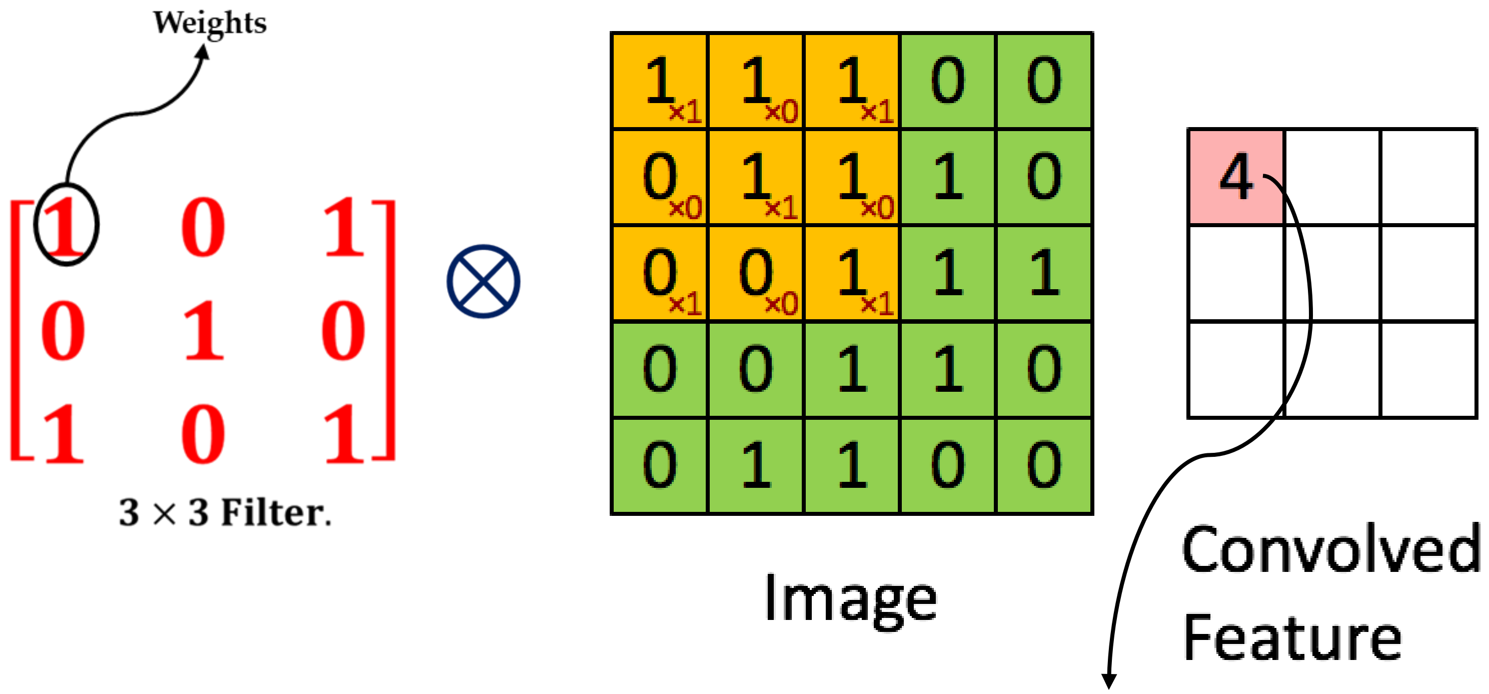
6CS012 – Artificial Intelligence and Machine Learning.
Tut – 06
Building a Good Image Classifier with CNN.

Siman Giri {Module Leader – 6CS012}

1.Recap – CNN.

1.1 A Convolution Operation.



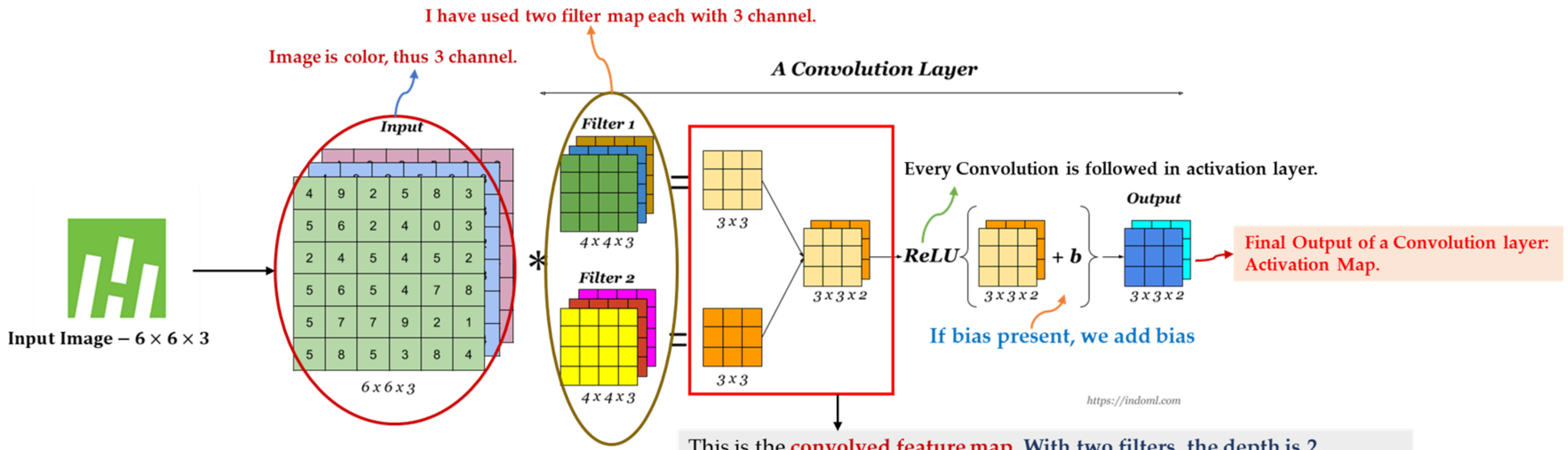
- **Hyper – Parameters:**
  - **Filter Dimension: 3 × 3.**
  - **Stride: (1, 1)**
  - **Padding: valid.**

$$\mathbf{O_{11} = 1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4}$$

$$\mathbf{(I * F)_{x,y} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I(x+i, y+j) \cdot F(i,j)}$$

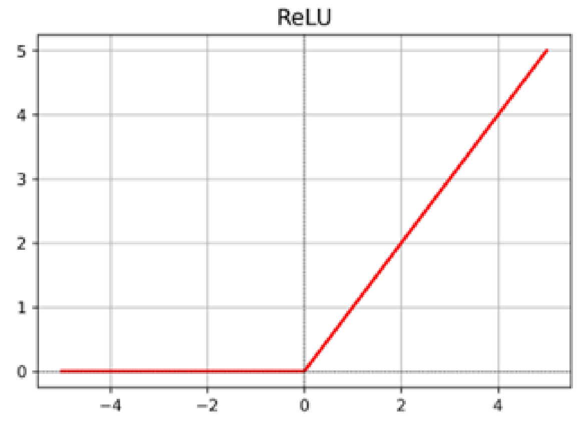
1.2 One Complete Convolution Operation.

- After the convolution operation we apply an activation function, the go to choice of activation function for CNN Models are ReLU.



This is the convolved feature map. With two filters, the depth is 2.
The input dimension of 6 × 6 × 3 is reduced to 3 × 3 × 2 after convolution.
In practice, depth is increased by using more filters.
Cautions: If we use same padding then dimension will not be reduced.

