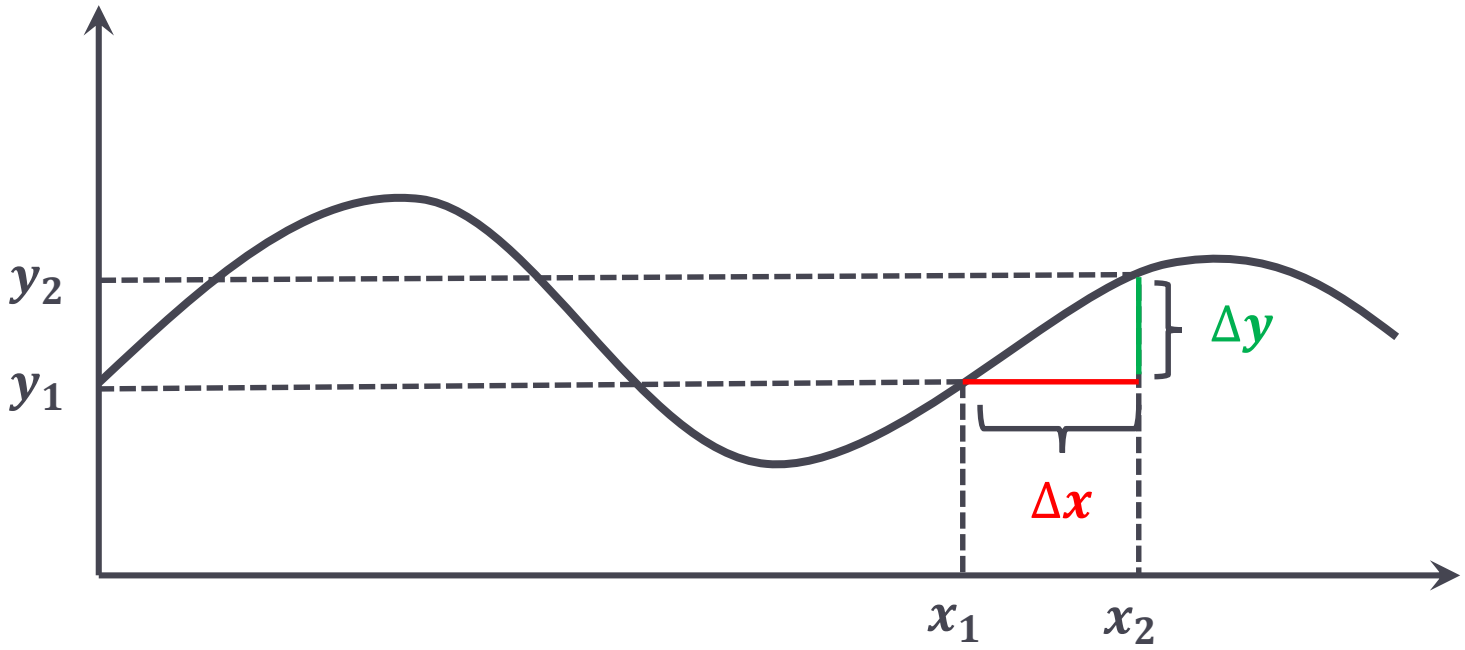


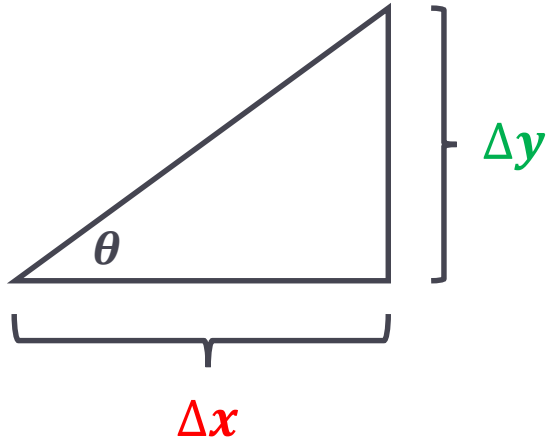
$$\frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

$$tan\theta = \frac{\Delta y}{\Delta x}$$

Intuition about derivatives

Slope of tangent of f(x) at x

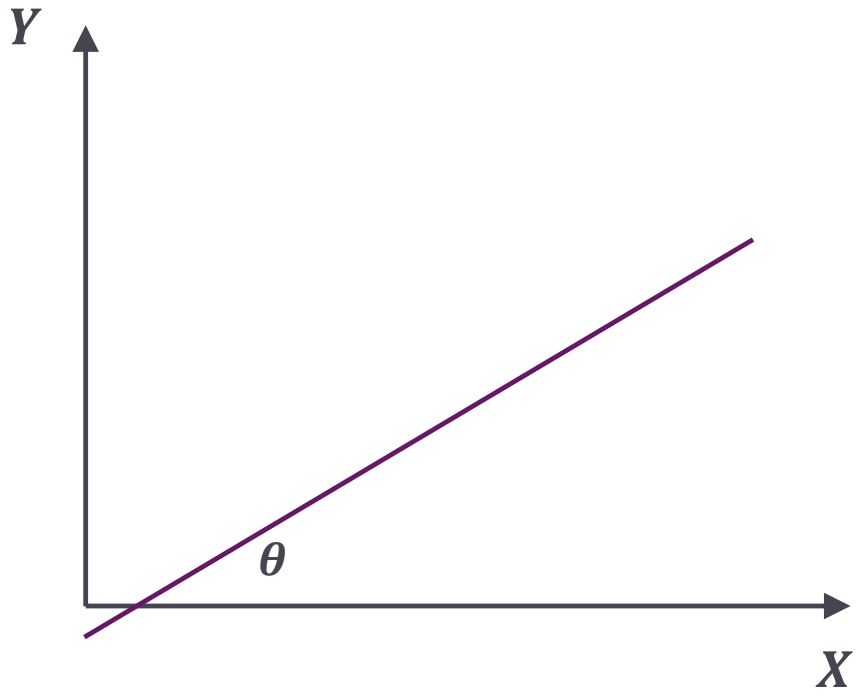




$$\frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

$$tan\theta = \frac{\Delta y}{\Delta x}$$

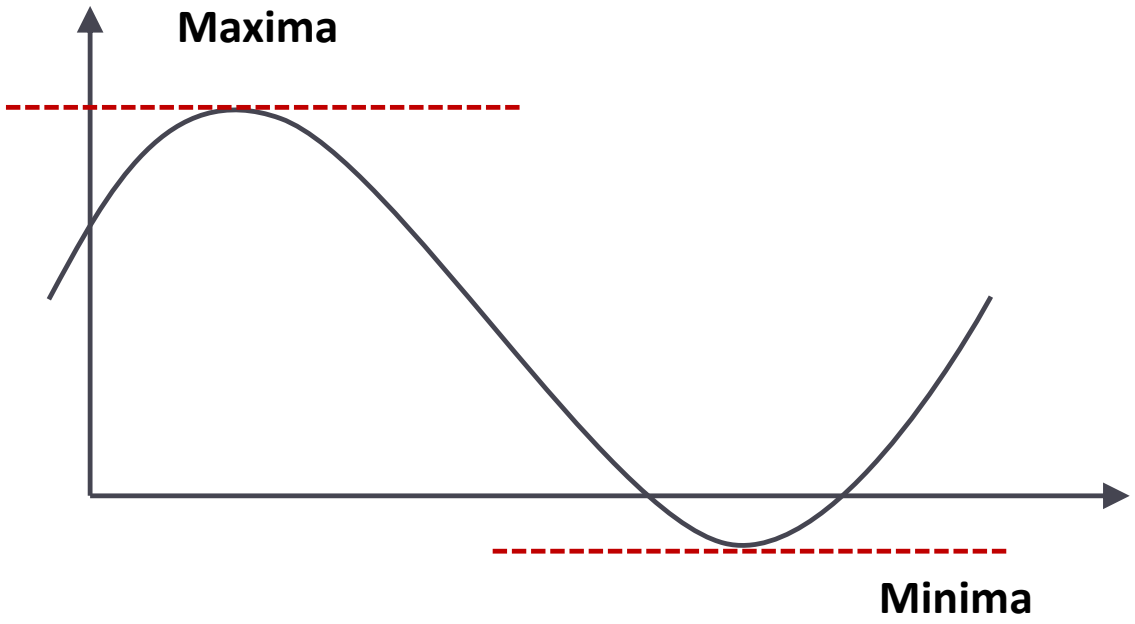
Intuition about derivatives



tanθ or slope value

1. +ve if $0 \leq \theta < 90$
2. 0 if $\theta = 0$
3. undefined if $\theta = 90$
4. -ve if $90 \leq \theta < 180$

Maxima and Minima



Maxima:
Maximum value that function takes

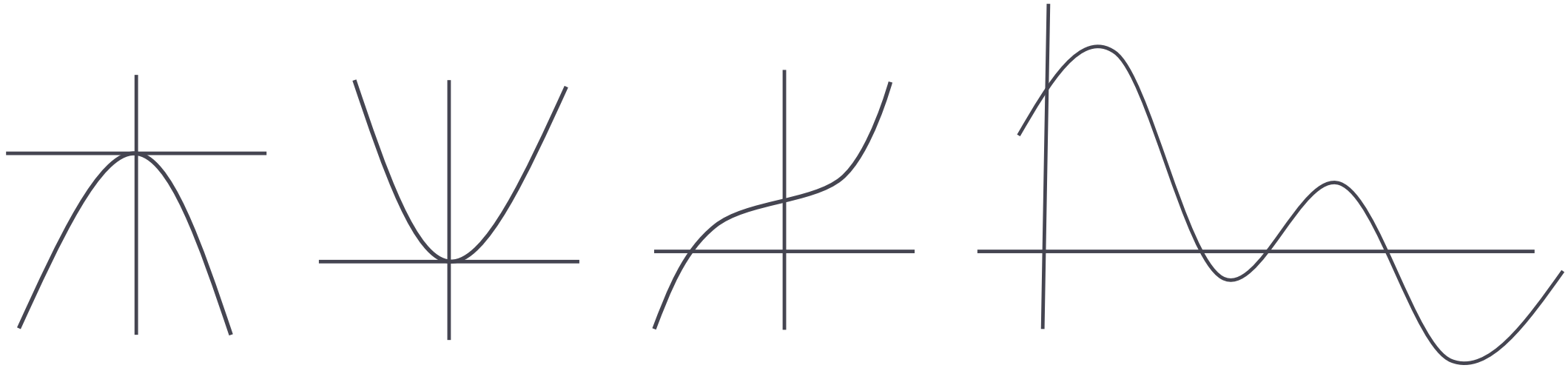
Minima:
Minimum value that function takes

Derivative/slope is zero at both maxima and minima

Maxima and Minima

There can be functions with

1. No minima
2. More than one minima (local/global)
3. No maxima
4. More than one maxima (local/global)
5. No maxima and no minima



Logistic Loss

$$f(x) = \log(1 + e^{(ax)})$$

$$\frac{df(x)}{dx} = \frac{ae^{(ax)}}{1 + e^{(ax)}}$$

$$\frac{df(x)}{dx} = 0$$

$$\frac{ae^{(ax)}}{1 + e^{(ax)}} = 0$$

Let f(x) is a loss function

To minimize loss take derivative w.r.t. x and equate to 0

Solving this equation is not trivial using Computer methods

For that we use another method i.e. Gradient Descent Method

Gradient descent

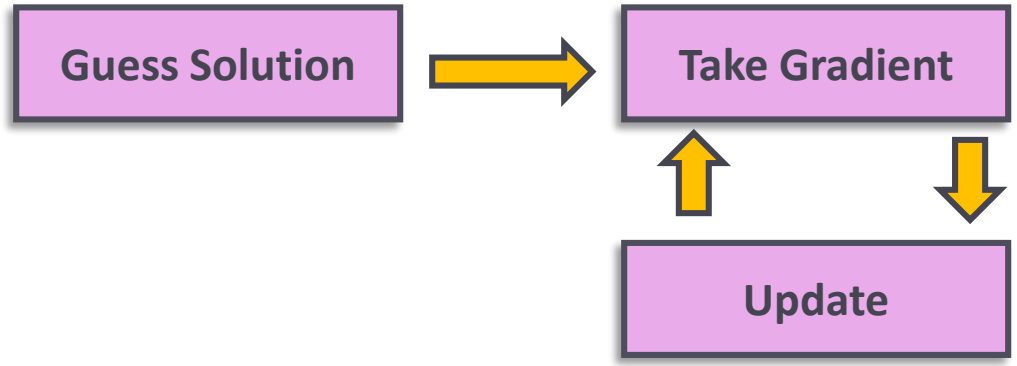
- Gradient descent is an optimization algorithm **to minimize the cost function** by iteratively adjusting parameters in the aiming to find the optimal set of parameters.

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

- The cost function measures how well the model fits the training data and defines the difference between the predicted and actual values.
- The cost function’s gradient is the derivative with respect to the model’s parameters.
- The algorithm starts with an initial set of parameters and updates them in small steps to minimize the cost function.
- The learning rate controls the step size, which determines how quickly the algorithm moves towards the minimum.

Gradient Descent Algorithm

1. Iterative Method – Gradually moves towards the solution
2. Easy to implement using computer



Backpropagation and Gradient Descent

our goal in training a neural network is to find weights and biases which minimize the quadratic cost function

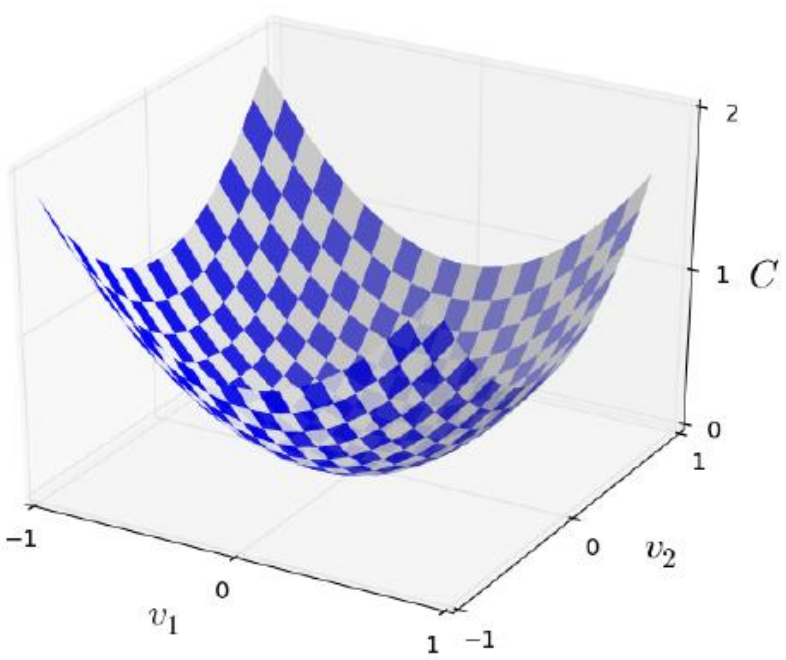
Gradient descent algorithm

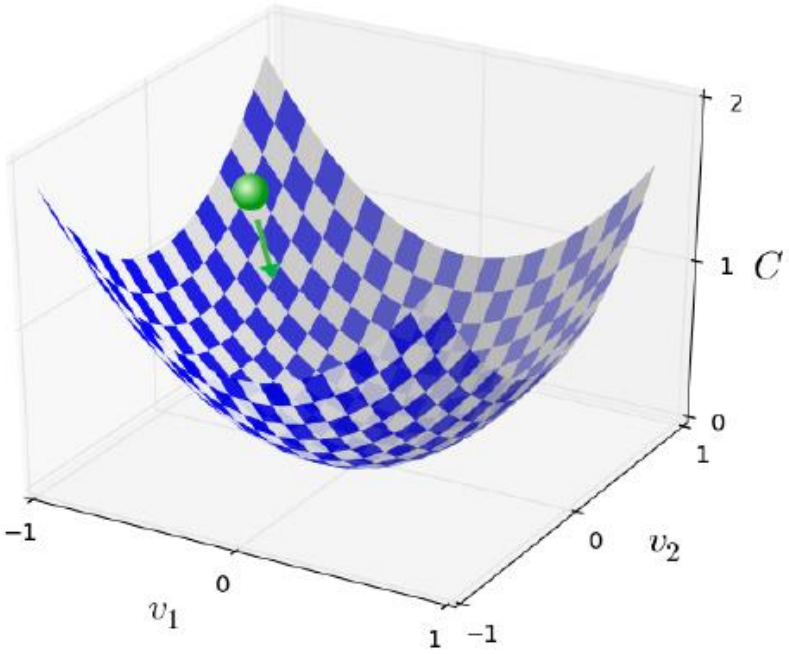
$$\text{repeat until convergence } \{$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

$$\}$$



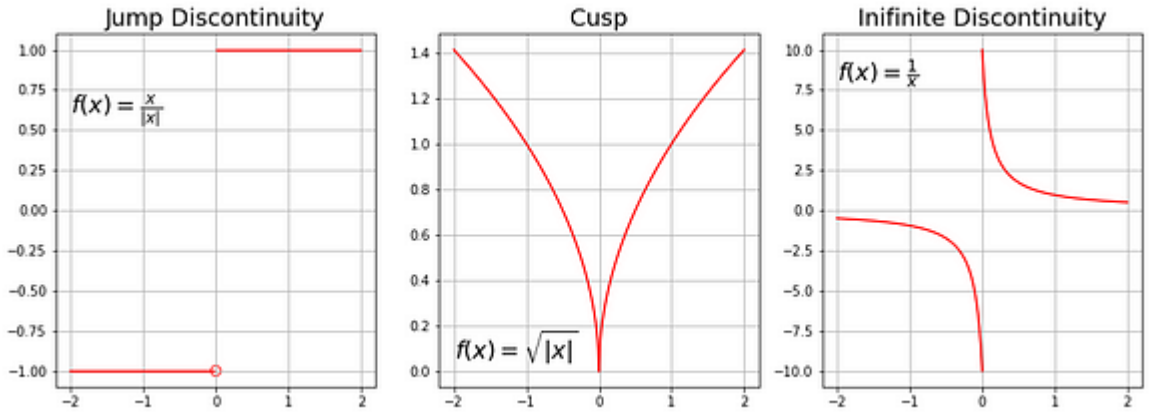


Function requirements

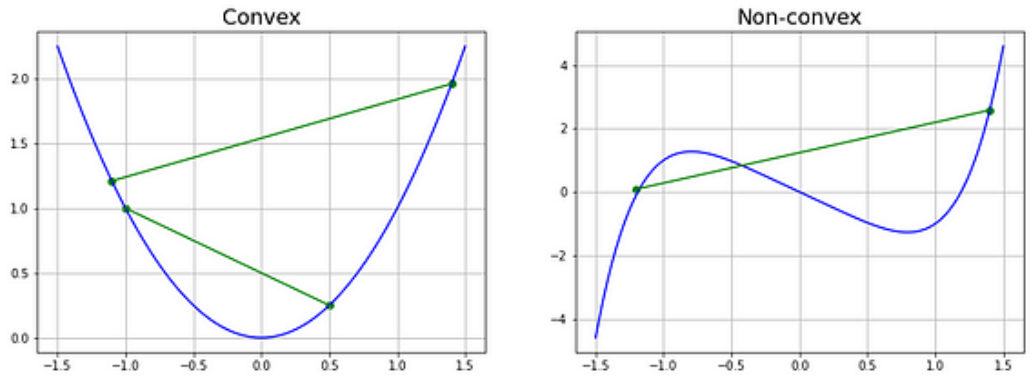
Gradient descent algorithm does not work for all functions. There are two specific requirements.

A function has to be:

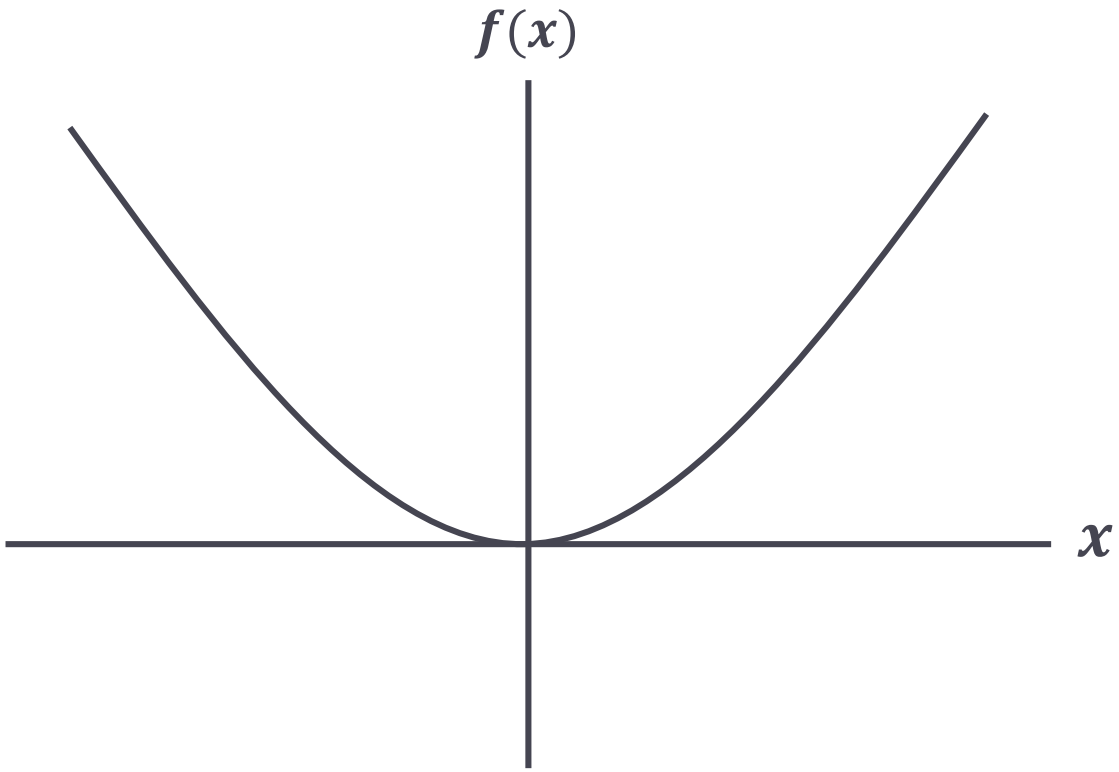
1. Differentiable



2. convex



Gradient Descent Algorithm

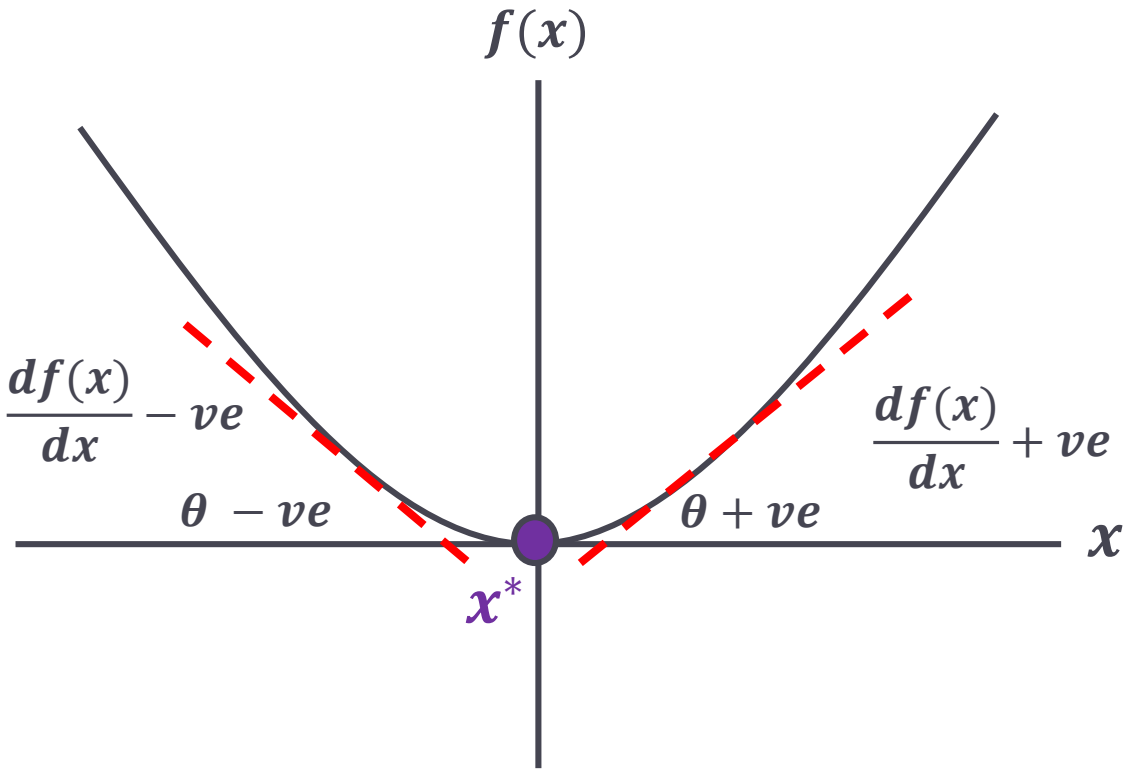


Minimization and Maximization

$$x^* = \min_x f(x)$$

Find *x* such that it minimizes *f(x)*

Gradient Descent Algorithm

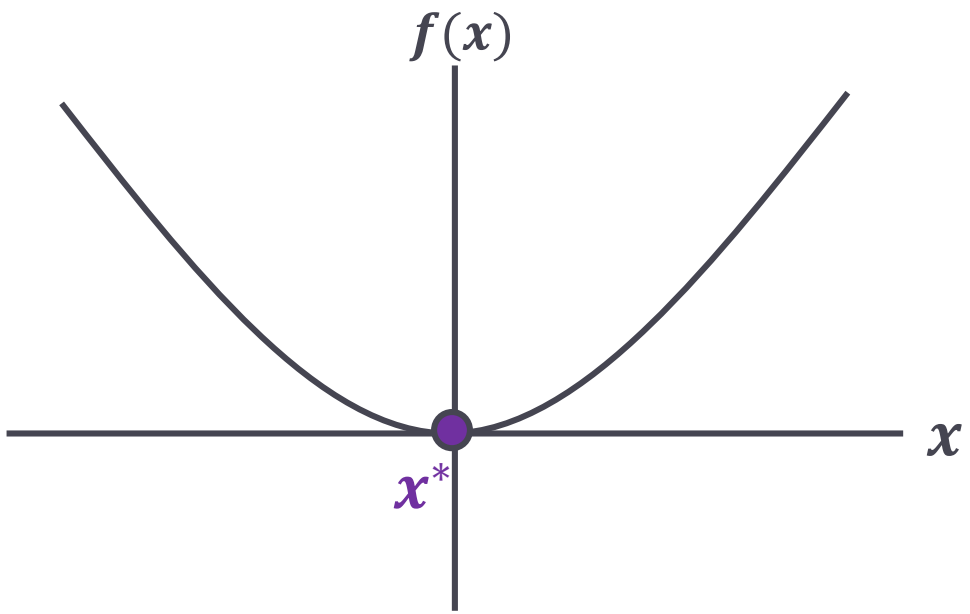


$$x^* = \min_x f(x)$$

Minima:

1. One side slope is positive
2. Other side slope is negative
3. Slope is zero at exactly minima

Gradient Descent Algorithm



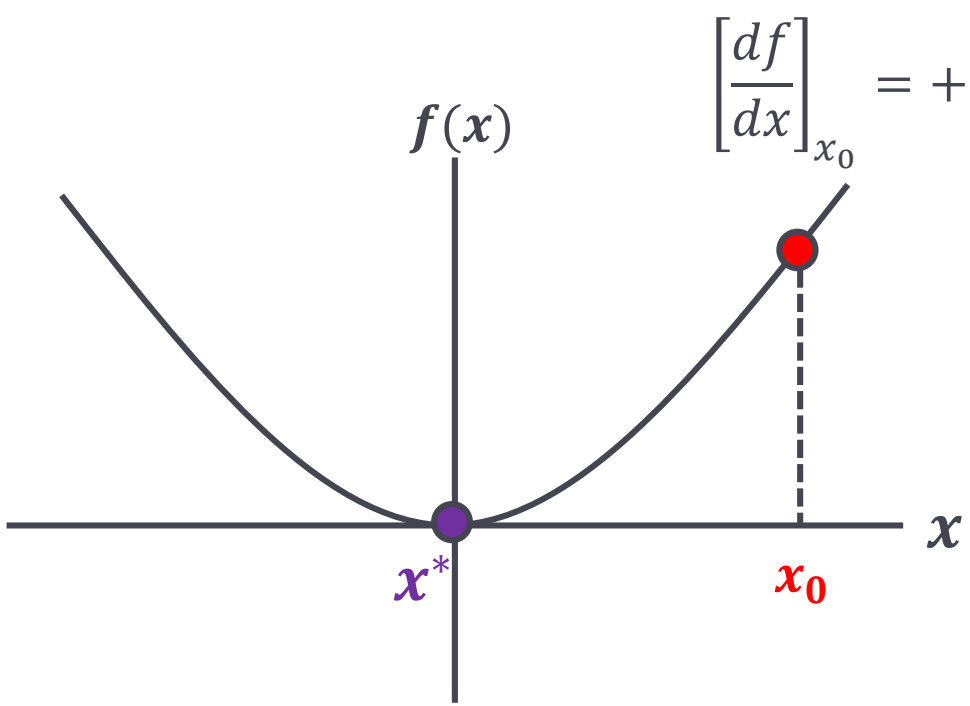
1. *Pick an initial point* x_0 *at random location*

