6CS012 – Artificial Intelligence and Machine Learning.
Lecture – 06

Getting Started with Deep Learning
Transfer Learning

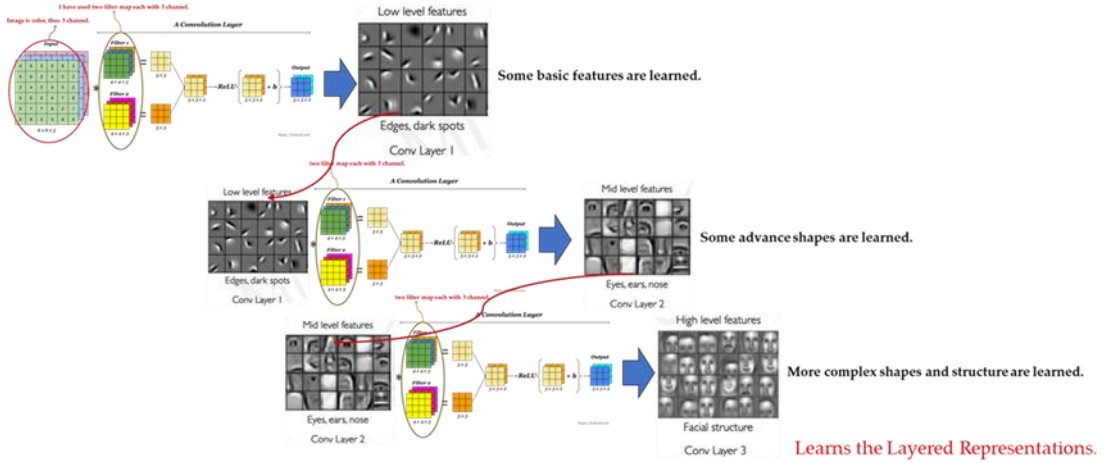
Siman Giri {Module Leader – 6CS012}

1. We solved the Challenges!!!!

{To Recap}

1.1 Challenges!!!!

- Challenge 1:
  - Automated Feature Learning with Model Parameters.



- Challenge 2:
  - Learning Non – Linear Decision Boundary.

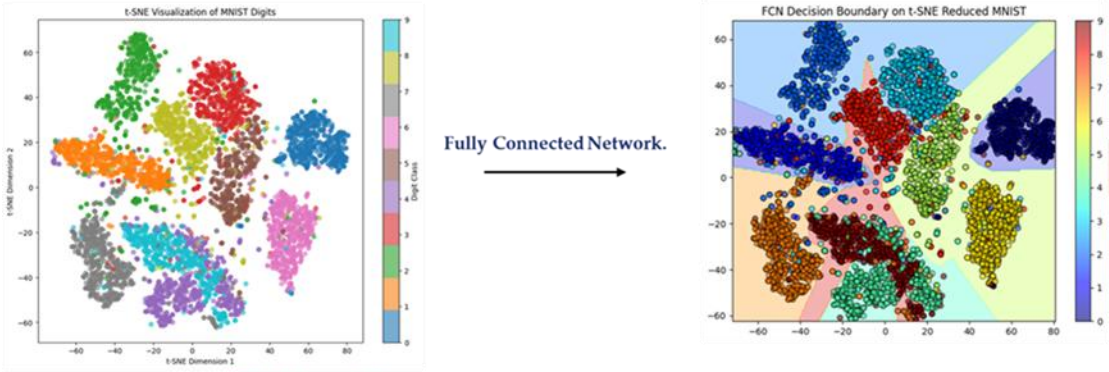


Fig: Non – Linear Decision Boundary and Multiclass Classification.

1.2 Where are We?

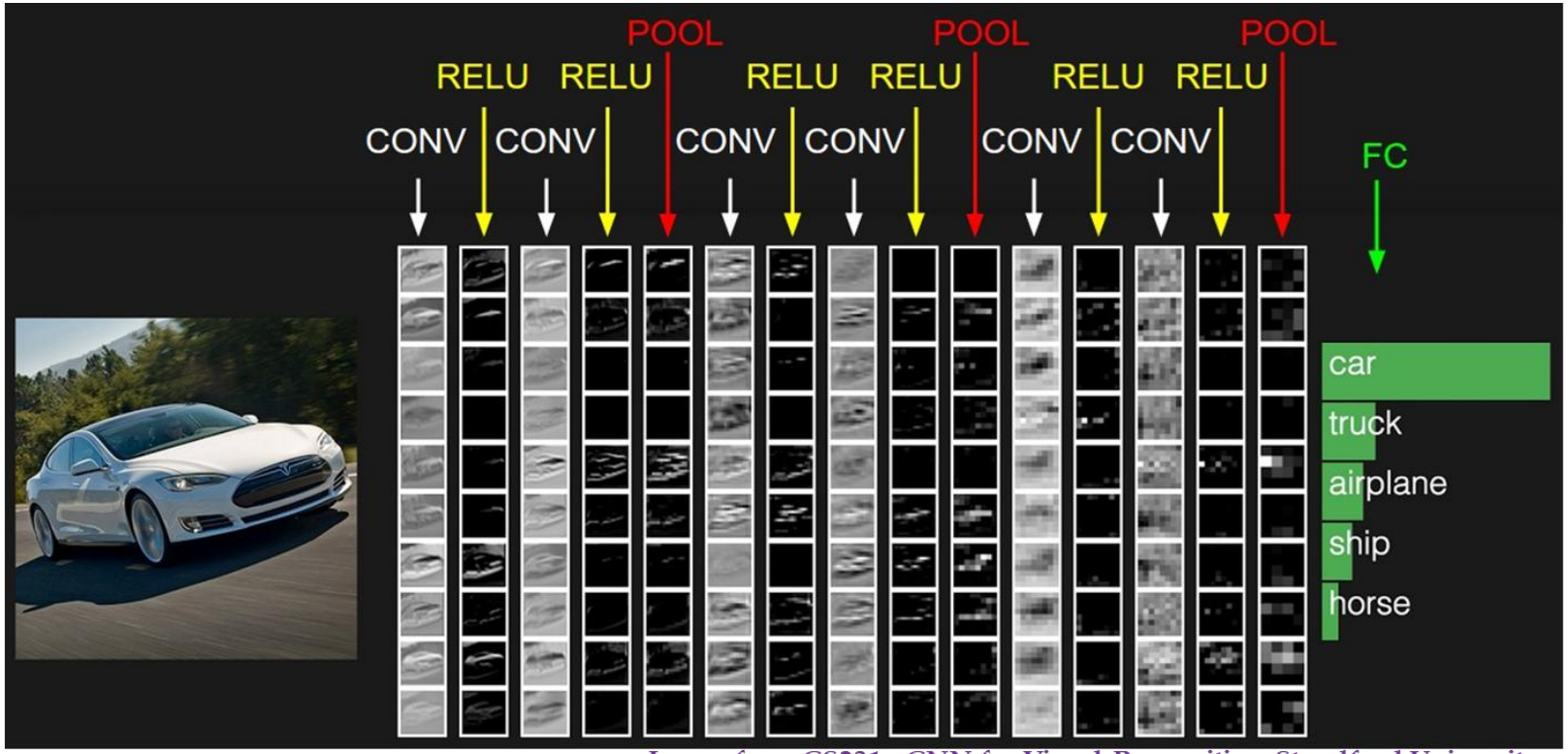


Image from CS231n CNN for Visual Recognition Standford University.

Fig: An example architecture:
{[Input] → CONV(ReLU) → CONV(ReLU) → Pool → CONV(ReLU) → CONV(ReLU) → Pool → CONV(ReLU) → CONV(ReLU) → Pool → Flatten → FCN → [Class Scores]}

1.3 An Architecture.

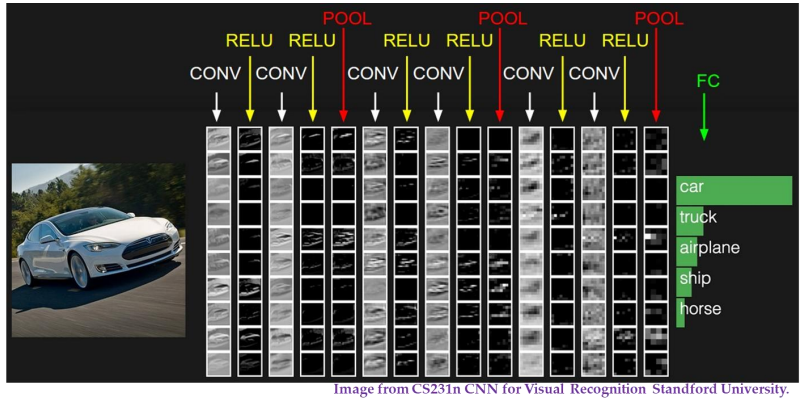


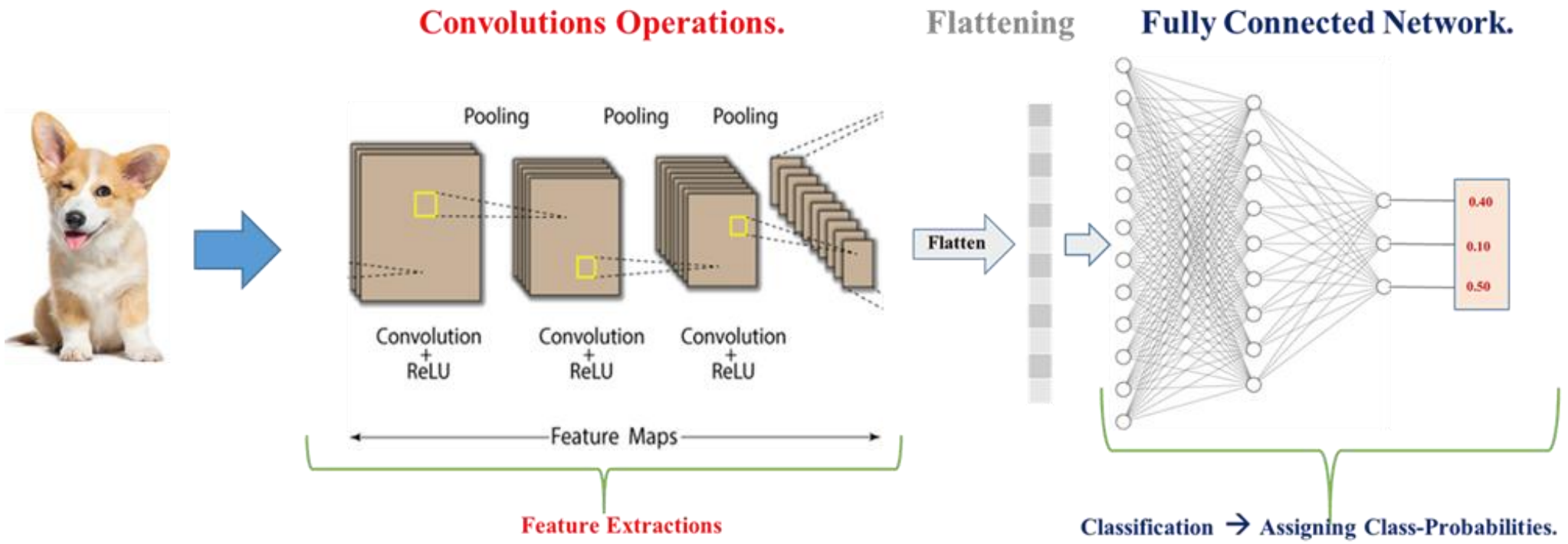
Image from CS231n CNN for Visual Recognition Standford University.

Fig: An example architecture:
{[Input] → CONV(ReLU) → CONV(ReLU) → Pool → CONV(ReLU) → CONV(ReLU) → Pool → CONV(ReLU) → CONV(ReLU) → Pool → Flatten → FCN → [Class Scores]}

A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (holding the class scores).