1.3 Activation Function: ReLU.



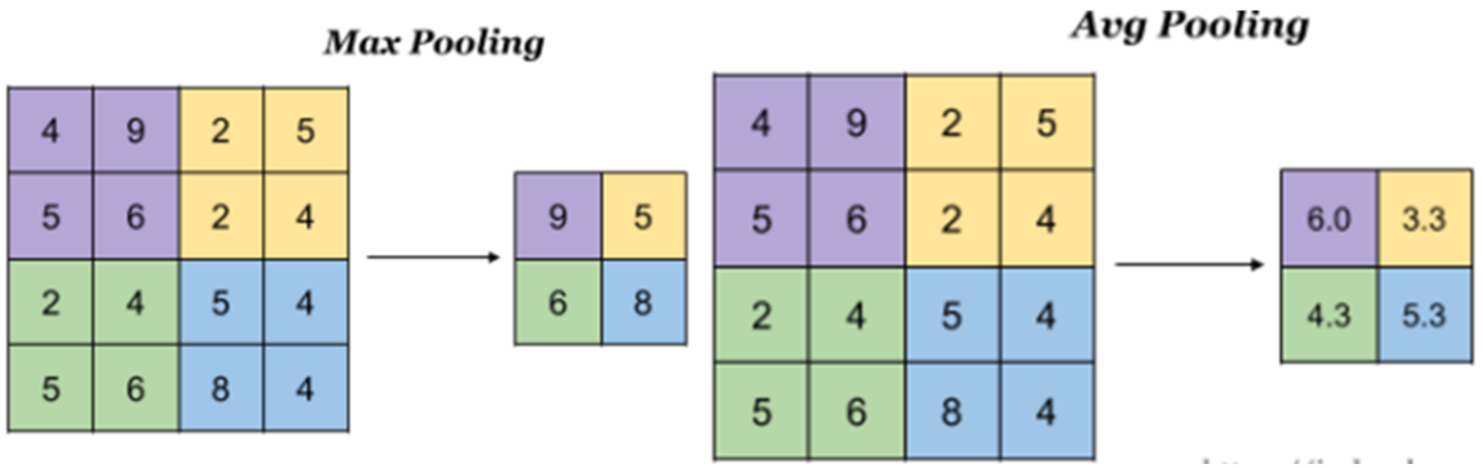
- **Mathematical Representation:**
  - $\mathbf{f(z) = max(0, z)}$
  - ReLU outputs the input directly if it's positive, otherwise, it outputs zero.
- **Pros:**
  - **Efficient Computation**: ReLU is computationally simple because it only involves a thresholding operation, which makes it faster to compute than sigmoid.
  - **No vanishing gradients:** For **positive values**, ReLU does not suffer from the vanishing gradient problem because its **derivative is 1** (for positive inputs).
  - **Sparsity:** Since negative inputs are set to zero, ReLU produces **sparse activations**, which can help with reducing overfitting and improve model efficiency.
- **Cons:**
  - **Dying ReLU problem**: When inputs are negative ReLU outputs zero, which means **the gradient is also zero**. If this happens too often, **some neurons may never activate (known as "dying ReLU"),** leading to situation where those **neurons are essentially useless during training**.
  - **Not zero – centered**: Although **ReLU provides non – zero values for positive inputs**, the negative outputs (zeros) can hinder **convergence as gradients are not zero – centered**.
- **When to use ?**
  - **Hidden layers in deep networks**: ReLU is widely **used hidden layers of deep neural networks**, despite the issue of dying ReLU problem **it is the go – to activation function for CNNs** because of its simplicity and performance

1.4 Output Dimension after Convolution Operation.

- A convolutional Layer is defined by following **hyperparameters**:
  - **Number of Filters (K):** The number of feature detectors applied to the input. Determines the depth of the output volume.
  - **Filter Size (F):** The spatial size of each Filter, **typically 3×3 or 5×5**, 7×7 are also used by some design, but above than 7 are used rarely.
  - **Stride (S):** The most common choice is 1.
  - **Padding (P):**
    - **Zero Padding (P):**The number of zero pixels added around the input's border.
    - **Same Padding ($P > 0$):** Keeps the output same size as input.
    - **Valid Padding($P = 0$):** No extra padding, output size shrinks.
- Output Shape Calculation:
  - Given an input of size $\mathbf{W_{in} \times H_{in} \times C_{in}}$ the output volume dimensions are:
    - $\mathbf{W_{oc} = \frac{(W_{in} - F + 2P)}{S} + 1}$
    - $\mathbf{H_{oc} = \frac{H_{in} - F + 2P}{S} + 1}$
    - $\mathbf{C_{oc} = K}$ (Number of filters, which determines the depth of the output)

1.5 Pooling Layer

- One of the objectives of Convolution Operation is also to reduce the spatial dimension (i.e. **height × width**) and increase the depth i.e. **channel**.
    - However, this reduction in spatial dimensions is not always guaranteed in convolution layer, especially when using **same padding**, which preserves **the spatial dimensions**.
- Therefore, a pooling layer is typically used **after convolution** to **reduce spatial dimensions** while **retaining essential information.**
- **Type of Pooling Operation:**



1.6 Output Dimension after Pooling.

- **Hyper-parameters in the Pooling Layer:**
  - **Size of Filters (F):** For max pooling, it is typically $\mathbf{2 \times 2}$.
  - **Stride (S):** When used $\mathbf{2 \times 2}$ **pooling filter stride of 2** is most common, this reduces the spatial dimension by half.
- **A pooling layer takes an input of volume:**
  - $\mathbf{W_{oca} \times H_{oca} \times C_{oca}}$ $\{\mathbf{oca \rightarrow Output\ after\ convolution\ and\ activation.}\}$
- And produces an output of volume:
  - $\mathbf{W_{ocap} \times H_{ocap} \times C_{ocap}}$ $\{\mathbf{ocap \rightarrow Output\ after\ convolution, activation\ and\ pooling.}\}$
- Where:
  - $\mathbf{W_{ocap} = \frac{W_{oca} - F}{S} + 1}$
  - $\mathbf{H_{ocap} = \frac{H_{oca} - F}{S} + 1}$
  - $\mathbf{C_{ocap} = C_{oca}}$ **(depth remains unchanaged)**
- The Pooling Layer does not have learnable parameters (weights or biases), so no weight updates are required during training.

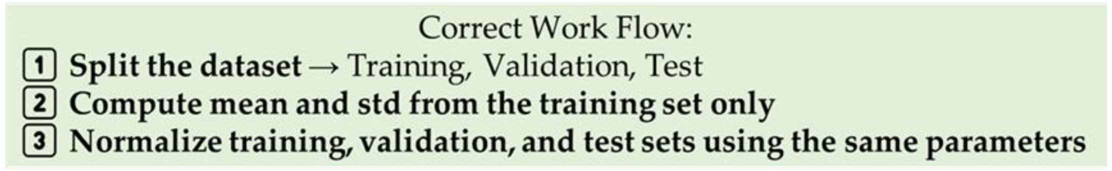
Exercise – 1.

2. Towards Stable Training and Just Right Model.

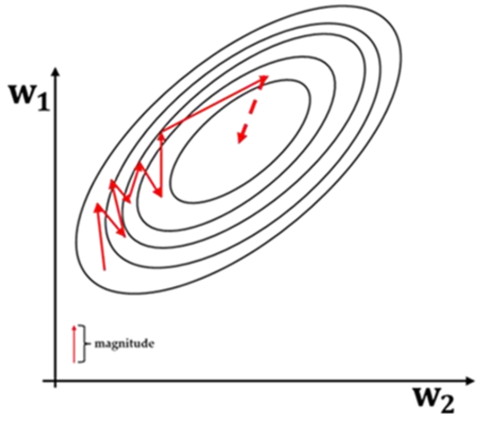
{Tips and Tricks}

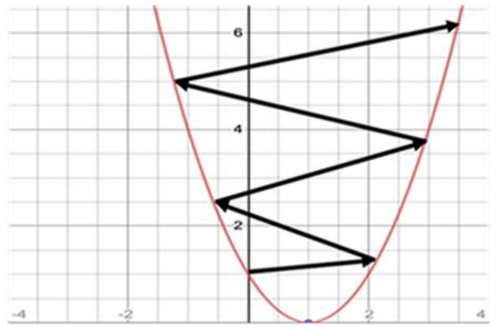
2.1 Data Pre – processing: Normalizations.

- The goal of normalization is to transform features to be on a similar scale.
- This {shall} improve the performance and training stability of the model.
- Some Popular techniques on performing Normalizations:
  - **scaling to a range {min-max Scaling}**
    - $\mathbf{x}' = \frac{(\mathbf{x} - \mathbf{x}_{\mathbf{minimum}})}{\mathbf{x} - \mathbf{x}_{\mathbf{maximum}}} == \frac{\mathbf{x.}}{\mathbf{255}}$
  - **z-score – Standard Normal Distributions aka standardizations.**
    - $\mathbf{X}' = \frac{\mathbf{x} - \mathbf{mean}}{\mathbf{standard\ deviation}}$
- Cautions: Always Normalize after proper train – Val and test split.
  - {Prevents Data Leakage and Helps in Generalizations.}

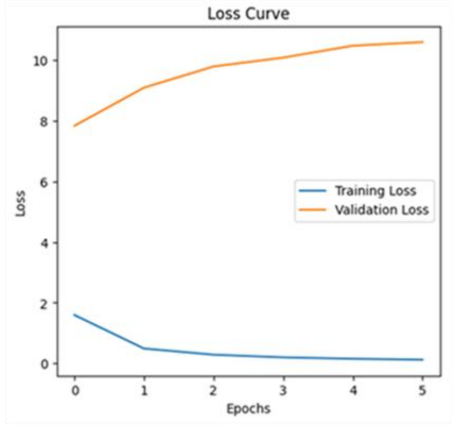

Correct Work Flow:
1 **Split the dataset** → Training, Validation, Test
2 **Compute mean and std from the training set only**
3 **Normalize training, validation, and test sets using the same parameters**

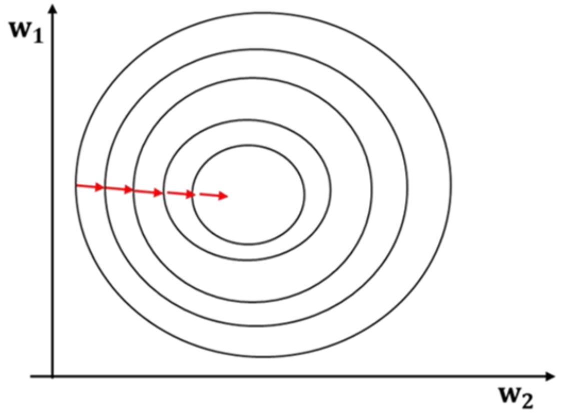
2.1.1 Normalizations Really Helps.