2. $$x_i = x_{i-1} - \upsilon * \left[\frac{df}{dx}\right]_{x_{i-1}}$$

$\upsilon \rightarrow Learning\ Rate$

$i \rightarrow 1\ to\ k$

Gradient Descent Algorithm



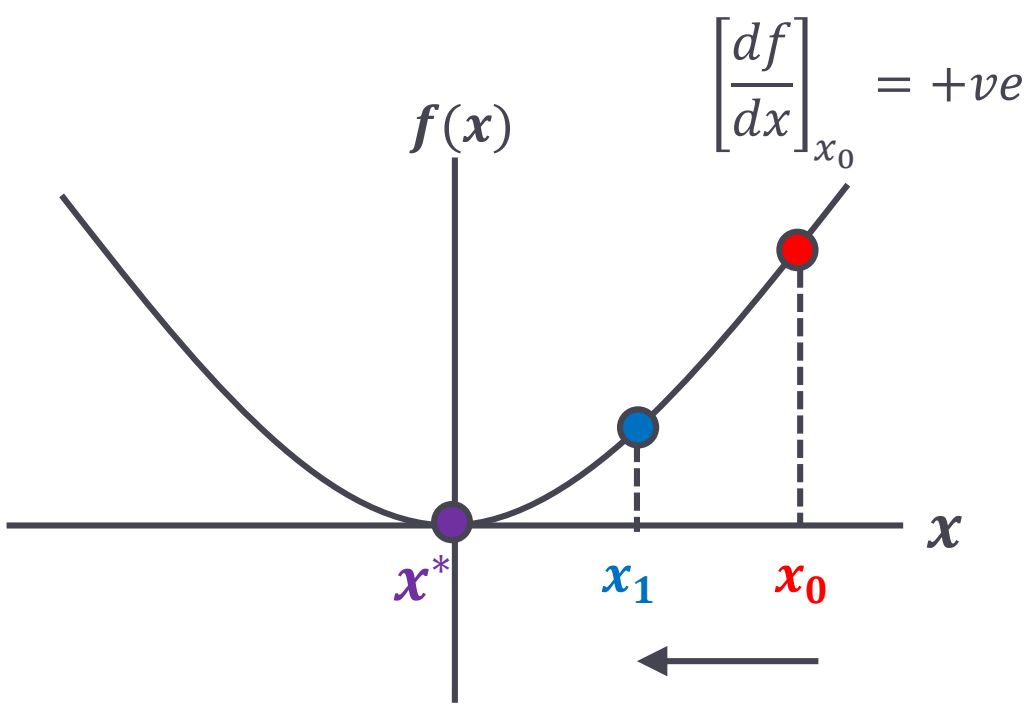
1. *Pick an initial point* x_0 *at random location*

$$x_1 = x_0 - v * \left[\frac{df}{dx}\right]_{x_0}$$

$$x_1 = x_0 - v * (some + ve\ value)$$

$$x_1 < x_0$$

Gradient Descent Algorithm



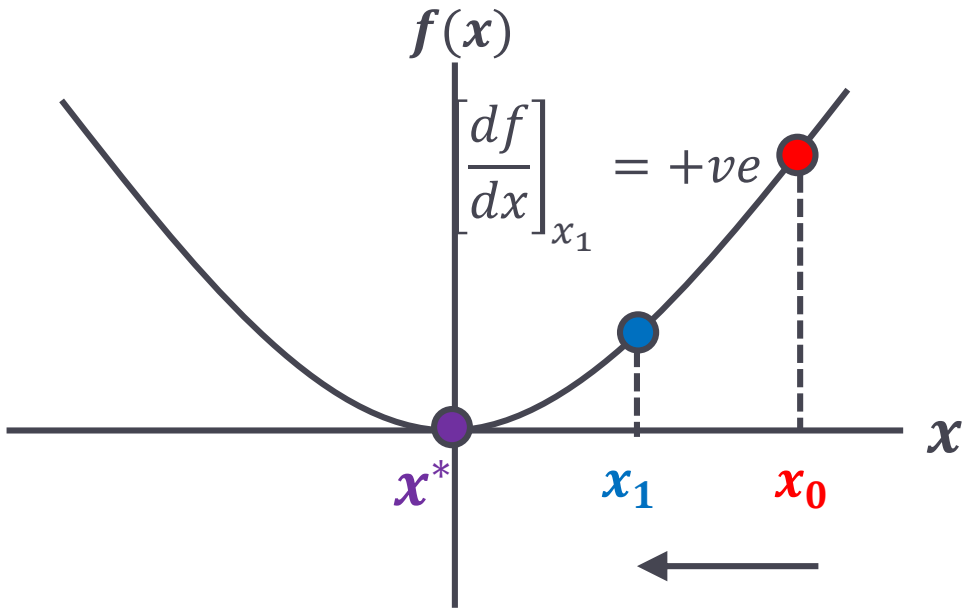
1. *Pick an initial point* x_0 *at random location*

$$x_1 = x_0 - v * \left[\frac{df}{dx}\right]_{x_0}$$

$$x_1 = x_0 - v * (some + ve\ value)$$

$$x_1 < x_0$$

Gradient Descent Algorithm



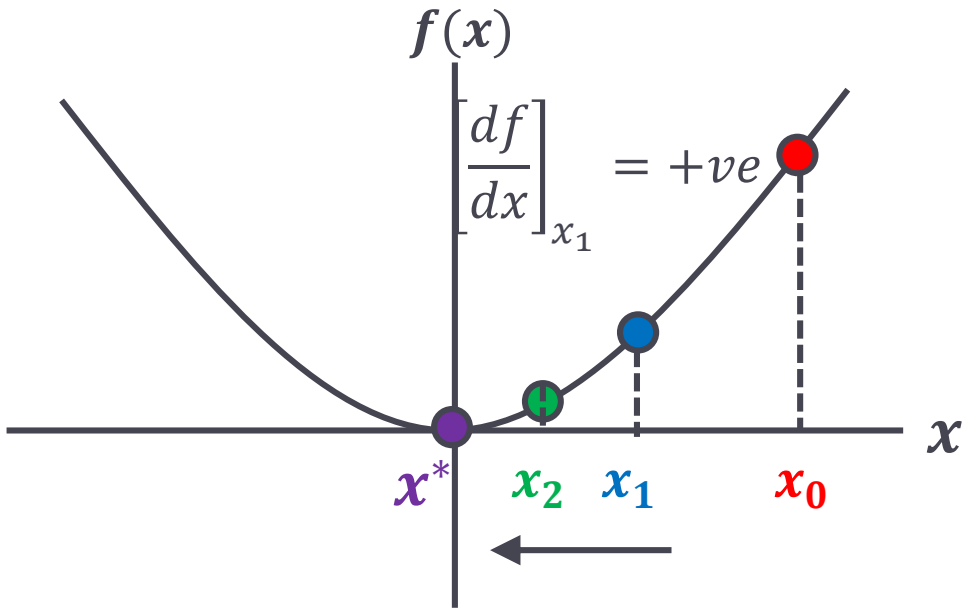
1. *Pick an initial point* x_0 *at random location*

$$x_2 = x_1 - v * \left[\frac{df}{dx}\right]_{x_1}$$

$$x_2 = x_1 - v * (some + ve\ value)$$

$$x_2 < x_1$$

Gradient Descent Algorithm



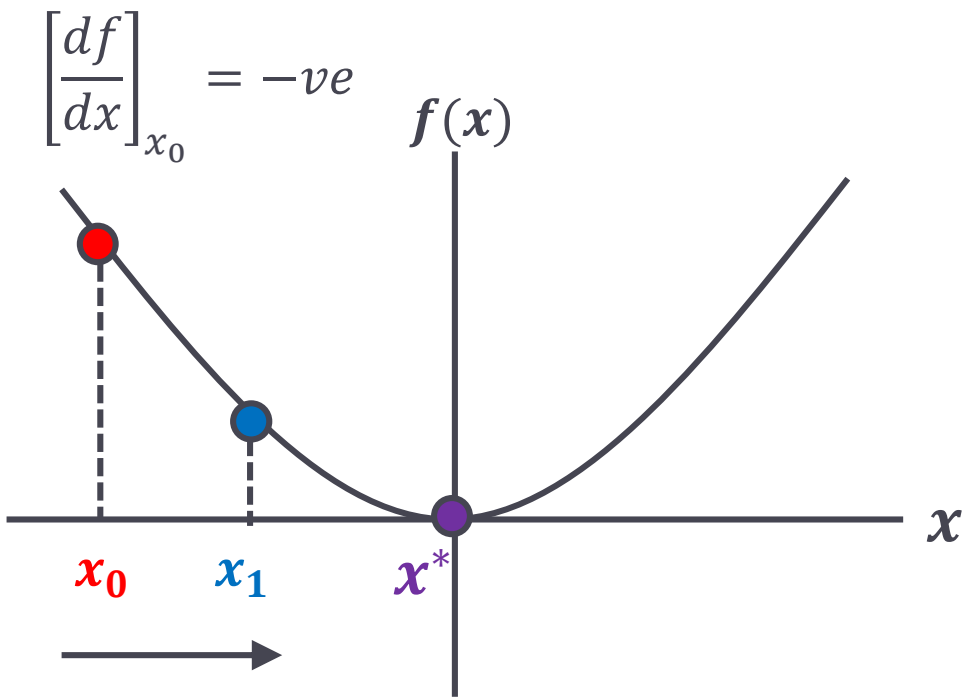
1. *Pick an initial point* x_0 *at random location*

$$x_2 = x_1 - v * \left[\frac{df}{dx}\right]_{x_1}$$

$$x_2 = x_1 - v * (some + ve\ value)$$

$$x_2 < x_1$$

Gradient Descent Algorithm



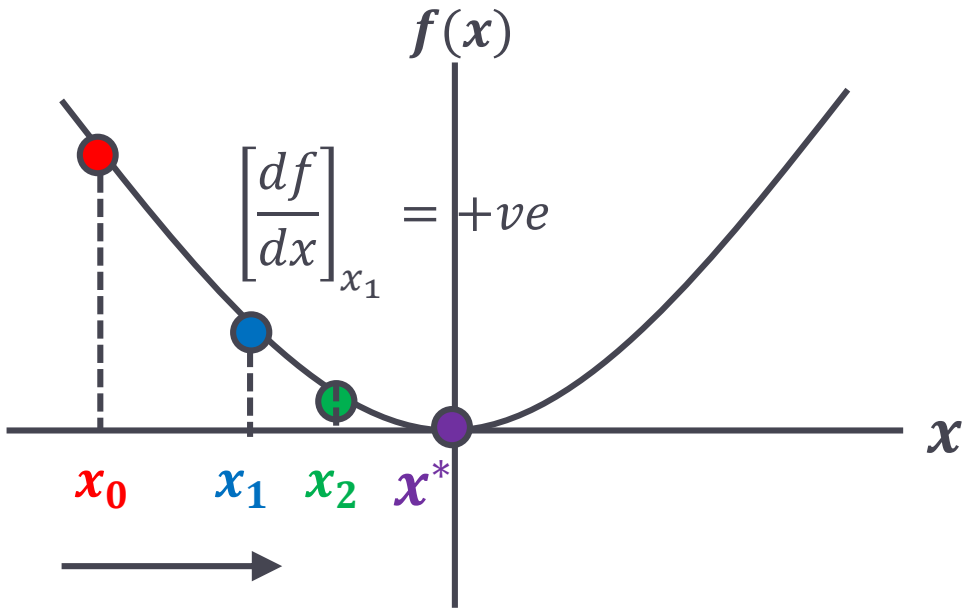
1. *Pick an initial point* x_0 *at random location*

$$x_1 = x_0 - v * \left[\frac{df}{dx}\right]_{x_0}$$

$$x_1 = x_0 - v * (some - ve\ value)$$

$$x_1 > x_0$$

Gradient Descent Algorithm



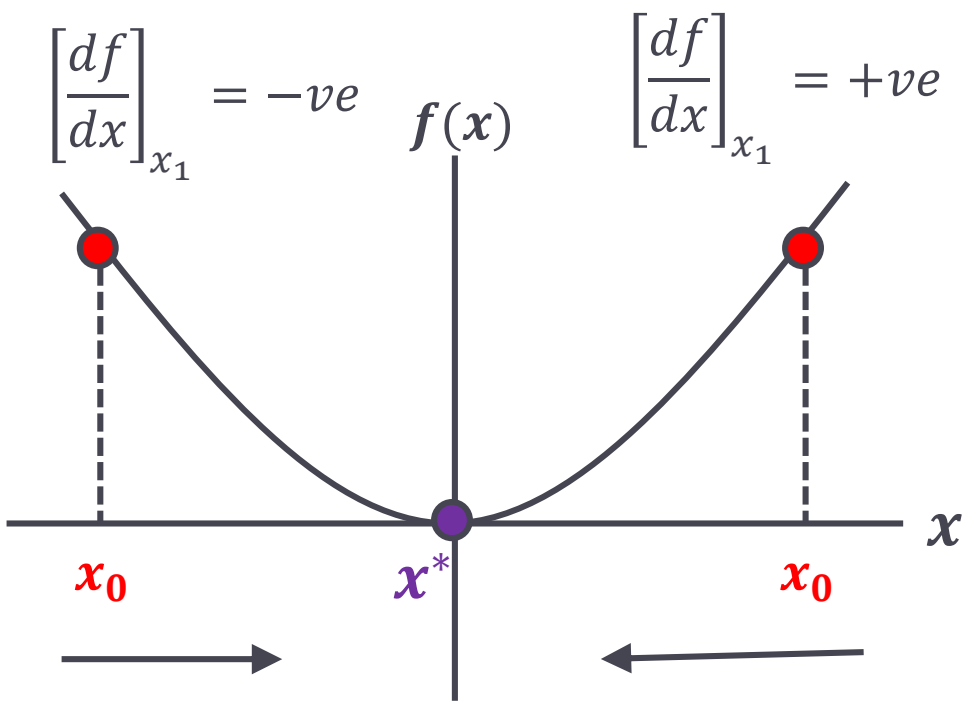
1. *Pick an initial point* x_0 *at random location*

$$x_2 = x_1 - v * \left[\frac{df}{dx}\right]_{x_1}$$

$$x_2 = x_1 - v * (some + ve\ value)$$

$$x_2 < x_1$$

Gradient Descent Algorithm

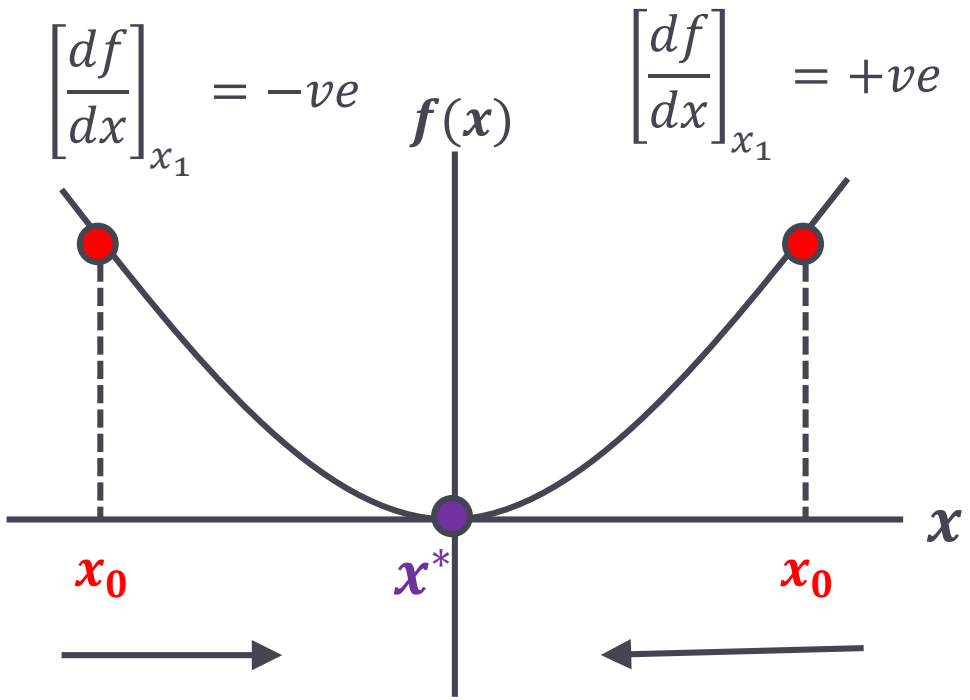


1. *Pick an initial point* x_0 *at random location*

2. $$x_i = x_{i-1} - \upsilon * \left[\frac{df}{dx}\right]_{x_{i-1}}$$

So basically start from anywhere GD Guarantees to reach at minima

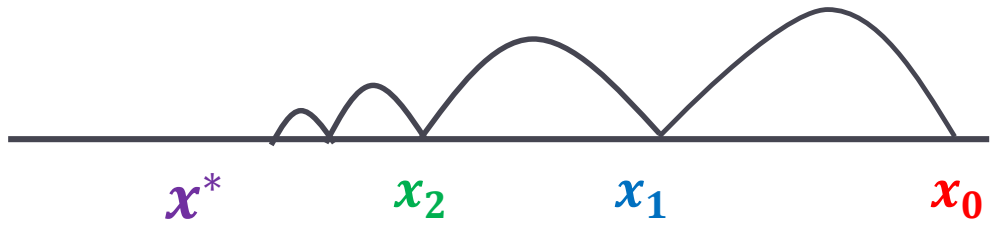
Gradient Descent Algorithm



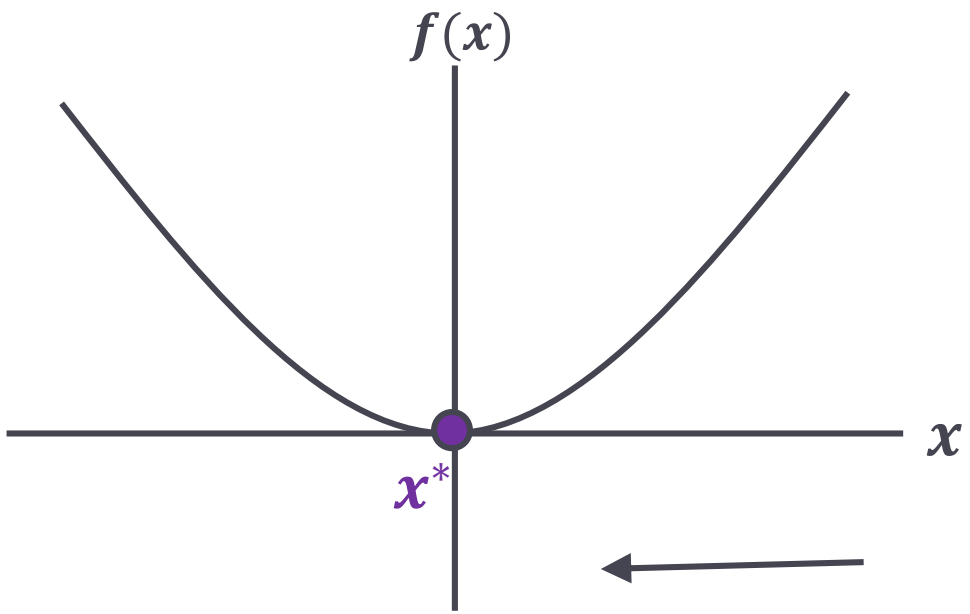
1. *Pick an initial point* x_0 *at random location*

2. $$x_i = x_{i-1} - \upsilon * \left[\frac{df}{dx}\right]_{x_{i-1}}$$

Learning Rate or Step size

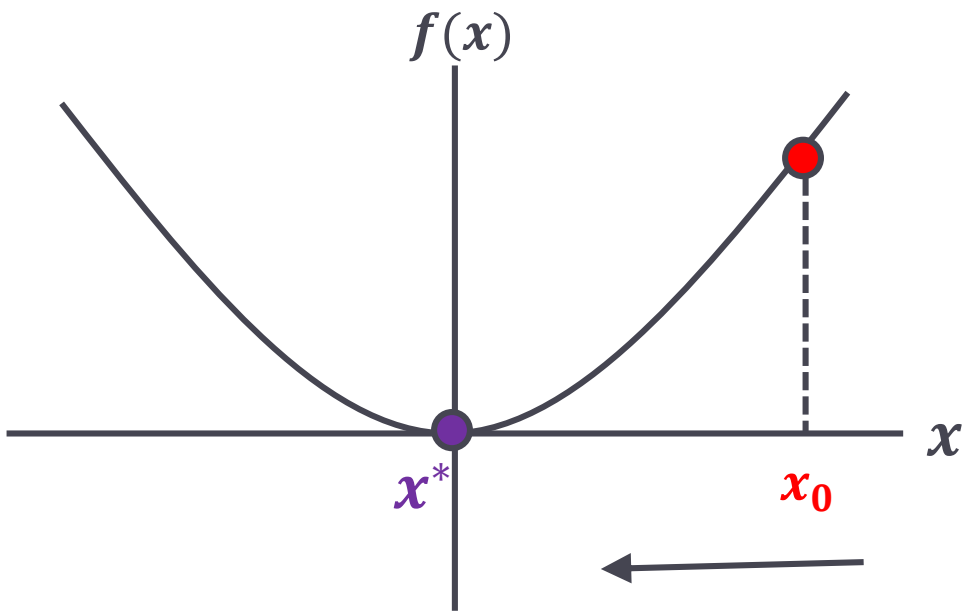


Gradient Descent Algorithm



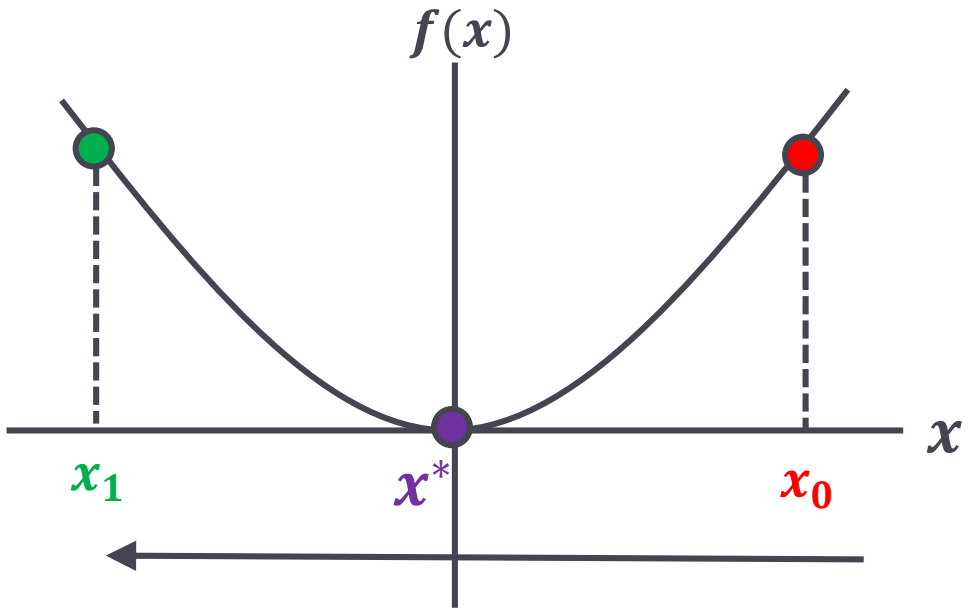
What if learning rate is too high?

Gradient Descent Algorithm

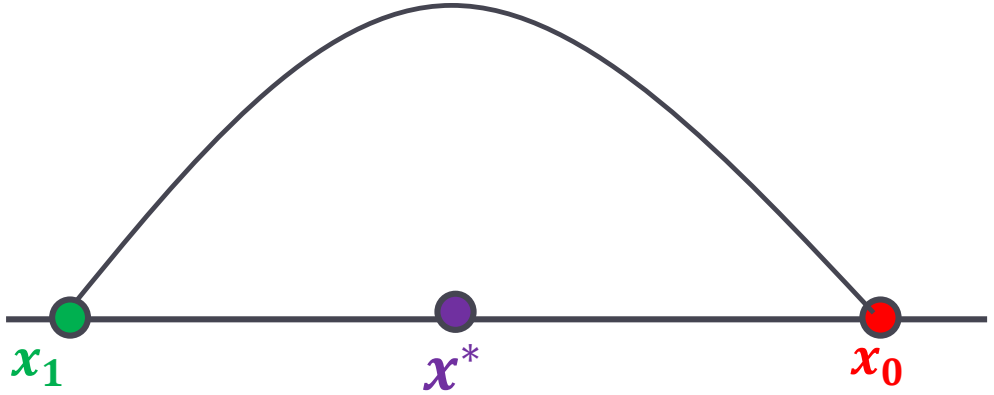


What if learning rate is too high?

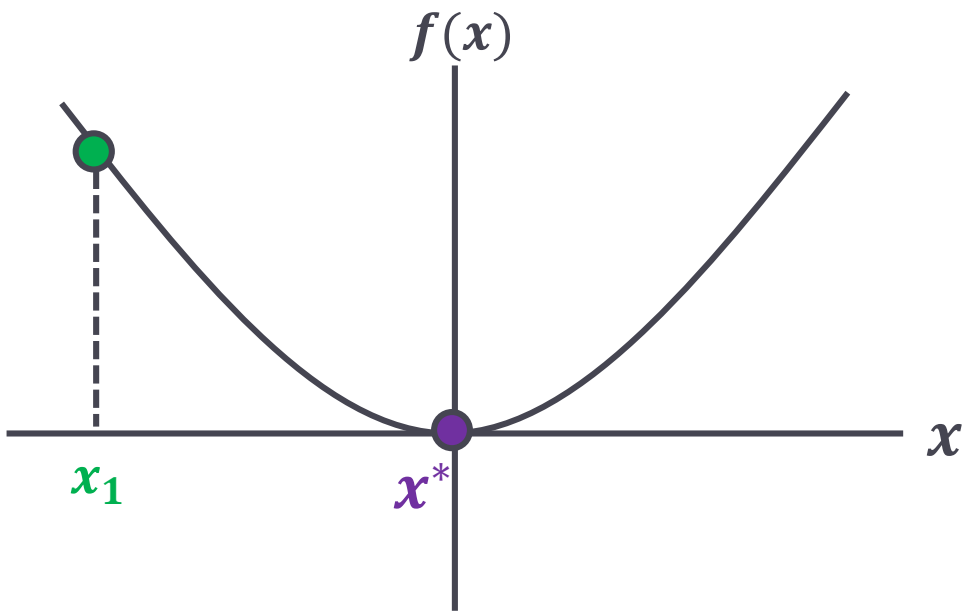
Gradient Descent Algorithm



What if learning rate is too high?

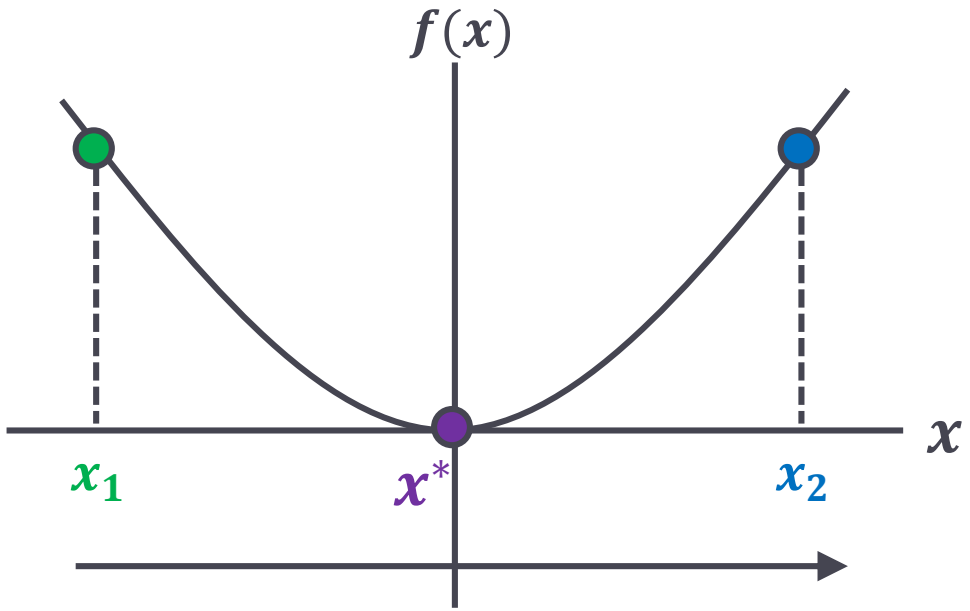


Gradient Descent Algorithm

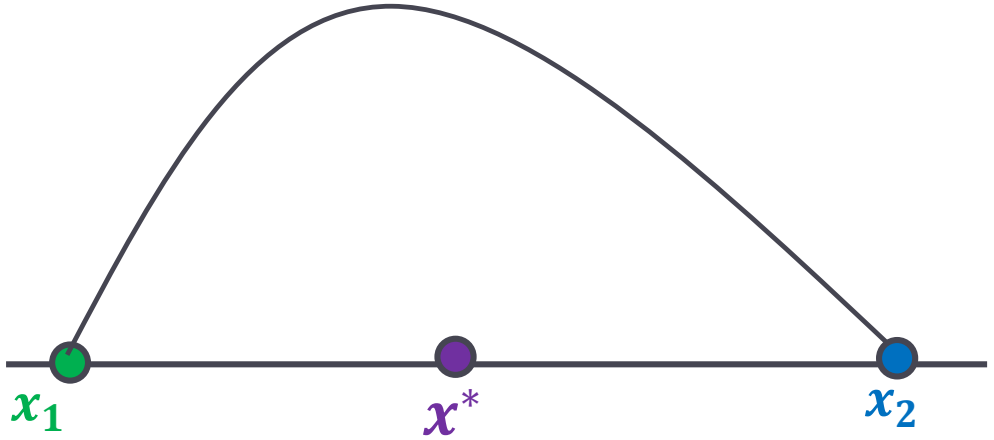


What if learning rate is too high?

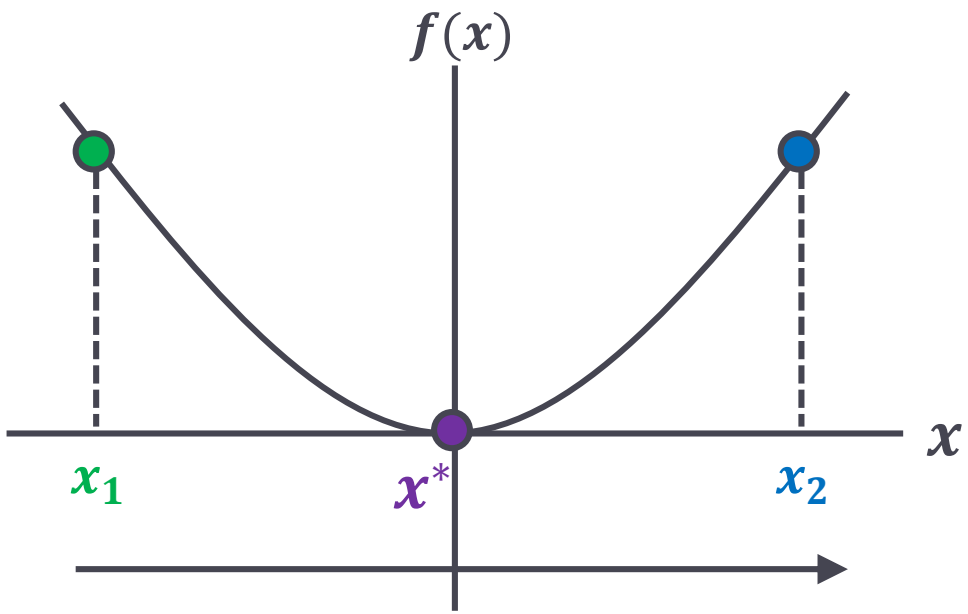
Gradient Descent Algorithm



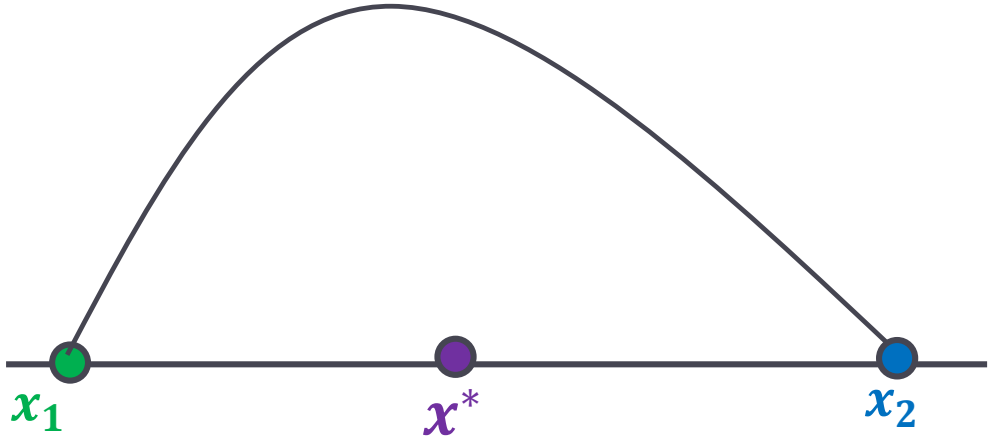
What if learning rate is too high?



Gradient Descent Algorithm

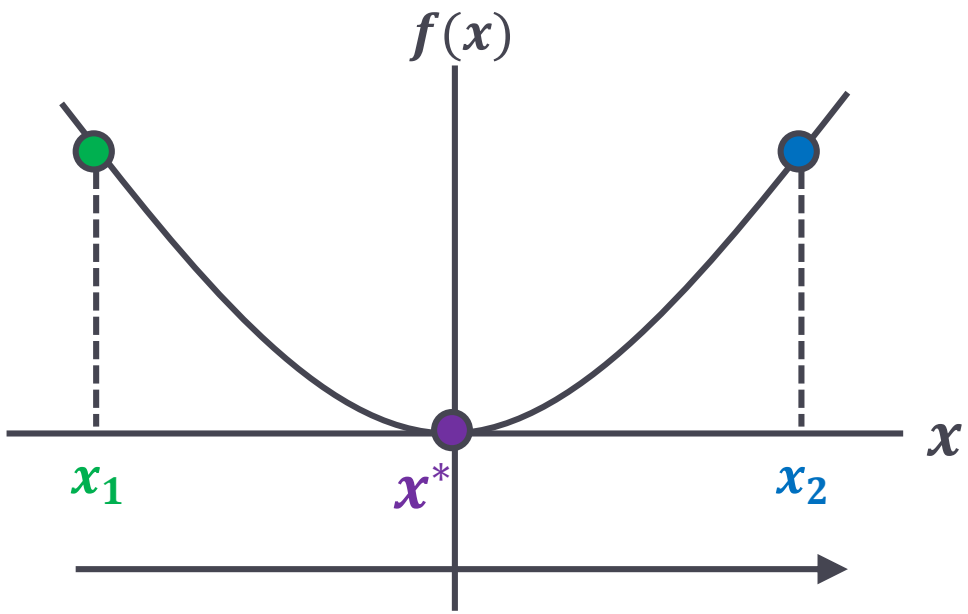


What if learning rate is too high?

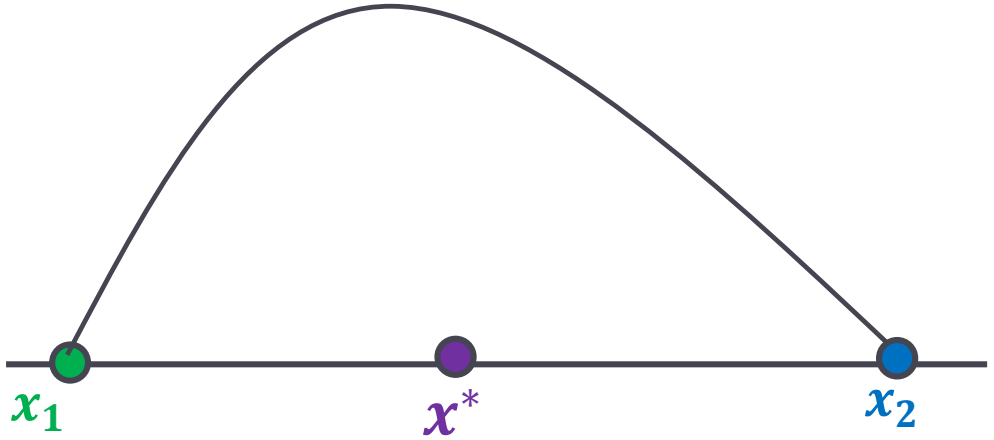


It will oscillates continuously around minima

Gradient Descent Algorithm



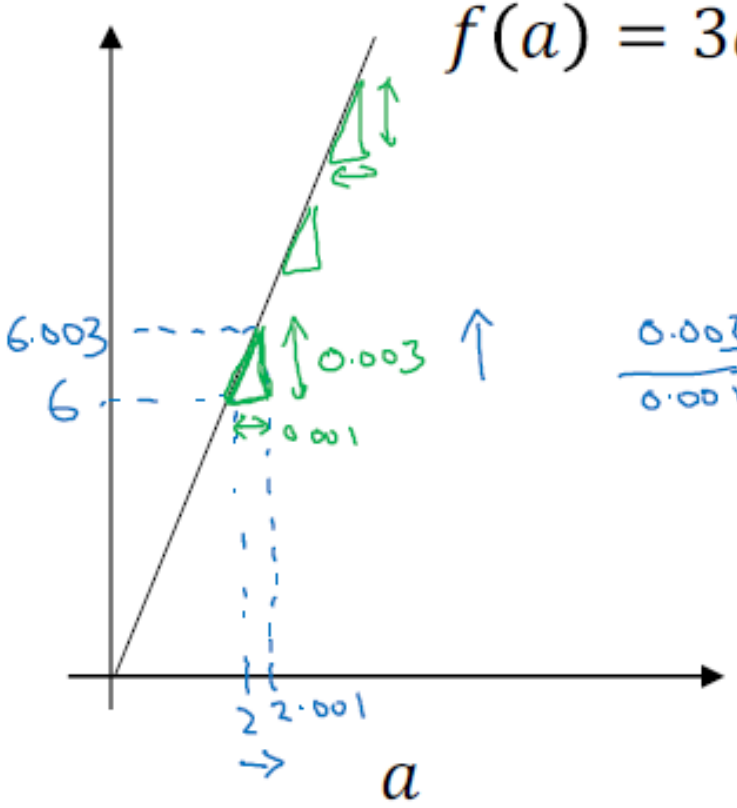
What if learning rate is too high?