- There are a few foundational layer in every ConvNet architecture which are **{CONV(ReLU), Pool, FCN}.**
- Each layer accepts an
  - input 3D volume and transforms it to an output 3D volume through a differentiable function.
    - **[Height × Width × Channel]**
- Each layer may or may not have parameters.
  - E.g. CoNV and FCN do, ReLU and Pool don't.
- Each layer may or may not have additional hyperparameters.
  - E.g. CoNV, Pool, FCN does, ReLU doesn't
- Fully Connected Layer will compute the class scores, resulting the volume of size
  - **[1 × 1 × Number of Classes]**

1.4 Recap CNN Architecture.



1.5 Training ConvNets.

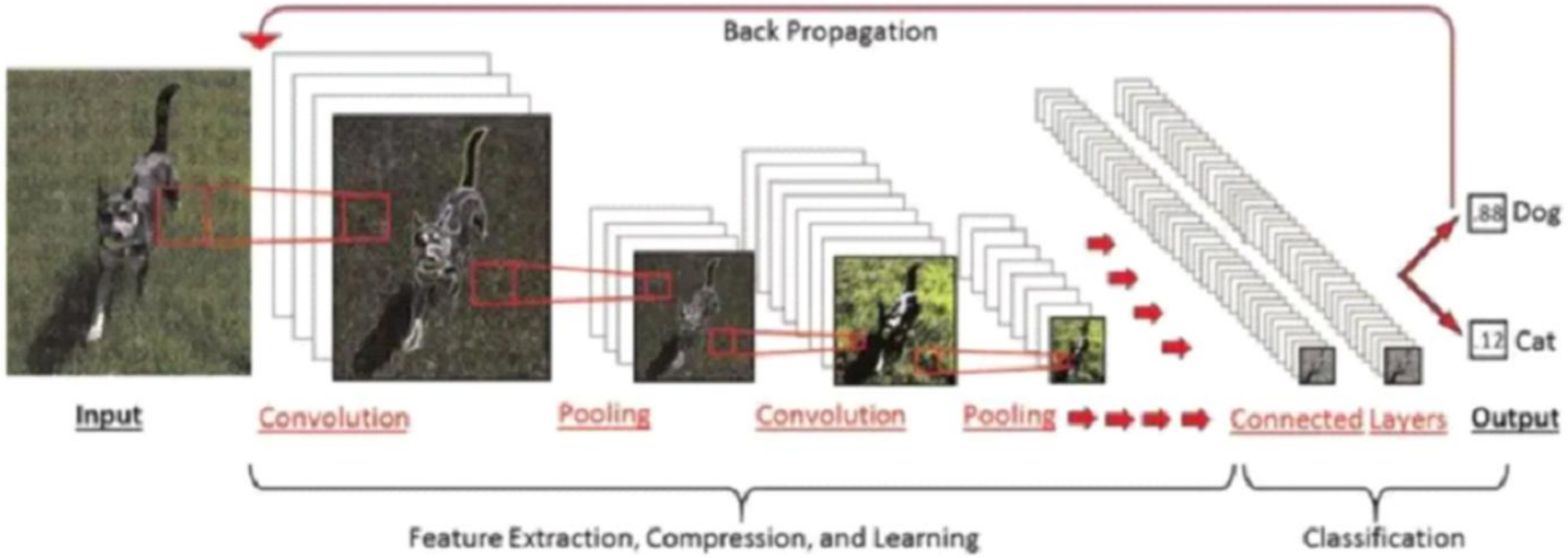


Image from Mickael Komendyak Blog post.

1.6 Learning Objective for the Week.

- **Challenges in Training CNNs: Avoiding Overfitting and Underfitting.**
  - Today, we'll explore key strategies to help Convolutional Neural Networks (CNNs) generalize well to unseen data.
    - One of the biggest challenges in training CNNs is finding the right balance
      - avoiding both overfitting (when the model memorizes training data but fails on new data)
        - and underfitting (when the model is too simple to capture patterns effectively).
- We'll discuss essential tips and techniques, including
  - data augmentation, regularization methods, architectural improvements, and optimization strategies, to improve the generalization of CNNs.

2. Tips and Tricks to Avoid Overfitting and Underfitting.

{Data and Data Pre – processing.}

2.1 Before Training : Train – Val – Test Split.

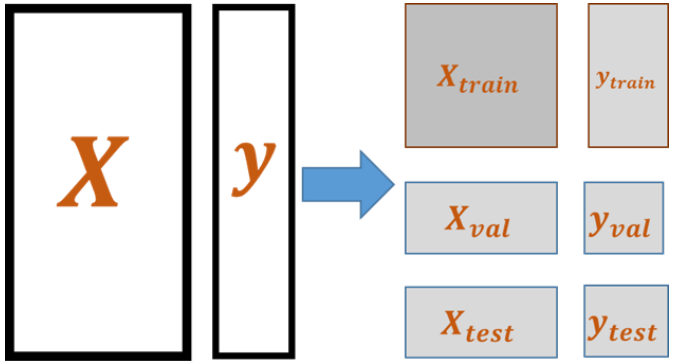
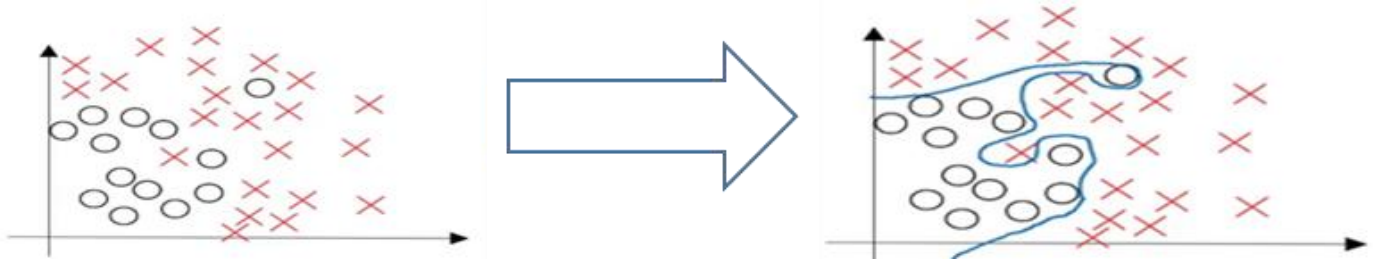


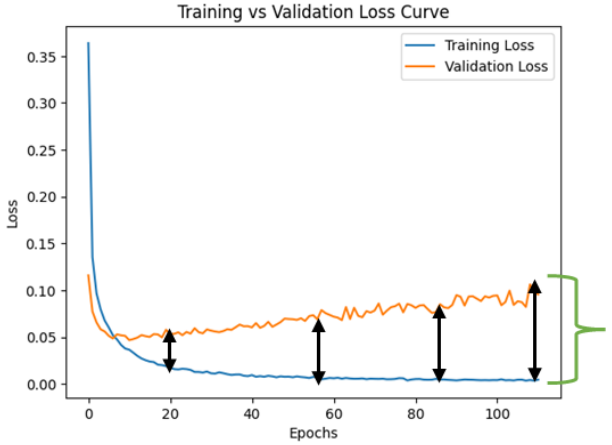
Fig: Three Way Holdout Methods.

- Before training a deep learning model, we split the dataset into three parts:
  - **Train Set:** Used for updating weights through backpropagation.
  - **Validation Set:** Used to monitor the model during training and detect overfitting.
  - **Test Set:** Used only after training to evaluate generalization performance.
- **How Train – Validation Split Helps Monitor Overfitting?**
  - During training, we observe the **Cost vs. Iteration (Epochs) Plot**:
    - If training loss continues decreasing while validation loss starts increasing, the model is overfitting.
    - Overfitting means the model **has started to memorize training data** instead of **learning the pattern**.
- Solution: **Apply Early Stopping**.
  - Early stopping **monitors validation loss** and **stops training if it starts increasing**.
    - This prevents the model from learning noise and memorizing training data.
- **Why do we need a Test Set?**
  - The test split is **never used during training (including during early stopping)**.
  - It gives an **unbiased measure of the model's final performance on unseen data**.

2.2 Recap Bias and Variance Tradeoff.

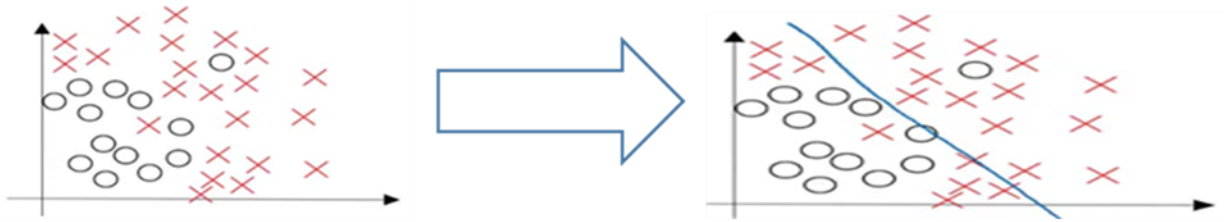


- **Overfitting → High Variance and Low Bias.**
- Characterized by:
  - Low error on Training Data But High error on Validation Data.

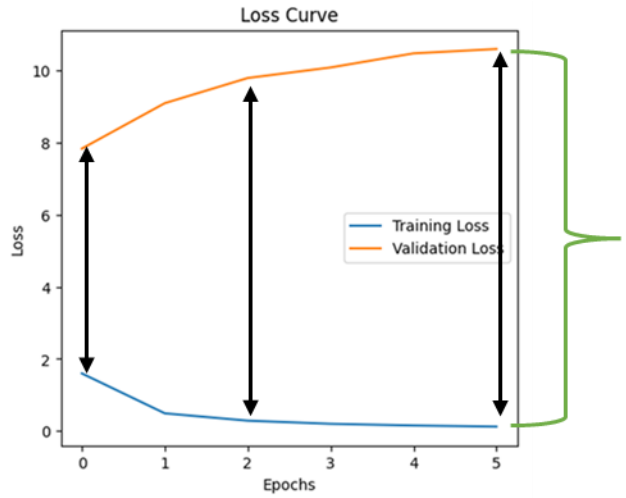


The widening gap between train and validation loss indicates the onset of overfitting.

2.2.1 Recap Bias and Variance Tradeoff.

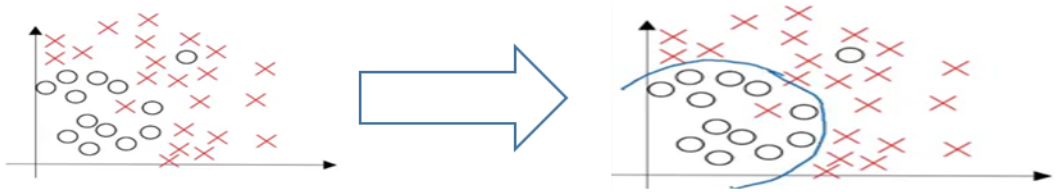


- **Underfitting → Low Variance and High Bias.**
- Characterized by:
  - High error on Training Data and Validation Data.

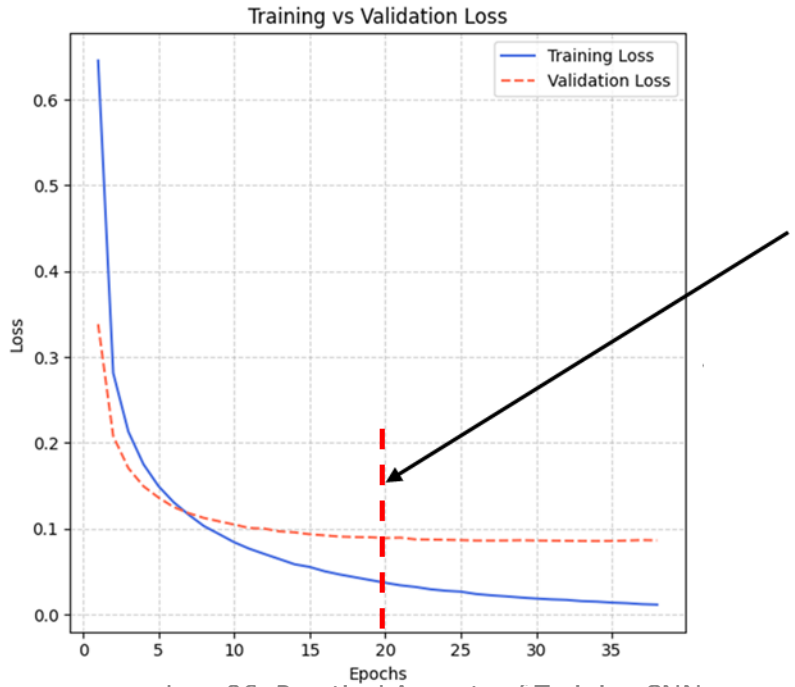


The increasing gap between train and validation loss suggests underfitting.

2.2.2 Recap Bias and Variance Tradeoff.



- Ideally we want our model to have Low Variance and Low Bias which is almost impossible,
  - Thus, as a tradeoff we pick a model which is "Just Right"



Apply Early Stopping Here.

2.1 Data: Get more Data.

- **Include more training data**
  - Simply introducing more data in the training set can remove overfitting of a complicated model as it previously may not have sufficient data to generalize well
- **Where to get more data?**
  - In most of the circumstances, **data are limited and collecting more data and labelling those data are labor intensive and extremely tedious task.**
- **Tips – 1: Apply Data Augmentation.**

2.1.1 Data Augmentations.

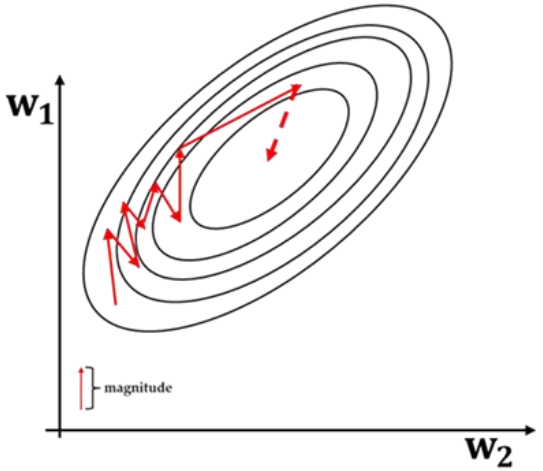
- **Data Augmentation** is a technique used in deep learning, especially in computer vision,
  - to artificially expand a dataset by creating modified versions of images.
- Why use **Data Augmentation**?
  - **Prevents Overfitting**: Increases dataset diversity without collecting new data.
  - **Improves Generalization**: Helps the model perform better on unseen data.
  - **Reduces Data Dependency**: Useful when collecting large datasets is difficult.
  - **Makes model more Robust**: Introduces real – world variations like lighting changes, rotations, and distortions.