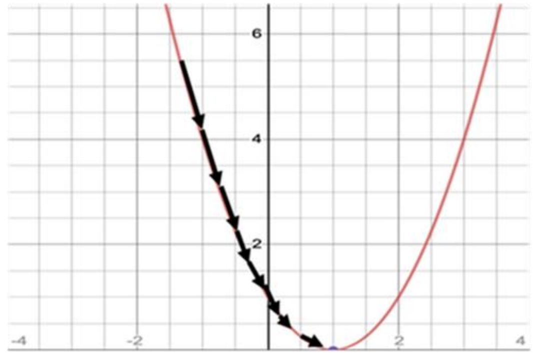




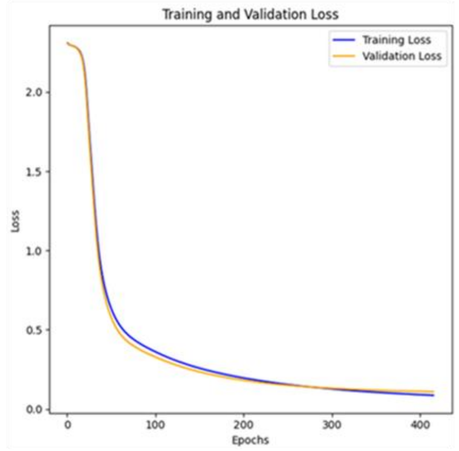
Without Normalizations.



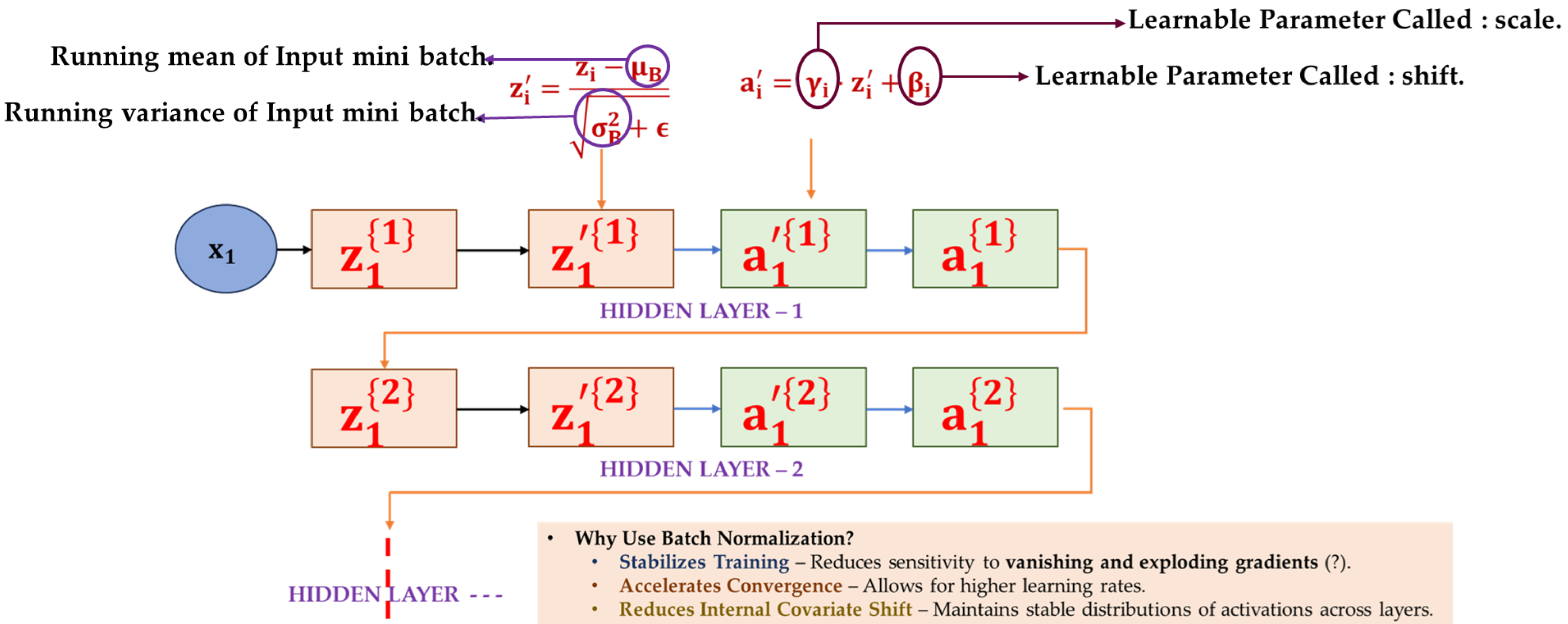




With Normalizations.



2.2 Batch Normalizations: An Idea.


Running mean of Input mini batch.
Running variance of Input mini batch.
$z_i' = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
$a_i' = \gamma_i z_i' + \beta_i$
Learnable Parameter Called : scale.
Learnable Parameter Called : shift.
x_1
$z_1^{\{1\}}$
$z_1'^{\{1\}}$
$a_1'^{\{1\}}$
$a_1^{\{1\}}$
HIDDEN LAYER – 1
$z_1^{\{2\}}$
$z_1'^{\{2\}}$
$a_1'^{\{2\}}$
$a_1^{\{2\}}$
HIDDEN LAYER – 2
HIDDEN LAYER - - -

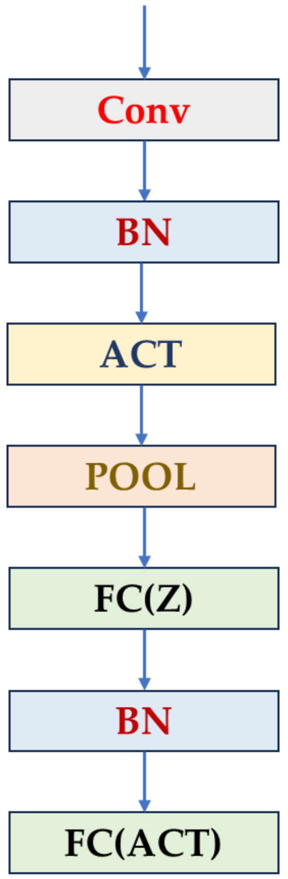
- **Why Use Batch Normalization?**
  - Stabilizes Training – Reduces sensitivity to **vanishing and exploding gradients** (?).
  - Accelerates Convergence – Allows for higher learning rates.
  - Reduces Internal Covariate Shift – Maintains stable distributions of activations across layers.
  - Acts as Regularization – Can reduce the need for **dropout** (?).

2.2.1 Batch Normalizations: Test Time.

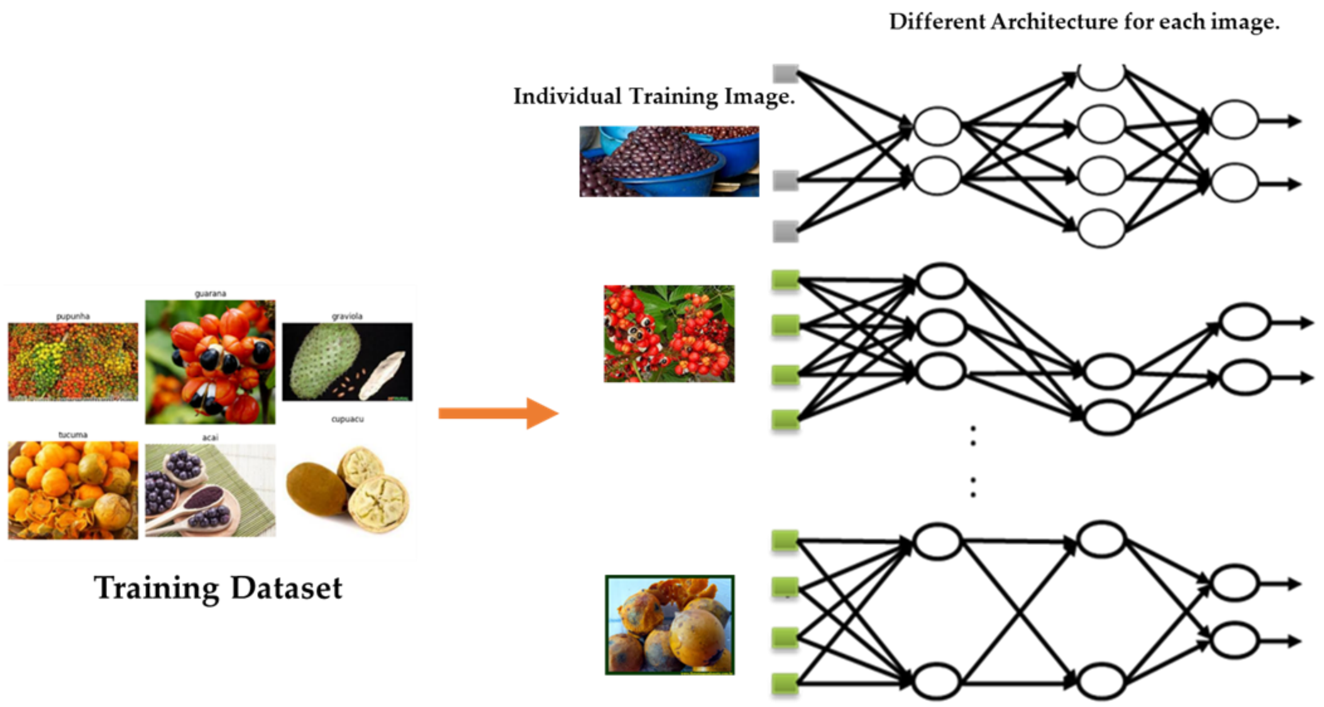
- Input: $\mathbf{z : N \times D \{ N \rightarrow number\ of\ samples, D \rightarrow Dimension(Height, Width, Channel)\}}$;
  - Applied before activation.
  - $\mathbf{\mu_B\ and\ \sigma_B^2}$ $\rightarrow$ are estimated based on the mini-batch;
    - however, since we do not have a mini-batch during testing, the running mean and variance from the last batch of training data are used.
  - Learnable scale and shift parameters: $\boldsymbol{\gamma, \beta} \rightarrow \{\mathbf{shape\ A\ vector\ of\ length\ D}\}$
    - learned during training.
- Application:
  - $\mathbf{z'_{i(H \times W)} = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \{z_{i(H \times W)} \rightarrow Every\ Pixel\ of\ Convolved\ Output, Shape \rightarrow N \times D\}}$
  - $\mathbf{a'_{i(H \times W)} = \gamma_i \cdot z'_i + \beta_i \{ a'_{i(H \times W)} \rightarrow Every\ Pixel\ of\ z'_i, Shape \rightarrow N \times D\}}$