It will oscillates continuously around minima

To avoid this we can decrease the LR as number of iterations increase

Intuition about derivatives



$\frac{0.003}{0.001}$ $\frac{\text{height}}{\text{width}}$

→ slope (derivative) of $f(a)$ at $a=2$ is 3

→ $a = 2$ $f(a) = 6$

$a = 2.001$ $f(a) = 6.003$

→ $a = 5$ $f(a) = 15$

$a = 5.001$ $f(a) = 15.003$

slope at $a=5$ is also 3

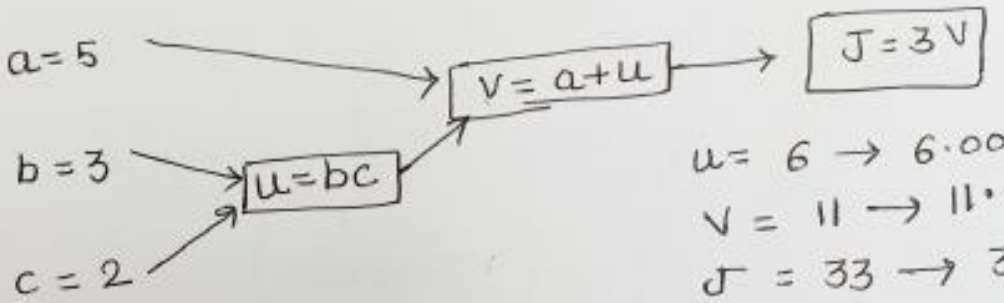
$\frac{d f(a)}{da} = 3 = \frac{d}{da} f(a)$

0.001
0.00000001
0.000000001

Andrew Ng

Computation Graph

Computing derivatives



$u = 6 \rightarrow 6.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$dv \rightarrow \frac{dJ}{dv}$ $\quad da = \frac{dJ}{da}$.

a affects v & v affects J

$$\frac{dJ}{da} = \frac{dJ}{dv} * \frac{dv}{da}$$

$$= 3 \cdot 1 .$$

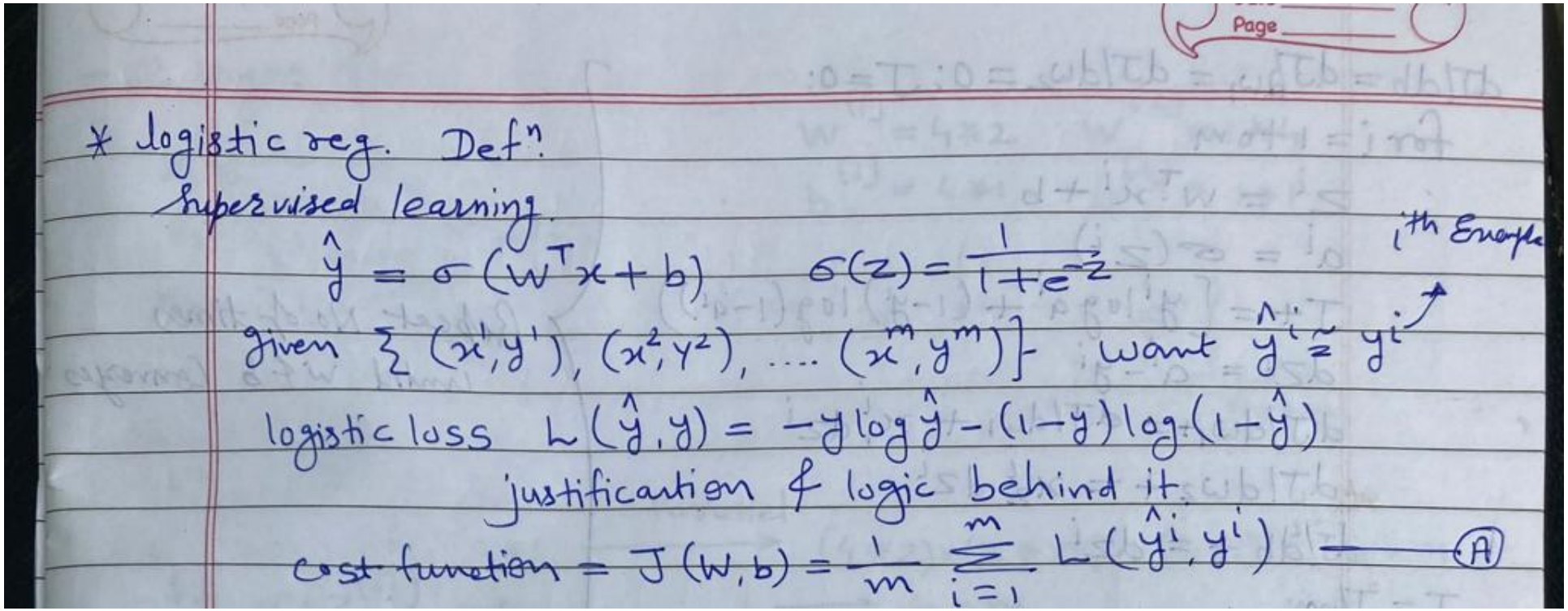
$$\frac{dJ}{db} = \frac{dj}{dv}\frac{dv}{du}\frac{du}{db} = 3 \cdot 1 \cdot c^{2} = \underline{6}$$

Single training sample → Loss function

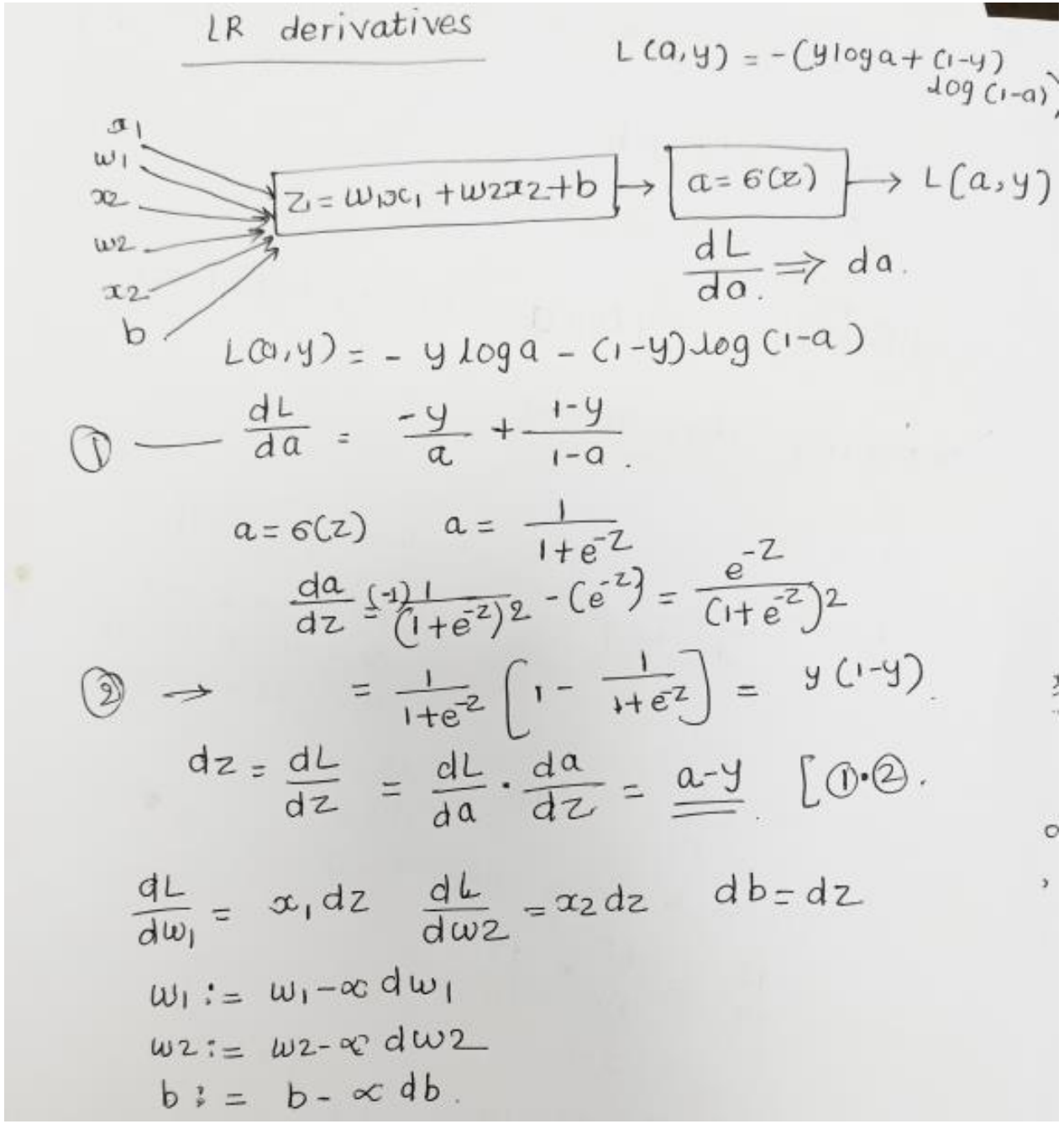
m training samples → Cost function.

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

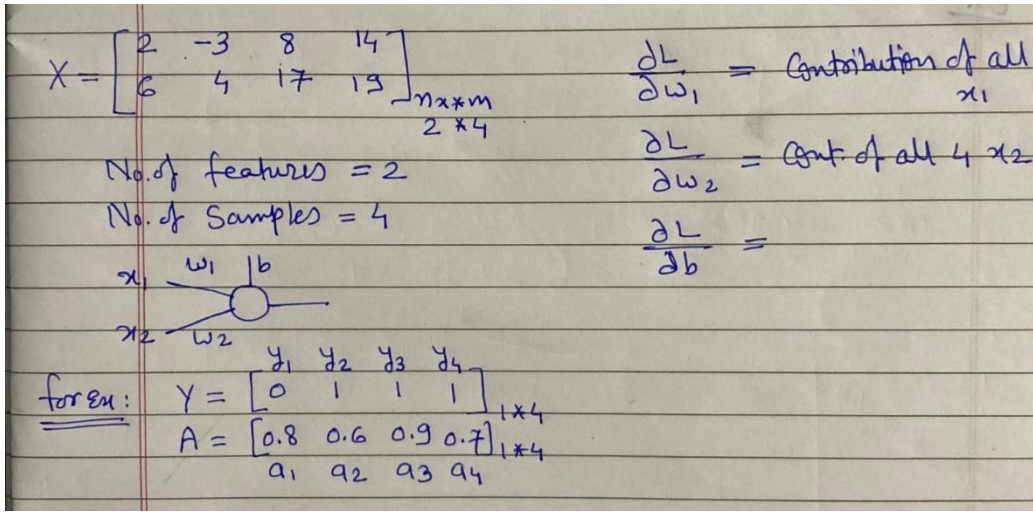
Logistic regression derivatives



Logistic regression derivatives



Gradient descent



- Gradient descent is an optimization algorithm to minimize the cost function by iteratively adjusting parameters in the aiming to find the optimal set of parameters.

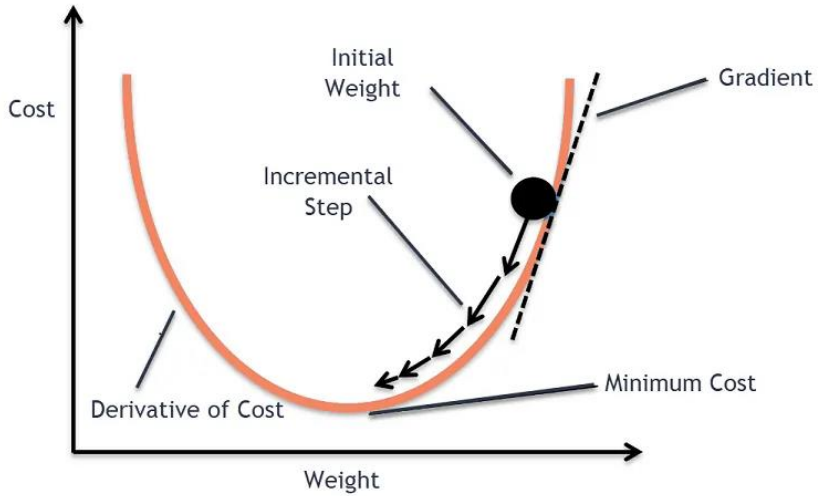
Gradient descent algorithm

$$\text{repeat until convergence } \{$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

$$\}$$



$$X = \begin{bmatrix} 2 & -3 & 8 & 14 \\ 6 & 4 & 17 & 19 \end{bmatrix}$$

$$Y = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} 0.8 & 0.6 & 0.9 & 0.7 \end{bmatrix}$$

$$\frac{\partial L}{\partial w_1} = \frac{1}{4} \sum^{4} (A_i - Y_i) x_{1i}$$

$$= \frac{1}{4}[(0.8 - 0) \cdot 2 + (0.6 - 1) \cdot (-3) + (0.9 - 1) \cdot 8 + (0.7 - 1) \cdot 14]$$

$$= \frac{1}{4}[1.6 + (-1.2) + (-0.8) + (-4.2)]$$

$$= \frac{1}{4}[1.6 - 1.2 - 0.8 - 4.2]$$

$$= \frac{1}{4}[-4.6] = -1.15$$

$$\frac{\partial L}{\partial w_2} = \frac{1}{4} \sum_{i=1}^{4} (A_i - Y_i) x_{2i}$$

$$= \frac{1}{4}[(0.8 - 0) \cdot 6 + (0.6 - 1) \cdot 4 + (0.9 - 1) \cdot 17 + (0.7 - 1) \cdot 19]$$

$$= \frac{1}{4}[4.8 + (-1.6) + (-1.7) + (-5.7)]$$

$$= \frac{1}{4}[4.8 - 1.6 - 1.7 - 5.7]$$

$$= \frac{1}{4}[-4.2] = -1.05$$

Gradient descent

- Gradient descent is an optimization algorithm to minimize the cost function by iteratively adjusting parameters in the aiming to find the optimal set of parameters.

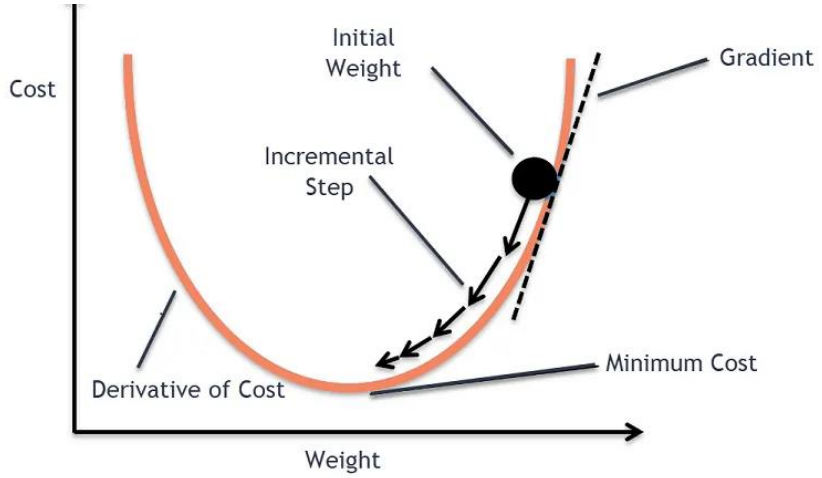
Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}



LR on m samples

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

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

$dw_1^{(i)}$, $dw_2^{(i)}$, $db^{(i)}$ — ith sample

$J = 0, dw_1 = 0, dw_2 = 0, db = 0$

for i = 1 to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

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

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 += x_1^{(i)} dz^{(i)}$

$dw_2 += x_2^{(i)} dz^{(i)}$ — two features

$db += dz^{(i)}$

more features for loop.

$J /= m \quad dw_1 /= m \quad dw_2 /= m ; \quad db /= m$

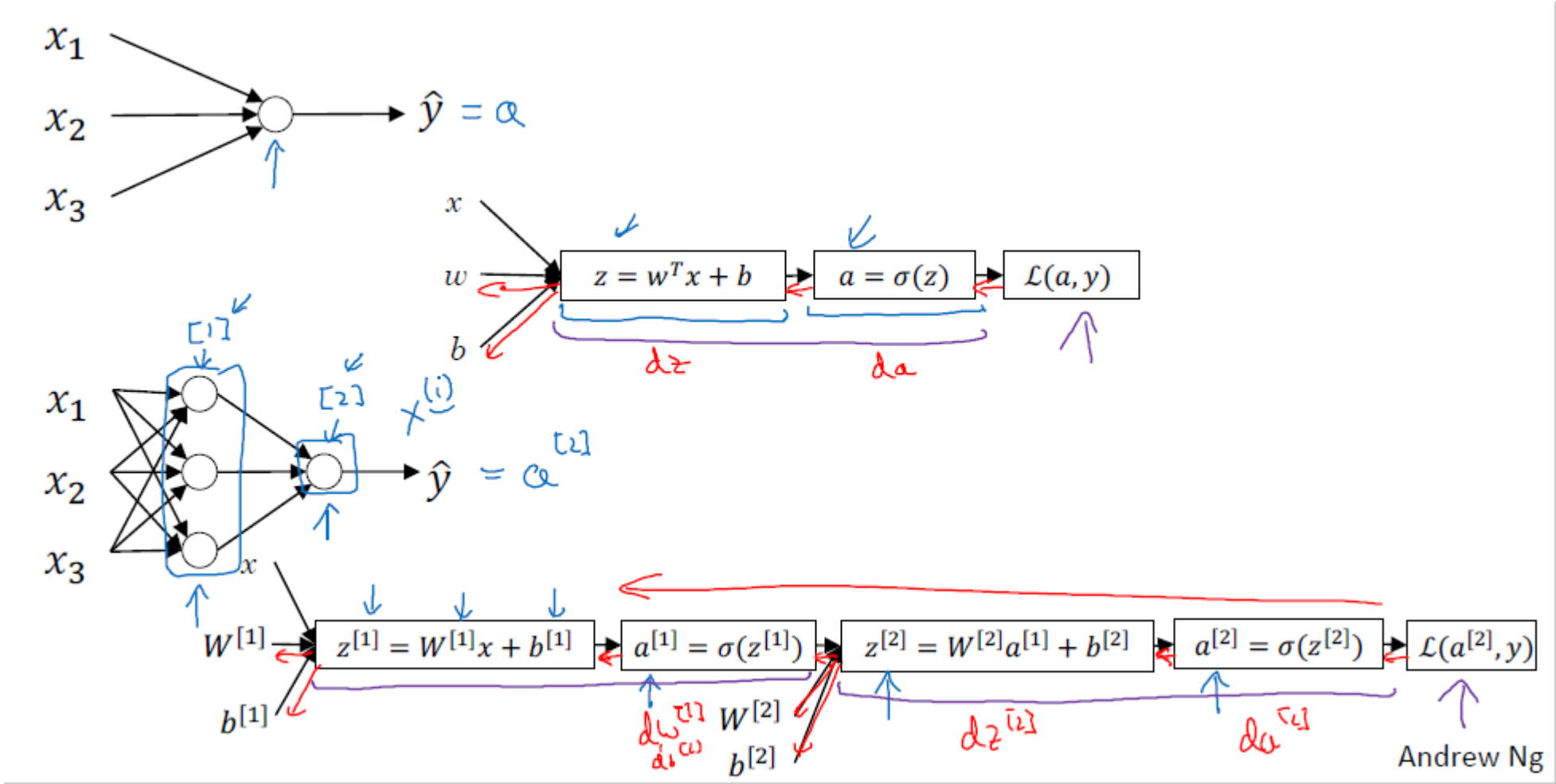
$dw_1 = \frac{\partial J}{\partial w_1}$ → overall derivative of w_1 w.r.t J

$w_1 := w_1 - \alpha\, dw_1$

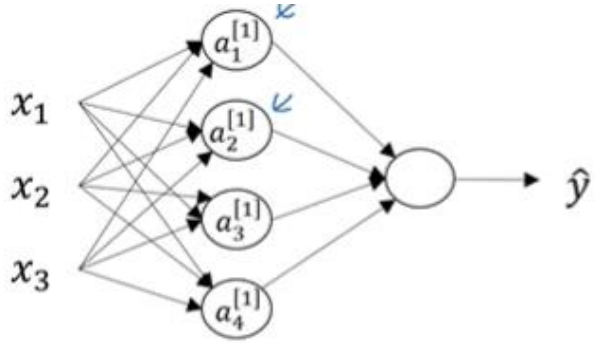
$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

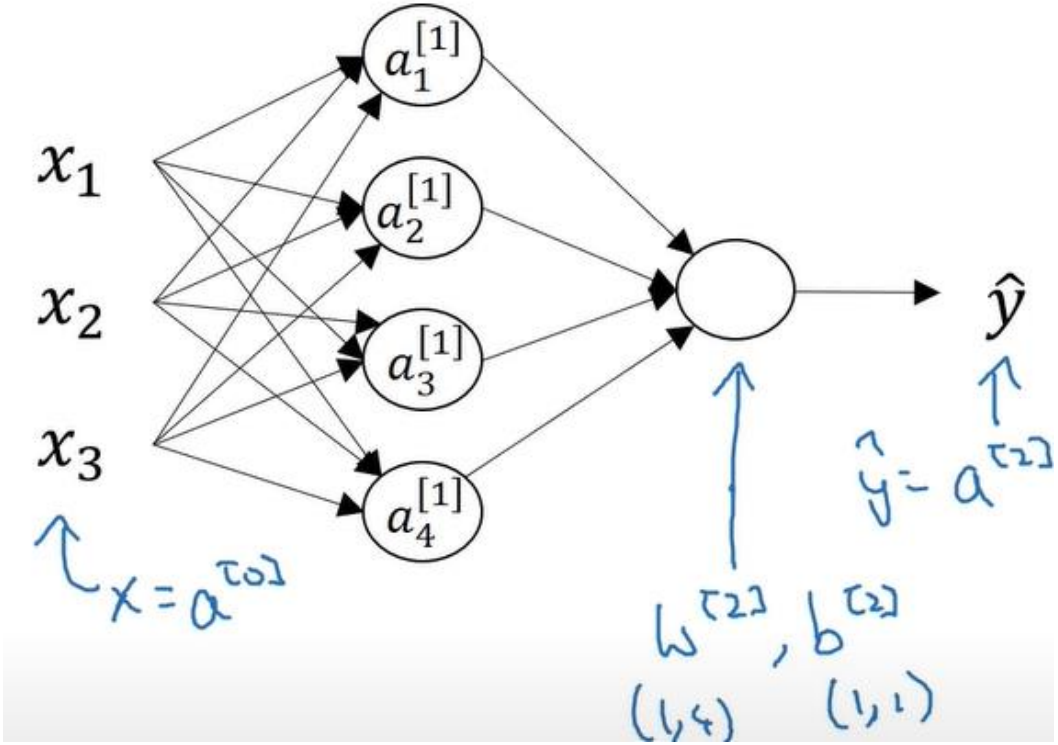
Neural Network Representation



Neural Network Representation



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$



Given input x:

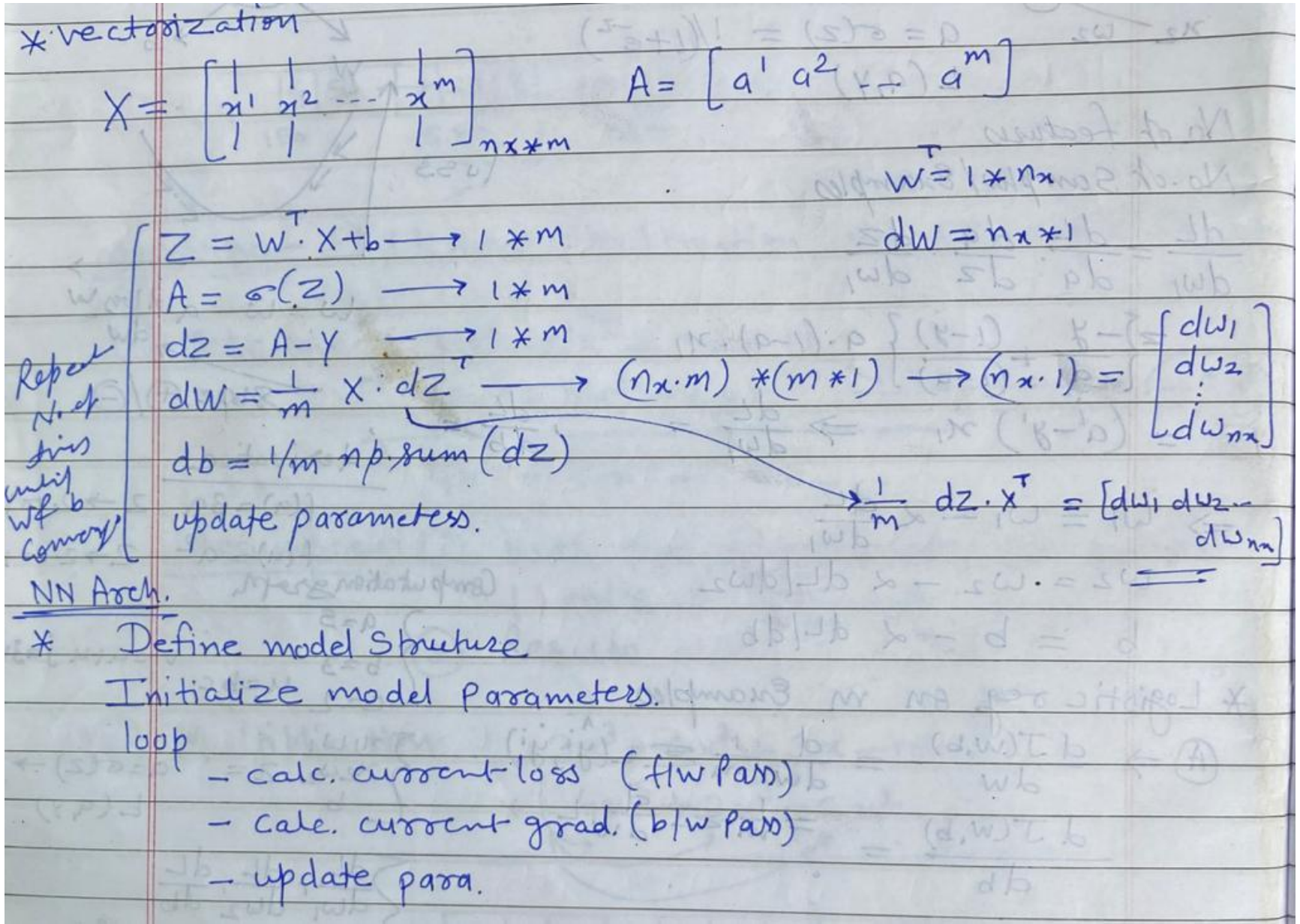
$$\rightarrow z^{[1]} = W^{[1]} a^{[0]} + b^{[1]} \quad (4,1)\ (4,3)\ (3,1)\ (4,1)$$

$$\rightarrow a^{[1]} = \sigma(z^{[1]}) \quad (4,1)\ (4,1)$$

$$\rightarrow z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \quad (1,1)\ (1,4)\ (4,1)\ (1,1)$$