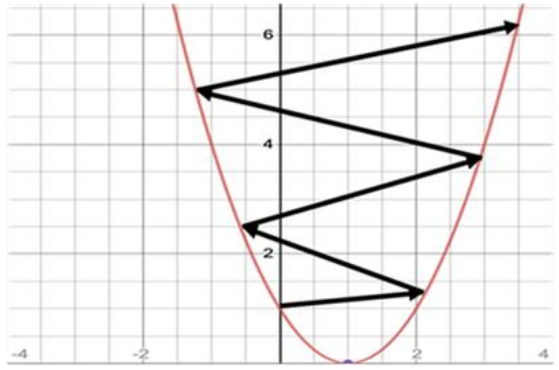
2.1.1.1 Common Techniques for Data Augementations.

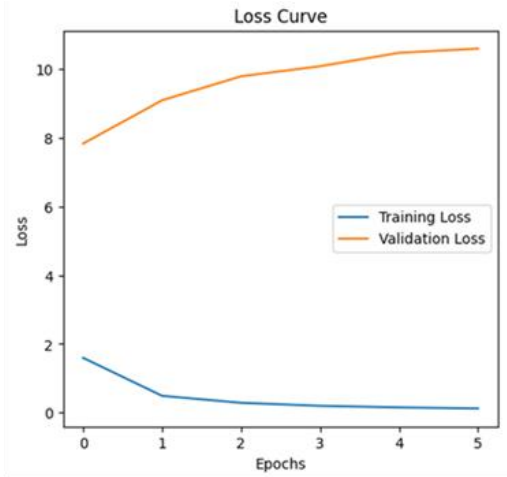
- Following are some common augmentation techniques used for CNNs are:
  - **Geometric Transformations: Some Examples are:**
    - Flipping: Horizontal or vertical reflection.
    - Rotation: Rotating images by small degree.
    - Cropping: Randomly selecting a part of the image.
    - Scaling: Resizing images while maintaining proportions.
    - Translation: Shifting images left, right, up or down.
  - **Color Transformations: Some Examples are:**
    - Brightness Adjustment: Making the image darker or brighter.
    - Contrast Change: Increasing or decreasing contrast.
    - Noise Addition: Some Examples are:
    - Adding random variations in pixel intensity.
  - **Occlusion and Information Loss: Some Examples are:**
    - Cutout: Hiding part of the image to force the model to focus on other details.

2.2 Data pre – processing: Normalizations.

- **Data Normalization** is a preprocessing technique used in machine learning and deep learning to **scale input data** so that it has a consistent range.
- Why consistent ranges are important {**Why Normalizations are important**}?
- **Keep in mind that we are using the same learning rate for all weights, so large parameters will dominate the updates**.







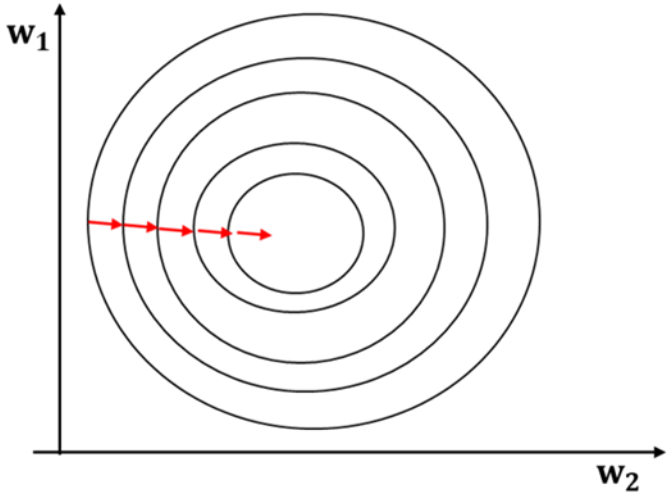
- The magnitude of the gradient depends on $\eta \times \frac{\partial L}{\partial W}$, which in turn depends on **w** .
  - If **w** is large, then $\eta \times \frac{\partial L}{\partial W}$ is also large (can cause **Exploding Gradient**),
    - and if **w** is small, $\eta \times \frac{\partial L}{\partial W}$ becomes small as well (can casus **Vanishing Gradient**).
- This both can affect **training stability causing model to underfit**.

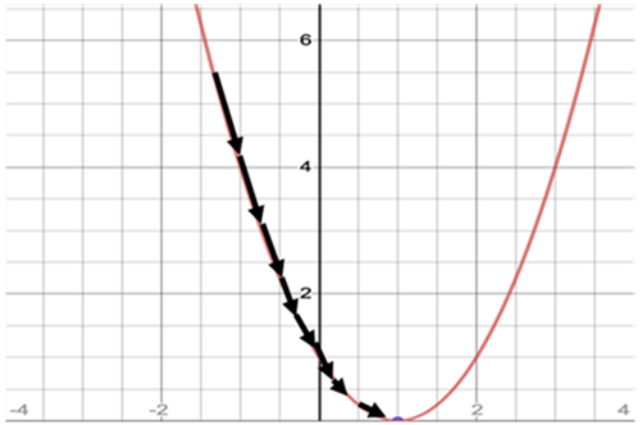
2.2.1 How to perform Normalizations?

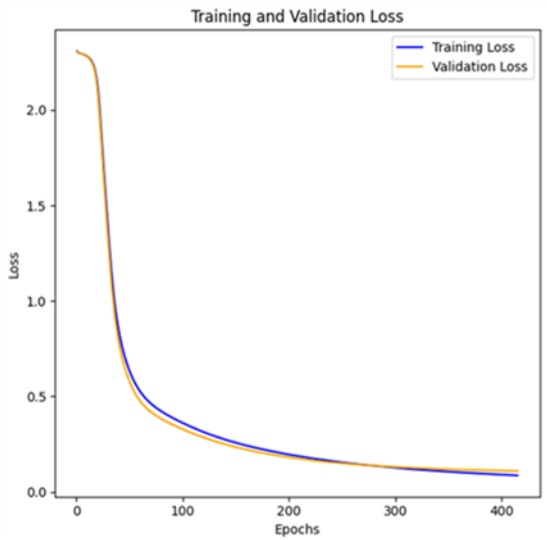
- The goal of normalization is to transform features to be on a similar scale.
- This {shall} improve the performance and training stability of the model.
- Some Popular techniques on performing Normalizations:
  - **scaling to a range {min-max Scaling}**
    - $\mathbf{x' = \frac{(x_i - x_{minimum})}{x_i - x_{maximum}} == \frac{x_{i.}}{255} == X \cdot \left(\frac{1.}{255}\right)}$
  - **z-score – Standard Normal Distributions aka standardizations.**
    - $\mathbf{X' = \frac{x_i - mean}{standard\ deviation}}$
- **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**

2.2.2 Normalizing really helps …







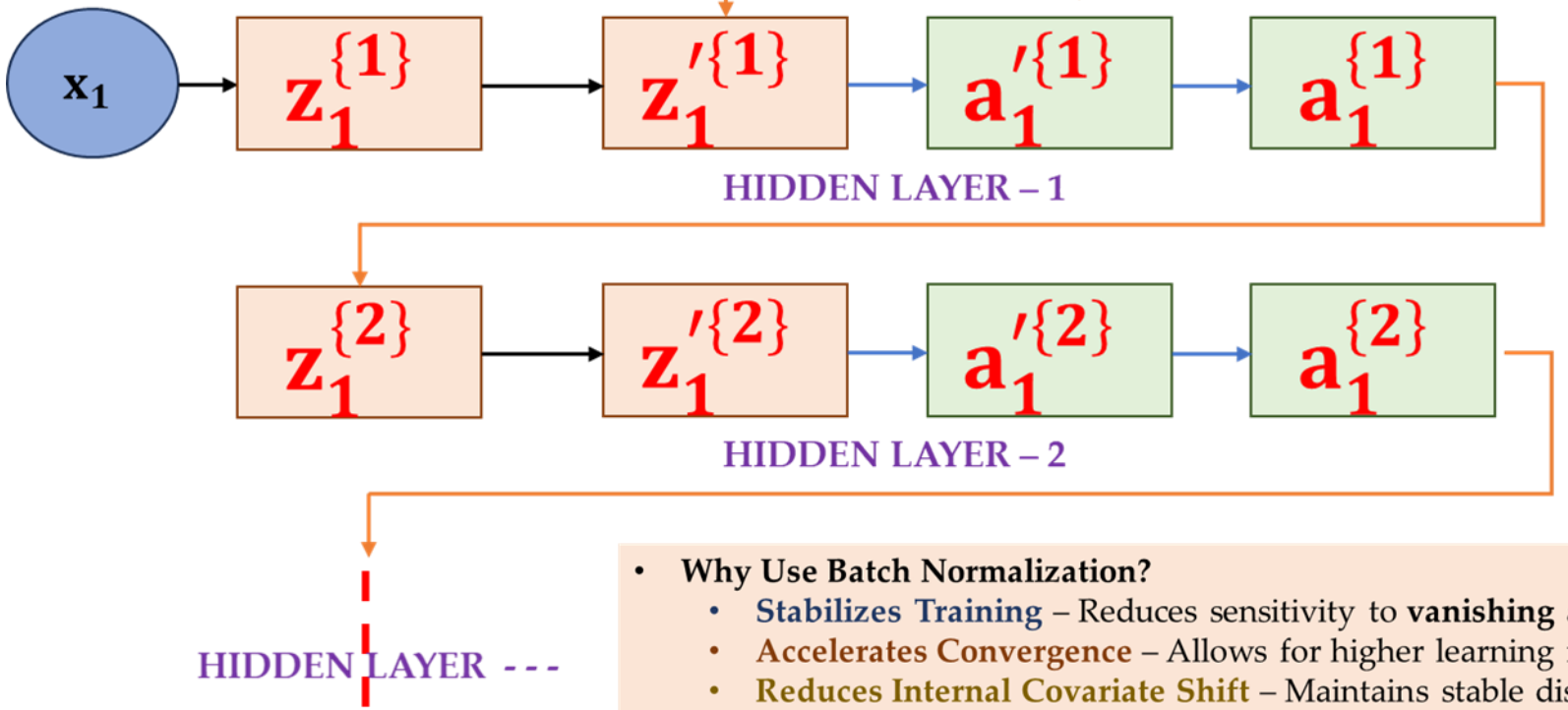
- **Q:** However, normalizing the inputs to the network only impacts the first hidden layer.
  - **What about the other hidden layers?**

2.3 Adding a Batch Normalizations Layer.

- Remember! we have been training our CNN and Fully Connected Neural Networks in **mini batches**.
  - {Cautions: Batch Normalizations does not mean normalizing at every mini batches.}
- How do we apply Batch Normalizations?
  - **BatchNorm Step 1: Normalize Net Inputs (z):**
    - Compute the Mean and Variance for the mini batch:
      - $\boldsymbol{\mu_B = \frac{1}{B}\sum_i^B z_i \; ; \sigma_B^2 = \frac{1}{B}\sum_i^B (z_i - \mu_B)^2}$ **{B → Batch Size}**
    - Use the Mean and Variance to normalize (standardize) net input (z)
      - $\boldsymbol{z_i' = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}}$ **{ϵ is very very small number added for numerical stability}**
  - **BatchNorm Step 2:**
    - Use Standardized $\boldsymbol{z_i'}$ to compute **Pre – activation Scaling** as:
      - $\boldsymbol{a_i' = \gamma_i \cdot z_i' + \beta_i}$
        - γ_i(scale → Controls the spread) and β_i(shift → Controls the mean) are learnable parameters.
- In a layman term, a BatchNorm Could learn to perform "**standardization**" with zero mean and unit variance.

2.3.1 To illustrate BatchNorm.

$$z_i' = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad a_i' = \gamma_i \cdot z_i' + \beta_i$$



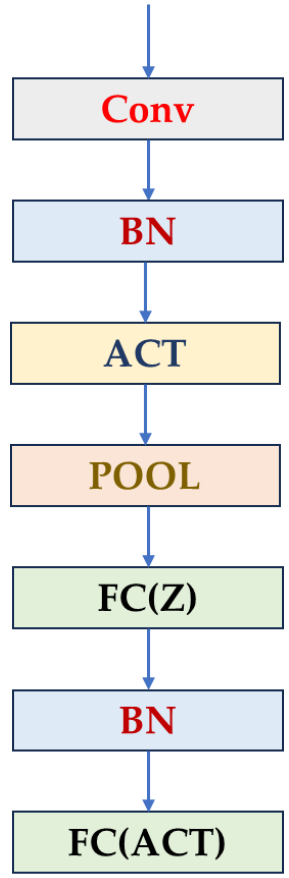
- **Why Use Batch Normalization?**
  - **Stabilizes Training** – Reduces sensitivity to **vanishing and exploding gradients** (?).
  - **Accelerates Convergence** – Allows for higher learning rates.
  - **Reduces Internal Covariate Shift** – Maintains stable distributions of activations across layers.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

2.3.2 Batch Normalizations: Test Time.

- Input: $\mathbf{z : N \times D}$ { $\mathbf{N \rightarrow}$ **number of samples,** $\mathbf{D \rightarrow}$ **Dimension(Height, Width, Channel)**};
  - Applied before activation.
  - $\boldsymbol{\mu_B}$ **and** $\boldsymbol{\sigma_B^2} \rightarrow$ 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: $\boldsymbol{\gamma, \beta} \rightarrow$ {**shape A vector of length D**}
    - learned during training.
- Application:
  - $\mathbf{z'_{i(H \times W)}} = \frac{\mathbf{z_i - \mu_B}}{\sqrt{\boldsymbol{\sigma_B^2 + \epsilon}}}$ $\left\{\mathbf{z_{i(H \times W)} \rightarrow}\right.$ **Every Pixel of Convolved Output, Shape** $\left.\mathbf{\rightarrow N \times D}\right\}$
  - $\mathbf{a'_{i(H \times W)} = \gamma_i \cdot z'_i + \beta_i}$ $\left\{\mathbf{a'_{i(H \times W)} \rightarrow}\right.$ **Every Pixel of** $\mathbf{z'_i}$**, Shape** $\left.\mathbf{\rightarrow N \times D}\right\}$

2.3.3 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.