2.2.2 Batch Normalization: Pros & Cons.

- **Pros:**
  - Makes deep networks much easier to train!
  - Improves gradient flow.
  - Allows higher learning rates for faster convergence.
  - Networks becomes more robust to initialization.
- **Cons:**
  - Behaves differently during training and testing: this is a very common source of bugs.



2.3 DropOut: An Idea.


Different Architecture for each image.
Individual Training Image.
pupunha
guarana
graviola
tucuma
acai
cupuacu
Training Dataset

- Is it Possible?
  - If Yes:
    - How?
      - Intuition Use DropOut.

2.3.1 DropOut

- In **DropOut**, different input images experience different **effective neural architectures** based on the dropout **probability p**.
- This happens because Dropout randomly deactivates neurons during each forward pass **only during training**, effectively creating a different sub-network for each input sample.
- **Key Points about DropOut:**
  - **During Training:**
    1. Each neuron is **dropped** (set to 0) with **probability p**.
    2. The remaining active neurons are **scaled by** $\frac{1}{1-p}$ to maintain expected activations.
    3. This creates different architectures for different training samples, introducing regularization.
  - **During Testing (Inference):**
    1. **Dropout is disabled** (all neurons are active).
    2. No neurons are randomly dropped.
    3. Scaling is **not applied**, ensuring consistent predictions.