$$\rightarrow a^{[2]} = \sigma(z^{[2]}) \quad (1,1)\ (1,1)$$

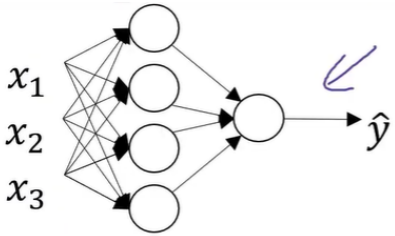
Vectorization



Vectorization

```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$



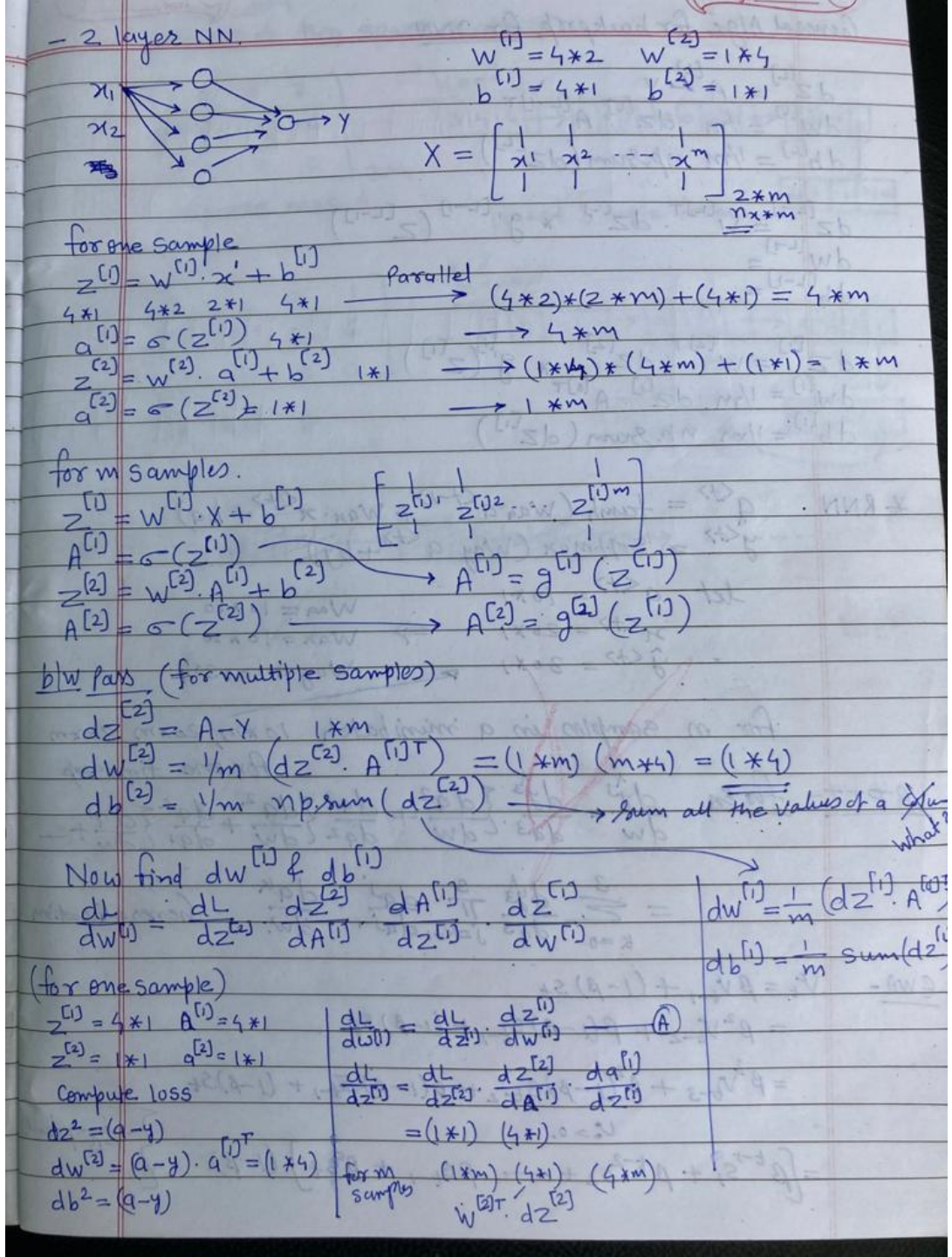
$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

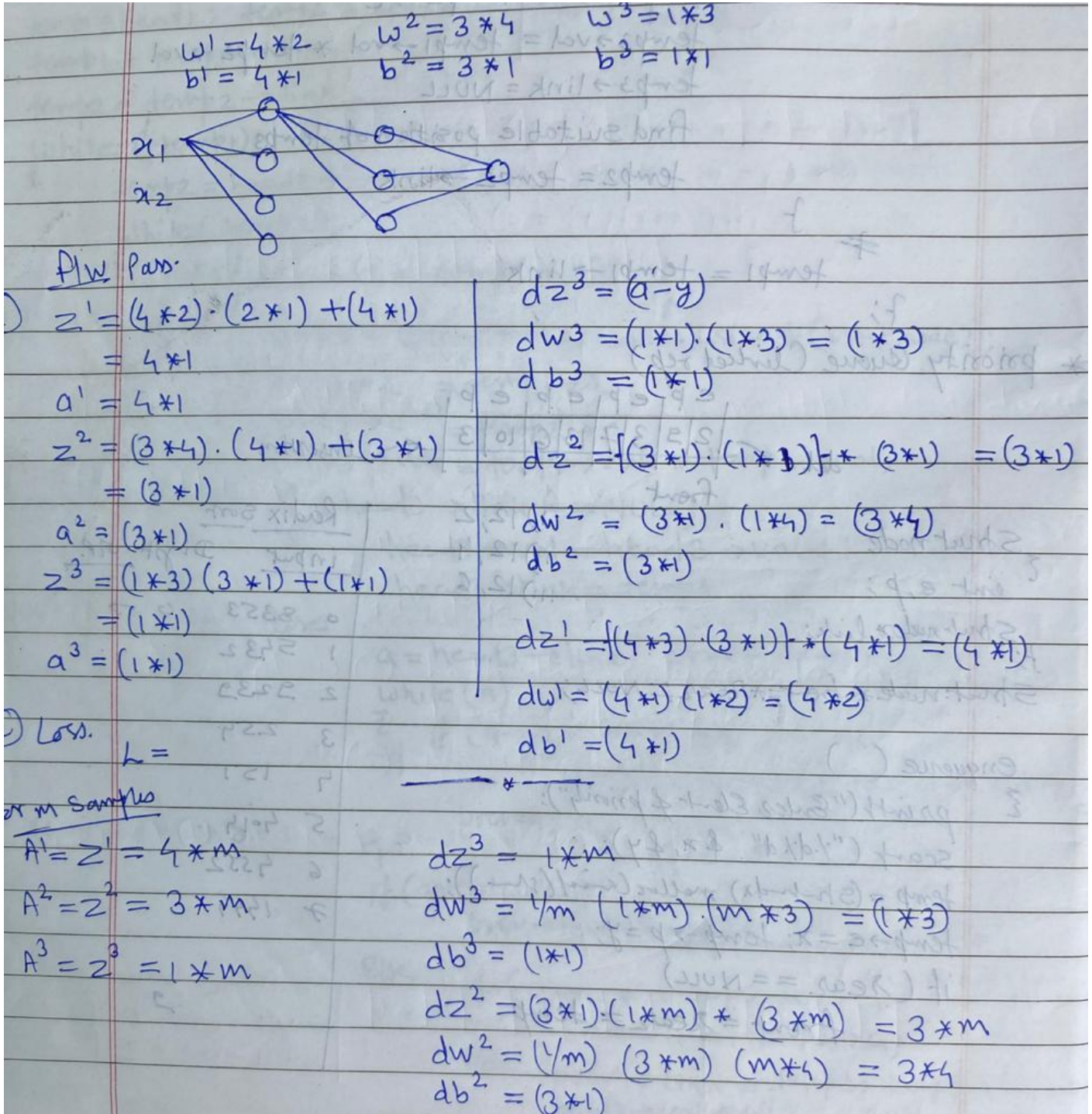
```
for i = 1 to m
```

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

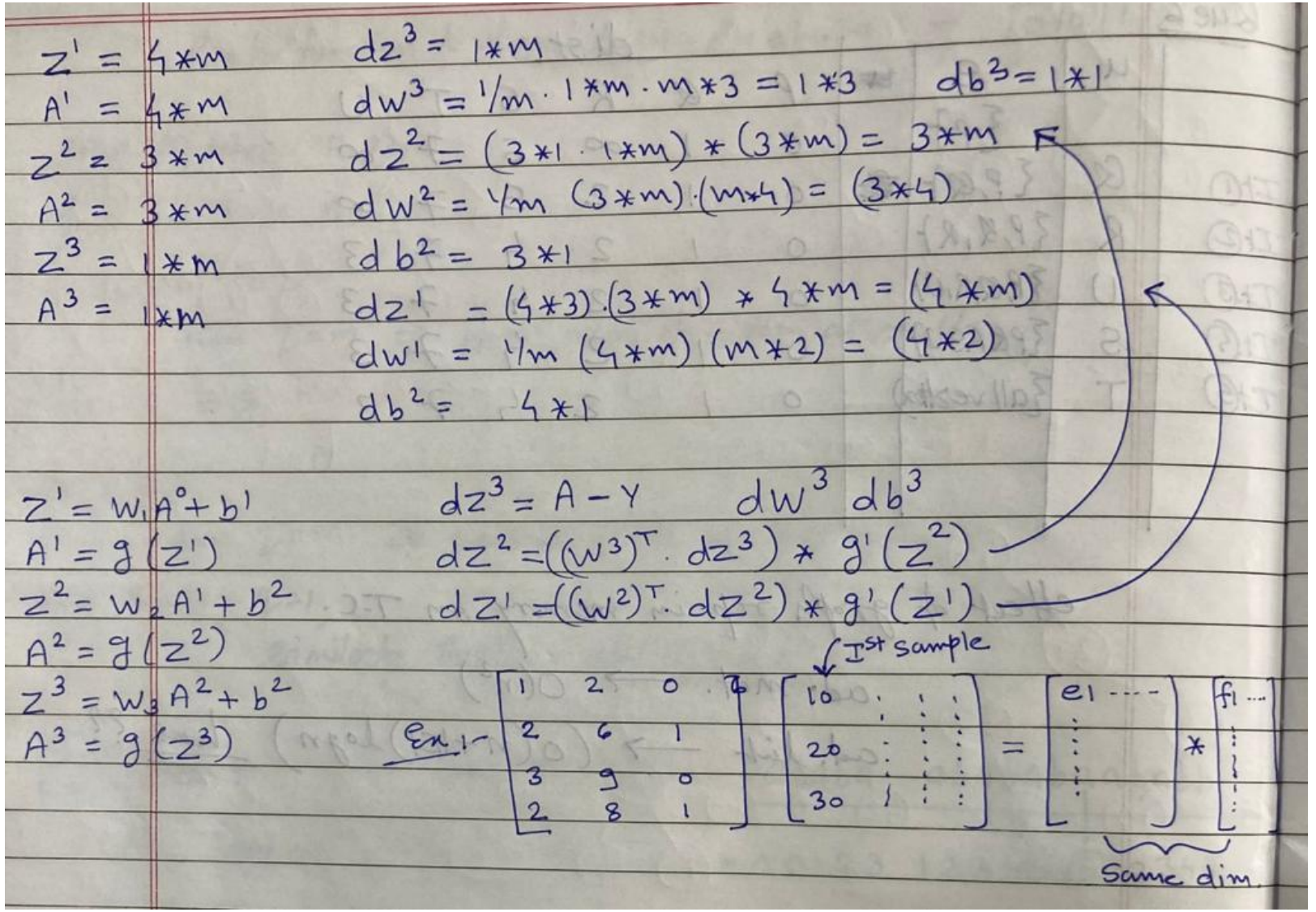
$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$



Vectorization



Vectorization



Summarization of Backpropagation

$$dZ^{[L]} = A^{[L]} - Y$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]^T}$$

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

$$dZ^{[L-1]} = W^{[L]^T} dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$$

$$\vdots$$

$$dZ^{[1]} = W^{[2]^T} dZ^{[2]} * g'^{[1]}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]^T}$$

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

Gradient Descent Algorithm

1. **Batch Gradient Descent**
2. **Stochastic Gradient Descent**
3. **Mini Batch Gradient Descent**

Batch Gradient Descent

Batch Gradient Descent: Batch Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data.

- **(Batch) Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost += compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

Stochastic Gradient Descent

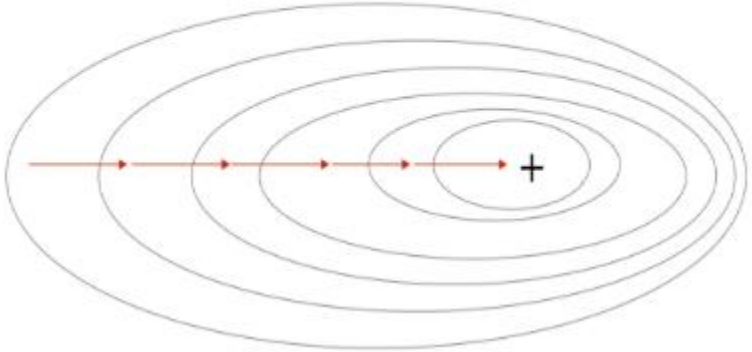
- **Stochastic Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost += compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

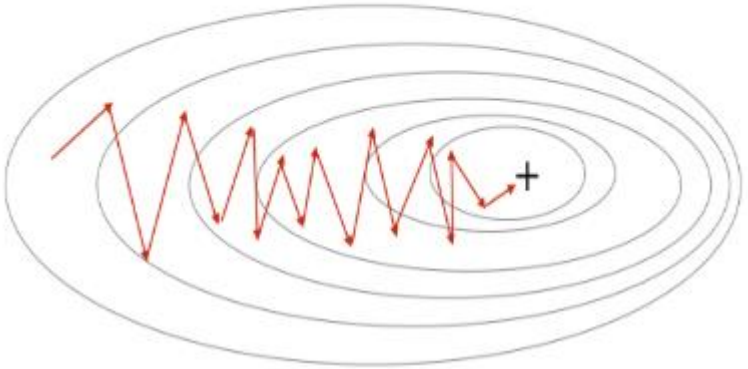
Stochastic Gradient Descent: Stochastic GD computes the gradients for each and every sample in the dataset and hence makes an update for every sample in the dataset.

GD, SGD, and MBGD

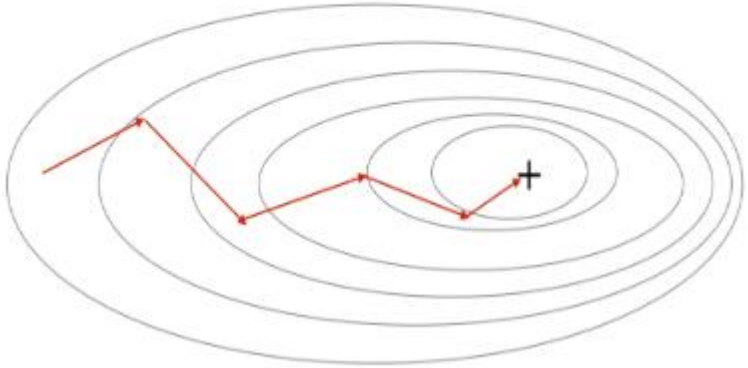
Gradient Descent



Stochastic Gradient Descent

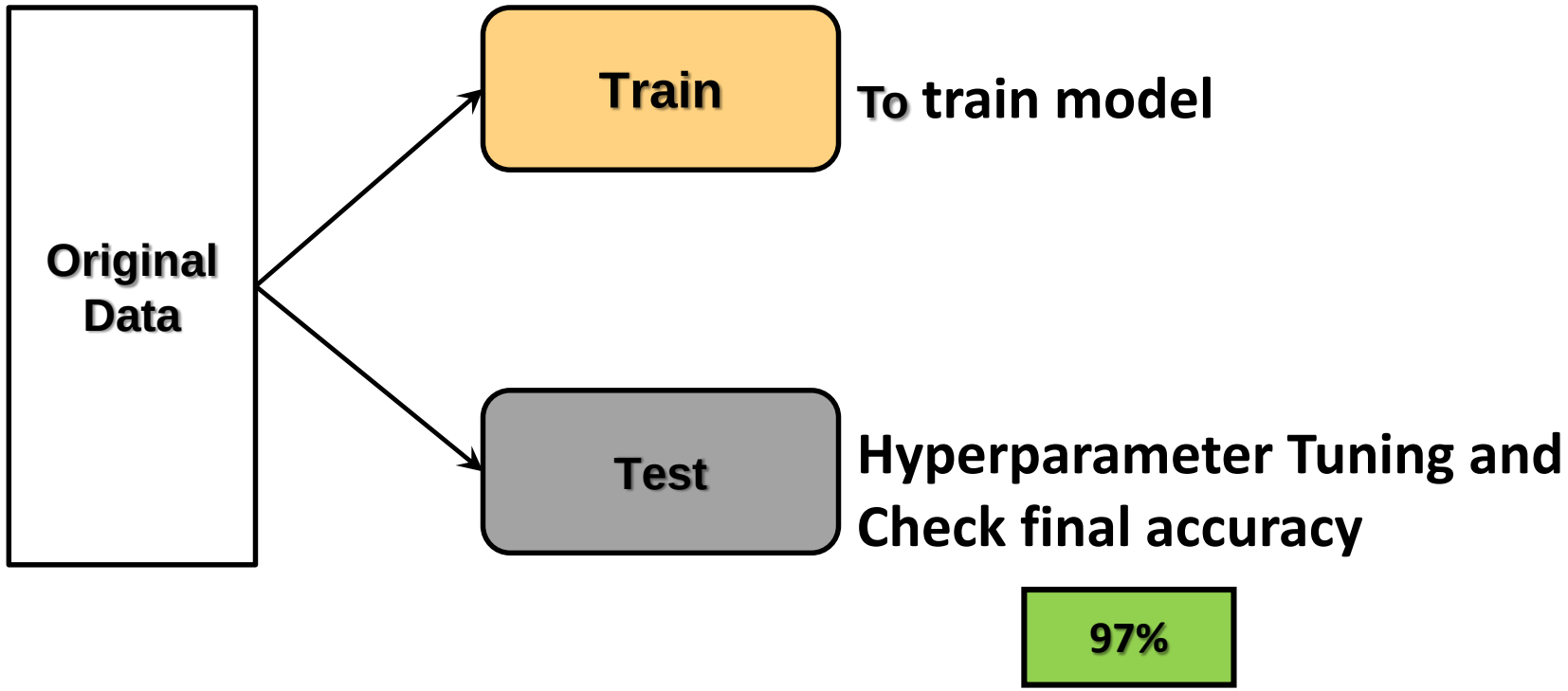


Mini-Batch Gradient Descent



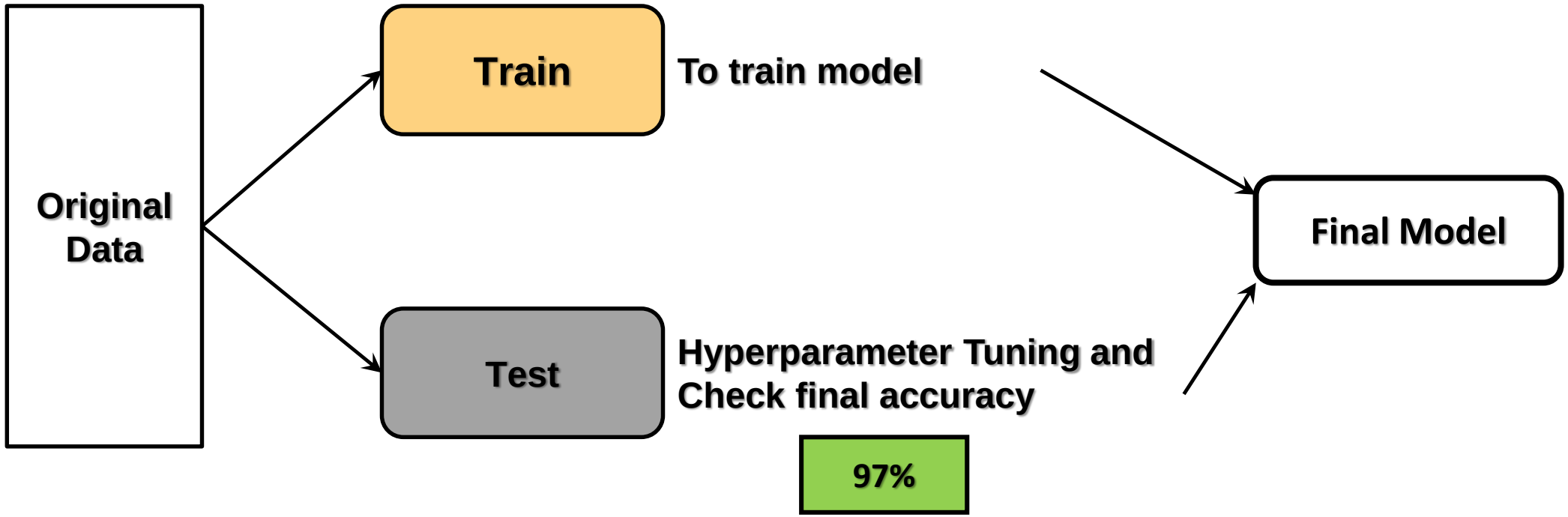
Cross Validation

How to test model performance?



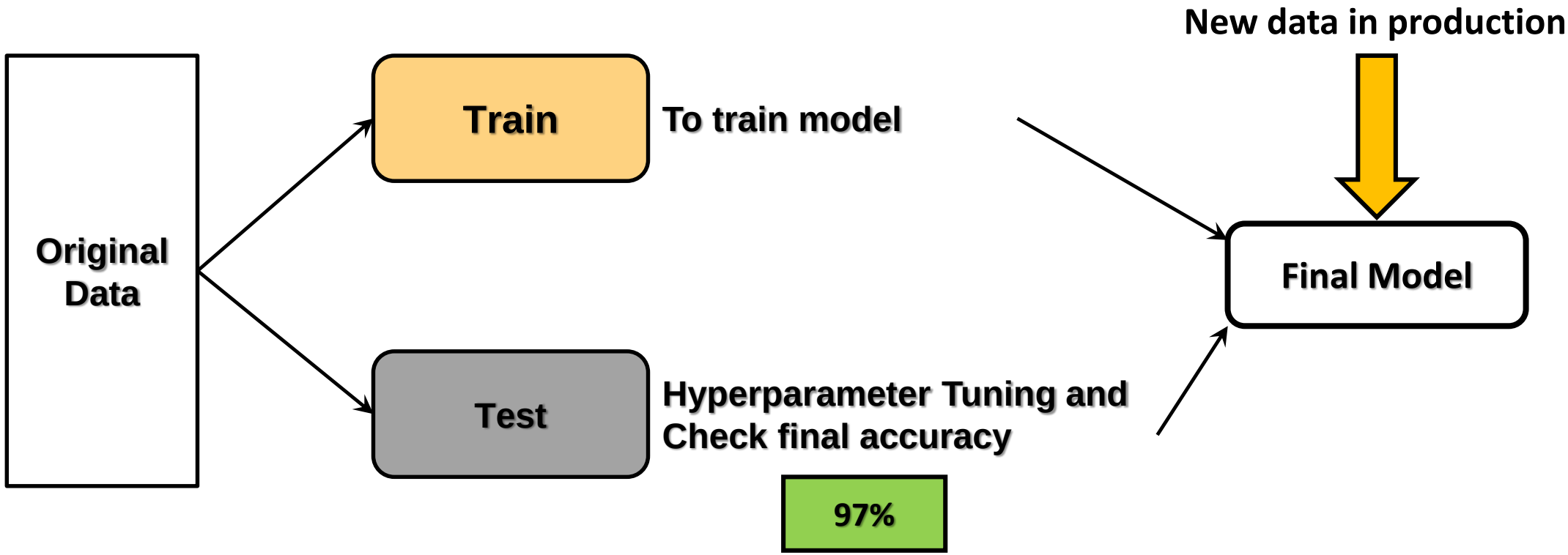
Cross Validation

How to test model performance?



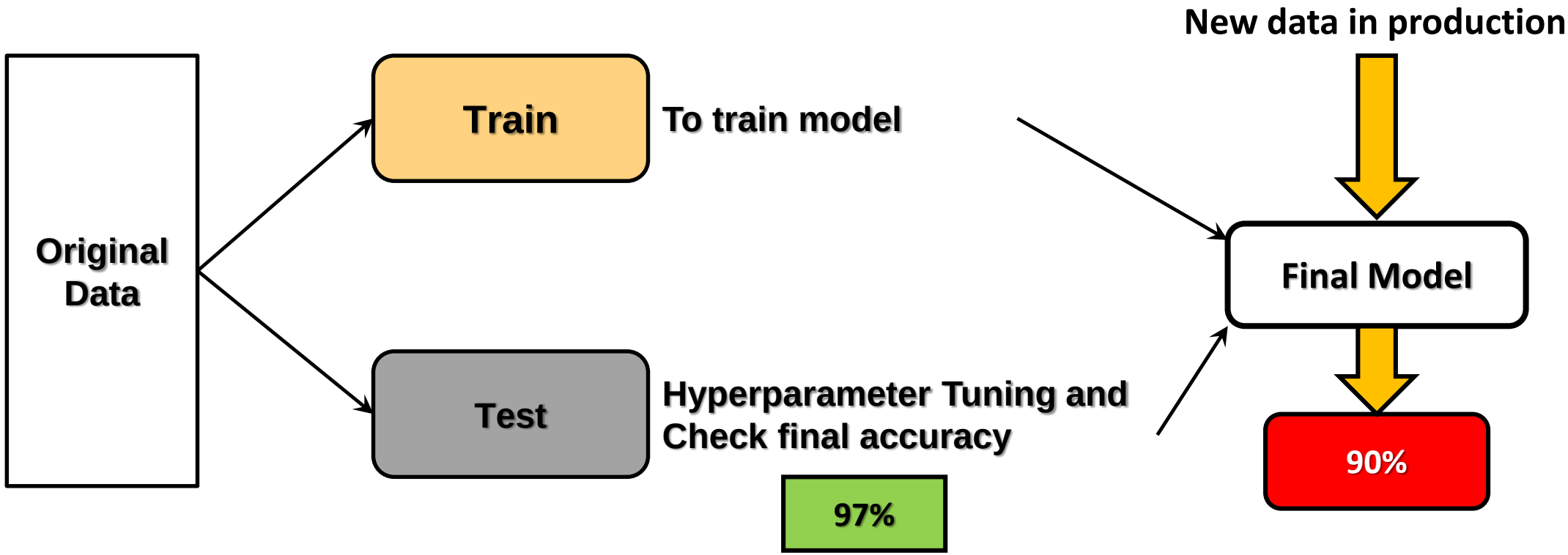
Cross Validation

How to test model performance?

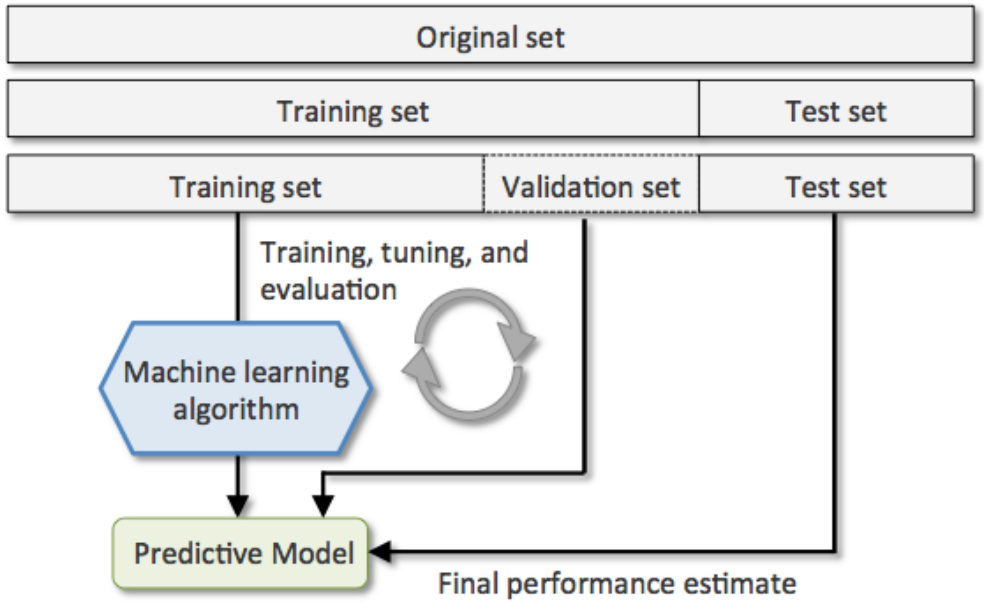


Cross Validation

How to test model performance?



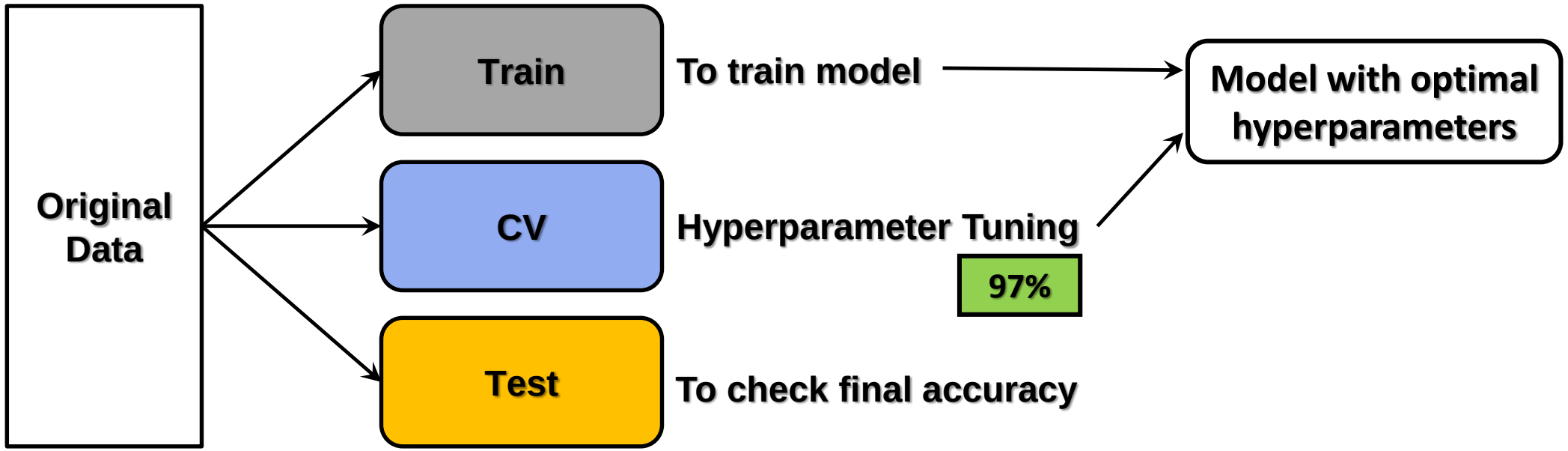
Cross Validation



- During Experimentation accuracy was **97%**
- In production accuracy is reduced to **90%**
- Tuning of hyperparameter and finding accuracy on **same data**
- Data was **seen** to model during hyperparameter tuning
- **Same data** used to test accuracy
- Solution is to **reserve** part of data as **unseen** during training as well as hyperparameter tuning
- Instead of two partitions make three partitions as **Train, CV** and **Test**

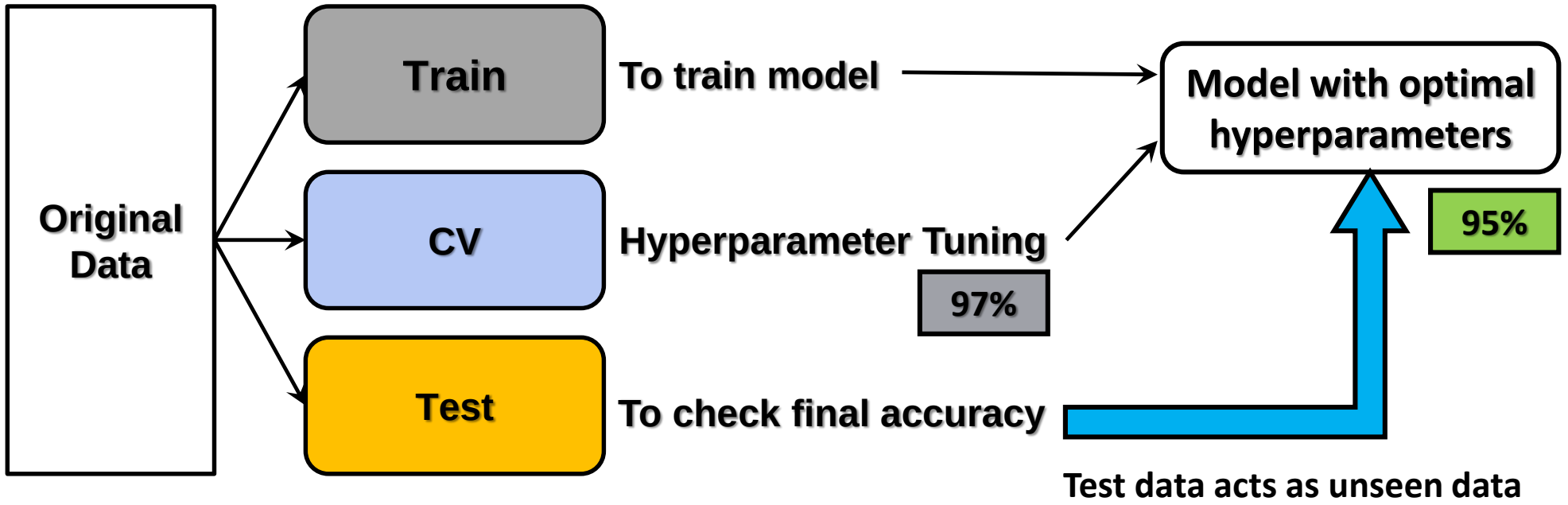
Cross Validation

Hyperparameter tuning

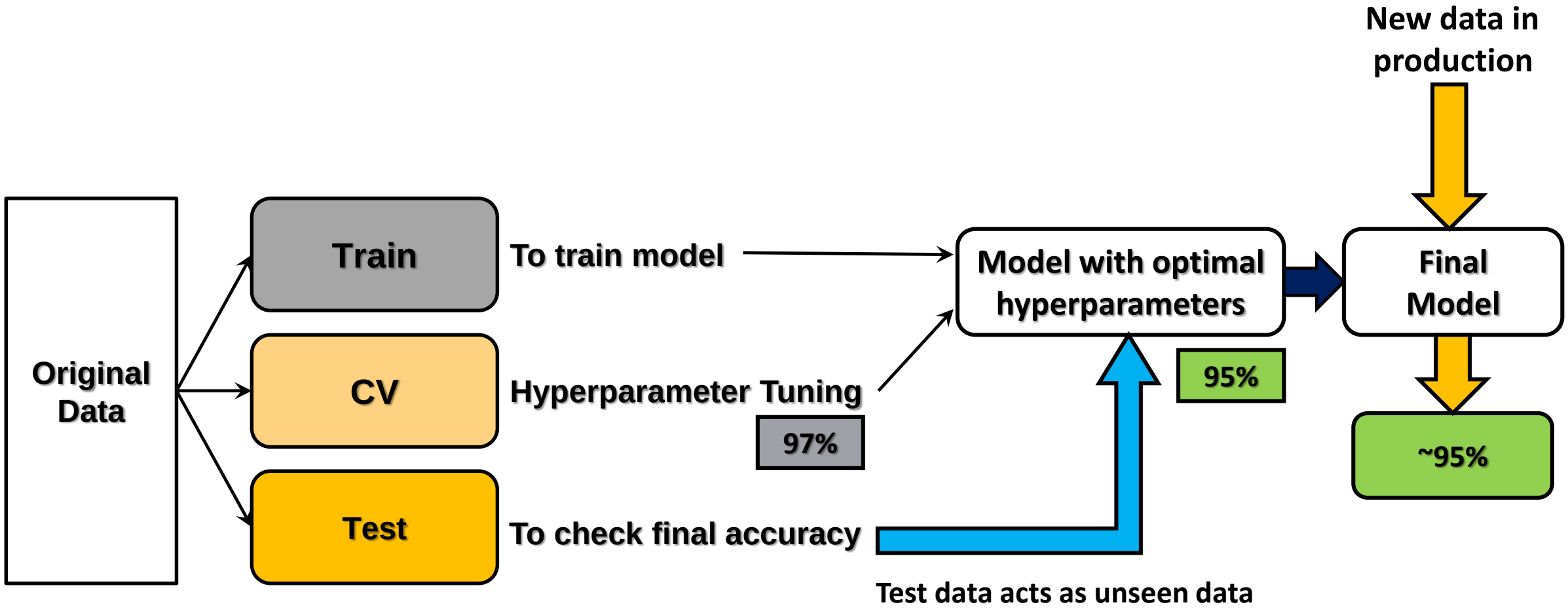


Cross Validation

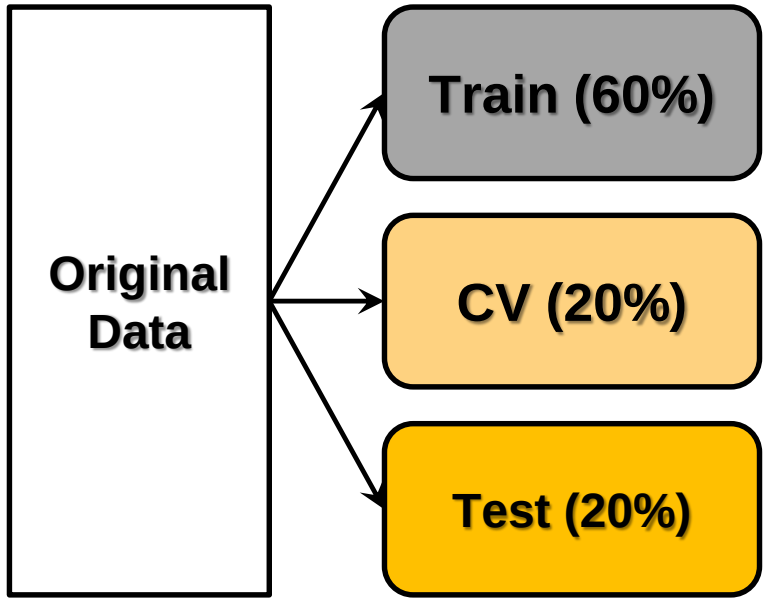
Hyperparameter tuning



Hyperparameter tuning

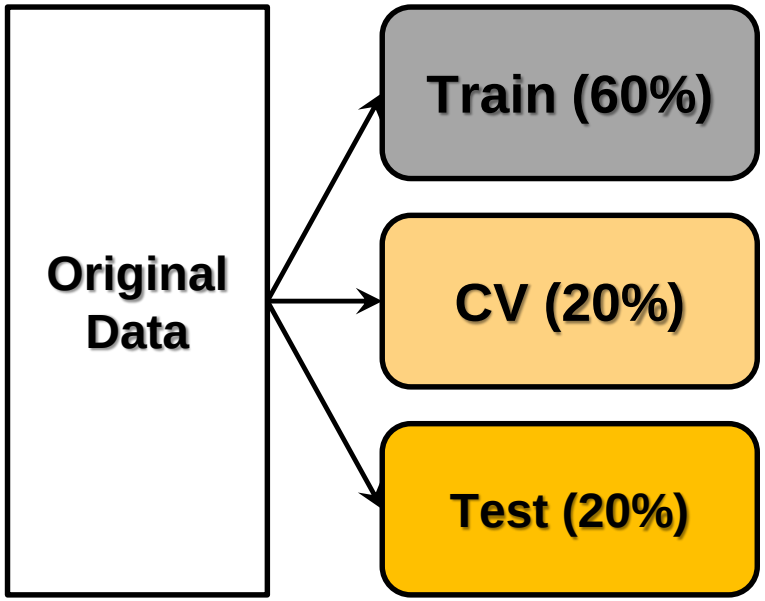


Cross Validation



40% of data is wasted in just tuning hyperparameters and for testing the model

K-Fold cross validation



40% of data is wasted in just tuning hyperparameters and for testing the model

Solution K-Fold cross validation

Bias

- **Bias** is the difference between the average prediction of our model and the correct value which we are trying to predict.

- Model with high bias pays very little attention to **the training data** and oversimplifies the model.

- It always leads to high error on training and test data.

Low and High Bias

- **Low Bias**: Suggests less assumptions about the form of the target function.
- **High-Bias**: Suggests more assumptions about the form of the target function.
- **Low-bias** ML Algorithms: Decision Trees, k-Nearest Neighbors and SVM. (Non-Linear algorithms)
- **High-bias** ML Algorithms: Linear Regression, LDA and Logistic Regression. (Linear algorithms)

Variance

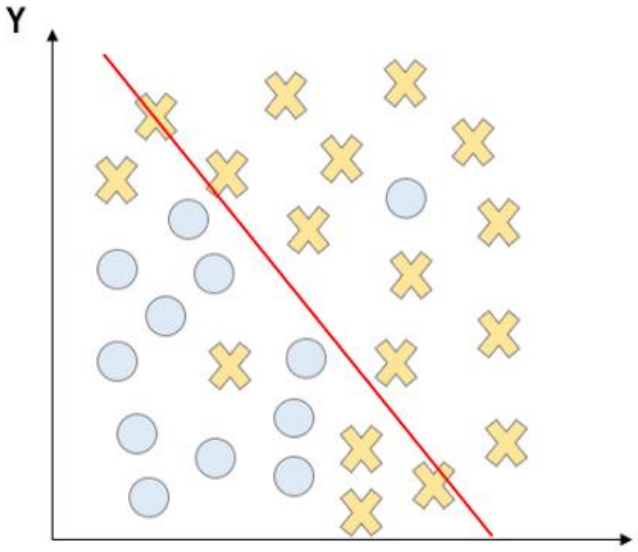
- **Variance** is the change in prediction accuracy of ML model between training data and test/validation data.

- Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn’t seen before.

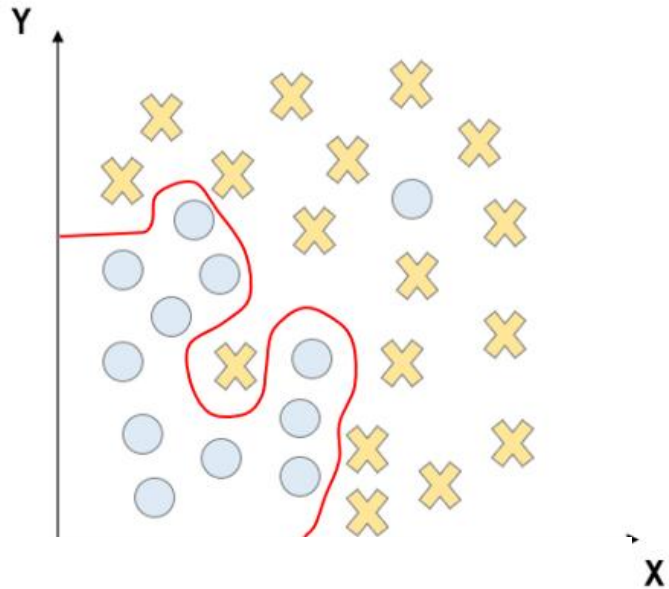
Low and High Variance

- **Low Variance**: Suggests small changes to the estimate of the target function with changes to the training dataset.
- **High Variance**: Suggests large changes to the estimate of the target function with changes to the training dataset.
- **Low-variance** ML Algorithms: Linear Regression, LDA and Logistic Regression.
- **High-variance** ML Algorithms: Decision Trees, k-Nearest Neighbors and Support Vector Machines.

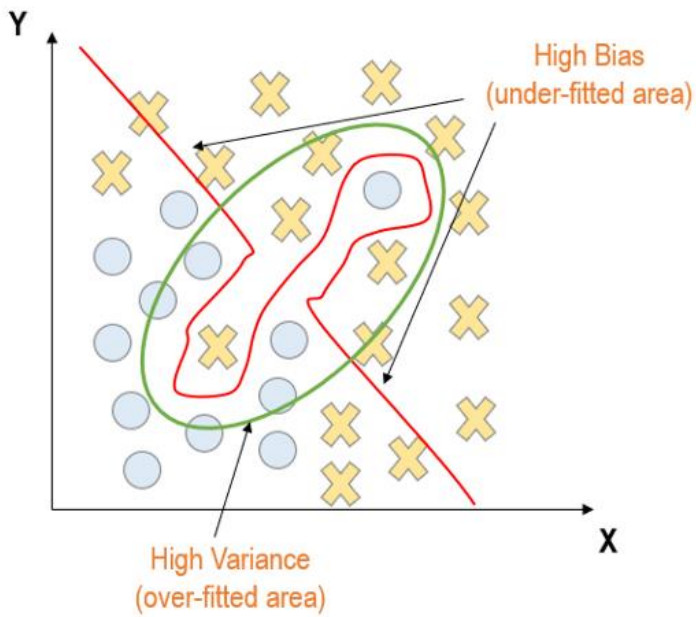
High Bias and Low Variance



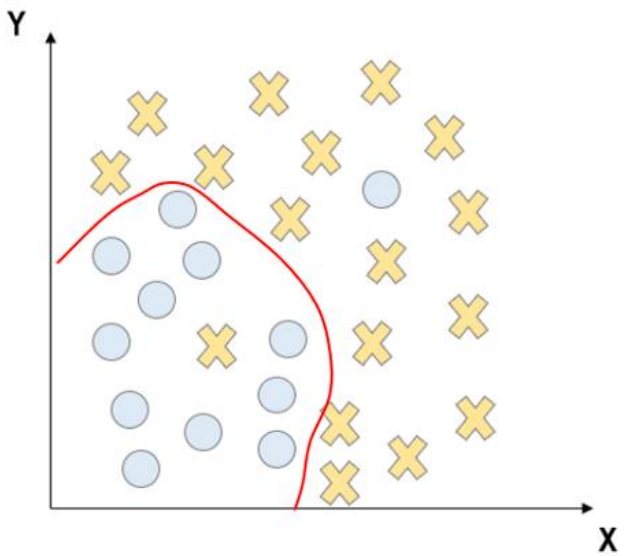
Low Bias and High Variance



High Bias and High Variance



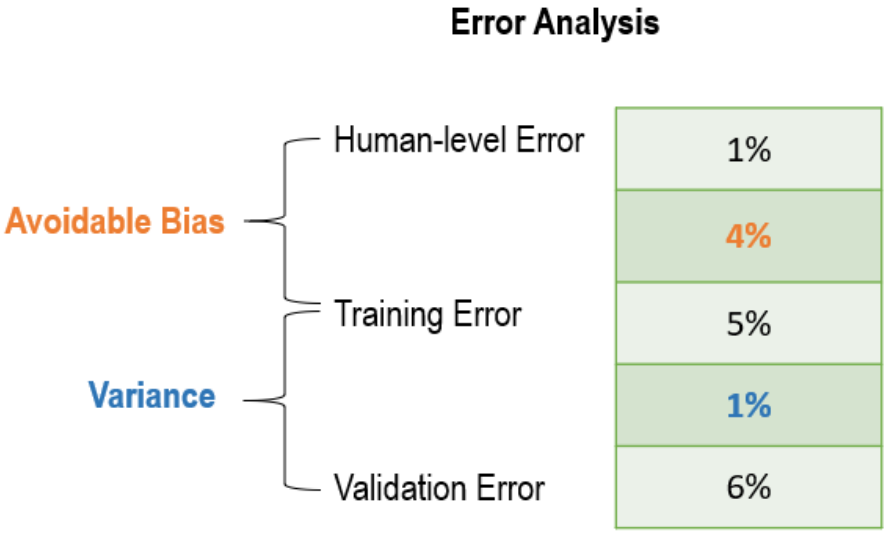
Low Bias and Low Variance

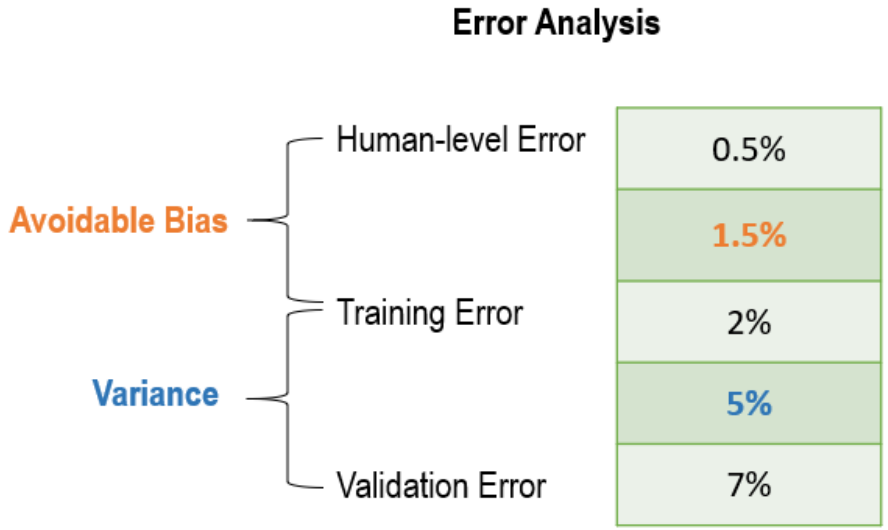


Several Important Points

- The basis for bias comparison is the assumption of human-level performance for that particular task (This is the best possible accuracy that a human expert can achieve on the given task.)
- It serves as a reference point to assess the model's performance.
- Model is a good fit if the model is performing at an equivalent level a human would perform.
- How do you measure human-level performance?
- The difference between your training error and human-level error is what is known as "avoidable bias".
- Human-level error is an approximation of Bayes error. In statistics, Bayes error is the lowest possible error for any classifier of a random outcome.

Several Important Points



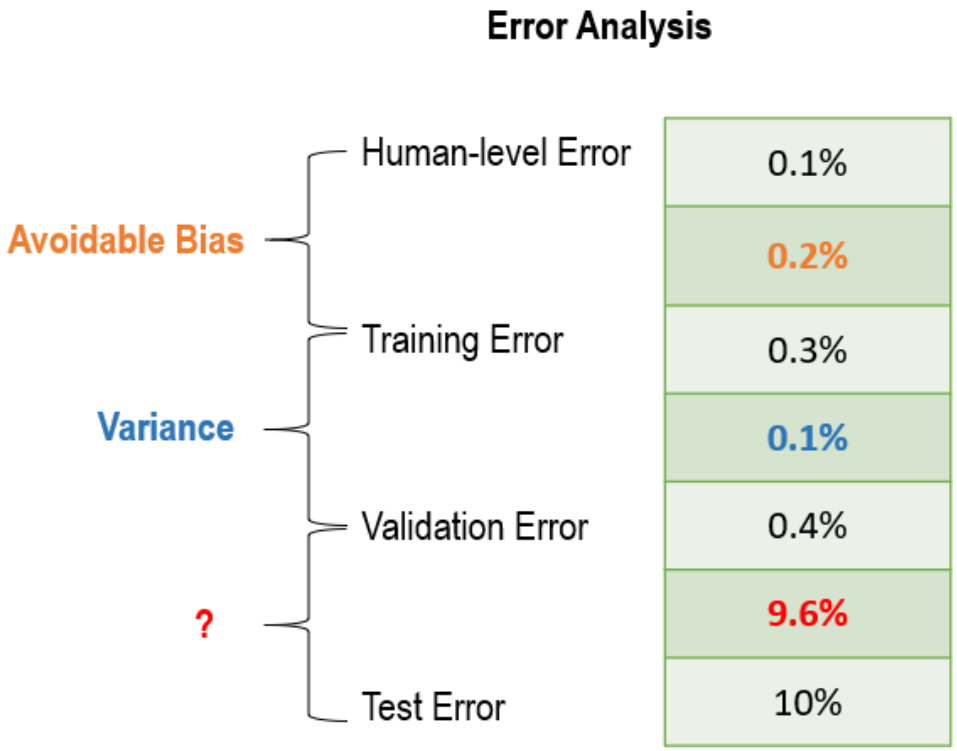


- Left Side (High Bias, Low Variance):
- The model struggles to learn from the data (underfitting issue).
- Validation error (6%) is close to training error (5%) → Low Variance

Right Side (Low Bias, High Variance)Training error (2%) is close to human-level error (0.5%) → Low Bias (Better Learning)Validation error (7%) is much higher than training error (2%) → High Variance (Overfitting)

- The goal is to balance bias and variance to optimize model performance.

Several Important Points

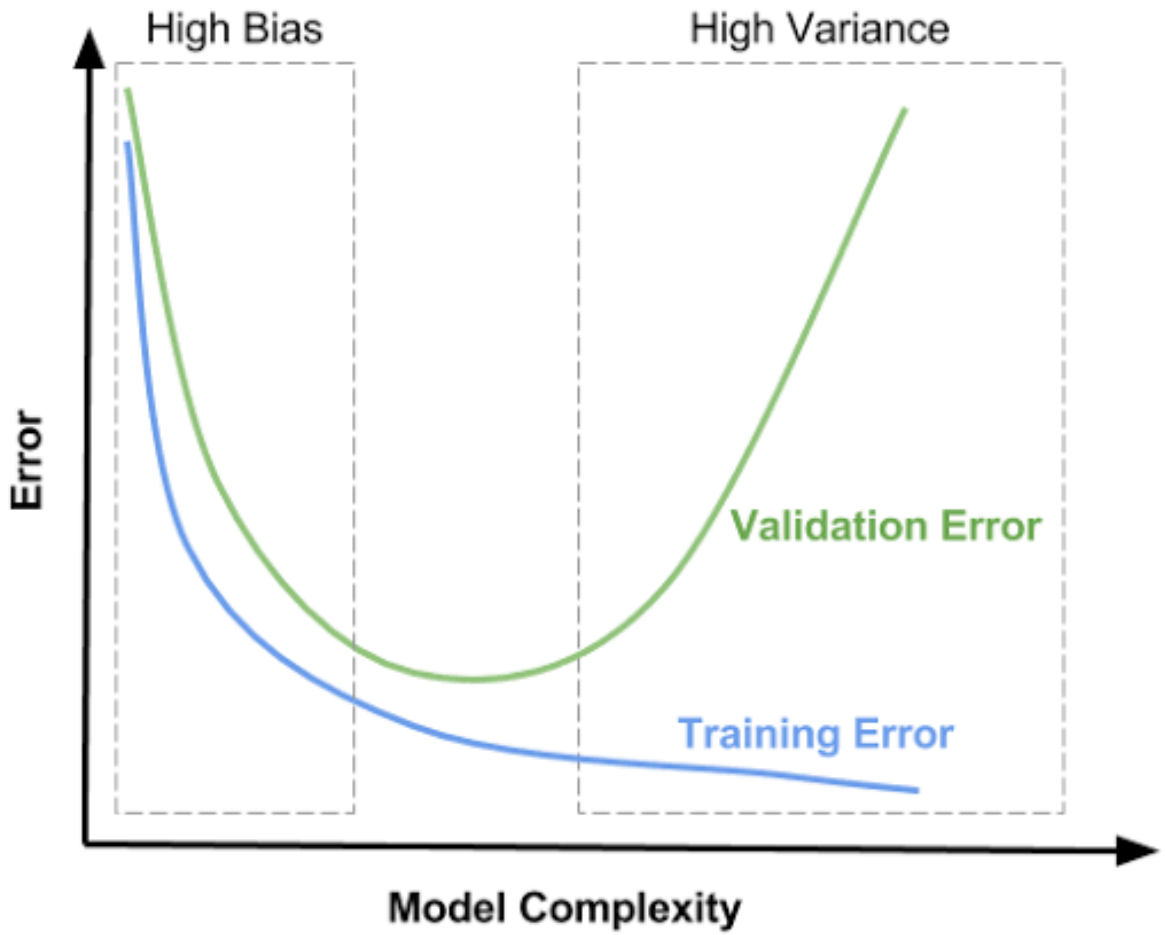


1. Does your test set come from the **same distribution** as your validation set? If you train a cat classifier, are your test set pictures of tigers?
2. Have you over-fitted on your validation set? Increase your validation set size. Remember to shuffle your data as well.
3. Your evaluation metric used during training/validation was not a good indicator. You will probably have to change the metric.

Bias Variance Trade-Off

- **Linear** machine learning algorithms often have a high bias but a low variance.
- **Nonlinear** machine learning algorithms often have a low bias but a high variance.
- Increasing the bias will decrease the variance.
- Increasing the variance will decrease the bias.

Bias Variance Trade-Off

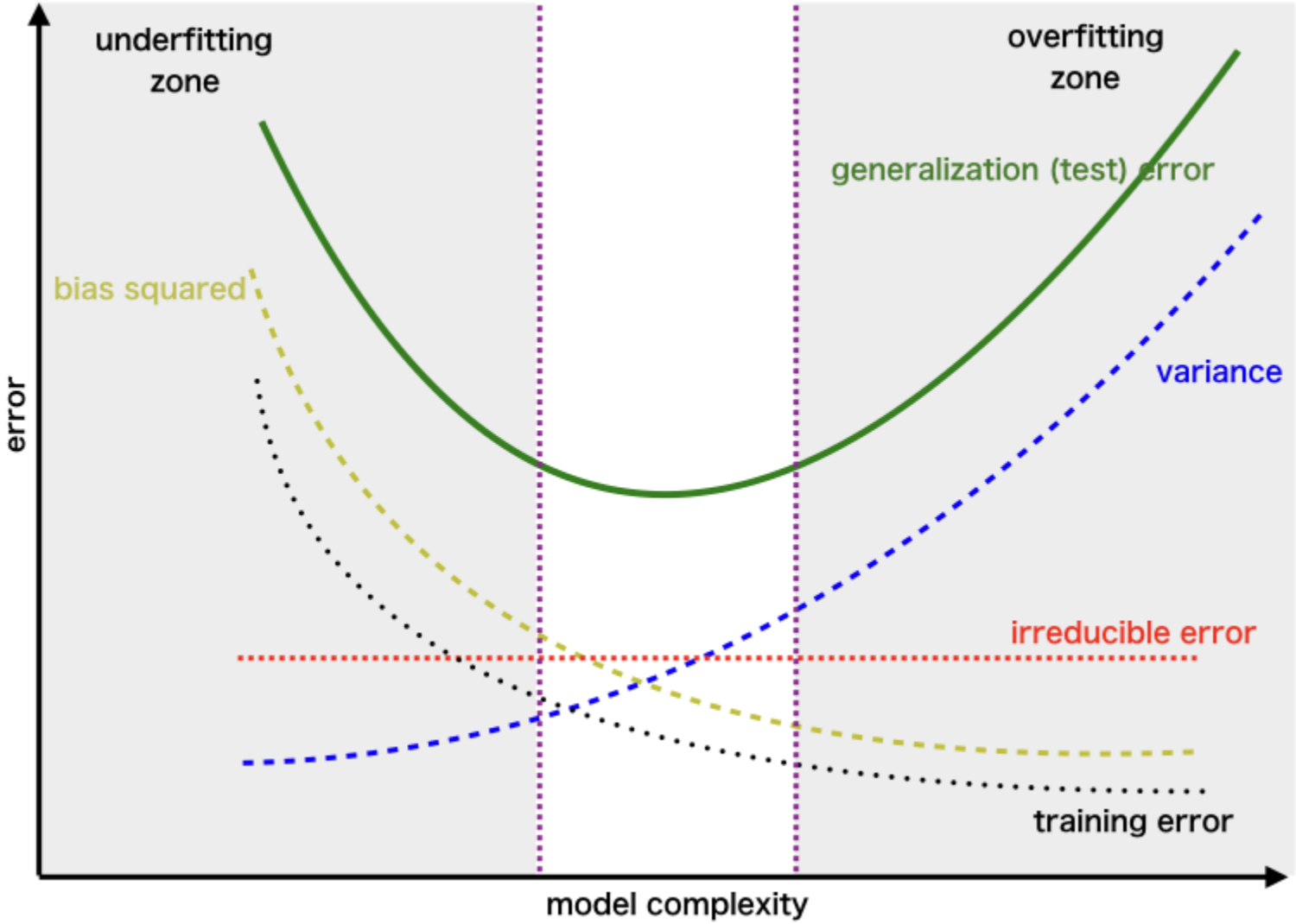


- The **left side** of the graph shows **high bias (underfitting)**, where both training and validation errors are high due to an overly simplistic model.

- The **right side** of the graph shows **high variance (overfitting)**, where training error is low, but validation error increases significantly due to poor generalization.

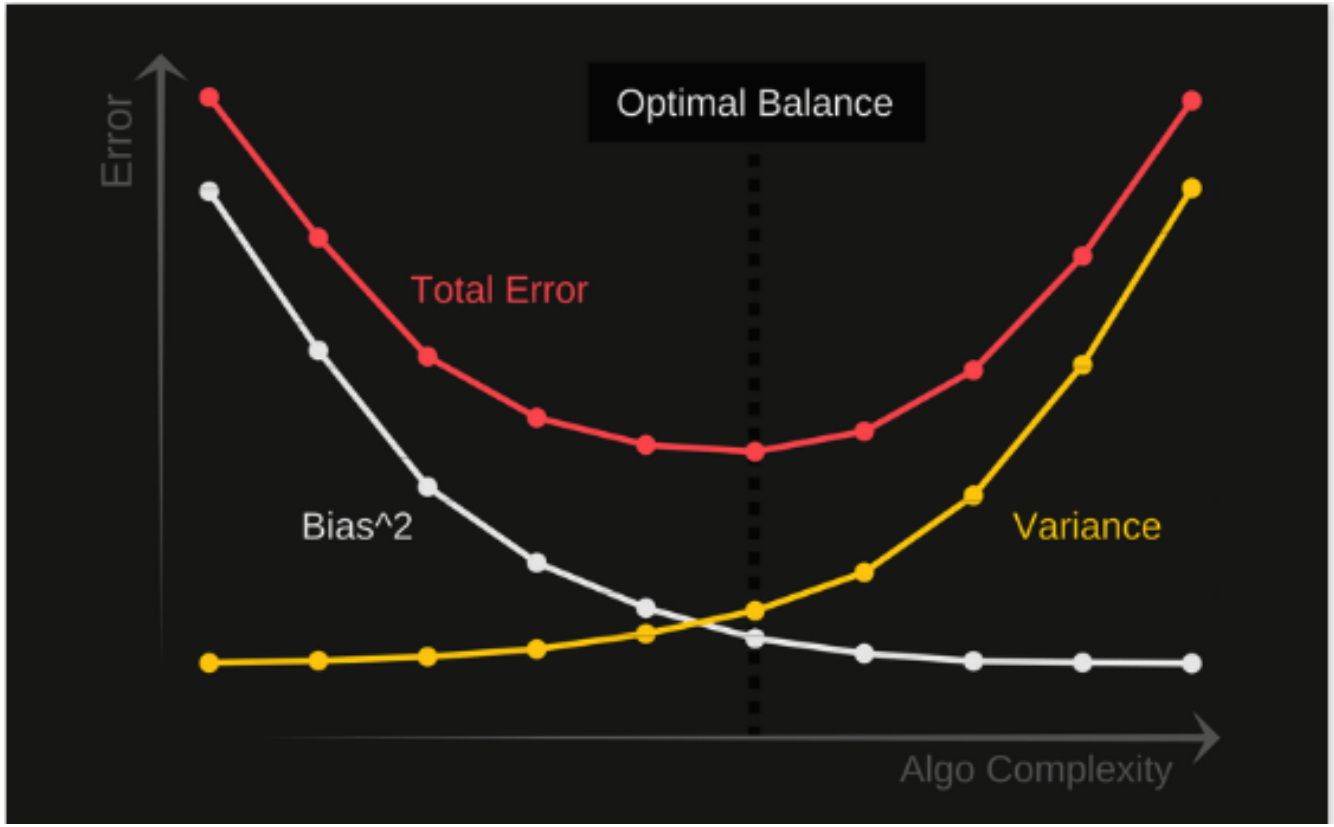
- The **optimal model complexity** is somewhere in the middle, where both training and validation errors are minimized.

Bias Variance Trade-Off



- High bias occurs in the **underfitting** zone (left side), where the model is too simple to capture data patterns.
- **Variance** measures how much predictions fluctuate with small changes in the training data.
- High variance occurs in the **overfitting** zone (right side), where the model is too complex and fits noise rather than patterns.
- **Training Error** Decreases as model complexity increases because a complex model can fit training data very well.
- **Irreducible Error** Represents noise or randomness in the data that no model can eliminate.

Total Error



Total Error (Red Line):The sum of bias² and variance.

•Too simple a model has high bias and low variance, leading to underfitting.

•Too complex a model has low bias but high variance, leading to overfitting.

•The best model is at the optimal balance, where total error is at its lowest.

- Optimal Balance: The ideal point where total error is minimized.
- A good tradeoff between bias and variance, avoiding both underfitting and overfitting.

The Reality

- We cannot calculate the real bias and variance error terms because we do not know the actual underlying target function.

- bias and variance provide the tools to understand the behavior of machine learning algorithms in the pursuit of predictive performance.