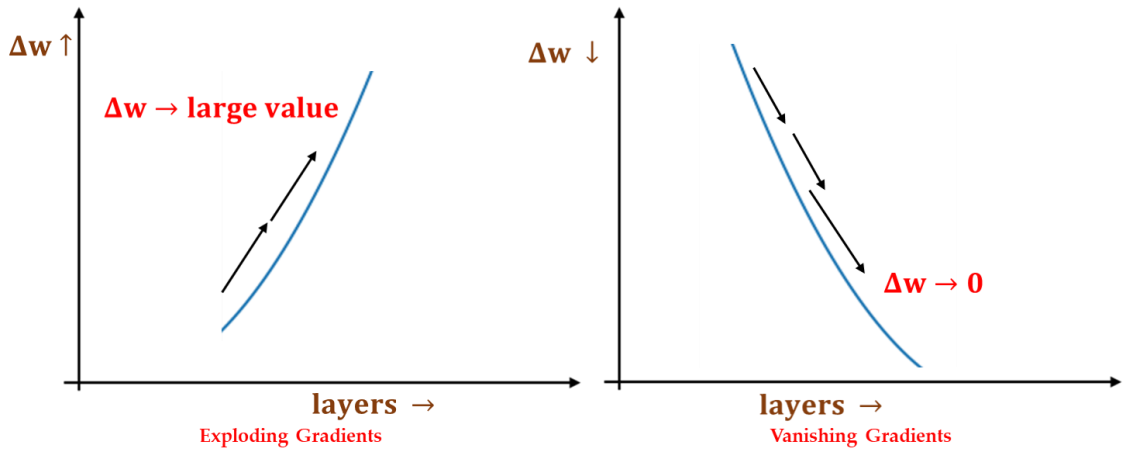


3. Tips and Tricks to Avoid Overfitting and Underfitting.

{Better Training Regime.}

3.1 Vanishing and Exploding Gradients.

- For very deep neural networks, **the activations** can **exponentially rise** or **diminish** gradients when backpropagating through the layers:
  - The **vanishing gradient** problem decays information/gradient as it goes deep into the network, making the network to never converge on a good solution
    - Neurons in the earlier layers learn more slowly than the ones in the latter layers
      - **Tips → Never use Sigmoid; use ReLU with monitoring of Dead neurons.**
  - The **exploding gradient** problem on the other hand makes the gradient bigger and bigger, and as a result forces the network to diverge
    - In this case, the earlier layers explode with very large gradients, making the model useless
      - **Tips → Use better initialization techniques.**



3.2 Weight Initializations.

- **Initializing weights to zero {Avoid}**
  - If **we initialize all our weights to zero**, **our Neural Network** will act as **a linear model** because all the layers are **learning the same thing**.
  - Therefore, the important thing to note with initializing your weights for Neural Networks is to not initialize all the weight to zero.
- **Initializing weights randomly**
  - Using **random initialization** defeats the problem caused by initializing weights to zero, as it prevents the neurons from learning the exact same features of their inputs.
    - Our aim is for each neuron to learn the different features of its input.
  - However, using this technique can also lead to vanishing or exploding gradients, due to incorrect activation functions not being used.
    - **It currently works effectively with the RELU activation function.**
  - **Some extensions to Random weights initializations.**

3.2.1 Weight Initializations: Techniques.

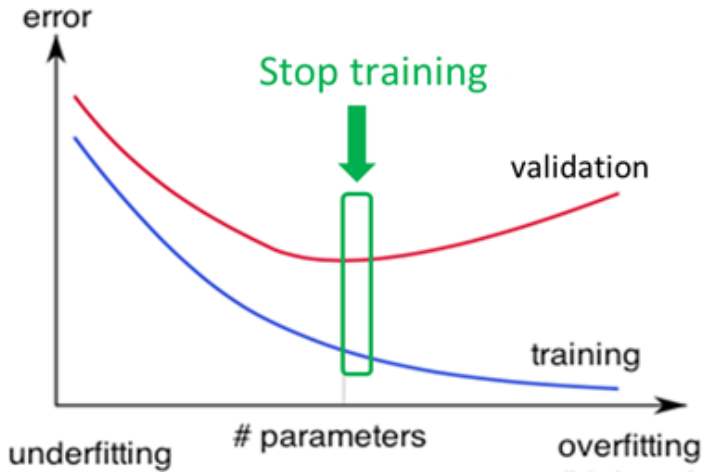
- **Small Random Initialization:**
  - In naïve random initialization, weights are assigned small random values from a uniform or normal distribution:
    - **From Normal Distributions** $\rightarrow$ $\mathbf{W} \sim \mathcal{N}(0, \sigma^2)$. **Or;**
    - **From Uniform Distributions** $\rightarrow \mathbf{W} \sim \mathcal{U}(-\epsilon, \epsilon)$
  - Issues: If weights are too small, gradients vanish; too large they explode during backpropagation.
- **Xavier (Glorot) Initialization:**
  - Xavier Initialization (Glorot and Bengio, 2010) ensures that the variance of activations remains constant across layers.
  - It assumes that both forward and backward signals maintain equal variance.
  - For a layer with n_{in} input neurons and n_{out} output neurons, weights are drawn from:
    - **From Normal Distributions** $\rightarrow$ $\mathbf{W} \sim \mathcal{N}\left(0, \frac{2}{n_{in}+n_{out}}\right)$. **Or;**
    - **From Uniform Distributions** $\rightarrow \mathbf{W} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$
  - **Why it works?**
    - Balance variance of activations across layers.
    - Works well for **sigmoid and Tanh**, where keeping activations in an **appropriate range is important**.

3.2.2 Weight Initializations: Techniques.

- **He Initialization (for ReLU and its Variants):**
  - He initialization (He et. al., 2015) is optimized for ReLU activations, which output non – negative values.
  - It modifies Xavier initialization to prevent the dying ReLU problem by scaling the variance appropriately.
  - Weights are initialized as:
    - **Form Normal Distributions:** $\mathbf{W \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right) : \{n_{in} \rightarrow \text{number of input neurons}\}}$; OR;
    - **For Uniform Distributions:** $\mathbf{W \sim U\left(-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right)}$
  - He Initialization preferred for ReLU and its variant:
    - because it accounts for the nature of ReLU's positive outputs and avoids the vanishing gradient and dead neuron problem by adjusting the variance accordingly
  - (**i.e. increases the magnitude of alive neurons to compensate for dead neurons.**)

3.3 Regularizations: An Idea.

- **What is Regularizations?**
  - "any **modification** to a **learning algorithm** to reduce **its generalization error** but not its **training error**"
  - Reduce generalization error even at the expense of increasing training error
    - E.g., **Limiting model capacity is a regularization method**
- **Regularization Techniques → Early Stopping.**
  - Most simple of Regularizations Techniques.
  - Stop when the **validation accuracy (or loss) has not improved** after ***n* epochs**
    - The parameter ***n*** is called patience.

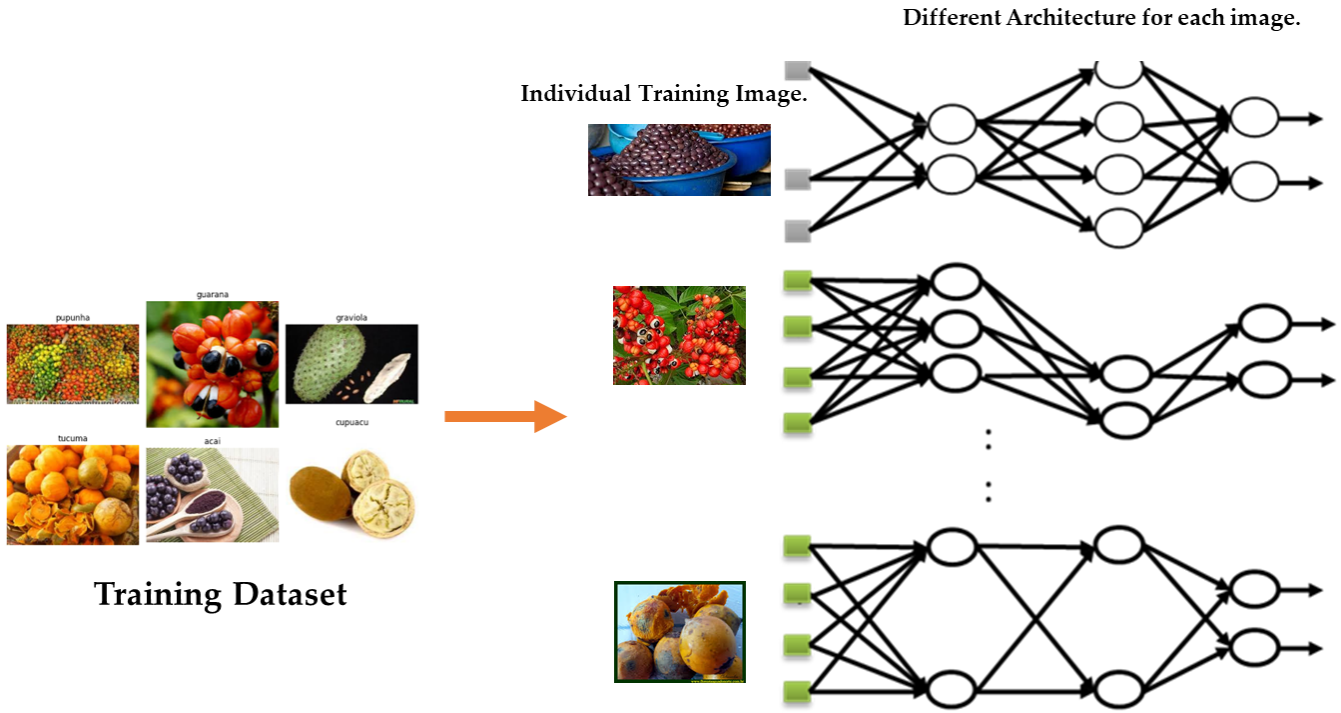


3.3.1 Regularizing : Weight and Biases.

- Remember L1 and L2 Regularizations we use for Linear Regression in 5CS037.
  - Idea → Add **a penalty term** to increase a loss/error which in-turn increases the bias reducing the variance.
- *ℓ_2 weight decay{Ridge Regularizations}*
  - A regularization term that penalizes large weights is added to the loss function
    - $\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k \theta_k^2$
  - For every weight in the network, we add the regularization term to the loss value
    - During gradient descent parameter update, every weight is decayed linearly toward zero
  - The weight decay coefficient λ determines how dominant the regularization is during the gradient computation
- **ℓ_1 weight decay{Lasso Regularizations}**
  - **It** is less common with NN
    - $\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$

3.3.2 Regularizing: Network Architecture.

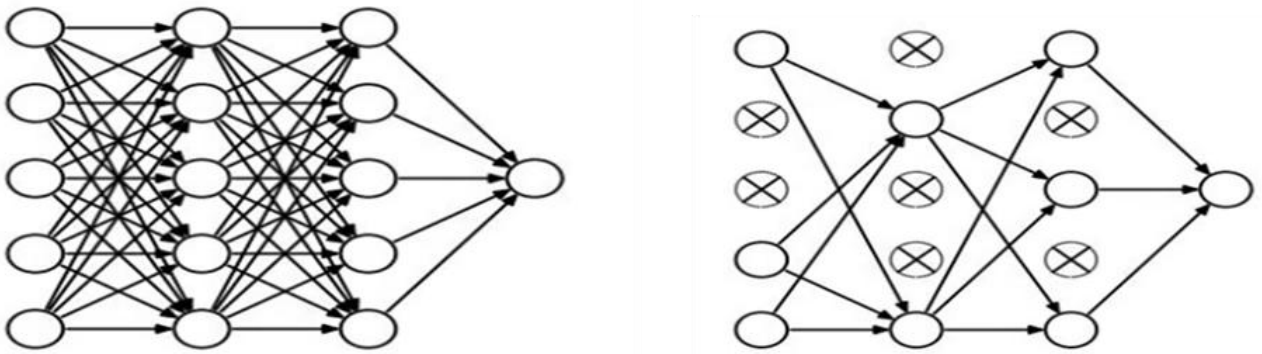
- Imagine we could train a different Neural Network Architecture for every different input:



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

3.3.3 DropOut Introduction.

- Dropout is a regularization technique used in neural networks to prevent overfitting, especially in deep neural networks.
- The primary idea behind dropout is to randomly "drop out" a fraction of neurons along with there connections during training, which forces the network to learn more robust and generalized features.
  - {*dropping out the neuron does not mean deleting the neurons, it just means making it 0, such that it does not learn and update the weight*}
- Each unit is retained with a fixed **dropout rate *p***, independent of other units
- The hyper-parameter *p* needs to be chosen (tuned)
  - Often, between 20% and 50% of the units are dropped



3.3.3.1 Understanding DropOut Parameter p.

- **Imagine a layer with 5 neurons:**
  - **[Neuron 1] [Neuron 2] [Neuron 3] [Neuron 4] [Neuron 5]**
- When we Apply a **DropOut layer with p = 0.5:**
- This means that:
  - In one training iteration, **half of these neurons will be dropped** (**set to 0**), while the remaining neurons will continue to pass their output forward.
  - The selection of dropped **neurons is random and changes in each iterations**.
- **Example Scenarios:**
  - One possible scenario after applying dropout might look like:
    - **[Neuron 1 (0)] [Neuron 2 (value)] [Neuron 3 (value)] [Neuron 4 (0)] [Neuron 5 (value)]**
  - Same layer in second iteration may look like:
    - **[Neuron 1 (value)] [Neuron 2 (value)] [Neuron 3 (0)] [Neuron 4 (0)] [Neuron 5 (0)]**
- Thus, **p = 0.5** in the context of dropout is interpreted as:
  - **p = 0.5** dropout means each neuron has a **50% chance of being dropped** independently, so the number of active neurons may vary per iteration.