2.4 Sample Implementations.

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Define a CNN model with Dropout and FCN with Batch Normalization
model = models.Sequential([
    # Convolutional Block
    layers.Conv2D(16, (3, 3), padding='same', activation=None, input_shape=(32, 32, 3)),
    layers.BatchNormalization(),  # Batch Normalization before dropout
    layers.Dropout(0.5),  # Dropout applied after BN
    layers.ReLU(),  # Activation after dropout
    layers.MaxPooling2D((2, 2)),  # Pooling after activation
    # Flatten the output of Conv2D layers before feeding into FCN
    layers.Flatten(),
    # Fully Connected Block
    layers.Dense(128, activation=None),  # Fully connected layer without activation yet
    layers.BatchNormalization(),  # Batch Normalization before dropout
    layers.Dropout(0.5),  # Dropout applied after BN
    layers.ReLU(),  # Activation after dropout
    layers.Dense(10, activation='softmax')  # Output layer for classification (10 classes)
])
# Model summary
model.summary()
```

A sample CNN Model with Batch Normalizations and DropOut.

2.5 Model Architecture and Parameters.

- In CNNs, the number of parameters in each type of layer can be computed based on the layer's structure.
- Below, is the break down the computations for **convolutional layers**, **fully connected layers**, and **pooling layers**.
  1. **Convolutional Layer Parameters:**
  - For a convolutional layer, the number of parameters depends on the kernel size, the number of input channels $(\mathbf{C_{in}})$, and the number of output channels $(\mathbf{K})$.
  - Formula for Convolutional Layer Parameters:
    - $\mathbf{Total\ Parameters = (F \times F \times C_{in} + 1) \times K}$
      - $\mathbf{F}$ → The size of the Filter or Kernel (e.g. $\mathbf{3 \times 3}$ **or** $\mathbf{5 \times 5}$)
      - $\mathbf{C_{in}}$ → The number of input channels for input layer after that it is depth of the convolved feature map volume)
      - $\mathbf{K}$→ The number of output channels for input layer after that it is number of filters applied at that layer.
      - $\mathbf{+1}$ → accounts for the bias term for each filter.
  - This expression is multiplied by $\mathbf{K}$ because each filter has its own set of weights and biases, and there are K filters.
- **Example: For first convolutional layer:**
  - Suppose we have: $\mathbf{F = 3 \times 3}$; $\mathbf{C_{in} = 1}$ input with gray scale image, $\mathbf{K = 16}$ i.e. 16 filters are applied.
  - The total parameters for this convolutional layer would be:
    - $\mathbf{Total\ Parameters = (F \times F \times C_{in} + 1) \times K = (3 \times 3 \times 1 + 1) \times 16 = (9 + 1) \times 16 = 160}$

2.5.1 Model Architecture and Parameters.

2. **Pooling Layer Parameters:**
    - Unlike convolutional or fully connected layers, **pooling layers** (such as MaxPooling or AveragePooling) do **not have trainable parameters**.
        - The pooling operation does not involve any weights or biases — its role is to reduce the spatial dimensions of the input (by performing operations like max or average) without learning from the data.
    - However, it does reduce the dimension;
        - for example, applying a **P × P pooling layer** with a **stride S** on an input of **size H × W** results in an output of size:
            - $\mathbf{H' = \frac{H-P}{S} + 1; W' = \frac{W-P}{S} + 1}$
        - For example:
            - applying a 2 × 2 maxpooling layer with **stride of 2** on a **32 × 32 feature map** will reduce its size to:
                - $\mathbf{H' = \frac{32-2}{2} + 1 = 16; W' = \frac{32-2}{2} + 1 = 16}$
                - Effectively halving the width and height.
    - This **reduction is crucial** as it determines the input size for the **next convolutional layer**, impacting the **count of parameters** in next layer.

2.5.2 Model Architecture and Parameters.

3. Fully Connected Layer Parameters:

- In a fully connected layer (FCN), each input neuron is connected to every output neuron.
  - The number of parameters depends on the number of input and output neurons as well as the bias term for each output neuron.
- Formula for Fully Connected Layer Parameters:
  - $\textbf{Total Parameters} = (\textbf{Input Units} + \mathbf{1}) \times \textbf{Output Units}.$
    - Input Units: The number of neurons in the input layer.
    - Output Units: The number of neurons in the output layer.
    - +1 : accounts for the bias term.
- Example: Suppose we have: 128 input neurons and 64 Output neurons,
  - The total parameters for this fully connected layer would be:
    - $\textbf{Total Parameters} = (\textbf{Input Units} + \mathbf{1}) \times \textbf{Output Units} = (\mathbf{128} + \mathbf{1}) \times \mathbf{64} = \mathbf{129} \times \mathbf{64} = \mathbf{8256}.$