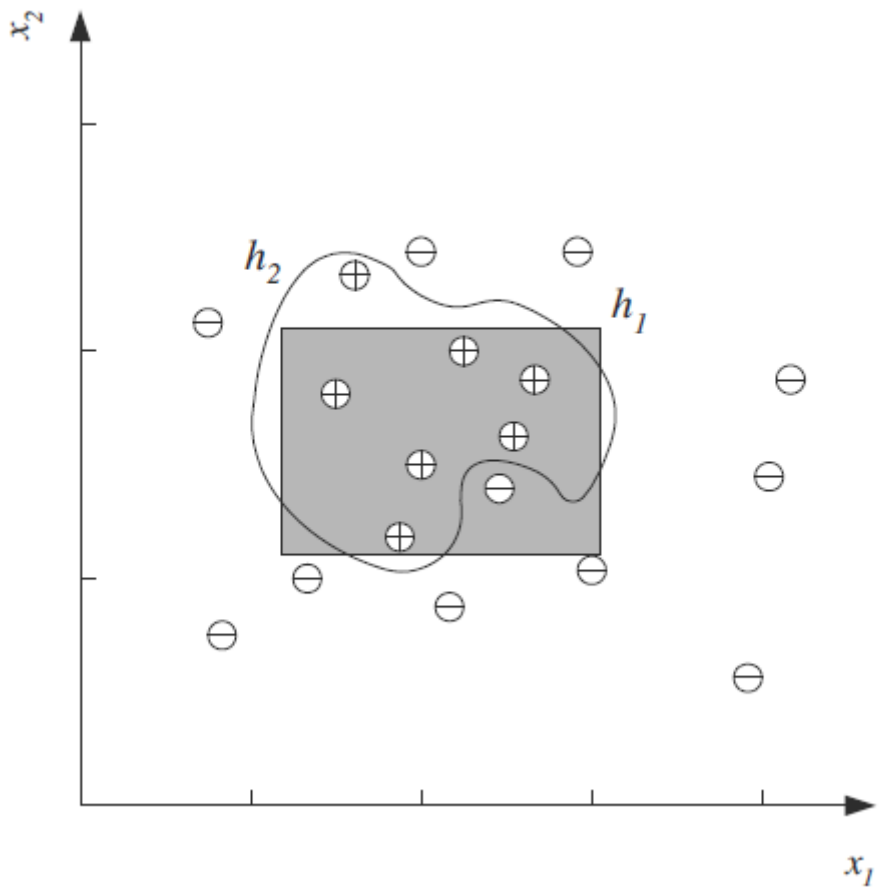
- Irreducible error is the error that can’t be reduced by creating good models. It is a measure of the amount of noise in our data.

Noise

- *Noise* is any unwanted anomaly in the data and due to noise, the class may be more difficult to learn and zero error may be infeasible with a simple hypothesis class.

1. Imprecision in recording the input attributes, which may shift the data points in the input space.
2. Errors in labeling the data points. This is sometimes called ***teacher noise***.
3. There may be additional attributes, which we have not taken into account, that affect the label of an instance.

Noise



Simple vs. Complex Hypothesis

- Rectangle can be defined by **four numbers**, but to define a more complicated shape one needs a more complex model with a much larger number of parameters.

- Using the simple rectangle makes more sense because of the following:

➢ It is a simple model to use. It is easy to check whether a point is inside or outside a rectangle.

➢ It is a simple model to train and has fewer parameters. It is easier to find the corner values of a rectangle.

➢ It is a simple model to explain. A rectangle simply corresponds to defining intervals on the two attributes.

Simple vs. Complex Hypothesis

- If indeed there is mislabeling or noise in input and the actual class is really a simple model like the rectangle, then the simple rectangle, because it has less variance and is less affected by single instances, will be a better discriminator than the wiggly shape.

- *Occam's razor*: simpler explanations are more plausible and any unnecessary complexity should be shaved off.

Regression

- When the output is a numeric value, what we would like to learn is not a class, $C(\boldsymbol{x}) \in \{0, 1\}$, but is a numeric function.

$$\mathcal{X} = \{\boldsymbol{x}^t, r^t\}_{t=1}^N$$

where $r^t \in \Re$. If there is no noise, the task is *interpolation*. We would like to find the function $f(x)$ that passes through these points such that we have

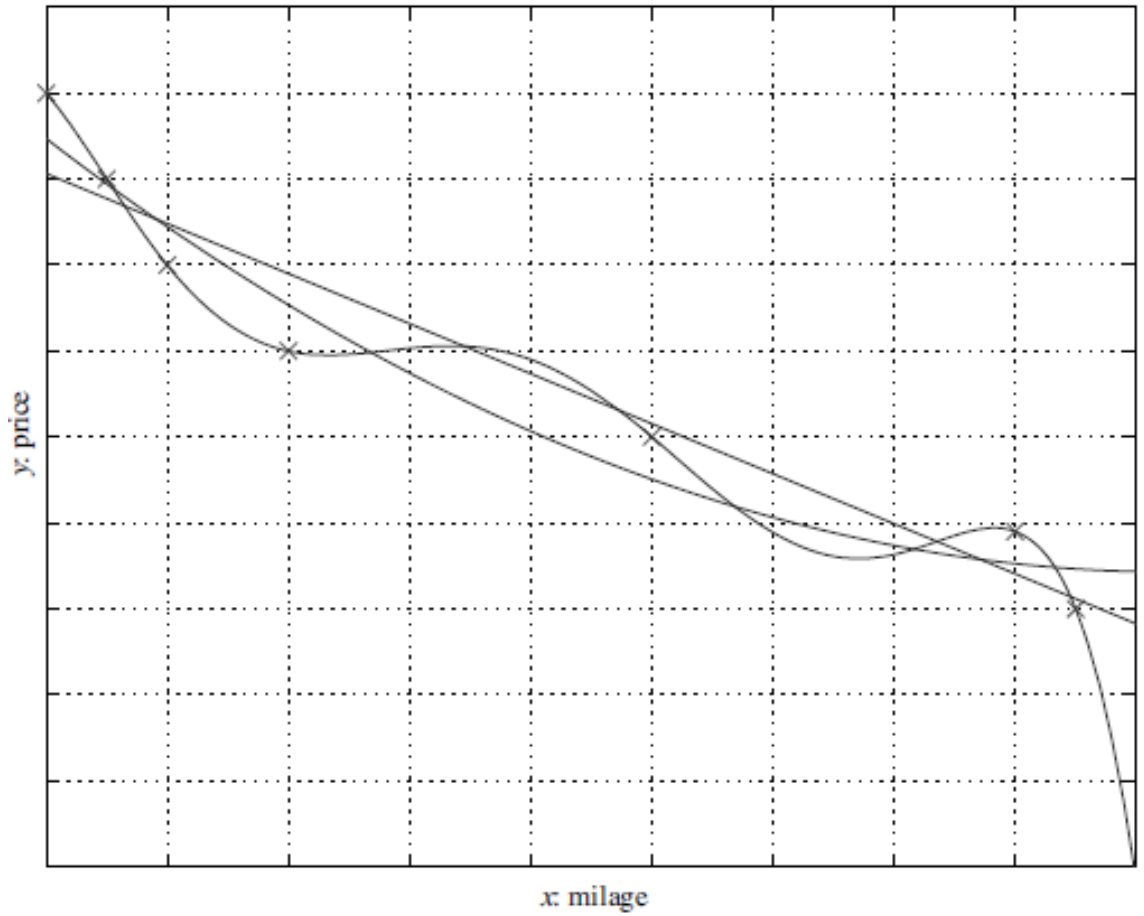
$$r^t = f(\boldsymbol{x}^t)$$

Interpolation vs. Extrapolation

- **Interpolation** is guessing data points that fall within the range of the data you have, i.e. between your existing data points.
- **Extrapolation** is guessing data points from beyond the range of your data set.

- In *polynomial interpolation*, given N points, we find the $(N-1)^{st}$ degree polynomial that we can use to predict the output for any $\boldsymbol{x}$.

- This is called extrapolation if $\boldsymbol{x}$ is outside of the range of $\boldsymbol{x}_t$ in the training set.



High-order polynomial follows individual examples closely, instead of capturing the general trend.

Occam’s razor also applies in the case of regression.

Model Selection and Generalization

- Learning a Boolean function.
- With d inputs, the training set has at most 2^d examples.
- 2^{2^d} possible Boolean functions of d inputs.
- Each distinct training example removes half the hypotheses.

x_1	x_2	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}	h_{13}	h_{14}	h_{15}	h_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Model Selection and Generalization

- In the case of a Boolean function, to end up with a single hypothesis we need to see *all* 2^d training examples.
- After seeing N example cases, there remain 2^{2^d-N} possible functions. This is an example of an ***ill-posed problem*** where the data by itself is not sufficient to find a unique solution.
- Generalization
- We should make some extra assumptions to have a unique solution with the data we have. The set of assumptions we make to have learning possible is called the ***inductive bias*** of the learning algorithm.
- Examples in classification and regression
- Learning is not possible without inductive bias, and now the question is how to choose the right bias. This is called ***model selection***, which is choosing between possible H.

Overfitting and Underfitting

- How well a model trained on the training set predicts the right output for new instances is called ***generalization***.
- For best generalization, we should match the complexity of the hypothesis class H with the complexity of the function underlying the data.
- If H is less complex than the function, we have ***underfitting***, for example, when trying to fit a line to data sampled from a third-order polynomial.
- when fitting a sixth-order polynomial to noisy data sampled from a third-order polynomial. This is called ***overfitting***.

Overfitting and Underfitting

- In supervised learning, **underfitting** happens when a model unable to capture the underlying pattern of the data.
- These models usually have high bias and low variance.
- It happens when we have very less amount of data to build an accurate model or when we try to build a linear model with a nonlinear data.
- Models are very simple to capture the complex patterns in data like Linear and logistic regression.

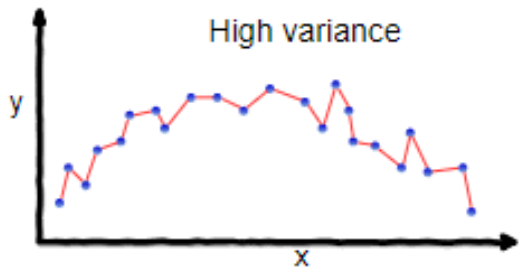
Overfitting and Underfitting

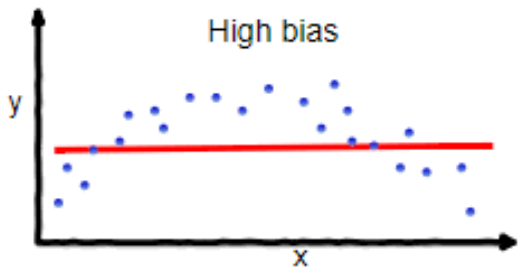
- In supervised learning, **overfitting** happens when our model captures the noise along with the underlying pattern in data.
- It happens when we train our model a lot over noisy dataset.
- These models have low bias and high variance. These models are very complex like Decision trees which are prone to overfitting.
- The data is simply noisy, that is the model is neither overfitting nor underfitting, and the high MSE is simply due to the amount of noise in the dataset.

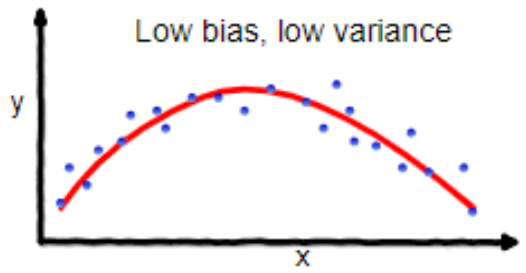
Overfitting and Underfitting

- Tension between optimization and generalization.
- *Optimization* refers to the process of adjusting **a model to get the best performance** possible on the training data.
- G*eneralization* refers to how well the trained model performs on data it has never seen before.
- It’s beginning to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data.
- Simpler models are less likely to overfit than complex ones.

Overfitting and Underfitting







overfitting

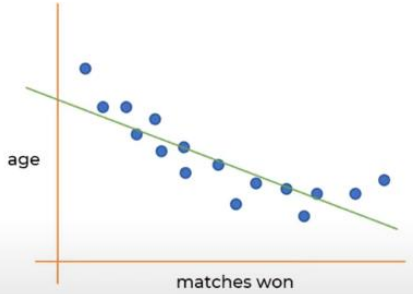
underfitting

Good balance

underfit

overfit

balanced fit

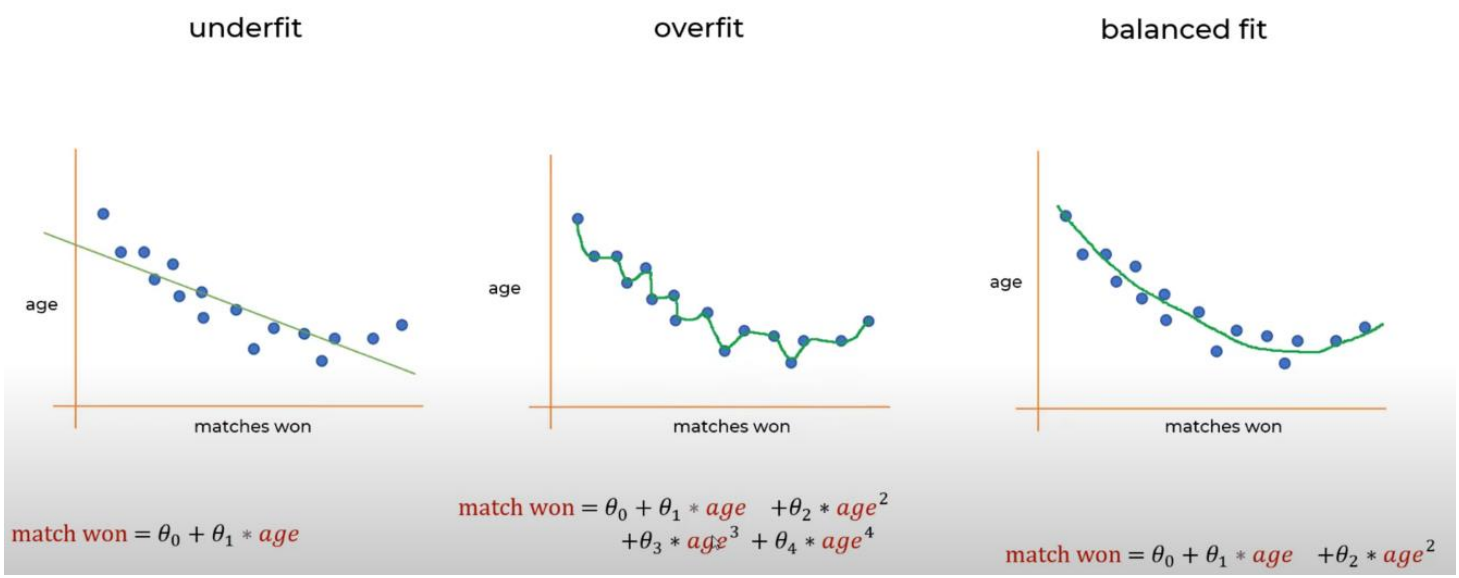


$\text{match won} = \theta_0 + \theta_1 * age$

$\text{match won} = \theta_0 + \theta_1 * age + \theta_2 * age^2 + \theta_3 * age^3 + \theta_4 * age^4$

$\text{match won} = \theta_0 + \theta_1 * age + \theta_2 * age^2$

No. of Parameters and Bias Variance



- If our model is too simple and has very few parameters then it may have high bias and low variance.
- If our model has large number of parameters then it's going to have high variance and low bias.
- So we need to find the right/good balance without overfitting and underfitting the data.

triple trade-off

1. the complexity of the hypothesis we fit to data, namely, the capacity of the hypothesis class,

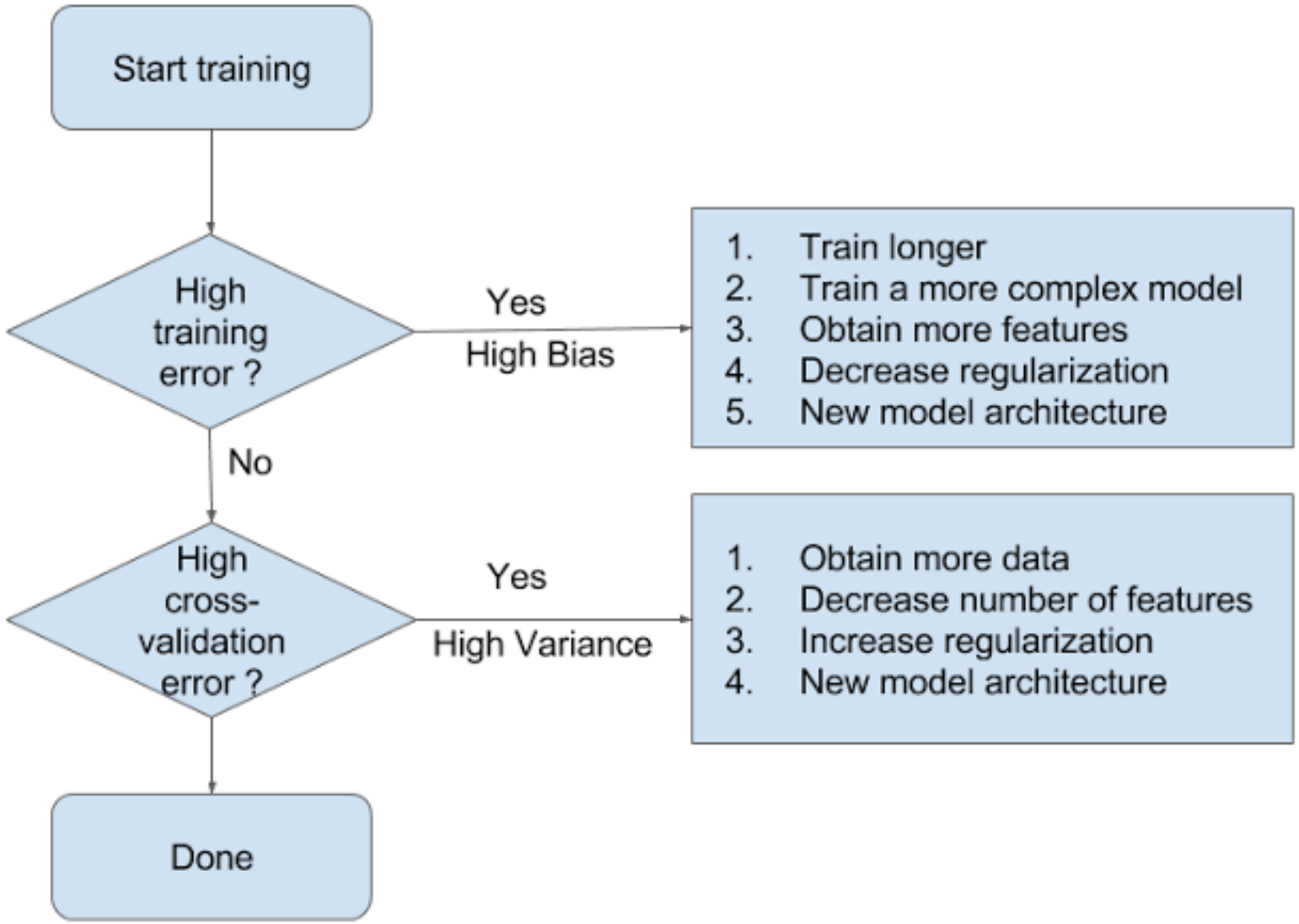
2. the amount of training data, and

3. the generalization error on new examples.

The Approach- (Bigger Network with more data)

Training Error	Valid Error	Cause	Solution
High	High	High bias	- Increase model complexity - Train for more epochs
Low	High	High variance	- Add more training data (e.g., **dataset augmentation**) - Use **regularization** - User **early stopping** (train less)
Low	Low	Perfect tradeoff	- You are done!

The Approach



Methods to avoid Over-fitting

- Cross-Validation
- Early Stopping
- Pruning
- Regularization

Regularization

- Technique to discourage the complexity of the model.
- As the degree of the input features increases the model becomes complex.

$$L(x, y) = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2$$

Regularization

- which degree will fit the data
- which features to penalize to get the best predictions for unseen data
- longer we train=more specialized the weights =overfitting the training data
- weights will grow in size in order to handle the specifics of the training examples
- small weights or even zero weights for less relevant or irrelevant inputs = focused learning

L1 Regularization

- LASSO is a form of regularization that penalizes the absolute magnitude of the coefficients in a regression model.
- Large weights in a neural network are a sign of a more complex network that has overfit the training data.
- This approach forces some weights to become exactly zero, effectively eliminating less important predictors from the model.
- When the optimization process minimizes the loss function, it often lands at a **corner of the diamond** where one or more coefficients are exactly zero.
- This alignment enables LASSO to perform **feature selection**, driving some coefficients to zero.

L1 Regularization or Lasso or L1 norm

- In Sparse solution majority of the input features have zero weights.
- very few features have non zero weights.