3.3.3.2 DropOut Key points.

- **Fixed Network Architecture:**
  - The FCN architecture remains the same i.e. the number of layers and the numbers of neurons in each layer.
  - Only the neurons activations { output which are made 0} are affected by dropout, not the structure of the network.
- **Random Dropout:**
  - For each new input during training, dropOut is applied independently.
  - Each neuron has a p probability of being dropped (set to zero) on that forward pass, and this selection is random for each input.
- **Different Input = Different DropOut Mask:**
  - Each new input data will result in **a different dropout mask** being applied, which means that a different set of neurons may be dropped out during each forward pass.
- **Training and Inference:**
  - **During training**, dropout is applied to the neurons as described, and the network adapts by learning robust features despite some neurons being randomly dropped.
  - **During inference**, dropout is not applied. All neurons are used during testing and validation, and the model uses all the weights may be scaled to account for dropout during training by multiplying the weights by $\frac{1}{1-p}$.

4. Tips and Tricks to Avoid Overfitting and Underfitting.

{ Some Popular Model Architecture.}

4.1 About ImageNet Competition.

- The **ImageNet Large Scale Visual Recognition Challenge (ILVRC)** was a highly influential annual competition in computer vision, running from 2010 t0 2017.
- It served as a benchmark for evaluating the performance of image classification and object detection models on the ImageNet dataset, which contains over 14 million labeled images across 1,000 categories.
- **Impact of ImageNet Competition:**
  - **Accelerated deep learning research:** Led to CNN dominance in vision tasks.
  - **Improved AI applications**: Helped power self-driving cars, medical imaging, facial recognition, and more.
  - **Drove industry adoption:** Companies like Google, Facebook, and Nvidia heavily invested in AI.
  - **It ended in 2017**, because the error rates dropped below human – level performance.

4.1.1 ImageNet Major Breakthrough.

Major Breakthrough in ILSVRC

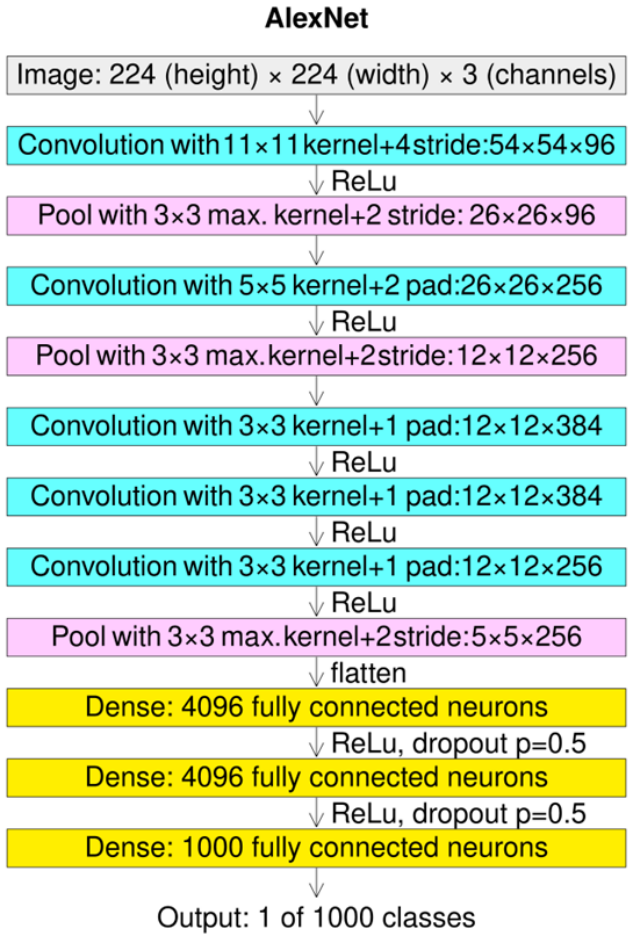
Year	Model	Top ~ 5 Error %	Key Innovations
2010	Traditional Methods	**28.2%**	Handcrafted features
2012	AlexNet	**16.4%**	Deep CNNs, ReLU activation, DropOut
2013	ZFNet	**11.2%**	Optimized AlexNet with better filters
2014	VGG - 16	**7.3%**	Deep networks (16 – 19 layers)
2014	GoogleLeNet (Inception v1)	**6.7%**	Inception Module
2015	ResNet – 152	**3.6%**	Deep residual learning (skip connections)
2016	Inception – v4 and ResNet	**3.1%**	Improved depth and efficiency
2017	SENet	**2.3%**	Squeeze and Extraction blocks for attention.

Note: Human – level performance is estimated at ~5% error.

4.2 AlexNet.

- **AlexNet** is a deep convolutional neural network (CNN) architecture that won the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012**.
  - It was developed by **Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton** and marked a breakthrough in deep learning for computer vision.
- **Key Features:**
  - **8 Layers Deep**: 5 Convolutional Layers + 3 Fully Connected Layers.
  - **ReLU activation**: Used instead of sigmoid to speed up training and mitigate the vanishing gradient problem.
  - **DropOut Regularization**: Applied in fully connected layers to prevent overfitting.
  - **Overlapping Max – Pooling**: Helps reduce spatial dimensions while maintaining important features.
  - **Data Augmentation**: Techniques like random cropping, flipping and color jittering were used to improve generalizations.
  - **Trained with GPU Acceleration**: AlexNet was one of the first deep networks trained on two NVIDIA GPUs to handle large – scale computations efficiently.

4.2.1 AlexNet Architecture Breakdown.



Layer	Type	Details
1	Conv + ReLU	96 filters (11×11), stride 4
2	Max Pooling	3×3, stride 2
3	Conv + ReLU	256 filters (5×5), stride 1
4	Max Pooling	3×3, stride 2
5	Conv + ReLU	384 filters (3×3), stride 1
6	Conv + ReLU	384 filters (3×3), stride 1
7	Conv + ReLU	256 filters (3×3), stride 1
8	Max Pooling	3×3, stride 2
9	Fully Connected	4096 neurons + ReLU + Dropout
10	Fully Connected	4096 neurons + ReLU + Dropout
11	Fully Connected	1000 neurons (softmax output)

4.2.2 AlexNet Parameters.

Parameters Calculation for Each layer.

Layer	Type	Filters/Neurons	Kernel Size	Parameters Calculation	Total Parameters
Conv1	**Conv2D**	96	11 × 11 × 3	**(11 × 11 × 3 × 96) + 96**	34,944
Conv2	**Conv2D**	256	5 × 5 × 96	**(5 × 5 × 96 × 256) + 256**	614,656
Conv3	**Conv2D**	384	3 × 3 × 256	**(3 × 3 × 256 × 384) + 384**	885,120
Conv4	**Conv2D**	384	3 × 3 × 384	**(3 × 3 × 384 × 384) + 384**	1,327,488
Conv5	**Conv2D**	256	3 × 3 × 384	**(3 × 3 × 384 × 256) + 256**	884,992
FC6	**Fully Connected**	4096	-	**(6 × 6 × 256 × 4096) + 4096**	37,752,832
FC7	**Fully Connected**	4096	-	**(4096 × 4096) + 4096**	16,781,312
FC8	**Fully Connected**	1000	-	**(4096 × 1000) + 1000**	4,097,000
Total					**≈ 60 Mil**

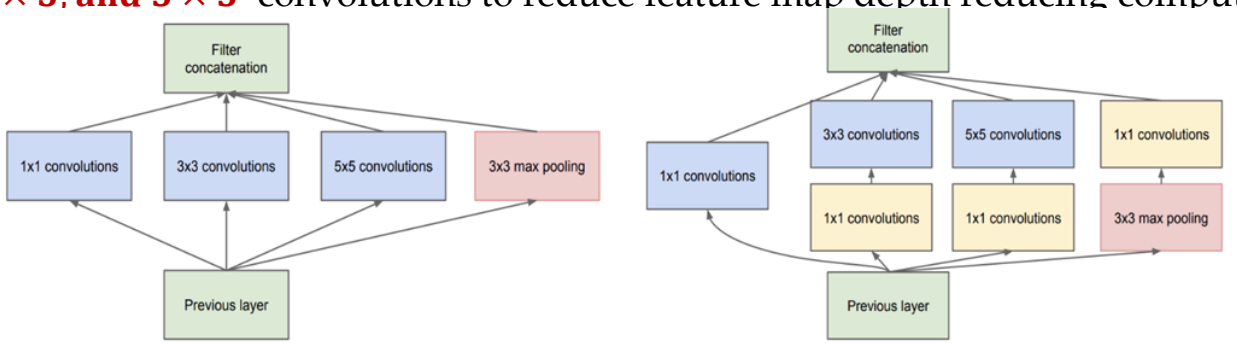
Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 55, 55, 96)	34,944
max_pooling2d (MaxPooling2D)	(None, 27, 27, 96)	0
conv2d_1 (Conv2D)	(None, 27, 27, 256)	614,656
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 256)	0
conv2d_2 (Conv2D)	(None, 13, 13, 384)	885,120
conv2d_3 (Conv2D)	(None, 13, 13, 384)	1,327,488
conv2d_4 (Conv2D)	(None, 13, 13, 256)	884,992
max_pooling2d_2 (MaxPooling2D)	(None, 6, 6, 256)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 4096)	37,752,832
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16,781,312
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 1000)	4,097,000

Total params: 62,378,344 (237.95 MB)
Trainable params: 62,378,344 (237.95 MB)
Non-trainable params: 0 (0.00 B)

4.3 Google LeNet – Inceptions V1.

- **GoogLeNet**, also known as **Inception v1**, was the winning model of the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014**.
- It achieved a **top-5 error rate of 6.7%**, significantly improving over previous architectures like **AlexNet** and **VGG-16**, while using far fewer parameters.
- **Key Innovations in Inception v1:**
  - **Inception Modules:**
    - Introduced a novel multi scale feature extraction approach.
    - Instead of using a single kernel size (**like VGG or AlexNet**), it used **1 × 1; 3 × 3; and 5 × 5** convolutions in **parallel**.
    - This allowed the network to capture fine details (small kernels) and broader patterns (large kernel) at the same time.
  - **1 × 1 Convolutions for Dimensionality Reduction:**
    - Used 1 × 1 convolutions before **3 × 3; and 5 × 5** convolutions to reduce feature map depth reducing computational cost significantly.



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

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.

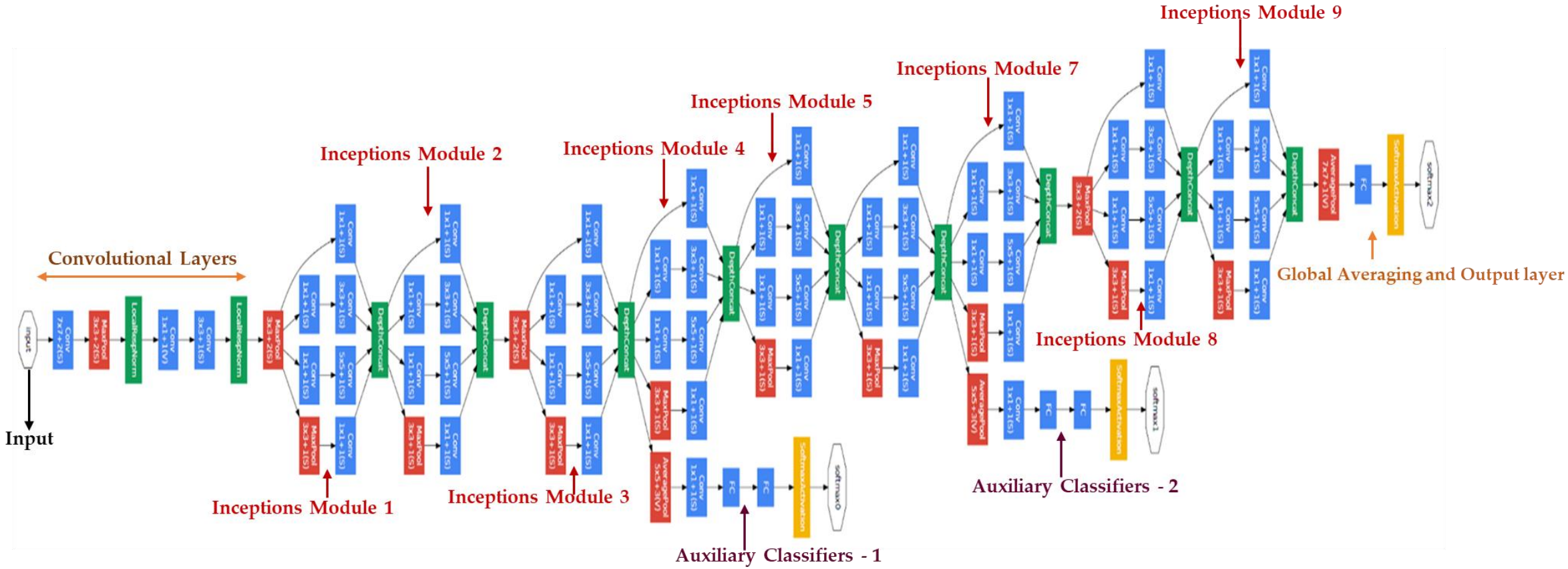
Special Operation 1×1 Convolution.