2.6 Special Parameters in Special Layer.

- **Batch Normalization Layer:**
  - {*In batch normalization layer we do not have added weights and biases, but it has its own two parameters (scale and shift which has to be learned) Thus counted towards total learnable parameter*.}
  - BN is used to normalize the activations of the network, which can speed up training and improve stability.
  - Parameters in BN: There are two learnable parameters:
    - **γ: The scale factor for each feature.**
    - **β: The shift parameter for each feature.**
  - Formula for parameters in Batch Normalization:
    - For each feature map (i.e. for each channel in the output of convolutional layer which depends on K number of filter applied), there are two learnable parameters:
      - $\mathbf{Total\ Parameters = K_{\gamma} + K_{\beta} = 2 \times K}$
    - Example: If $\mathbf{K = 16}$, then the parameters for $\mathbf{\gamma = 16\ and\ for\ \beta = 16}$, Thus the total parameter is:
      - $\mathbf{Total\ parameters = 2 \times K = 2 \times 16 = 32}$.

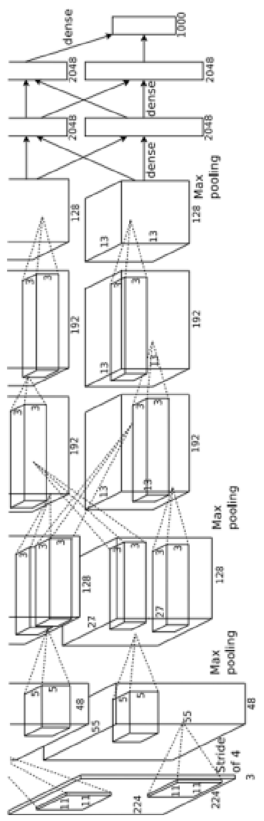
2.7 Special Operation 1×1 Convolution.

- A **1x1 convolution** is a special case of a convolutional operation where the **kernel size is 1x1,**
    - meaning the filter operates on a single pixel of the input at a time, but with the depth (number of channels) considered.
    - Despite its simplicity, 1x1 convolutions are widely used in modern deep learning models, such as **ResNet** and **Inception Networks**, for various purposes,
        - **including dimensionality reduction**, increasing the depth, and introducing non-linearity.
- **How does 1×1 Convolution Work?**
    - **Input:** The input to the 1×1 convolution is a feature map of size:
        - $\mathbf{H \times W \times C_{in}}$ where H is the height, W is the width, and $\mathbf{C_{in}}$ is the number of input channels.
    - Filter: The Filter size is $\mathbf{1 \times 1}$ with depth of **K** i.e. number of filter.
        - $\mathbf{1 \times 1 \times K}$
    - Operation: For each spatial location, the filter performs a weighted sum over the C_{in} input channels at the position, essentially transforming the input depth to a new depth, K.
- A 1×1 convolution operates across the channel dimension without affecting the spatial dimensions (height and width) of the feature map.

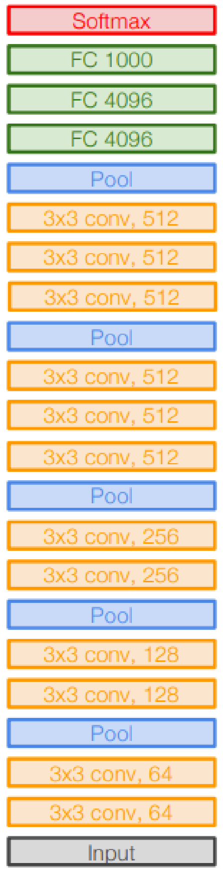
2.7.1 Special Operation 1×1 Convolution.

- Impact on Computational Complexity:
  - **Without $\mathbf{1 \times 1}$ convolution:**
    - $\mathbf{5 \times 5 \times 128}$ **convolution on $\mathbf{32 \times 32 \times 256}$ has:**
      - $\mathbf{5 \times 5 \times 256 \times 128 = 820,000}$ **parameters.**
  - **With $\mathbf{1 \times 1}$ convolution:**
    - Before applying
      - $\mathbf{5 \times 5 \times 128}$ **convolution on $\mathbf{32 \times 32 \times 256}$,**
      - let's first apply **$\mathbf{1 \times 1 \times 64}$ convolution**:
        - $\mathbf{32 \times 32 \times 256 \rightarrow}$ **apply $\mathbf{1 \times 1 \times 64}$ Convolution** $\mathbf{\rightarrow 32 \times 32 \times 64}$
    - Now apply **$\mathbf{5 \times 5 \times 128}$ convolution on $\mathbf{32 \times 32 \times 64}$**
    - Total parameters would be:
      - $\mathbf{16,384}$ **(for $\mathbf{1 \times 1}$ convolution) + $\mathbf{102,400}$ (for $\mathbf{5 \times 5}$ convolution) = $\mathbf{118,784}$ parameters.**

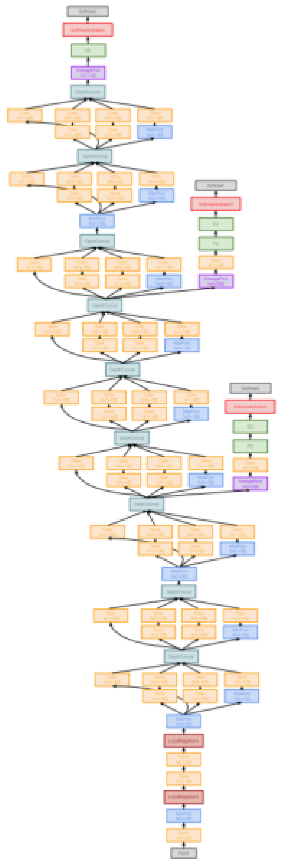
2.8 Some Popular Architecture.



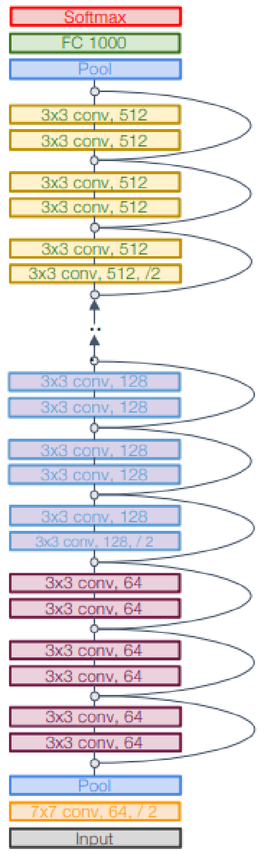
AlexNet



VGG



GoogLeNet



ResNet

Exercise – 2.

3. Better Optimization Algorithm.

{ Variants of Gradient Descent Algorithm.}

3.1 Before Gradient Descent.

- **Understanding Loss/Error Surface:**
  - It is the surface obtained by plotting average loss or error $\mathbb{E}$ against weights **W**?
  - The shape of the plot depends on the loss function we use, ideally, we want our plot to be convex as shown in picture.

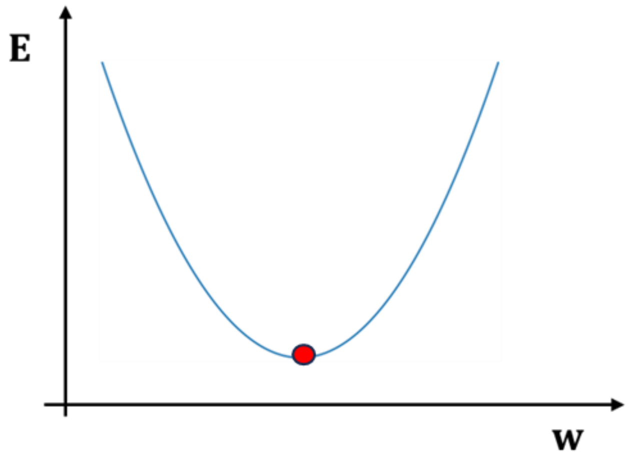


Fig: Expected Convex Loss Surface.

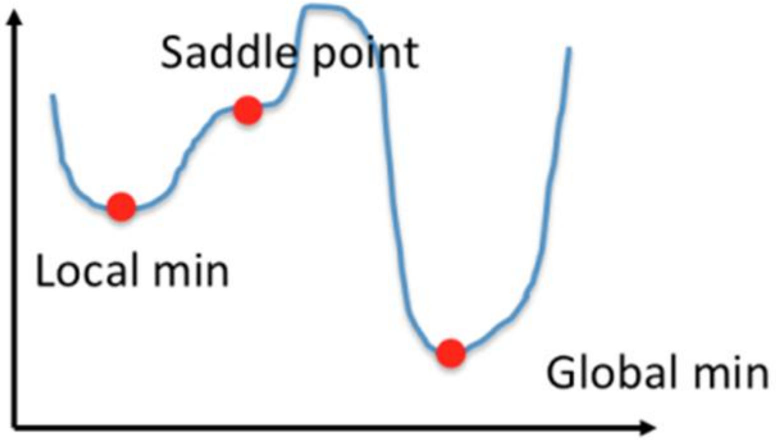
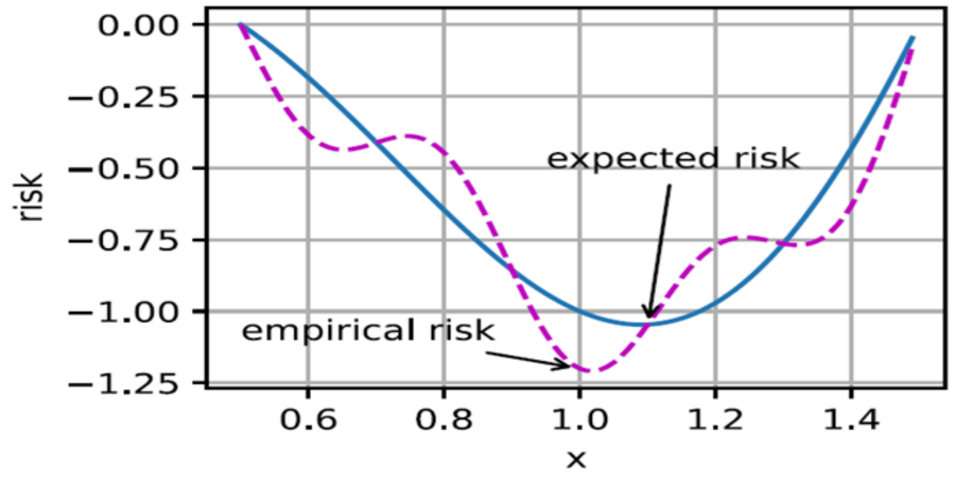


Fig: In Practice Non-Convex Loss Surface

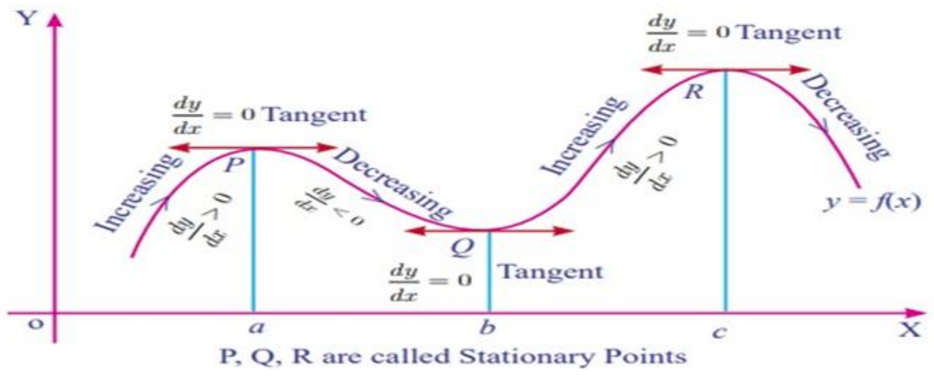
3.2 Challenges: Loss Surface in Practice.

- For a given empirical function *g* (dashed purple curve), optimization algorithms attempt to find the point of minimum empirical risk
- The expected function *f* (blue curve) is obtained given a limited amount of training data examples
- DL algorithms attempt to find the point of minimum expected risk, based on minimizing the error on a set of testing examples