- we penalize the absolute value of the weights.

$$Cost\ function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

$$J_{\text{regularized}} = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(a^{[L](i)}) + (1-y^{(i)})\log(1-a^{[L](i)})\right] + \frac{\lambda}{m}\sum_{l}\sum_{k}\sum_{j}\left|W_{kj}^{[l]}\right|$$

L2 Regularization or Ridge Regularization

- L2 regularization is also known as weight decay.
- L2 regularization penalizes sum of square weights.
- L2 is not robust to outliers
- L2 gives better prediction when output variable is a function of all input features
- L2 regularization is able to learn complex data patterns
- L2 penalizing larger weights more severely, but resulting in less sparse weights.
- L2 (use a simpler model)
- The regularization parameter controls the amount of attention that the learning process should pay to the penalty.

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

L2 Regularization or Ridge Regularization

Initial weight: $W = 0.5$

Gradient from loss function: $\frac{\partial L}{\partial W} = 0.1$

Learning rate: $\eta = 0.01$

Regularization strength: $\lambda = 0.1$

$$W_{\text{new}} = W - \eta \cdot \frac{\partial L}{\partial W}$$

$$W_{\text{new}} = 0.5 - (0.01 \times 0.1)$$

$$W_{\text{new}} = 0.5 - 0.001 = 0.499$$

Without Regularization

$$W_{\text{new}} = W - \eta \cdot \left(\frac{\partial L}{\partial W} + \lambda W\right)$$

$$W_{\text{new}} = 0.5 - 0.01 \times (0.1 + 0.1 \times 0.5)$$

$$W_{\text{new}} = 0.5 - 0.01 \times (0.1 + 0.05)$$

$$W_{\text{new}} = 0.5 - 0.0015 = 0.4985$$

With L2 Regularization (Weight Decay)

he weight is slightly smaller compared to standard gradient descent, reducing overfitting.

$$W_{\text{new}} = W - \eta \cdot \left(\frac{\partial L}{\partial W} + \lambda \cdot \text{sign}(W)\right)$$

$$W_{\text{new}} = 0.5 - 0.01 \times (0.1 + 0.1 \times 1)$$

$$W_{\text{new}} = 0.5 - 0.01 \times (0.1 + 0.1)$$

$$W_{\text{new}} = 0.5 - 0.002 = 0.498$$

(some weights may become zero

Regularization	Weight Update Formula	New Weight W_{new}
No Regularization	$W - \eta \cdot \frac{\partial L}{\partial W}$	0.499
L2 Regularization	$W - \eta(\frac{\partial L}{\partial W} + \lambda W)$	0.4985
L1 Regularization	$W - \eta(\frac{\partial L}{\partial W} + \lambda \cdot \text{sign}(W))$	0.498

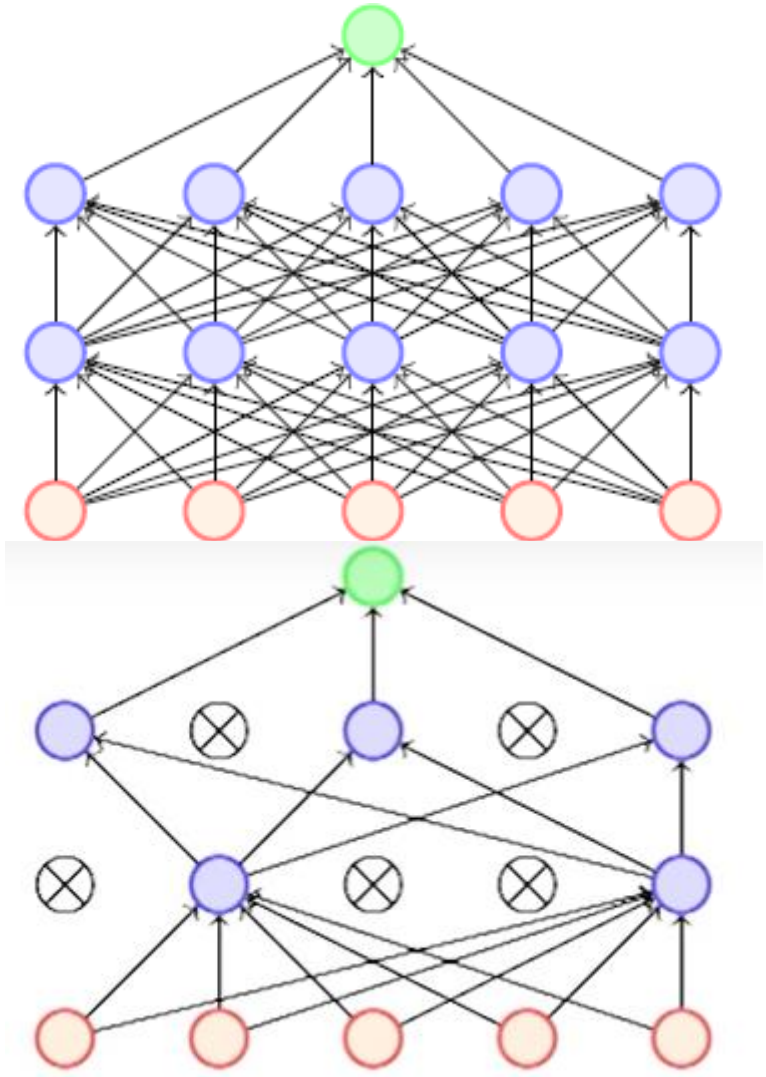
L2 Regularization or Ridge Regularization

- **L2 (Weight Decay)** makes small adjustments, shrinking weights but **never setting them to zero**.

- **L1 (Weight Selection)** reduces weight more aggressively and **can push weights to zero**, encouraging sparsity.

- The larger **λ** is, the stronger the regularization effect.

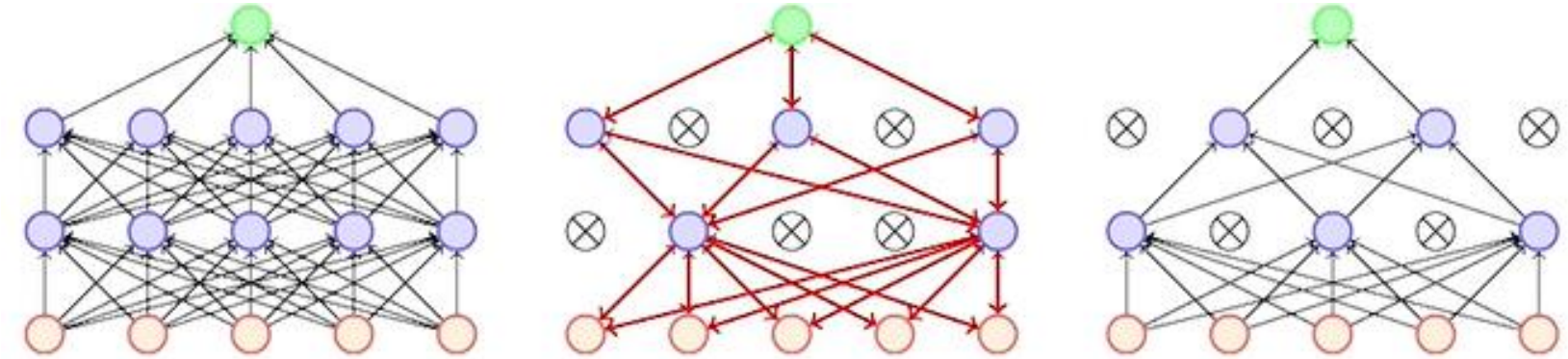
Dropout Regularization



drop 5 nodes to get a thinned network

- Dropout refers to dropping out units
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network. Any weight updates are not applied to these neuron on the backward pass.
- Each node is retained with a fixed probability (typically p = 0:5) for hidden nodes and p = 0:8 for visible nodes.
- compute the loss and backpropagate Only those which are active.

Dropout Regularization

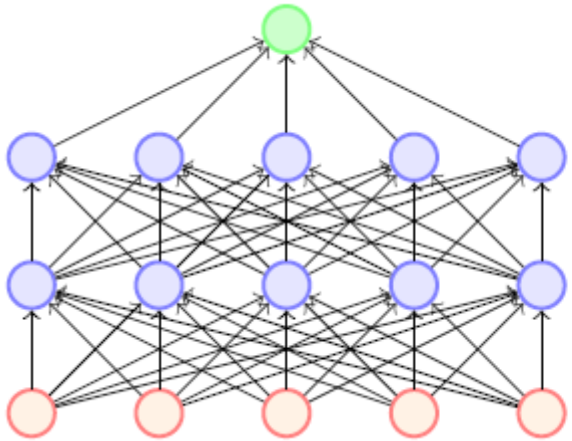


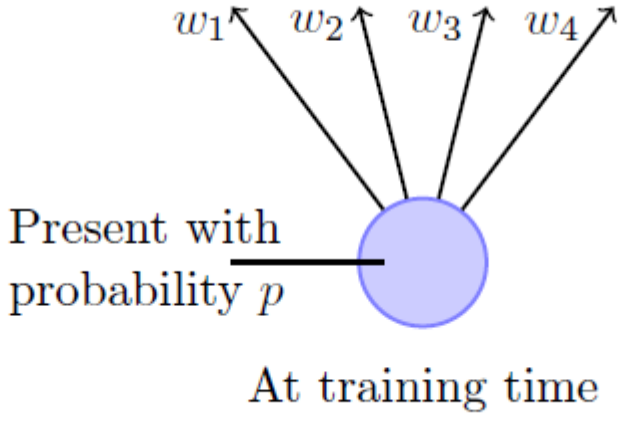
- For the second training instance (or mini-batch), we again apply dropout resulting in a different thinned network

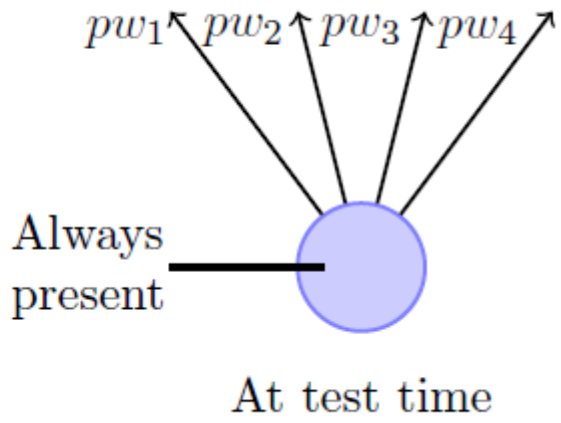
- We again compute the loss and backpropagate to the active weights If the weight was active for both the training instances then it would have received two updates by now

- If the weight was active for only one of the training instances then it would have received only one updates by now

Dropout Regularization



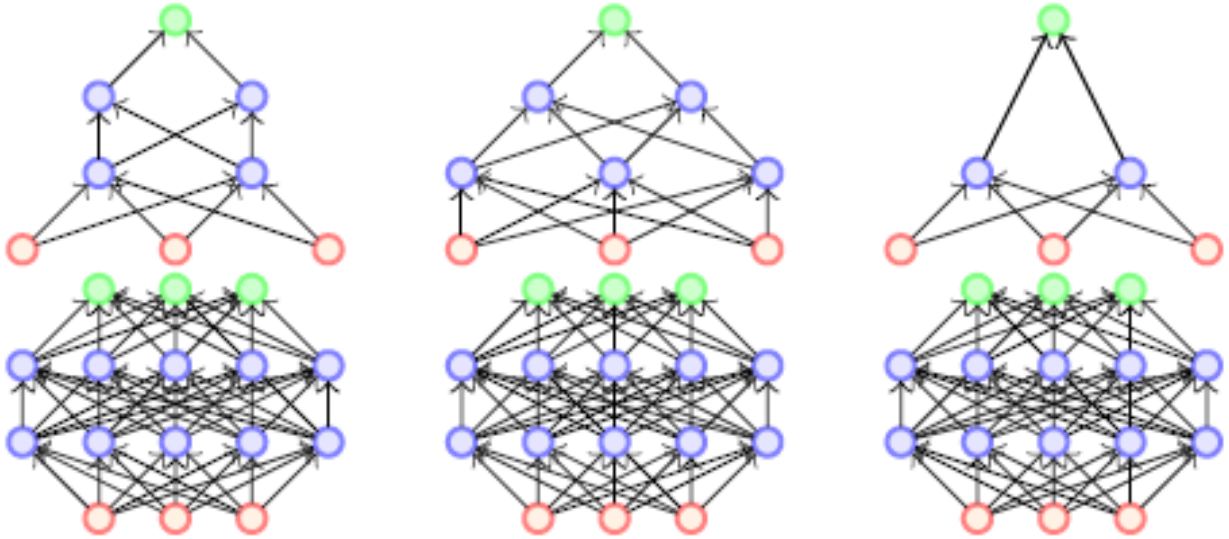




- Instead we use the full Neural Network and scale the output of each node by the fraction of times it was on during training
- Training Time (Left Circle)
  - Each neuron is present with a probability p (e.g., p=0.5 means each neuron has a 50% chance of being dropped).
  - The weights w1,w2,w3,w4 are applied only to the remaining neurons.
- Testing Time (Right Circle)
  - All neurons are always present (i.e., no dropout during testing).
  - The weights are scaled by p, i.e., pw1,pw2,pw3,pw4 to compensate for the reduced number of active neurons during training.

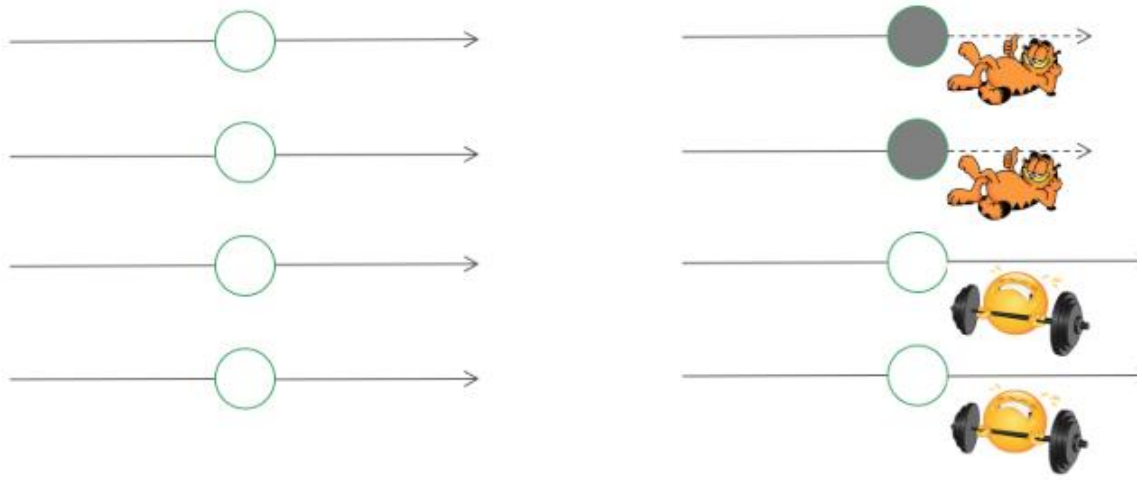
Dropout Regularization

- Generally, use a small dropout value of 20%-50% of neurons. Use a larger network.
- Use dropout on incoming (visible) as well as hidden units.
- Use a large learning rate with decay and a large momentum.
- it allows training several neural networks without any significant computational overhead.



Co-Adaptation

- Some of the connections will have more predictive capability.
- Powerful connections are learned more while the weaker ones are ignored.
- This is not solved by traditional regularization, like L1 and L2 (regularize based on the predictive capability of the connections).
- Strong gets stronger and the weak gets weaker.



Dropout Regularization

- Dropout forces a neural network to learn more robust features.
- Dropout roughly doubles the number of iterations required to converge.
- Training time for each epoch is less.
- With h hidden units, each of which can be dropped, we have 2^h possible models.

Dropout Regularization

- Dropout is not used after training when making a prediction with the fit network.
- *If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time.*
- The rescaling of the weights can be performed at training time instead, after each weight update at the end of the mini-batch. This is called "*inverse dropout*".

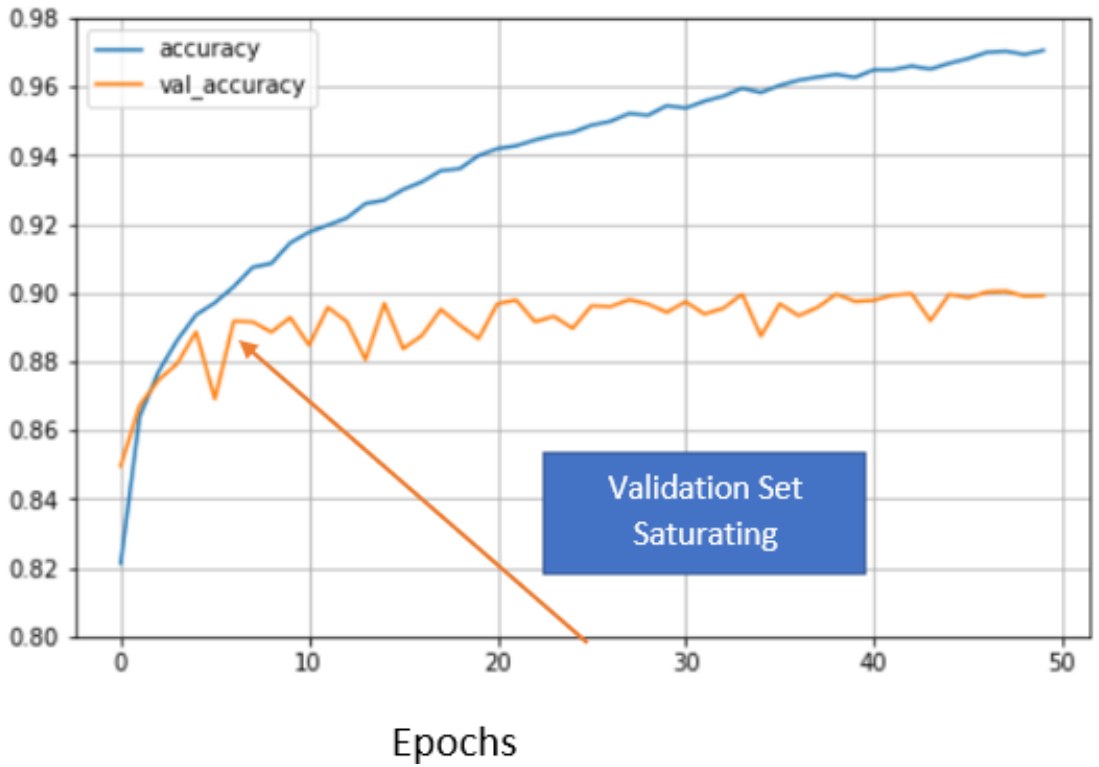
Implementation of Dropout

- At each iteration, shut down some neurons with prob. (1-p) where p is known as dropout rate.
- Forward Pass- Apply mask on the activation and divide activation by p. (Inverted dropout)
- Backward Pass- Apply the same mask (used in f/w pass) on the derivatives of activation and again divide it by p.
- Update parameters.
- Is it possible to use both *dropout* and L1/L2 reg.?

Tips for Using Dropout Regularization

- Use With All Network Types.
- A good value for dropout in a hidden layer is between 0.5 and 0.8. (where 1.0 means no dropout).
- Use a Larger Network-If *n* is the number of hidden units in any layer and *p* is the dropout rate, a good dropout net should have at least n/p units
- Grid Search Parameters-test values between 1.0 and 0.1 in increments of 0.1.
- Use a Weight Constraint.
- Use With Smaller Datasets-Problems where there is a large amount of training data may see less benefit from using dropout.

Early Stopping



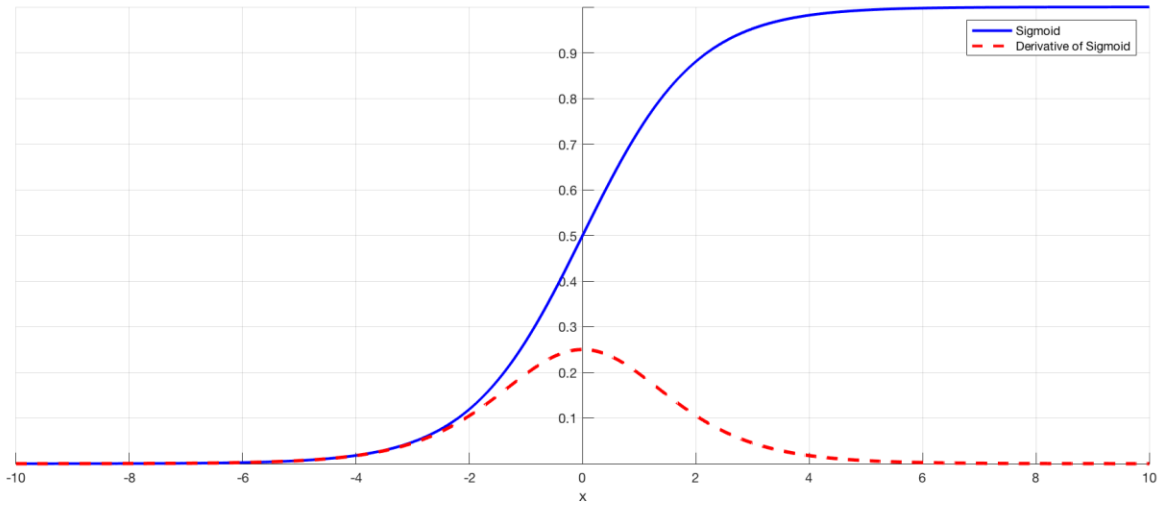
- Early stopping is a form of regularization that halts the training process when the performance of the model on a validation dataset starts to degrade.
- The training set accuracy continues to increase, through all the Epochs
- The validation set accuracy, however, saturates between 8 to 10 epochs. This is where the model can be stopped training.
- Early Stopping, hence does not only protect against overfitting but needs considerably less number of Epoch to train.

Vanishing and Exploding Gradient

- If your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small.
- This makes training difficult, especially if your gradients are exponentially small, then gradient descent will take tiny little steps.
- It will take a long time for gradient descent to learn anything.
- Exploding gradients are a problem where large error gradients accumulate and result in very large updates to neural network model weights during training.
- By the chain rule, the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers.

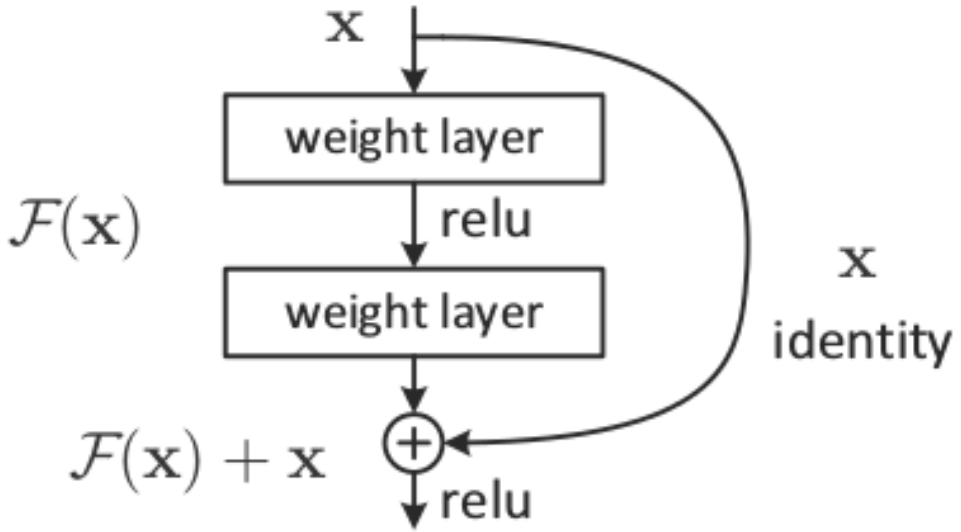
Vanishing and Exploding Gradient

- Sigmoid, tanh, and ReLU – (Small derivative problem)
- Squashing functions

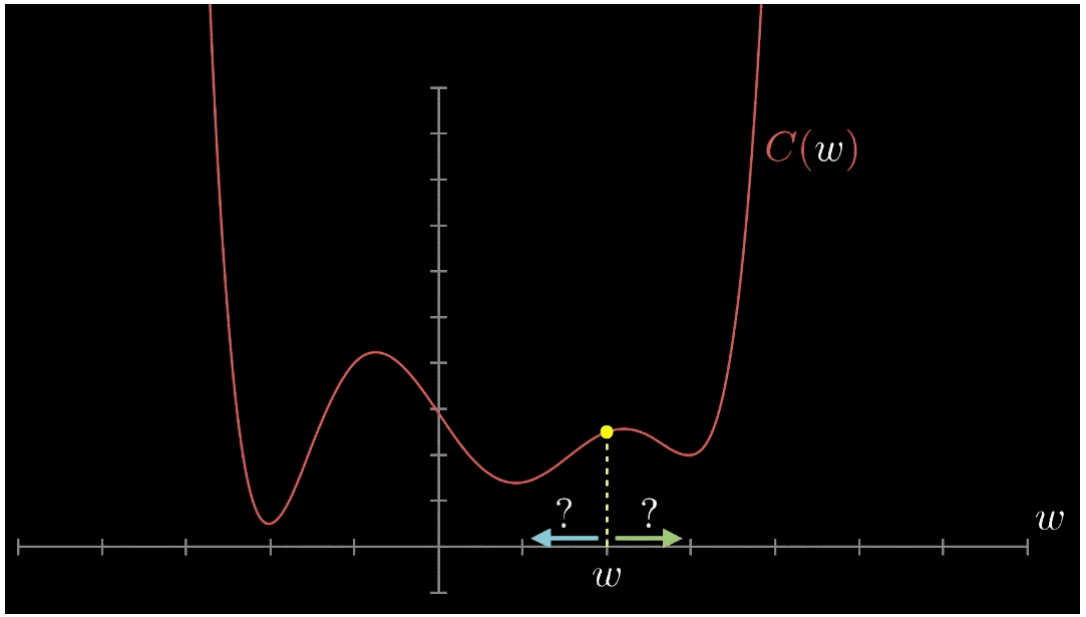


Solutions

- Use ReLU
- Residual networks
- batch normalization
- Gradient clipping



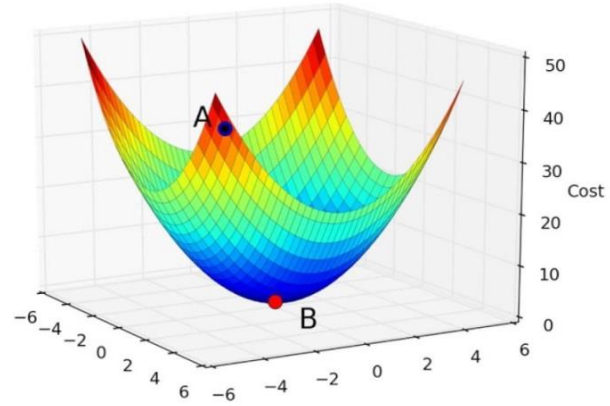
Challenges to Gradient Descent

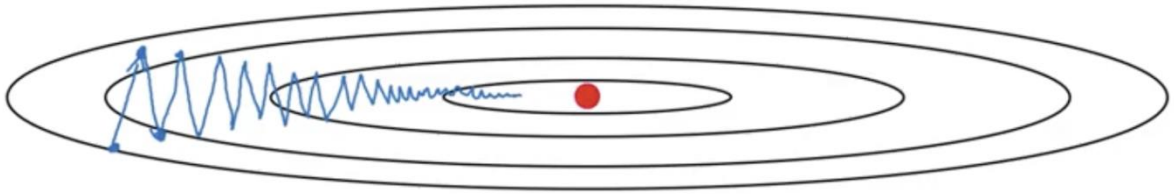


- Choosing a proper learning rate can be difficult.
- Learning rate schedules
- Same learning rate applies to all parameter update
- Minimizing highly non-convex error functions: trapped in their numerous suboptimal local minima

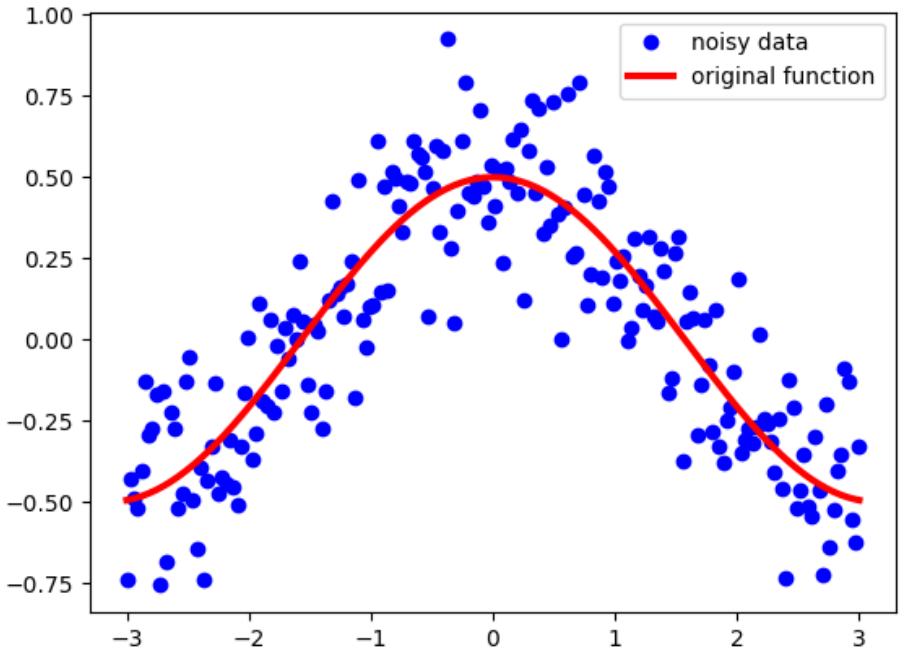
Gradient Descent Optimizer

- The objective of all optimizers is to reach the global minima where the cost function attains the least possible value.
- Gradient Descent with Momentum
- RMSprop Optimizer
- Adam's Optimizer
- Effect of Learning Rate
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- AdaMax
- Nadam
- AMSGrad





Gradient Descent Optimizing methods (Faster Convergence)

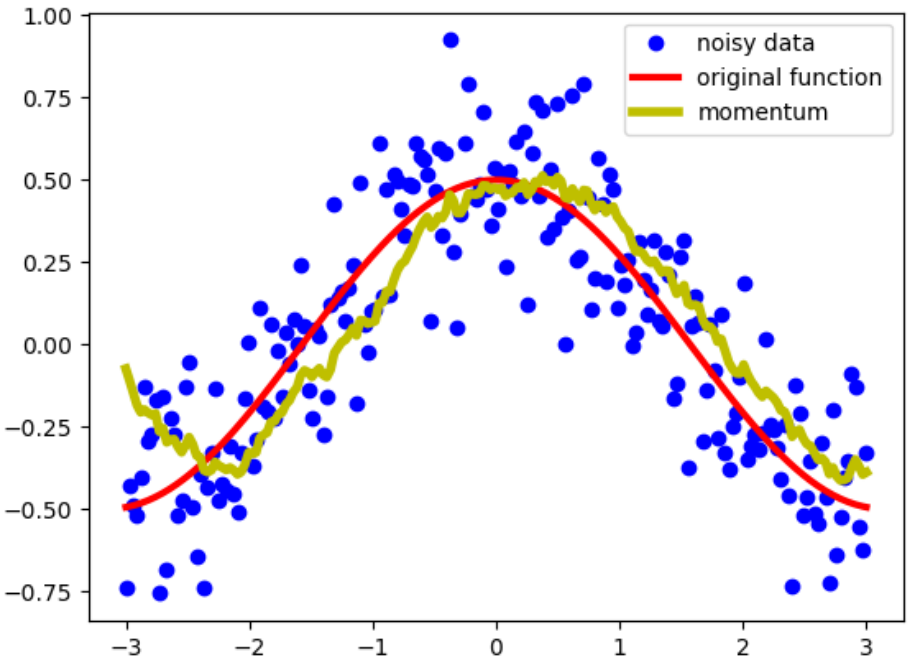