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

4.3.1 Google LeNet – Inceptions V1.

- **Key Innovations in Inception v1 (Contd.):**
  - **Deeper Network with Fewer Parameters:**
    - 22 layers deep compared to AlexNet (8 layers) and VGG – 16 (16 layers).
    - Used only 6 million parameters, compared to AlexNet's 60 million parameter.
  - **Global Average Pooling Instead of Fully Connected Layers:**
    - Replaced traditional fully connected layers with average pooling, reducing parameters and overfitting.
  - **Auxiliary Classifiers (Extra Loss Branches):**
    - Added two smaller classifier branches at intermediate layers to help gradient flow, reducing the vanishing gradient problem in deep networks.

4.3.2 Google LeNet – Inceptions V1 - Architecture.



4.3.3 Inceptions – V1 – Parameters.

- Layer - wise parameter computation in GoogleLeNet:
  - **Stage 1: Initial Convolutions:**

Layer	Filter Size	Input Channels	Output Channels	Parameters
Conv1	7×7	3	64	(7×7×3+1)×64=9,664(7×7×3 +1)×64 = 9,664(7×7×3+1)×64=9,664
Conv2a	1×1	64	64	(1×1×64+1)×64=4,160(1×1×64 +1)×64 = 4,160(1×1×64+1)×64=4,160
Conv2b	3×3	64	192	(3×3×64+1)×192=110,784(3×3×64 +1)×192 = 110,784(3×3×64+1)×192=110,784

Total for Initial Convolutions: 124, 608 Parameters.

4.3.4 Inceptions – V1 – Parameters.

- Layer - wise parameter computation in GoogleLeNet:
  - **Stage 2: Inceptions Modules:**
    - Each **Inception module** consists of **1×1, 3×3, and 5×5 convolutions**, plus a **1×1 projection layer** for dimensionality reduction.
    - We compute parameters per module:

Branch	Filter Size	Input Channels	Output Channels	Parameters
1×1 Conv	1×1	192	64	(1×1×192+1)×64=12,352(1×1×192 +1)×64 = 12,352(1×1×192+1)×64=12,352
3×3 Conv	1×1 → 3×3	192 → 128	128	(1×1×192+1)×128+(3×3×128+1)×128=16,512+147,584=164,096(1×1×192 +1)×128 + (3×3×128 +1)×128 = 16,512 + 147,584 = 164,096(1×1×192+1)×128+(3×3×128+1)×128=16,512+147,584=164,096
5×5 Conv	1×1 → 5×5	192 → 32	32	(1×1×192+1)×32+(5×5×32+1)×32=6,176+25,632=31,808(1×1×192 +1)×32 + (5×5×32 +1)×32 = 6,176 + 25,632 = 31,808(1×1×192+1)×32+(5×5×32+1)×32=6,176+25,632=31,808
1×1 Projection	1×1	192	32	(1×1×192+1)×32=6,176(1×1×192 +1)×32 = 6,176(1×1×192+1)×32=6,176

- **Total for Inception 3a: 214,432 Parameters.**
- **We follow the same process for all 9 Inceptions modules. The Total Count from all inception modules sums up to 5 million parameters.**

4.3.5 Inceptions – V1 – Parameters.

- Layer - wise parameter computation in GoogleLeNet:
  - **Stage 3: Final Layers:**
    - **Is a fully connected layers with 1000 softmax neurons.**

Layer	Type	Input Size	Output Size	Parameters
Fully Connected	FC	1024	1000	(1024+1)×1000=1,025,000(1024+1)×1000 = 1,025,000(1024+1)×1000=1,025,000

Total for Fully Connected Layers: 1,025,000 Parameters.

4.3.6 Inceptions – V1 Total – Parameters.

- Layer - wise parameter computation in GoogleLeNet:
  - **Final Parameter Count in GoogleNet.**

Components	Parameters
Initial Convolutions	124, 608
Inceptions Module (9 Total)	~5, 000, 000
Full Connected Output Layers	1, 025, 00
Total	~6 Million Parameters

Dimensionality reduction (1×1 convolutions) plays a major role in parameter efficiency.

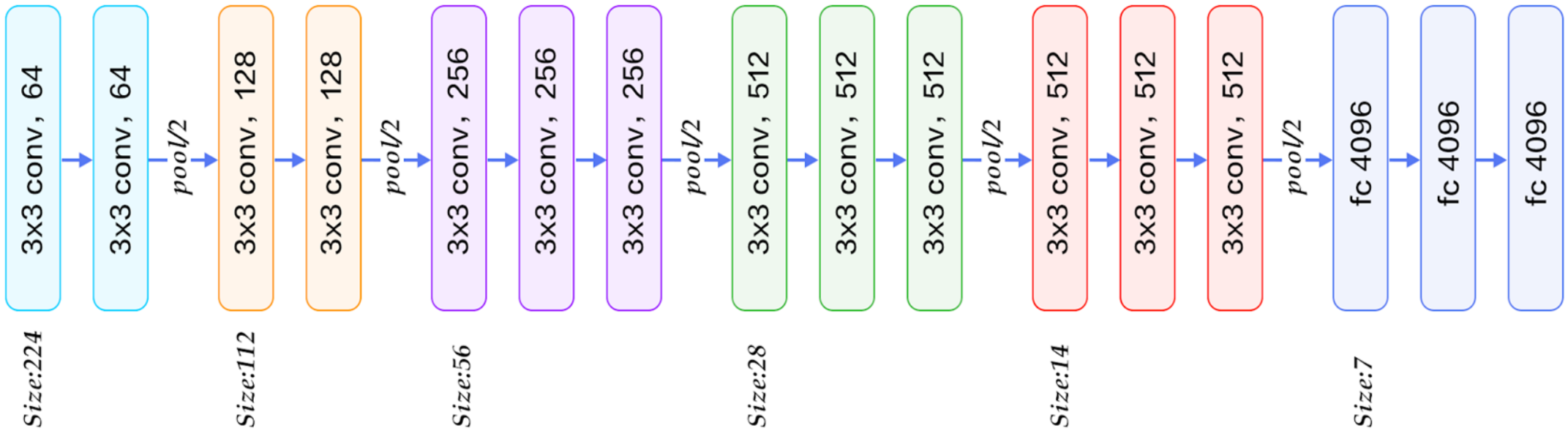
4.4 VGG 16

- VGG16 is a deep convolutional neural network (CNN) that emphasizes simplicity by using very small convolutional filters (3x3) stacked on top of each other.
  - It primarily uses **convolutional layers** followed by **fully connected (FC) layers** for classification.
    - {Came second place in 2014 behind Inception, My Favourite Model to build a prototype}
    - Not any key new features, a clean trick to build deeper network with cleaner architecture.
- Here's the breakdown:
- **Feature Extraction:**
  - **Convolutional Layers:**
    - VGG 16 is composed of **13 convolutional layers**.
    - Each of these layers uses **3 × 3 filters (with a stride of 1 and padding of 1)** to process the input image.
      - The use of small filters ensures that the receptive field gradually increases as you go deeper into the network, allowing the network to capture **increasingly complex features**.
    - There are also max – pooling layers (**typically 2 × 2 pooling with stride of 2**) placed after certain Convolutional layers to reduce the spatial dimensions of the feature maps and prevent overfitting.

4.4.1 VGG 16

- **Classification:**
  - **Fully Connected (FC) Layer:**
    - After passing through the convolutional and pooling layers, the output is flattened into a $1D$ vector.
    - The flattened vector is passed through three fully connected layers.
    - The **first two FC layers each have 4096 neurons,**
    - The output of the final FC layer is passed into a softmax layer.
    - **Final FC layer - Softmax Layer:**
      - 1000 softmax neurons as ImageNet has 1000 different classes.
      - The softmax function is applied to the output of the last fully connected layer. This convnets the network's output into a probability distribution across the classes.
      - The class with the highest probability is selected as the predicted label.

4.4.2 VGG – 16 Architecture.



4.4.3 VGG – 16 Parameters.

Feature Extraction.

Layer	Type	Filters/Neurons	Kernel Size	Parameters Calculation	Total Parameters
Conv1_1	Conv2D	64	3x3	(3 × 3 × 3 × 64) + 64	1,792
Conv1_2	Conv2D	64	3x3	(3 × 3 × 64 × 64) + 64	36,928
MaxPool1	MaxPooling2D	-	2x2	-	0
Conv2_1	Conv2D	128	3x3	(3 × 3 × 64 × 128) + 128	73,856
Conv2_2	Conv2D	128	3x3	(3 × 3 × 128 × 128) + 128	147,584
MaxPool2	MaxPooling2D	-	2x2	-	0
Conv3_1	Conv2D	256	3x3	(3 × 3 × 128 × 256) + 256	295,168
Conv3_2	Conv2D	256	3x3	(3 × 3 × 256 × 256) + 256	590,080
Conv3_3	Conv2D	256	3x3	(3 × 3 × 256 × 256) + 256	590,080
MaxPool3	MaxPooling2D	-	2x2	-	0
Conv4_1	Conv2D	512	3x3	(3 × 3 × 256 × 512) + 512	1,180,160
Conv4_2	Conv2D	512	3x3	(3 × 3 × 512 × 512) + 512	2,359,808
Conv4_3	Conv2D	512	3x3	(3 × 3 × 512 × 512) + 512	2,359,808
MaxPool4	MaxPooling2D	-	2x2	-	0
Conv5_1	Conv2D	512	3x3	(3 × 3 × 512 × 512) + 512	2,359,808
Conv5_2	Conv2D	512	3x3	(3 × 3 × 512 × 512) + 512	2,359,808
Conv5_3	Conv2D	512	3x3	(3 × 3 × 512 × 512) + 512	2,359,808
MaxPool5	MaxPooling2D	-	2x2	-	0

Classification.

Layer	Type	Filters/Neurons	Kernel Size	Parameters Calculation	Total Parameters
Flatten	Flatten	-	-	-	0
FC1	Fully Connected	4096	-	(7 × 7 × 512 × 4096) + 4096	102,764,544
FC2	Fully Connected	4096	-	(4096 × 4096) + 4096	16,781,312
FC3	Fully Connected	1000	-	(4096 × 1000) + 1000	4,097,000
Total				Total Parameters	138,357,544

4.5 ResNet (Residual Networks)

- **ResNet** (Kaiming He et. al 2015 winner.) introduces the concept of **residual connections** or **skip connections**, which help mitigate the vanishing gradient problem and enable training of deeper networks.
- Residual or Skip Connections:
  - **Layer inputs** are propagated and added to the **layer output.**
  - Mitigate the problem of vanishing gradients during training
    - Allow training very deep NN (**with over 1,000 layers**)
- Several ResNet variants exist: 18, 34, 50, 101, 152, and 200 layers
- Are used as base models of other state-of-the-art NNs.

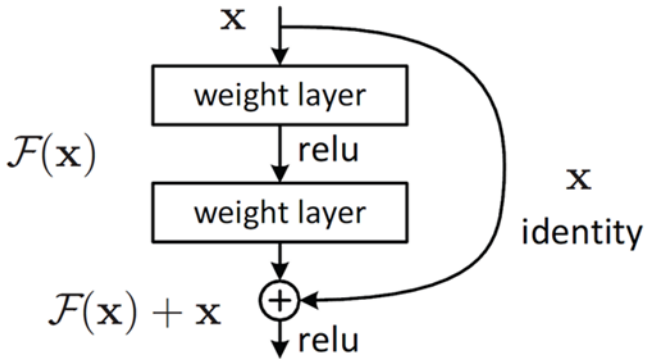


Fig: Skip Connections.

4.5.1 Workings of ResNet.

- **Feature Extraction**:
  - **Convolutional Layers**:
    - Like VGG16, ResNet uses convolutional layers to extract features from the input image, but it also includes **residual blocks**.
    - **Residual Block**:
      - A **residual block** consists of two or more convolutional layers,
        - with a **shortcut or skip connection** that bypasses these layers and **adds the input to the output**.
      - This helps the model learn the **residual** mapping
        - the difference between the input and the output; instead of the direct mapping, which aids in training very deep networks.
    - ResNet uses **batch normalization** after each convolution to stabilize learning and speed up convergence.