  - Which may be at a different location than the minimum of the training examples
  - And which may not be minimal in a formal sense



3.3 Some Terminology: Stationary Points

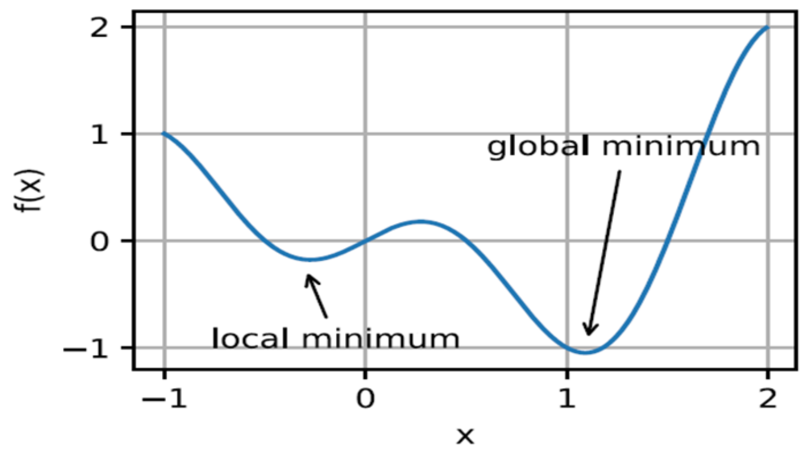
- **Stationary points** (or critical points) of a differentiable function f(x) of one variable are the points where the derivative of the function is zero, i.e., $f'(x) = 0$
- The stationary points can be:
  - **Minimum**, a point where the derivative changes from negative to positive
  - **Maximum**, a point where the derivative changes from positive to negative
  - **Saddle point**, derivative is either positive or negative on both sides of the point
- The minimum and maximum points are collectively known as extremum points



P, Q, R are called Stationary Points

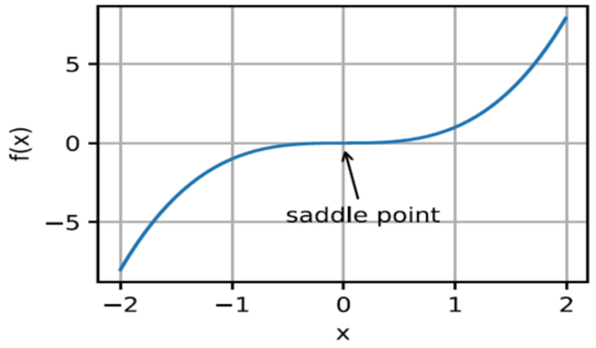
3.3.1 Some Terminology: Local Minima.

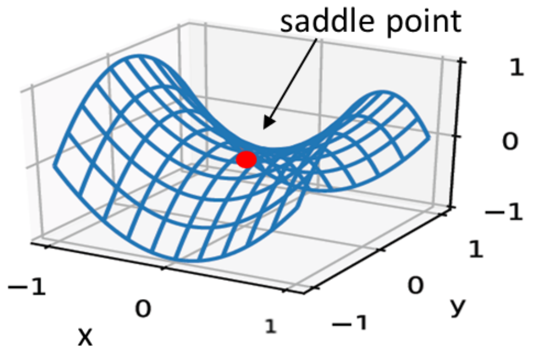
- Among the challenges in optimization of model's parameters in DL involve local minima, saddle points, vanishing gradients
- For an objective function $f(x)$, if the value at a point x is the minimum of the objective function over the entire domain of x, then it is the ***global minimum***
- If the value of $f(x)$ at x is smaller than the values of the objective function at any other points in the vicinity of x, then it is the ***local minimum***
    - The objective functions in DL usually have many local minima
        - When the solution of the optimization algorithm is near the local minimum, the gradient of the loss function approaches or becomes zero (vanishing gradients)
        - Therefore, the obtained solution in the final iteration can be a local minimum, rather than the global minimum



3.3.2 Some Terminology: Saddle Points.

- The gradient of a function $f(x)$ at a saddle point is 0, but the point is not a minimum or maximum point
  - The optimization algorithms may stall at saddle points, without reaching a minima
- Note also that the point of a function at which the sign of the curvature changes is called an inflection point
  - An inflection point ($f''(x) = 0$) can also be a saddle point, but it does not have to be
- For the 2D function (right figure), the saddle point is at (0,0)
  - The point looks like a saddle, and gives the minimum with respect to x, and the maximum with respect to y





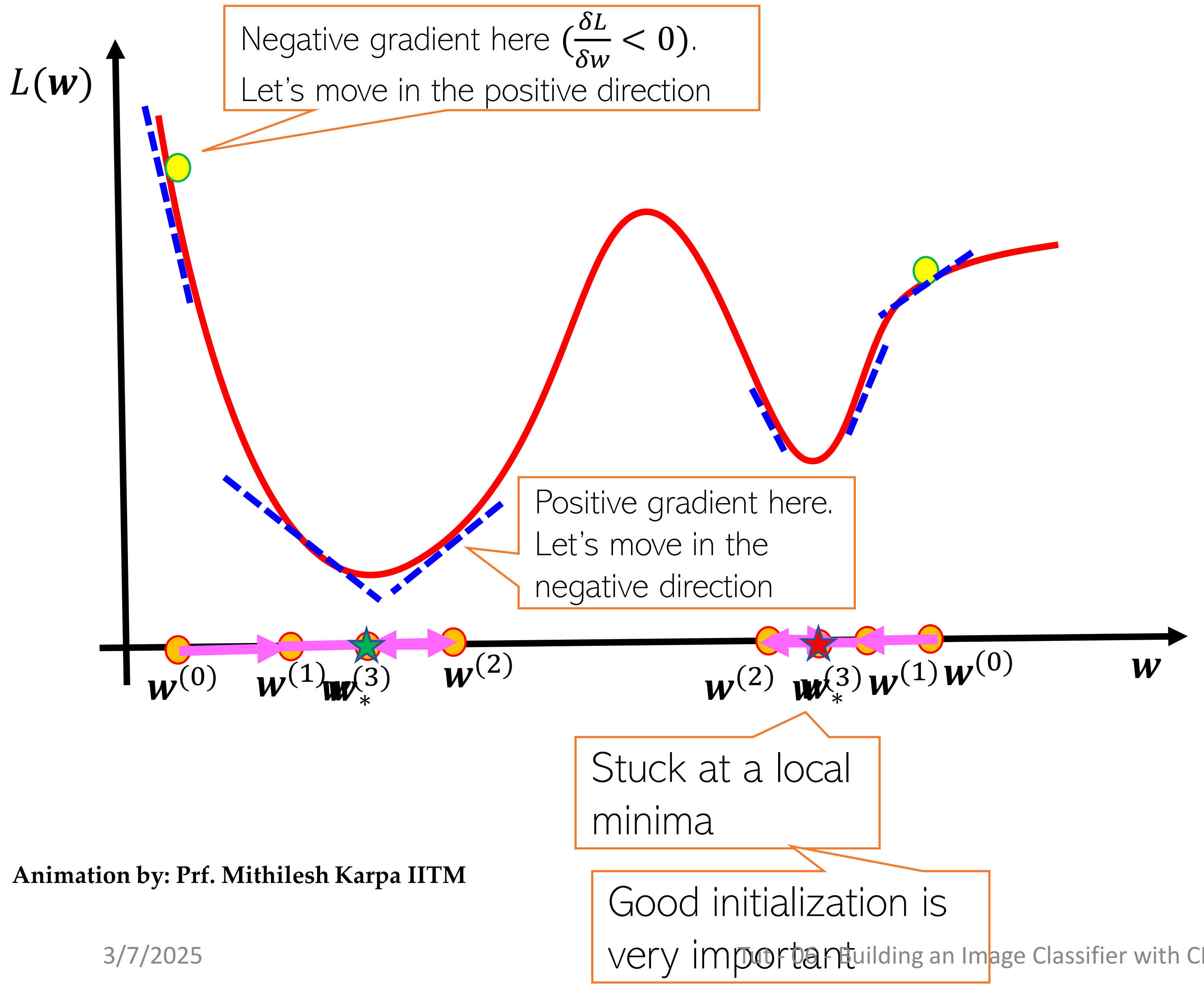
3.4 Remember: Gradient Descent.

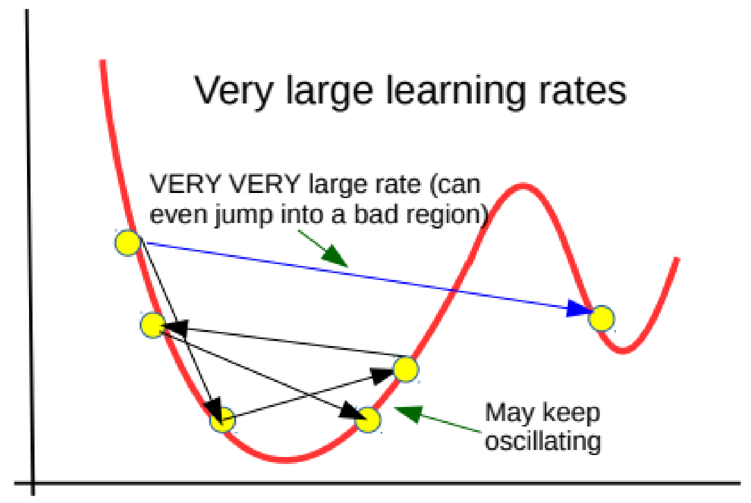
- It is an iterative methods used to compute minimum.
- The gradient ∇L at any point is the direction of the steepest increase. The negative gradient is the direction of steepest decrease.
- By following the –ve gradient, we can eventually find the lowest point.
- This method is called Gradient Descent.
- Algorithm:
  - For some cost/loss functions: $\mathbb{E}(w_0, \ldots, w_d)$.
  - Start off with some guesses for $w_0, \ldots, w_d$
    - It does not really matter what values you start off with, but a common choice is to set them all initially to zero
  - Repeat until Convergence:{

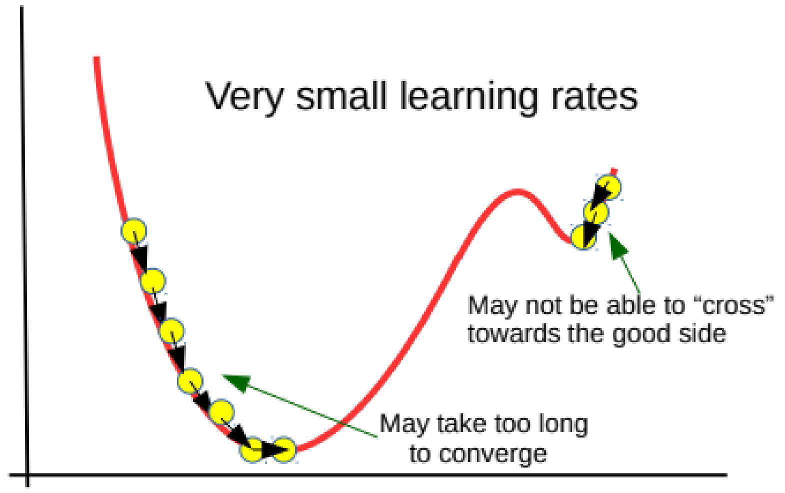
$$w_{new} \coloneqq \mathrm{w}_{\mathrm{old}} - \alpha \frac{\partial\, \mathbb{E}(w_0, \ldots, w_d)}{\partial w}$$

}

3.4.1 Gradient Descent: An Illustration


$L(\mathbf{w})$
Negative gradient here $(\frac{\delta L}{\delta w} < 0)$.
Let's move in the positive direction
Positive gradient here.
Let's move in the negative direction
$\mathbf{w}^{(0)}$ $\mathbf{w}^{(1)}$ $\mathbf{w}^{(3)}$ $\mathbf{w}_*$ $\mathbf{w}^{(2)}$
$\mathbf{w}^{(2)}$ $\mathbf{w}^{(3)}$ $\mathbf{w}_*$ $\mathbf{w}^{(1)}$ $\mathbf{w}^{(0)}$
$\mathbf{w}$
Stuck at a local minima
Good initialization is very important

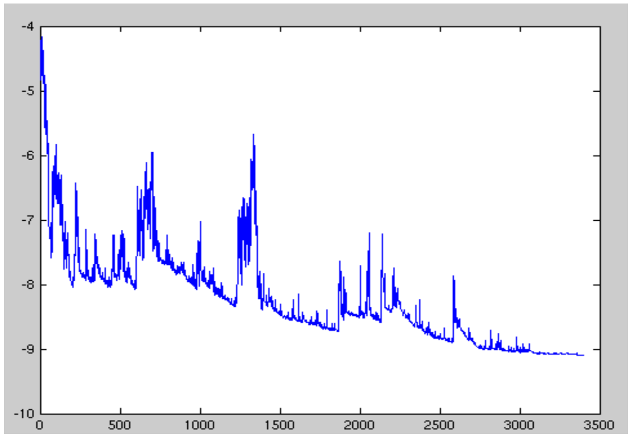

Very large learning rates
VERY VERY large rate (can even jump into a bad region)
May keep oscillating


Very small learning rates
May not be able to "cross" towards the good side
May take too long to converge

Animation by: Prf. Mithilesh Karpa IITM

3.5 Variants of GD: Stochastic Gradient Descent.

- In contrast SGD performs a parameter update **for each training example {loss function}** i.e. $\boldsymbol{x^i}$ **and label** $\boldsymbol{y^i}$.
  - **Update Rule:**
  - $\boldsymbol{w = w - \alpha \nabla_w J(w; x^i; y^i)}$.
- It is usually **much faster compared to BGD**, and also **can be used to learn online**.
- SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in image:
  - It can enable it to jump to new and potential better local minima.
  - It can also ultimately complicate the convergence to the exact minimum.



3.5.1 Variants of GD: Mini Batch Gradient Descent.

- It is the mixture of BGD and SGD i.e. it updates the parameter for every mini batch of n training examples:
  - **Update Rule:**
  - $\mathbf{w = w - \alpha \nabla_w J(w; x^{(i:i+n)}; y^{(i:i+n)}}$.
- Merits:
  - Reduces the variance of the parameter updates, which can lead to more stable convergence;
  - Efficient computation for large models.
- Common mini-batch size range between 50 and 256.
- {Recommended}

4. Speeding up mini – Batch Gradient Descent.

4.1 Error Landscape.

- A **Loss Surface** is a simplified 2D visualization showing how loss changes with respect to one or two model parameters, often depicted as a **curve or contour plot**.
- In contrast, an **Error Landscape** (or Loss Landscape) represents the loss function in a high-dimensional space, encompassing all possible parameter values in complex models like neural networks.
- While a loss surface provides an intuitive understanding of optimization paths,
  - the error landscape is inherently multidimensional, requiring dimensionality reduction techniques for visualization.

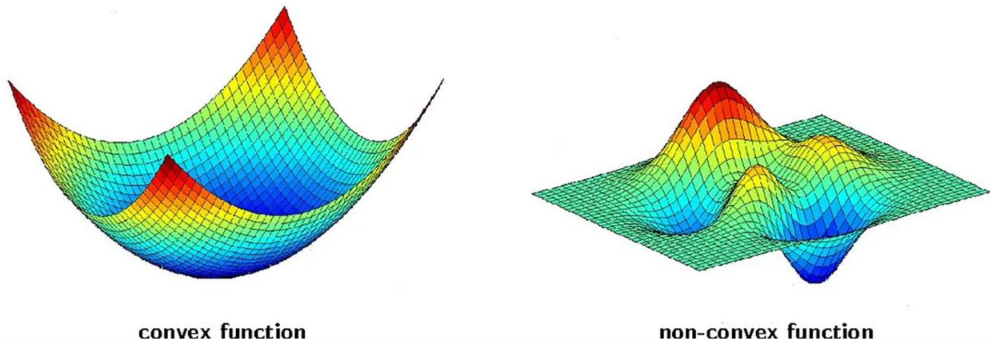


Fig: Error Landscape.

4.2 Momentum Optimization.

- Momentum helps **speed up gradient descent by accumulating past gradients**, reducing oscillations, and improving convergence.
- **Key Idea:**
  - Instead of updating parameters directly with gradients, Momentum smooth updates using a moving average of past gradients.
  - It helps in faster convergence and reduces oscillations in high – curvature regions.
- **Mathematical Formulation:**
  - Let v_t be the velocity term at time step t:
  - Compute velocity (exponentially weighted moving average of gradients):
    - $v_t = \beta v_{t-1} + (1-\beta)\Delta \mathrm{W}_{\mathrm{t}}$
      - Where β is the hyper parameter called momentum coefficient typically set at 0.9.
  - Update weights:
    - $w_t = w_{t-1} - \eta v_t$
- **Advantages:**
  - Faster Convergence.
  - Reduces oscillations in high – curvature areas.
  - Helps escape local minima.

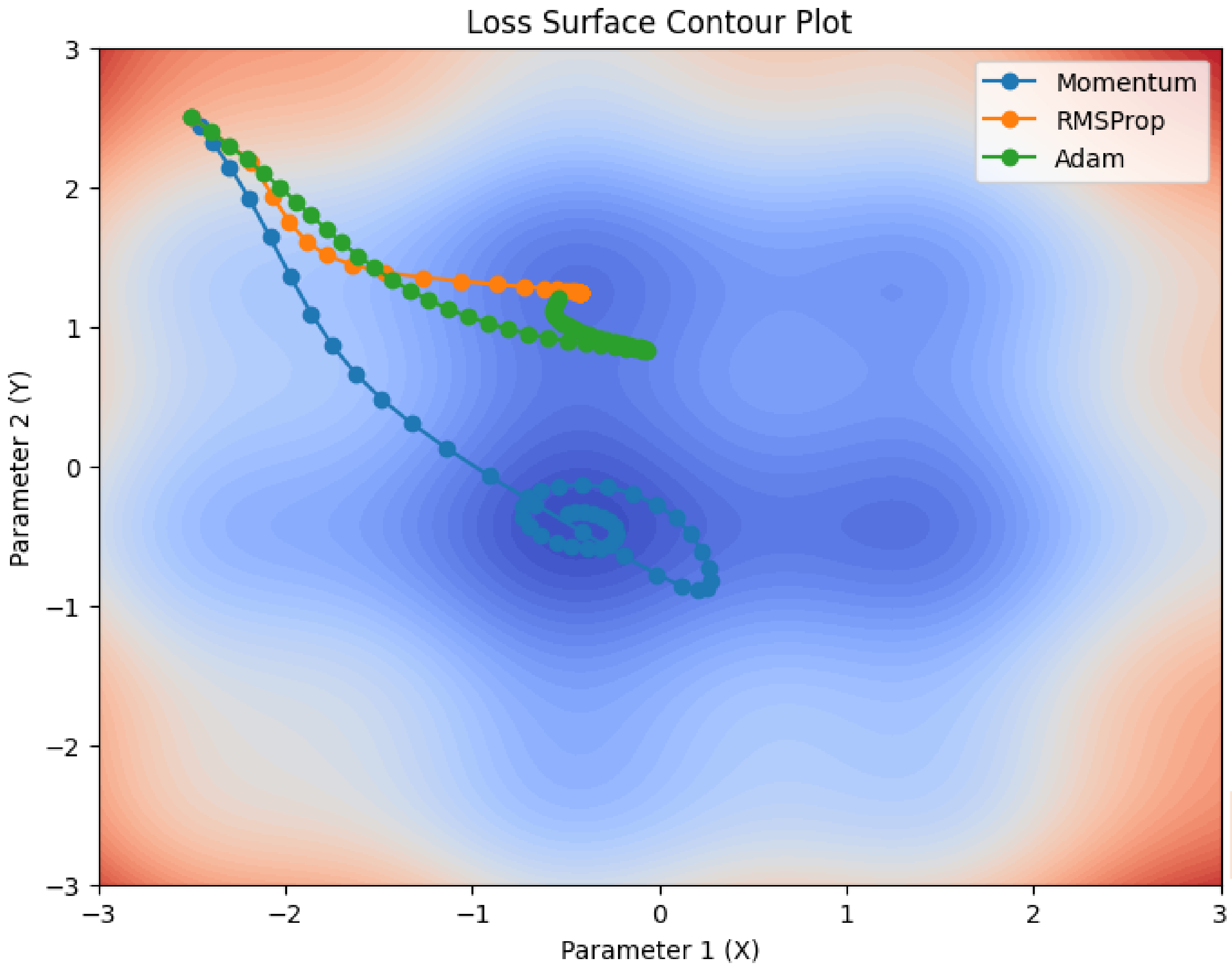
4.3 RMSProp (Root mean Square Propagation).

- RMSProp is designed to deal with the **vanishing learning rate problem** in **adaptive gradient methods** by maintaining an **exponentially decaying average of past squared gradients**.
  - Preferred Optimizer for RNN and Sequential Learning.
- **Key Idea:**
  - RMSProp accumulates an exponentially weighted moving average of past gradients.
- **Mathematical Formulation:**
  - Compute squared gradient moving average:
    - $v_t = \beta v_{t-1} + (1-\beta)(\Delta W_t)^2$ where β is typically set 0.9.
  - Update weights:
    - $w_t = w_{t-1} - \frac{\eta}{\sqrt{v_t + \epsilon}} v_t$
- **Advantages:**
  - Adapts learning rate for each parameter.
  - Works well for non-stationary problems (Hence preferred for Sequential input like text).
  - Avoids aggressive decay in learning rate.

4.4 Adam (Adaptive Moment Estimation).

- Adam is a combination of Momentum and RMSProp, making it one of the most popular optimization methods.
- Key Idea:
  - Uses Momentum to maintain moving averages of gradients.
  - Uses RMSProp to maintain moving averages of squared gradients.
  - Provides an adaptive learning rate for each parameter.
- Mathematical formulation:
  - Compute Momentum term:
    - $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\Delta(\mathrm{W_t})$
  - Compute RMSProp term:
    - $v_t = \beta_2 v_{t-1} + (1 - \beta_1)\left(\Delta(W_t)\right)^2$
  - Apply bias correction:
    - $\widehat{m}_t = \frac{m_t}{1-\beta_1^t}; \widehat{v}_t = \frac{v_t}{1-\beta_2^t}$
  - Update weights:
    - $W_t = W_{t-1} - \frac{\eta \cdot \widehat{m}_t}{\sqrt{\widehat{v}_t + \epsilon}}$
- Advantage:
- Works well on both NLP and Image Recognition Task , Preferred for CNN.

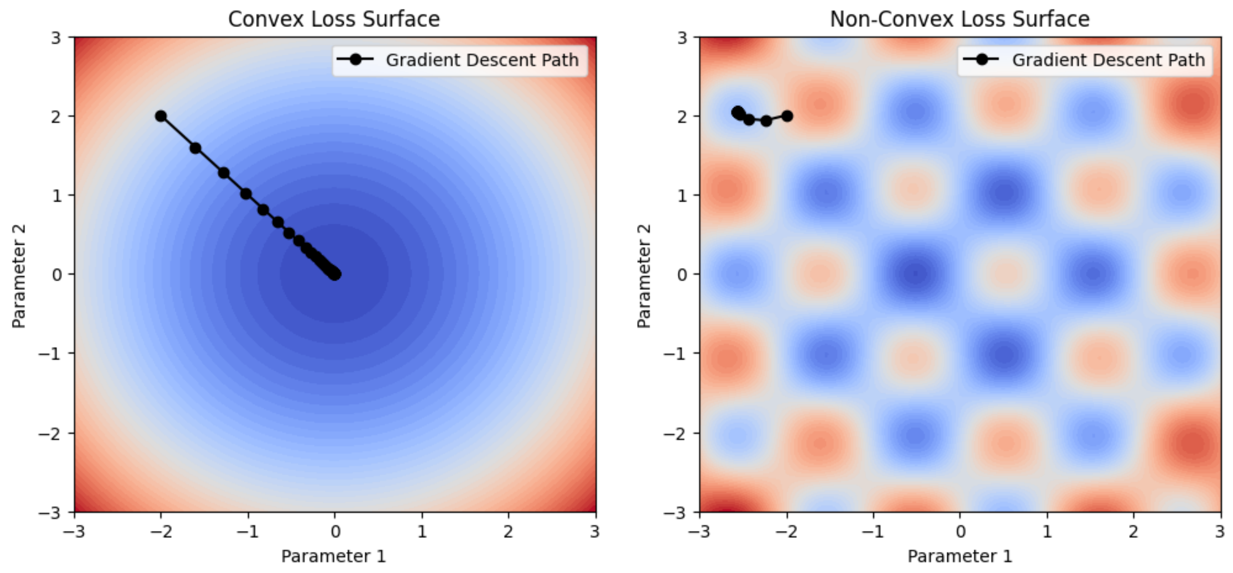
4.5 Choose Your Own Poisson.



- **My suggestion Start with mini – Batch Gradient Descent without adaptive learning rate.**
- **Then Try ADAM.**

4.6 Finally Good to Know.

- A Contour Plot:
  - A **contour plot** is a 2D representation of a **3D surface**, where each contour line connects points with the same function value (e.g., loss or error). It helps visualize how a function varies over two parameters.
  - The **closer the contour lines**, the **steeper** the slope, indicating rapid changes in the function (e.g., high gradients in optimization).
  - Widely spaced lines indicate **flatter regions** where the function changes gradually.
- Here's a **contour plot** illustrating a **Loss Surface** for both a convex and non-convex function:
- **Interpretation**
  - **Convex Loss**: Smooth, bowl-shaped function with a single global minimum.
  - **Non-Convex Loss**: Contains multiple local minima, making optimization more challenging.



Thank You.