- **SGD with Momentum**-helps accelerate gradients vectors in the right directions, thus leading to faster converging.
- Momentum reduce these oscillations toward convergence, we can set a higher learning rate.
- Exponentially Weighted Averages-deal with sequences of numbers.

Challenges:

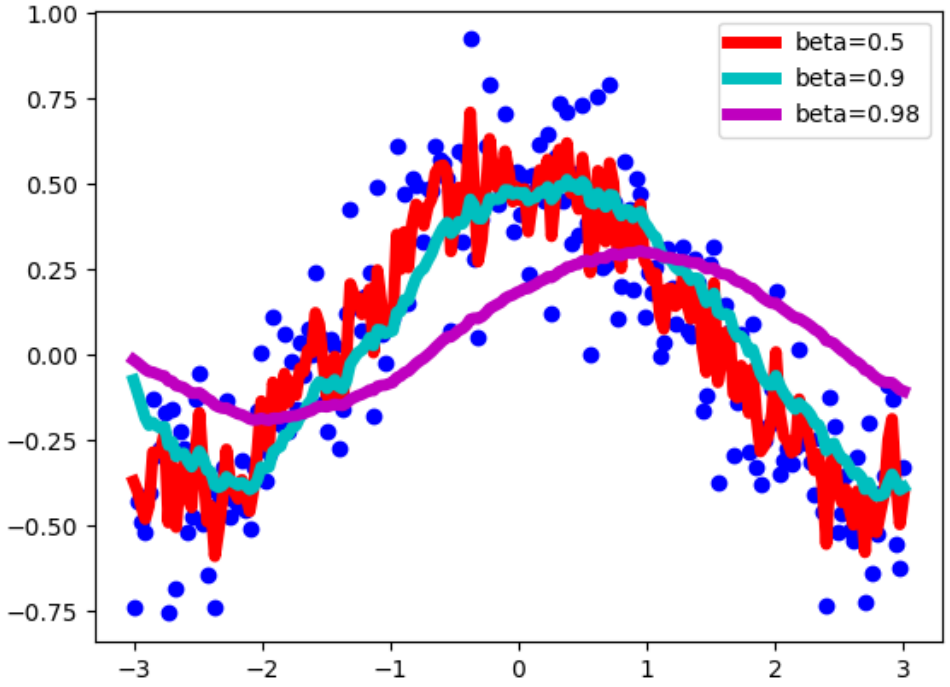
- **Slow Convergence:** If the gradient oscillates (especially in valleys or ravines), SGD takes small steps in the right direction but zigzags sideways.
- **Noise:** Since SGD updates weights using a small batch of data, it can be noisy.

Exponentially Weighted Averages



- We want some kind of ‘moving’ average which would ‘denoise’ the data and bring it closer to the original function.
- **Momentum**
- Momentum helps by **accumulating past gradients** to create a smoother update.

Exponentially Weighted Averages



- Exponentially weighed averages define a new sequence V with the following equation.

- *Beta* is another hyper-parameter which takes values from 0 to one. We're approximately averaging over last *1 / (1-beta)* points of sequence.

$$V_t = \beta V_{t-1} + (1-\beta) S_t$$
$$\beta \in [0, 1]$$

If $\beta = 0.9$, then $\frac{1}{1-0.9} = 10 \rightarrow$ The last **10 data points** significantly contribute.

If $\beta = 0.98$, then $\frac{1}{1-0.98} = 50 \rightarrow$ The last **50 data points** significantly contribute.

If $\beta = 0.5$, then $\frac{1}{1-0.5} = 2 \rightarrow$ The last **2 data points** significantly contribute.

- **Higher β** → longer memory, smoother updates.
- **Lower β** → shorter memory, quicker adaptation.

Exponentially Weighted Averages

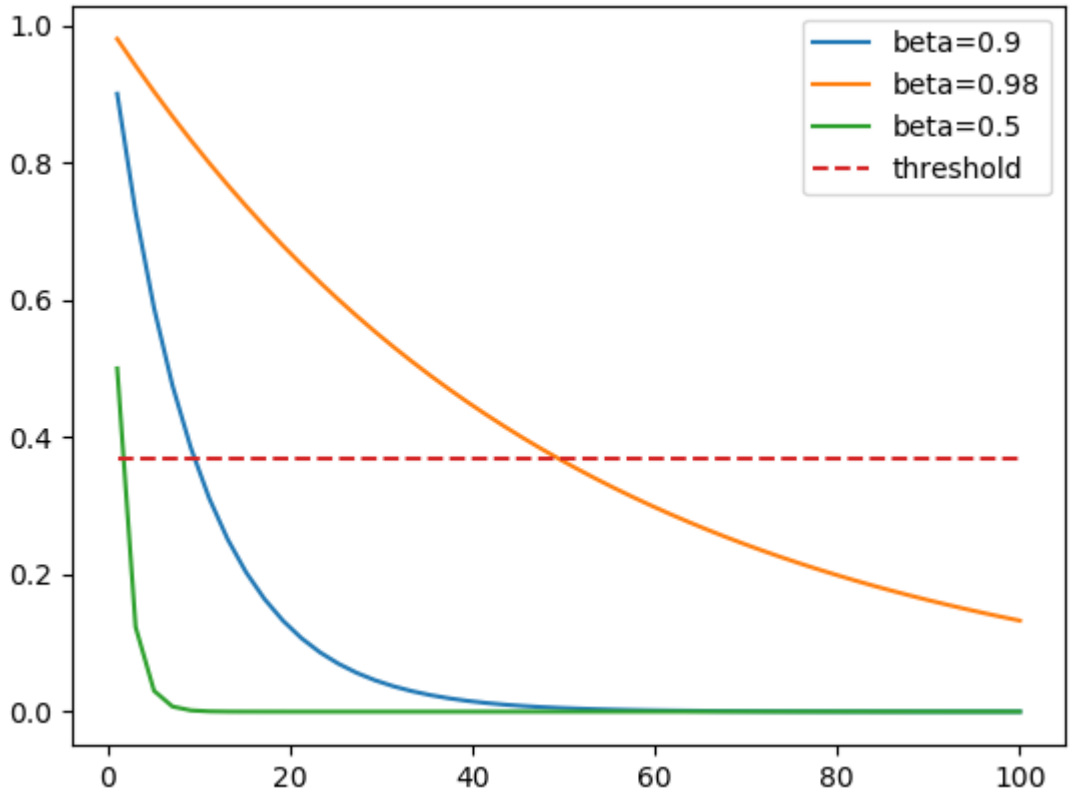
- With smaller beta, the new sequence turns out to be fluctuating a lot, because we're averaging over smaller number of examples and therefore are 'closer' to the noisy data.

- With bigger beta, we get much smother curve, but it's a little bit shifted to the right, because we average over larger number of example.

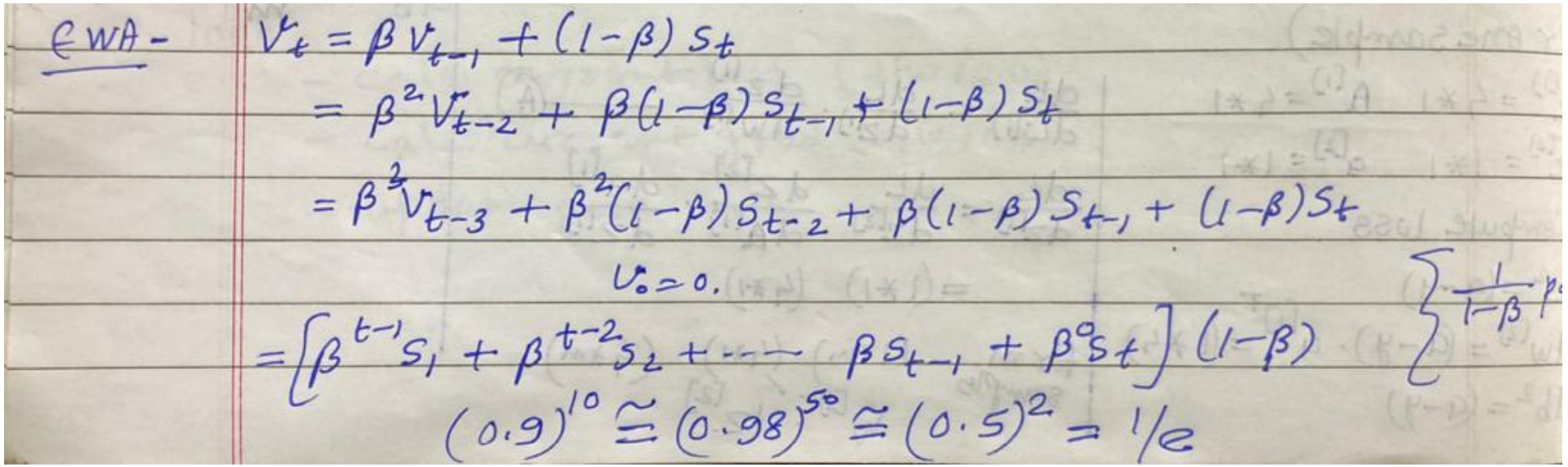
- *Beta = 0.9* provides a good balance between these two extremes.

- Mathematical Intuition

Exponentially Weighted Averages



Overfitting and Underfitting



Vt is the exponentially weighted average at time t,
β (0 < β < 1) is the smoothing factor,
St is the current input (e.g., gradient or loss),
V0=0 (initial condition).

$$\sum_{i=0}^{\infty} \beta^i \approx \frac{1}{1-\beta}$$

comes from the sum of an infinite geometric series.

MBGD with Momentum

- During backward propagation, we use dW and db to update our parameters W and b as follows:
    - W = W – learning rate * dW
    - b = b – learning rate * db

- In momentum, instead of using dW and db independently for each epoch, we take the exponentially weighted averages of dW and db.
    - $V_{dW} = \beta \times V_{dW} + (1 - \beta) \times dW$
    - $V_{db} = \beta \times V_{db} + (1 - \beta) \times db$

- $W = W - \text{learning rate} * V_{dW}$

- $b = b - \text{learning rate} * V_{db}$

Bias Correction

- First couple of iterations will provide a pretty bad averages.
- instead of using V, we can use what's called bias-corrected version of V.

$$V_t = \frac{V_t}{1 - b^t}$$

- You are using exponentially weighted average (EWA) for the computation of derivatives of cost w.r.to a weight w. The values of these derivatives for the first four mini-batches are 0.81, 0.9, 0.96, and 0.34 respectively. Comment on your observation on the computation of EWA for the above derivatives with and without bias correction for a) beta=0.9

Bias Correction

Given derivatives: **0.81, 0.9, 0.96, 0.34**

$V_0 = 0$, and $\beta = 0.9$.

Iteration 1 (t = 1)

$$V_1 = (0.9 \times 0) + (0.1 \times 0.81) = 0.081$$

Iteration 2 (t = 2)

$$V_2 = (0.9 \times 0.081) + (0.1 \times 0.9) = 0.1629$$

Iteration 3 (t = 3)

$$V_3 = (0.9 \times 0.1629) + (0.1 \times 0.96) = 0.24261$$

Iteration 4 (t = 4)

$$V_4 = (0.9 \times 0.24261) + (0.1 \times 0.34) = 0.25235$$

Now, we apply bias correction:

$$\hat{V}_t = \frac{V_t}{1 - 0.9^t}$$

Iteration 1 (t = 1)

$$\hat{V}_1 = \frac{0.081}{1 - 0.9^1} = \frac{0.081}{0.1} = 0.81$$

Iteration 2 (t = 2)

$$\hat{V}_2 = \frac{0.1629}{1 - 0.9^2} = \frac{0.1629}{0.19} = 0.8574$$

Iteration 3 (t = 3)

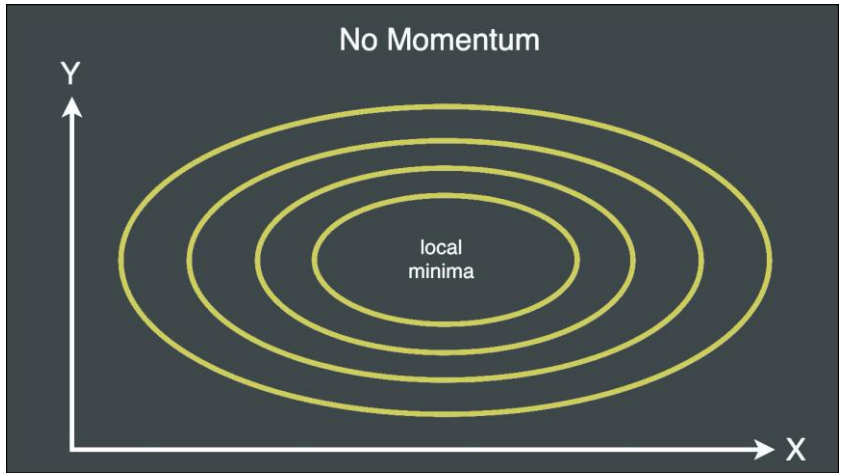
$$\hat{V}_3 = \frac{0.24261}{1 - 0.9^3} = \frac{0.24261}{0.271} = 0.8953$$

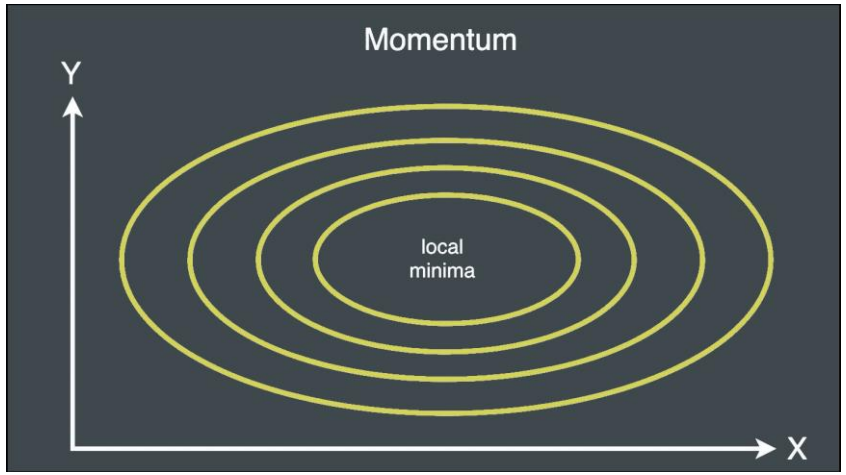
Iteration 4 (t = 4)

$$\hat{V}_4 = \frac{0.25235}{1 - 0.9^4} = \frac{0.25235}{0.3439} = 0.7343$$

Why Momentum Works?

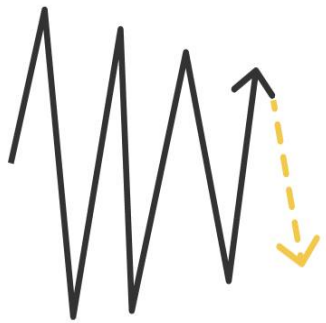
- We might not going in the optimal direction by just considering a mini batch.
- SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum.





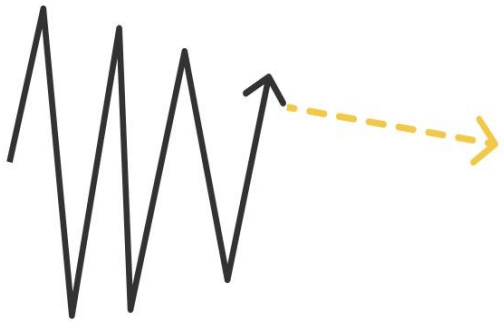
RMSprop Optimizer

Gradient descent



the gradient computed on the current iteration does not prevent gradient descent from oscillating in the vertical direction

Momentum



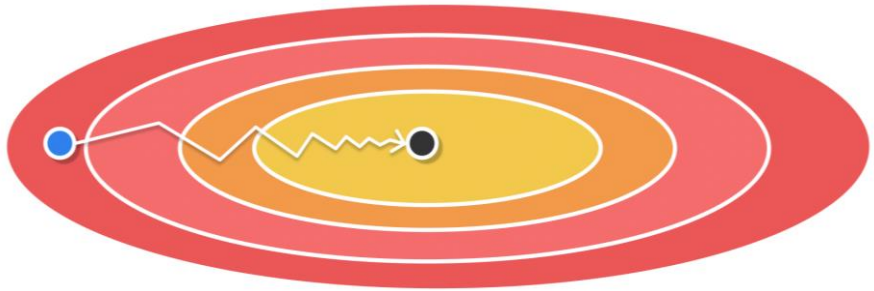
the average vector of the horizontal component is aligned towards the minimum

the average vector of the vertical component is close to 0

AdaGrad (Adaptive Gradient Algorithm)

$$v_t = v_{t-1} + dw_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t} + \varepsilon} dw_t$$



- AdaGrad is another optimizer with the motivation to adapt the learning rate to computed gradient values. Independently adapting the learning rate for each weight component.
- If gradients corresponding to a certain weight vector component are large, then the respective learning rate will be small. Inversely, for smaller gradients, the learning rate will be bigger.
- Adagrad accumulates element-wise squares **dw^2** of gradients from all previous iterations.
- During weight update, instead of using normal learning rate α, AdaGrad scales it by dividing α by the square root of the accumulated gradients **$\sqrt{v_t}$**. Additionally, a small positive term ε is added to the denominator to prevent potential division by zero.

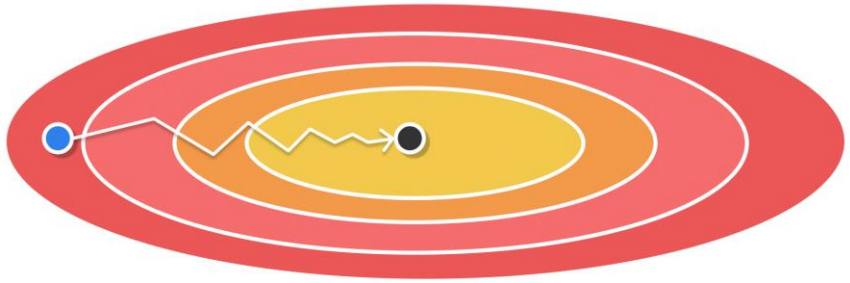
RMSprop Optimizer

- RMSProp was elaborated as an improvement over AdaGrad which tackles the issue of learning rate decay. It modifies Adagrad by using an exponentially weighted moving average of the squared gradients instead of a cumulative sum.
- RMSprop optimizer is similar to the gradient descent algorithm with momentum.

- Accelerate the learning process by penalizing the update of those parameters that make the estimate of the cost function oscillate too much. Avoid the gradient descent algorithm adapt too quickly to changes in these parameter.
- The RMSprop optimizer restricts the oscillations in the ~~vertical direction~~.
- Could take larger steps in the ~~horizontal direction~~ converging faster.
- We can increase our learning rate.
- Why it is called RMS?

RMSprop Optimizer

- However, instead of storing a cumulated sum of squared gradients dw^2 for v_t, the exponentially moving average is calculated for squared gradients dw^2.



Optimization with RMSProp

Momentum vs. RMSProp

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db$$

$$W = W - \alpha \cdot v_{dw}$$

$$b = b - \alpha \cdot v_{db}$$

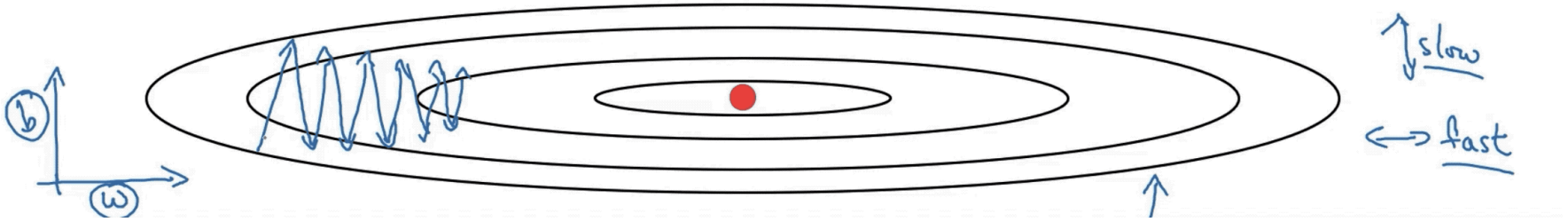
$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

RMSprop



Adam's Optimization(Adaptive Moment Estimation)

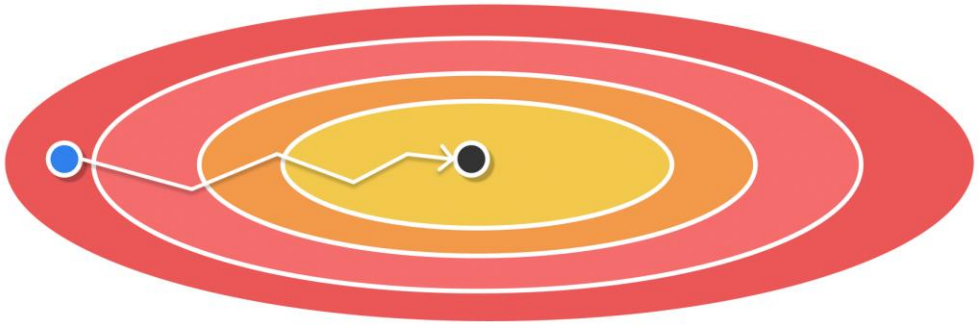
- Combination of Momentum and RMSProp

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) dw_t \xrightarrow{\text{bias correction}} \hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) dw_t^2 \xrightarrow{\text{bias correction}} \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w_t = w_{t-1} - \frac{\alpha \hat{v}_t}{\sqrt{\hat{s}_t} + \varepsilon} dw_t$$

$\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 1e\text{-}8$.



Steps to implement:

- Initialize V_{dW}, S_{dW}, V_{db} and S_{db} to zero.
- On iteration T, compute the derivatives dw & db using current mini-batch.
- Update V_{dW} and V_{db} like momentum.

$$V_{dW} = ß_1 \times V_{dW} + (1- ß_1) \times dW$$

$$V_{db} = ß_1 \times V_{db} + (1 - ß_1) \times db.$$

- Update S_{dW} and S_{db} like Rmsprop.

$$S_{dW} = ß_2 \times S_{dW} + (1- ß_2) \times dW^2$$

$$S_{db} = ß_2 \times S_{db} + (1 - ß_2) \times db^2.$$

- In Adam optimization implementation, we do implement bias correction.

$$V_{dW}^{corrected} = V_{dW} / (1- ß_1^t)$$

$$V_{db}^{corrected} = V_{db} / (1- ß_1^t)$$

$$S_{dW}^{corrected} = S_{dW} / (1- ß_2^t)$$

$$S_{db}^{corrected} = S_{db} / (1 - ß_2^t)$$

- Update parameters W and b.

$$W = W - \text{learning rate} \times (V_{dW}^{corrected} / \text{sqrt}(S_{dW}^{corrected} + \varepsilon))$$

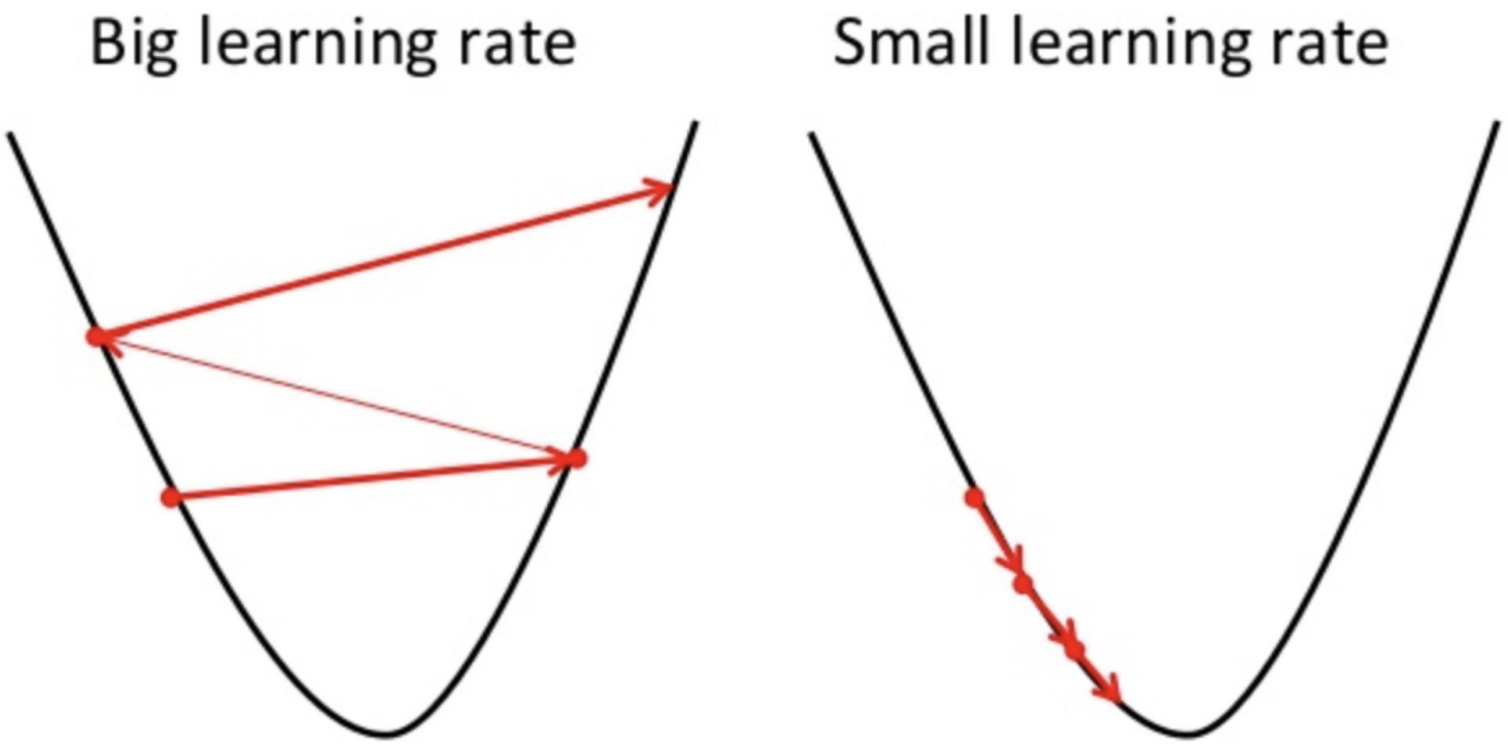
$$b = b - \text{learning rate} \times (V_{db}^{corrected} / \text{sqrt}(S_{db}^{corrected} + \varepsilon))$$

Adam Configuration Parameters

- **alpha**. Larger values (e.g. 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) slow learning right down during training