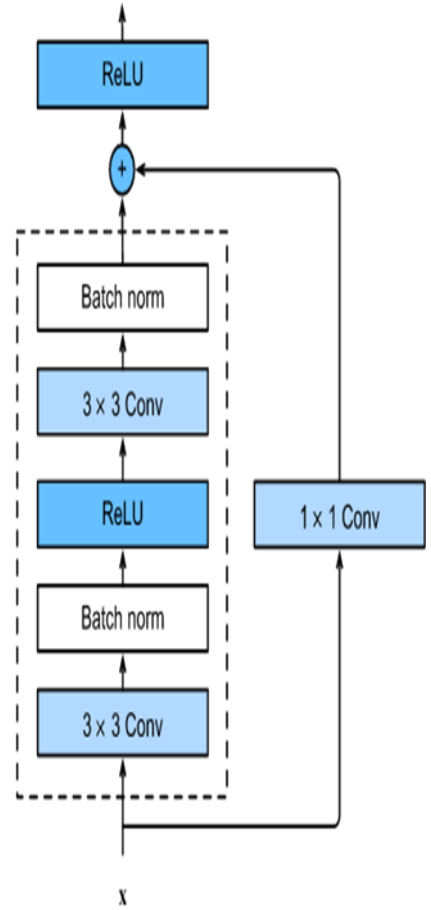
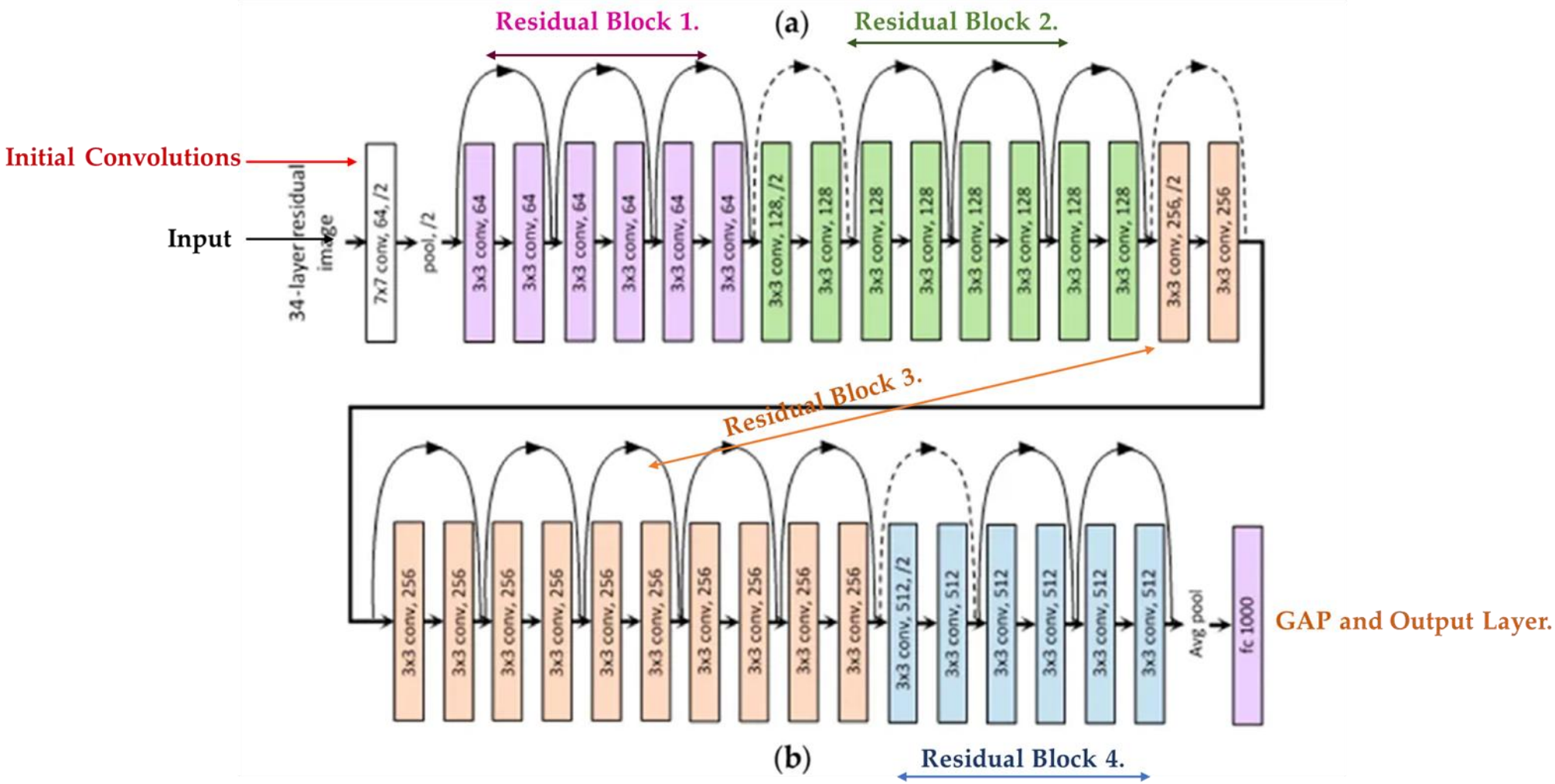


Fig: A Residual Block.

4.5.2 Workings of ResNet.

- **Feature Extraction**:
  - **Global Average Pooling (GAP)**:
    - Instead of using fully connected layers like VGG16, ResNet uses **Global Average Pooling (GAP)** to reduce the feature maps from the convolutional layers.
    - GAP averages each feature map into a single scalar, drastically reducing the number of parameters and preventing overfitting.
- **Classification:**
  - **Softmax Layer**:
    - The output from GAP is a 1D vector, which is passed into the **softmax layer**.
    - The softmax function converts the feature map into a probability distribution over the possible classes.
    - The class with the highest probability is selected as the predicted label.

4.5.3 ResNet Architecture.



4.5.4 ResNet – 50 Parameters.

- **Stage – 1 – Initial Convolutions:**

Layer	Filter Size	Input Channels	Output Channels	Parameters Calculation	Total Parameters
Conv1	7×7	3	64	(7×7×3+1)×64	9,472
Total for Initial Convolutions	-	-	-	-	**9,472**

- **Stage – 2 – Residual Block – 1:**

Layer	Filter Size	Input Channels	Output Channels	Parameters Calculation	Total Parameters
Conv2_1 (1×1)	1×1	64	64	(1×1×64+1)×64	4,160
Conv2_2 (3×3)	3×3	64	64	(3×3×64+1)×64	36,928
Conv2_3 (1×1)	1×1	64	256	(1×1×64+1)×256	16,640
Total for 3 layers in Conv2 block	-	-	-	-	57,728
Total for 3 Residual Blocks (Conv2_x)	-	-	-	-	173,184

4.5.4 ResNet – 50 Parameters.

- Stage – 3 – Residual Block 2:

Layer	Filter Size	Input Channels	Output Channels	Parameters Calculation	Total Parameters
Conv3_1 (1×1)	1×1	256	128	(1×1×256+1)×128	32,896
Conv3_2 (3×3)	3×3	128	128	(3×3×128+1)×128	147,584
Conv3_3 (1×1)	1×1	128	512	(1×1×128+1)×512	66,048
Total for 3 layers in Conv3 block	-	-	-	-	246,528
Total for 4 Residual Blocks (Conv3_x)	-	-	-	-	986,112

- Stage – 4 – Residual Block 3:

Layer	Filter Size	Input Channels	Output Channels	Parameters Calculation	Total Parameters
Conv4_1 (1×1)	1×1	512	256	(1×1×512+1)×256	131,328
Conv4_2 (3×3)	3×3	256	256	(3×3×256+1)×256	590,080
Conv4_3 (1×1)	1×1	256	1024	(1×1×256+1)×1024	263,168
Total for 3 layers in Conv4 block	-	-	-	-	984,576
Total for 6 Residual Blocks (Conv4_x)	-	-	-	-	5,907,456

4.5.4 ResNet – 50 Parameters.

- Stage – 5 – Residual Block 4:

Layer	Filter Size	Input Channels	Output Channels	Parameters Calculation	Total Parameters
Conv5_1 (1×1)	1×1	1024	512	(1×1×1024+1)×512	526,848
Conv5_2 (3×3)	3×3	512	512	(3×3×512+1)×512	2,359,808
Conv5_3 (1×1)	1×1	512	2048	(1×1×512+1)×2048	1,051,648
Total for 3 layers in Conv5 block	-	-	-	-	3,938,304
Total for 3 Residual Blocks (Conv5_x)	-	-	-	-	11,814,912

- Stage – 6 – Final Layer:

Layer	Filter Size	Input Channels	Output Channels	Parameters Calculation	Total Parameters
FC	Fully Connected	2048	1000	(2048+1)×1000	2,049,000
Total for Final Layers	-	-	-	-	2,049,000

4.5.4 ResNet – 50 Final Parameters.

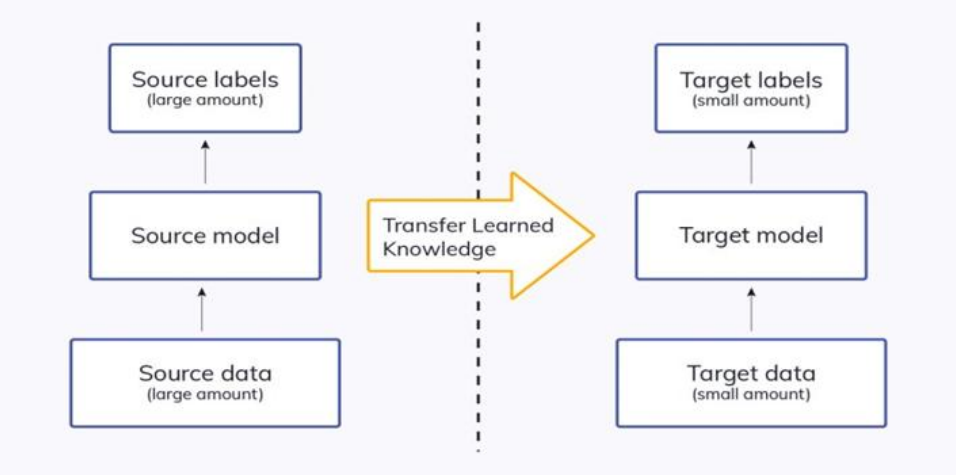
Component	Parameters
Initial Convolutions	9,472
Residual Blocks 1 (Conv2_x)	173,184
Residual Blocks 2 (Conv3_x)	986,112
Residual Blocks 3 (Conv4_x)	5,907,456
Residual Blocks 4 (Conv5_x)	11,814,912
Final Layers	2,049,000
Total	25.0M Parameters

5. Better Result in Practice.

{Transfer Learning}

5.1 Transfer Learning.

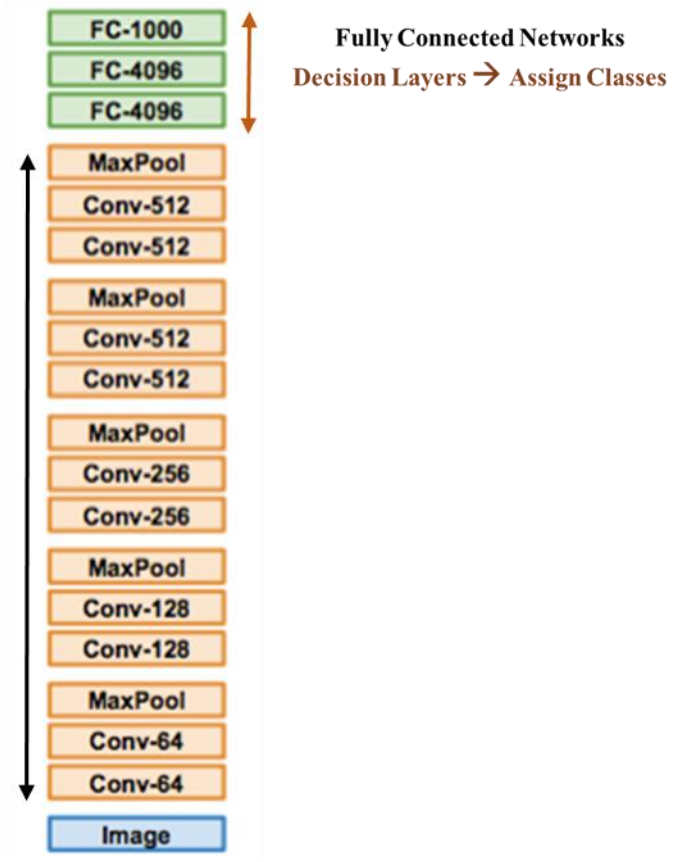
- **Improvement of learning** in a **new task** through the ***transfer of knowledge*** from a **related task** that has already been learned {**Trained**}.
- Cheaper, faster way of **adapting a neural network by exploiting** their **generalization properties**
- Training your own model can be difficult with limited data and other resources
  - It is a laborious task to manually annotate your own training dataset
  - Why not reuse already pre-trained models?