- **beta1**. The exponential decay rate for the first moment estimates (e.g. 0.9).
- **beta2**. The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with a sparse gradient (e.g. NLP and computer vision problems).
- **epsilon**. Is a very small number to prevent any division by zero in the implementation (e.g. 10E-8).

Which Optimizer to use??????????????

Effect of Learning Rate & Learning Rate Decay & Adaptive Learning Rate



Batch Norm

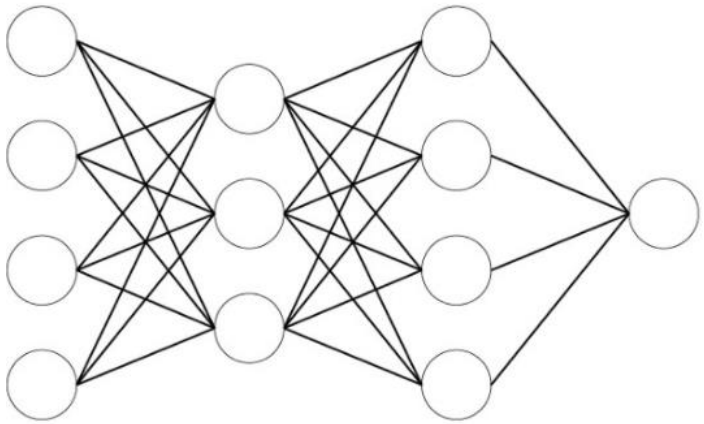
- Method which makes the training of Deep Neural Networks (DNN) faster and more stable. It consists of normalizing activation vectors from hidden layers using the first and the second statistical moments (mean and variance) of the current batch.
- This normalization step is applied right before (or right after) the nonlinear function. Usually inserted as a standard layer in a DNN.

Advantages:

1. Normalized signals
2. Mitigating interdependency between distributions
3. Fast and stable training

During model training

- Batch normalization is computed differently during the training and the testing phase.
- Third step, each neuron's output follows a standard normal distribution across the batch.
- Fourth step, linear transformation with γ and β, two trainable parameters. This allows the model to choose the optimum distribution for each hidden layers, by adjusting those two parameters : γ allows to adjust the standard deviation ; β allows to adjust the bias, shifting the curve on the right or on the left side.
- It trains γ and β through gradient descent, using an Exponential Moving Average (EMA)

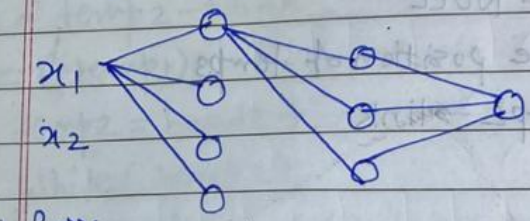


$$(1)\quad \mu = \frac{1}{n}\sum_i Z^{(i)} \qquad (2)\quad \sigma = \frac{1}{n}\sum_i (Z^{(i)} - \mu)$$

$$(3)\quad Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 - \epsilon}} \qquad (4)\quad \check{Z} = \gamma * Z^{(i)}_{norm} + \beta$$

$w^1 = 4*2$ $w^2 = 3*4$ $w^3 = 1*3$

$b^1 = 4*1$ $b^2 = 3*1$ $b^3 = 1*1$



FW Pass.

$z^1 = (4*2)\cdot(2*1) + (4*1)$

$= 4*1$

$a^1 = 4*1$

$z^2 = (3*4)\cdot(4*1) + (3*1)$

$= (3*1)$

$a^2 = (3*1)$

$z^3 = (1*3)(3*1) + (1*1)$

$= (1*1)$

$a^3 = (1*1)$

Loss.

$L =$

For m samples

$A^1 = Z^1 = 4*m$

$A^2 = Z^2 = 3*m$

$A^3 = Z^3 = 1*m$

$dz^3 = (a-y)$

$dw^3 = (1*1)\cdot(1*3) = (1*3)$

$db^3 = (1*1)$

$dz^2 = [(3*1)\cdot(1*3)] * (3*1) = (3*1)$

$dw^2 = (3*1)\cdot(1*4) = (3*4)$

$db^2 = (3*1)$

$dz^1 = [(4*3)\cdot(3*1)] * (4*1) = (4*1)$

$dw^1 = (4*1)(1*2) = (4*2)$

$db^1 = (4*1)$

$dz^3 = 1*m$

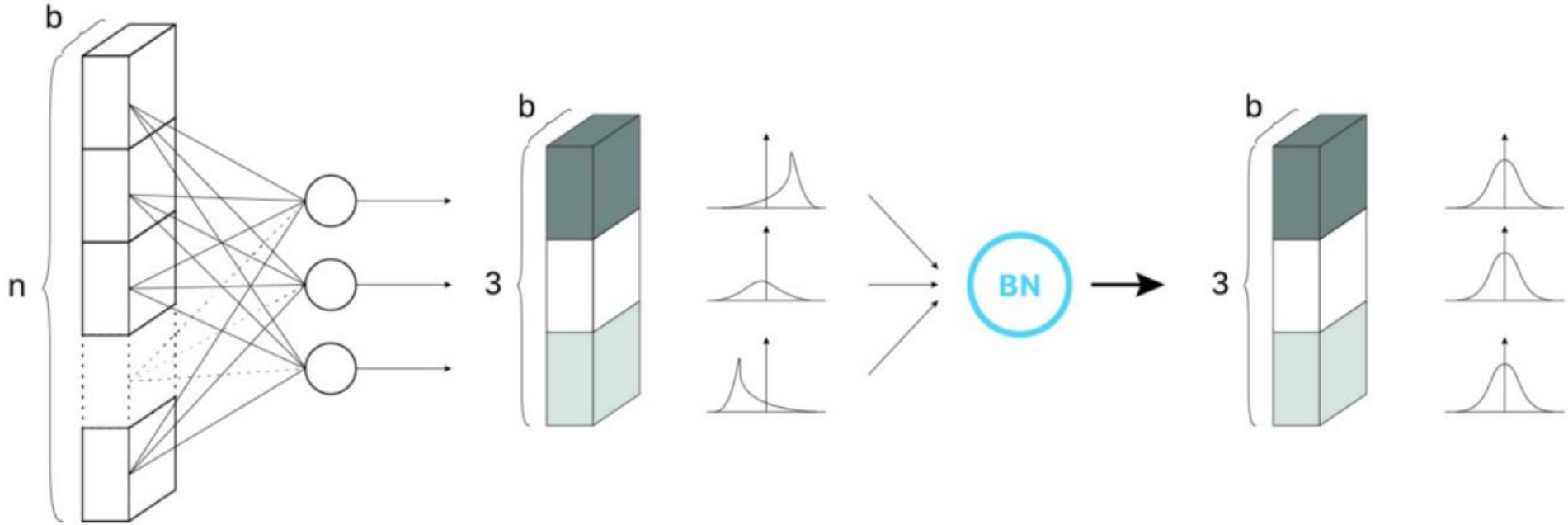
$dw^3 = 1/m\ (1*m)\cdot(m*3) = (1*3)$

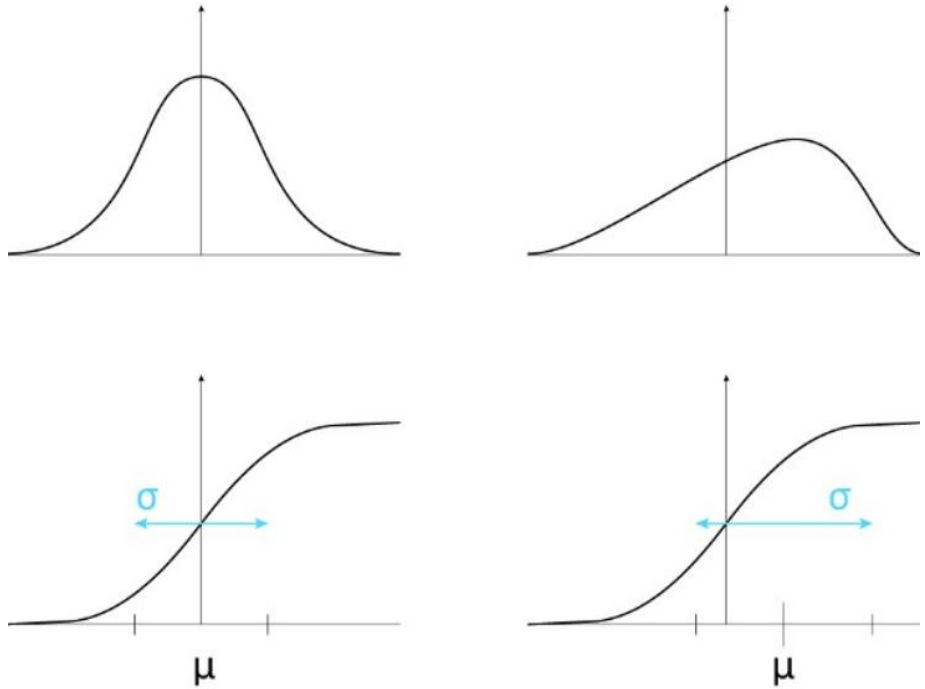
$db^3 = (1*1)$

$dz^2 = (3*1)\cdot(1*m) * (3*m) = 3*m$

$dw^2 = (1/m)\ (3*m)\ (m*4) = 3*4$

$db^2 = (3*1)$


b
n
b
3
BN
b
3


σ
μ
σ
μ

During model evaluation

- We may not have a full batch to feed into the model during the evaluation phase.
- To tackle this issue, we compute (μ_pop , σ_pop), such as:
- μ_pop : estimated mean of the studied population
- σ_pop : estimated standard-deviation of the studied population.
- Those values are computed using all the (μ_batch , σ_batch) determined during training, and directly fed into equation (3) during evaluation.
- *The shifting of distribution between training and testing set is called "covariate shift".*

Advantages

- Adding BN layers leads to faster and better convergence
- On large dataset, the improvement is much more significant than the one observed on the smaller dataset.
- Adding BN layers allows us to use higher learning rate (LR) without compromising convergence: Inception-with-BN network using a 30 times higher learning rate than the original one. Generally, model don’t converge if we increase LR more than 5 times.
- Introduce regularization: Output strongly tied to the current batch statistics (mean and var.), add noise.
- Shouldn’t rely on BN for regularization

Working behind BN: H1

- BN effectiveness is due to the **reduction** of **internal covariate shift** (ICS), *refer to the change in the distributions of internal nodes of a deep network, in the course of training.*
- Covariate shift - in the distributional stability prospective - describes the shifting of a model input distribution.
- By extension, the internal covariate shift describes this phenomenon when it happens in the hidden layers of a deep neural network.
- *From the model point-of-view, training images are - statistically - too different from testing images. There is a **covariate shift**.*
- It is well known that **linear regression** models are easier to optimize when the input values are normalized (i.e. making its distribution close to ($\boldsymbol{\mu}$ = 0, $\boldsymbol{\sigma}$ = 1)) : that's why **we usually normalize the input values of a model.**

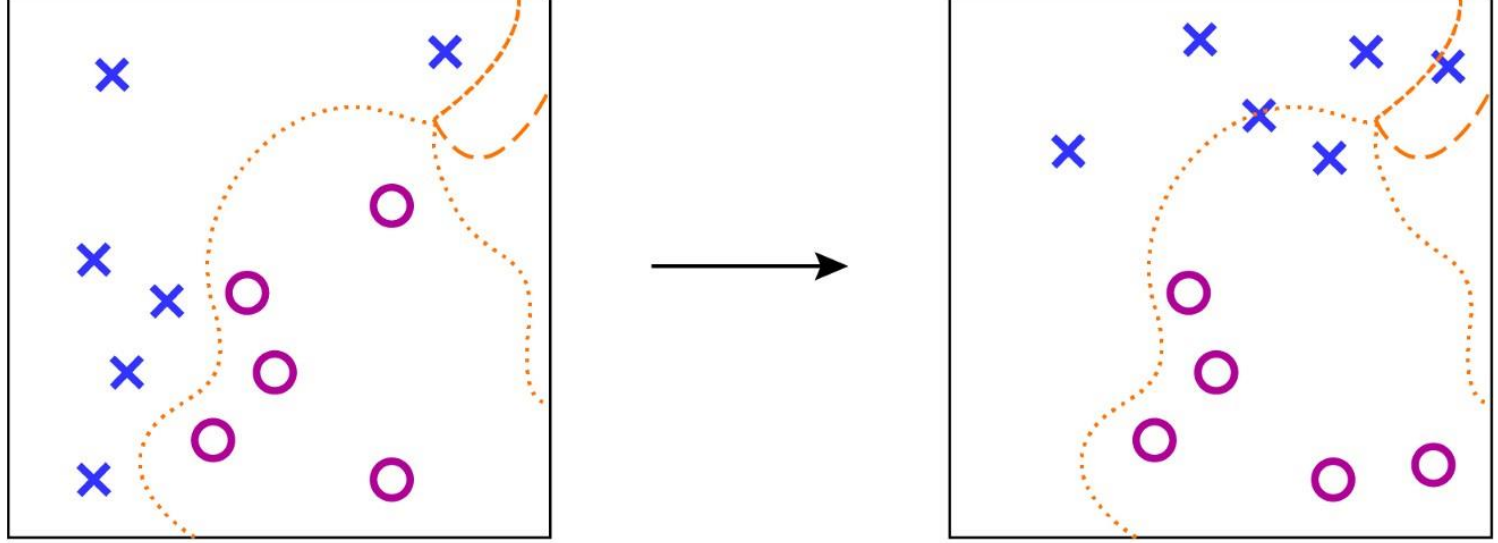
Working behind BN: H1

- If there is a huge covariate shift in the input signal, **the optimizer will have trouble generalizing well**.

- If the input signal always follows a standard normal distribution, the optimizer will more easily generalize.

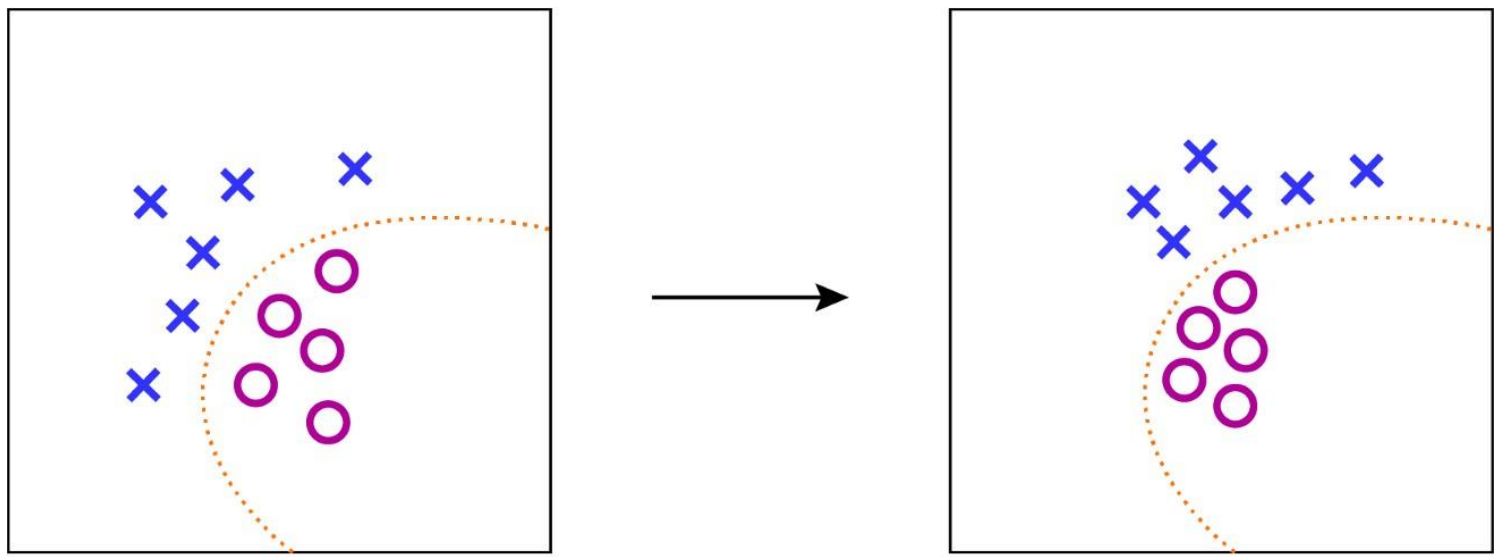
- Applying the strategy of normalizing the signal in the hidden layers forces to (μ = 0, σ = 1) the intermediates signal distribution will **help the network generalization in the "conceptual" levels of features**.

- *BN → **Normalization** of the input signal inside hidden units → Adding two trainable parameters to **adjust the distribution** and get the most out of the nonlinearities → Easier training*



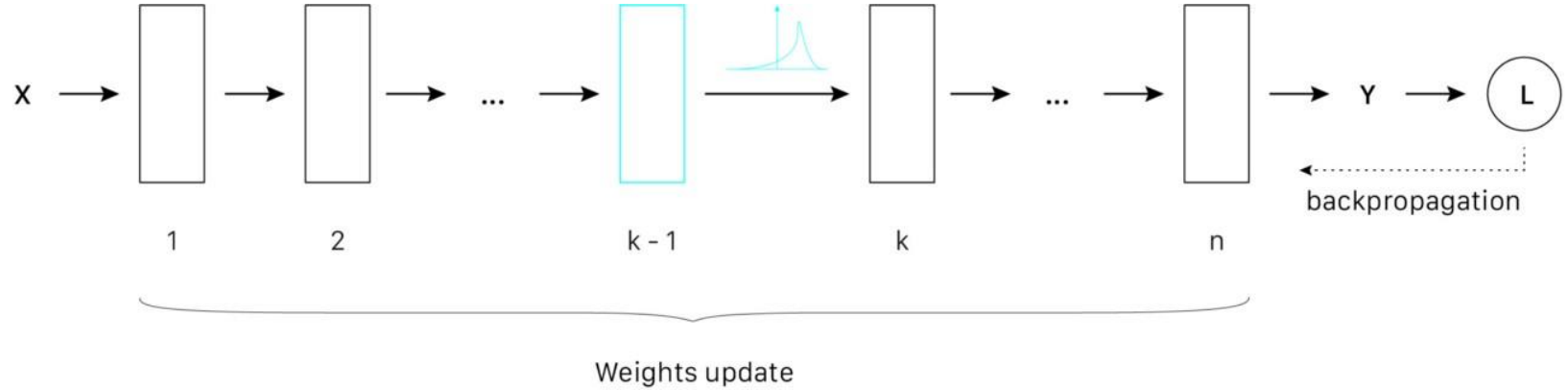
Training

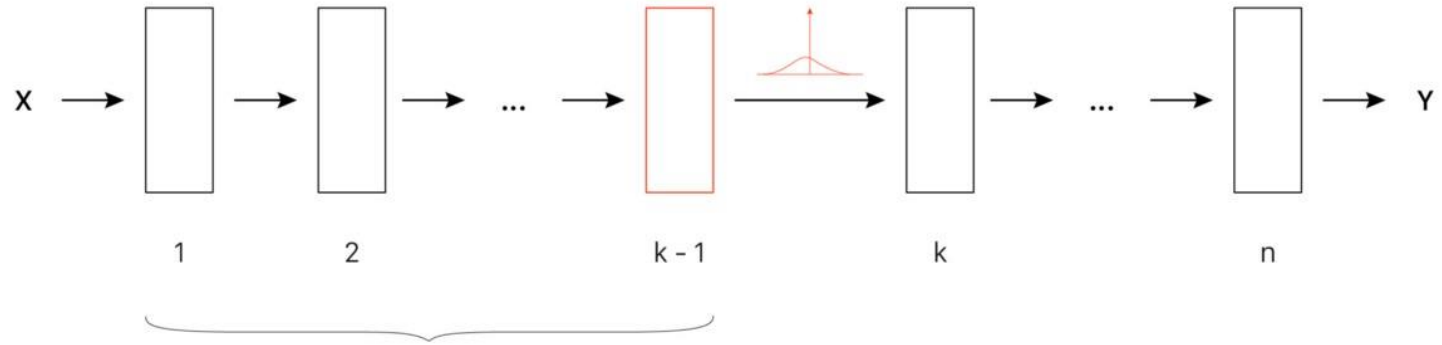
Test



Training
normalized

Test
normalized

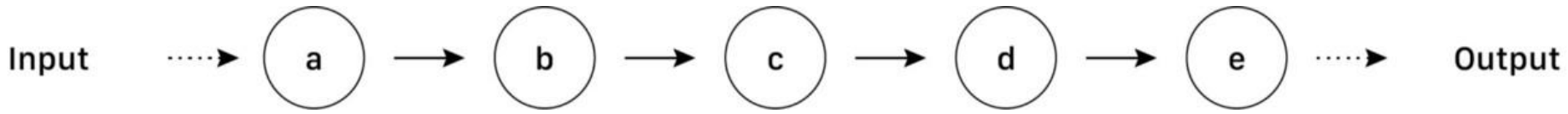




Weights update **modifies the input distribution of k** → ICS

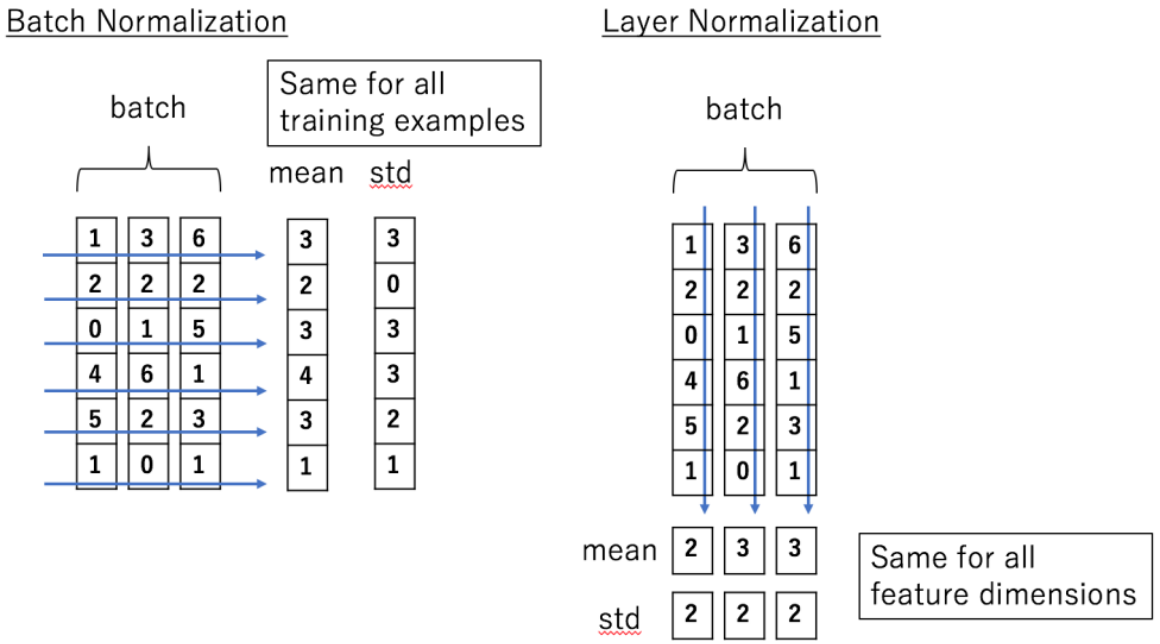
Working behind BN: H2

- BN **mitigates the interdependency between layers during training**.
- (a), (b), (c), (d) and (e) are the sequential layers of the network. Let's assume we want to train such model using SGD.
- If all gradients are large, grad(a) will be very large. On the contrary, if all gradients are small, grad(a) will be almost negligible.
- Modification of (a) weights will modify the input distribution of (b) weights, which will eventually modify the input signal of (d) and (e).
- This interdependency could be problematic for training stability : **if we want to adjust the input distribution of a specific hidden unit, we need to consider the whole sequence of layers**.



$$grad(a) = grad(b) * grad(c) * grad(d) * grad(e)$$

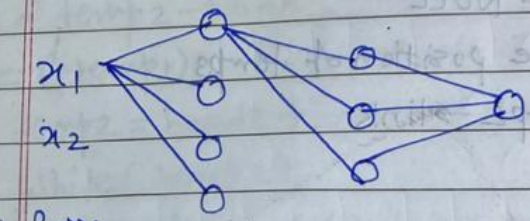
BN vs. LN



- In batch normalization, **input values of the same neuron for all the data in the mini-batch are normalized**. Whereas in layer normalization, input values for all neurons in the same layer are normalized for each data sample. the normalization is performed using other features from a single example instead of using the same feature across all current batch examples.
- For convolutional networks (CNN) : Batch Normalization (BN) is better
- For recurrent network (RNN) : Layer Normalization (LN) is better

$w^1 = 4*2$ $w^2 = 3*4$ $w^3 = 1*3$

$b^1 = 4*1$ $b^2 = 3*1$ $b^3 = 1*1$



FW Pass.

$z^1 = (4*2)\cdot(2*1) + (4*1)$

$= 4*1$

$a^1 = 4*1$

$z^2 = (3*4)\cdot(4*1) + (3*1)$

$= (3*1)$

$a^2 = (3*1)$

$z^3 = (1*3)(3*1) + (1*1)$

$= (1*1)$

$a^3 = (1*1)$

Loss.

$L =$

For m samples

$A^1 = Z^1 = 4*m$

$A^2 = Z^2 = 3*m$

$A^3 = Z^3 = 1*m$

$dz^3 = (a-y)$

$dw^3 = (1*1)\cdot(1*3) = (1*3)$

$db^3 = (1*1)$

$dz^2 = [(3*1)\cdot(1*3)] * (3*1) = (3*1)$

$dw^2 = (3*1)\cdot(1*4) = (3*4)$

$db^2 = (3*1)$

$dz^1 = [(4*3)\cdot(3*1)] * (4*1) = (4*1)$

$dw^1 = (4*1)(1*2) = (4*2)$

$db^1 = (4*1)$

$dz^3 = 1*m$

$dw^3 = 1/m\ (1*m)\cdot(m*3) = (1*3)$

$db^3 = (1*1)$

$dz^2 = (3*1)\cdot(1*m) * (3*m) = 3*m$

$dw^2 = (1/m)\ (3*m)\ (m*4) = 3*4$

$db^2 = (3*1)$

Advantages of Batch Normalization Layer

1. Batch normalization improves the training time and accuracy of the neural network.
2. It decreases the effect of weight initialization.
3. It also adds a regularization effect on the network.
4. It works better with the fully Connected Neural Network (FCN) and Convolutional Neural Network.

Universal workflow of machine learning

- Define the problem at hand and the data on which you'll train. Collect this data, or annotate it with labels if need be.
- Choose how you'll measure success on your problem. Which metrics will you monitor on your validation data?
- Determine your evaluation protocol: hold-out validation? K-fold validation? Which portion of the data should you use for validation?
- Develop a first model that does better than a basic baseline: a model with statistical power.
- Develop a model that overfits.
- Regularize your model and tune its hyperparameters, based on performance on the validation data. A lot of machine-learning research tends to focus only on this step—but keep the big picture in mind.