5.2 Transfer Learning-Process.

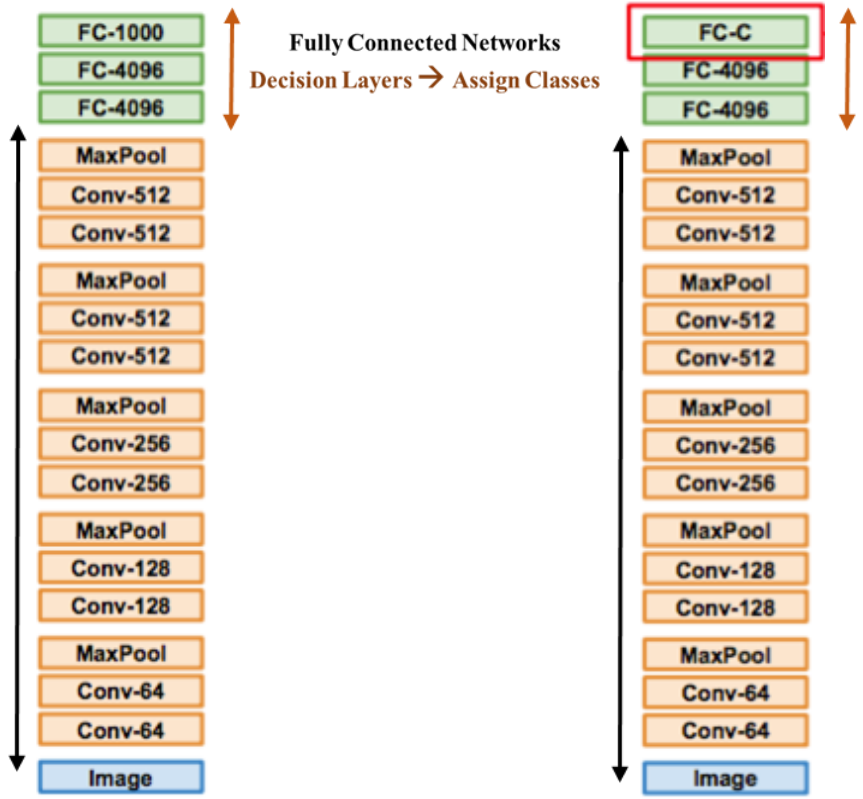
- Start with pre-trained **base network**.
  - Base model is initiated using popular architecture such as ResNet/VGG etc.
  - Generally Trained on Imagenet Dataset.
  - Downloads the pre-trained weights.

Convolution Operations For Feature Extractions.



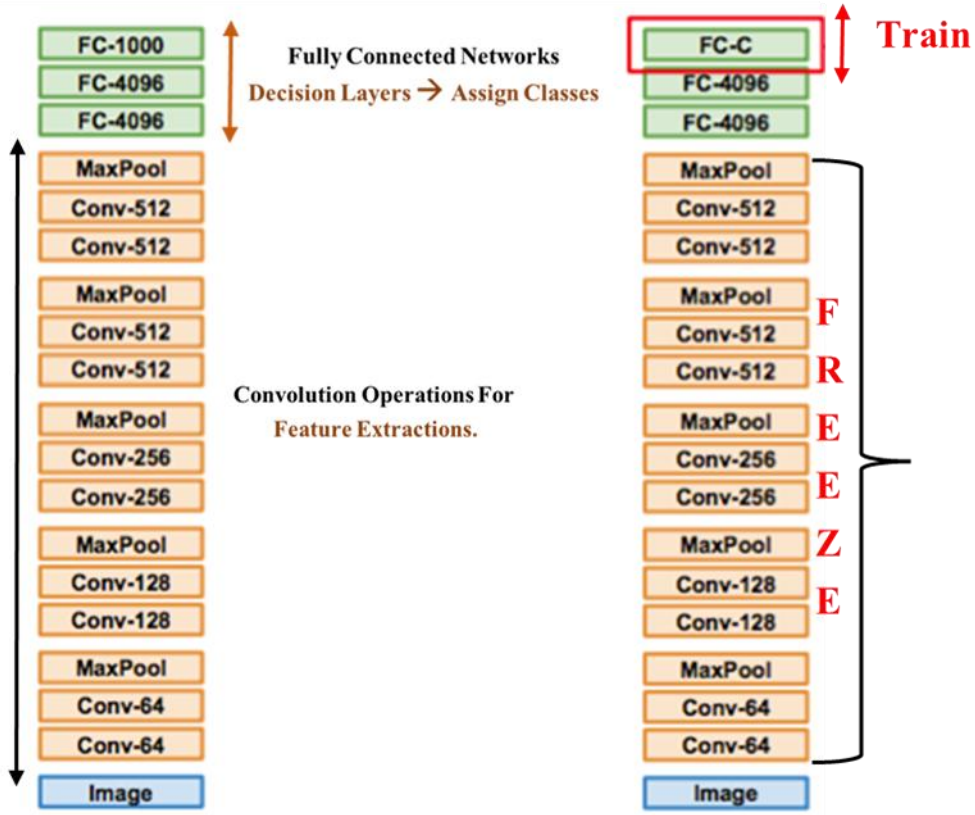
5.2.1 Transfer Learning – Process.

- **Redesign** a decision layer of base model, i.e. add a final layer that is compatible with problem in hand.
  - **New dataset with C classes**



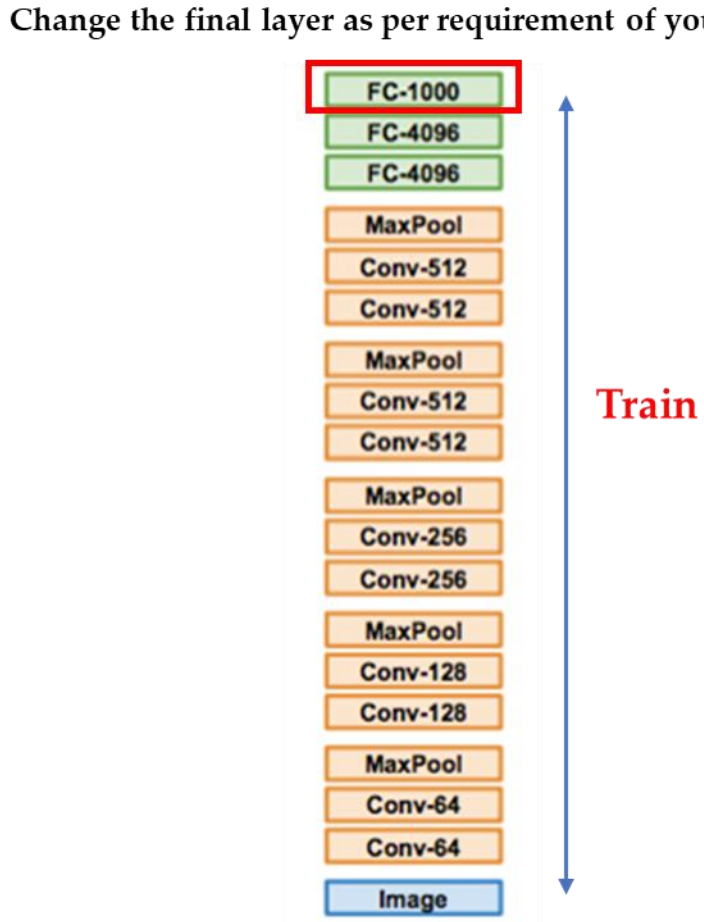
5.2.2 Transfer Learning – Process.

- **Redesign** a decision layer of base model, i.e. add a final layer that is compatible with problem in hand.
  - **New dataset with C classes**
- Training of the Model:
  - Two options:
    - **Options 1: If the dataset is Small:**
      - Freeze Weights and {**Fine Tune**} more layers of base network.
        - Since the dataset is small; no need to train on many layers unfreezing few layers may work.
      - **Use a Small Learning Rate:**
        - This is crucial!
        - A **small learning rate (e.g., 1e-4 or 1e-5)** prevents drastic weight updates and helps retain useful features learned from the original dataset.



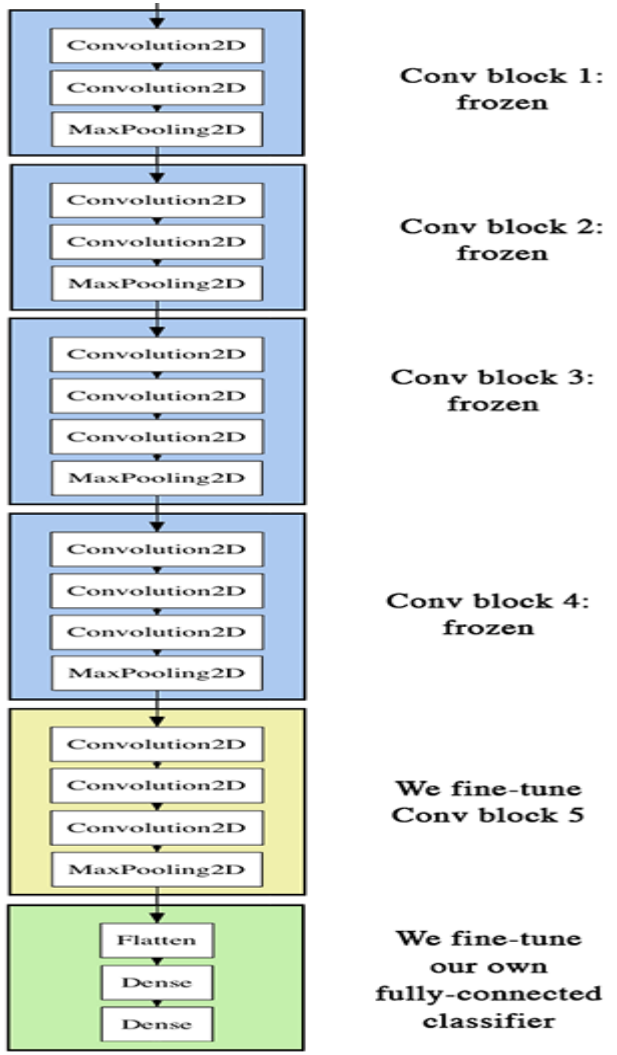
5.2.3 Transfer Learning – Process.

- **Redesign** a decision layer of base model, i.e. add a final layer that is compatible with problem in hand.
  - **New dataset with C classes**
- Training of the Model:
  - Two options:
    - **Options 2: If the dataset is big:**
      - **Unfreeze the weights** and re-train{fine – tune} on as many layer as possible only (linear) classifier with new data.



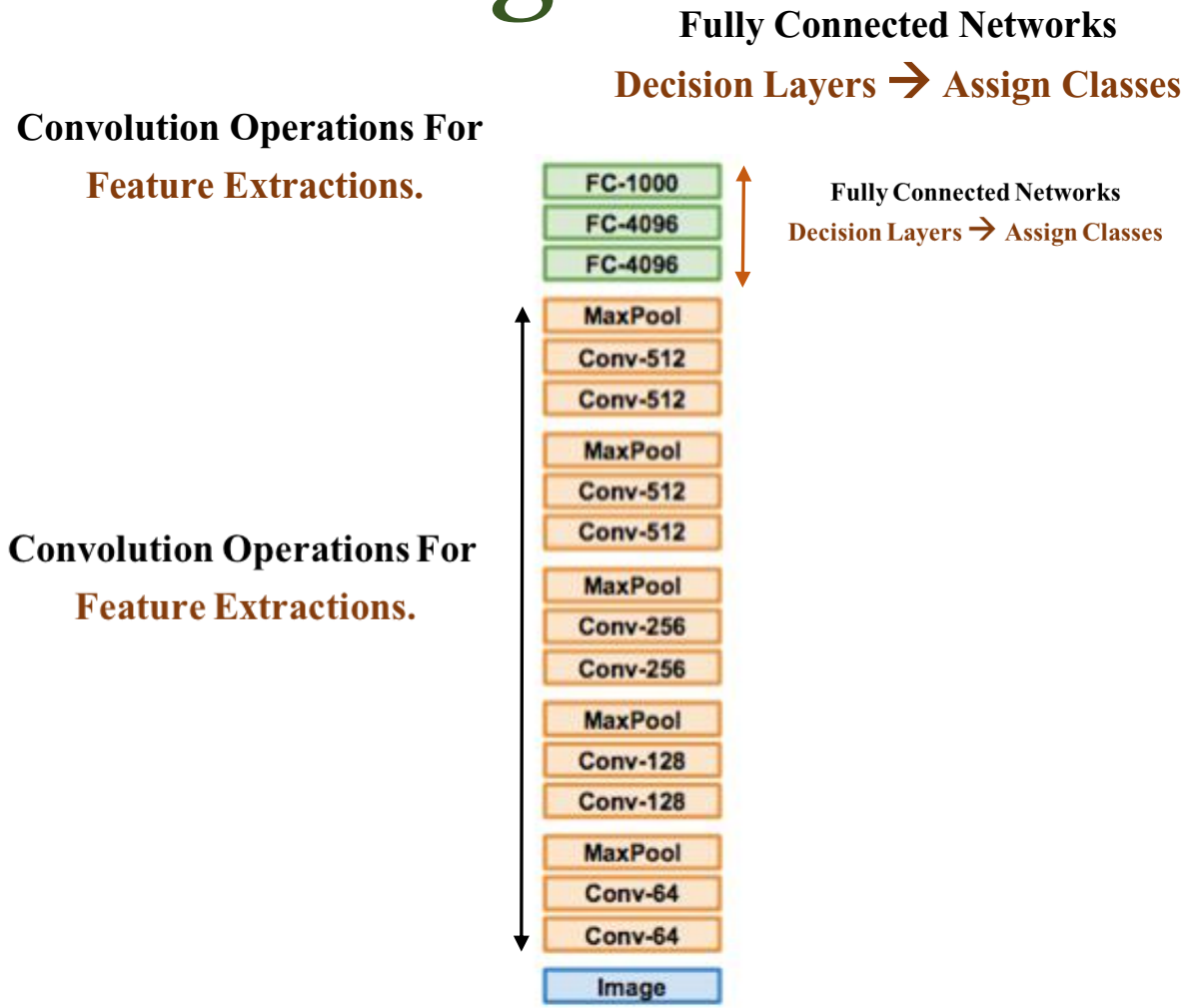
5.3 Freezing and Fine Tuning.

- Freeze layers do not change weights during training.
- Fine-tuning is done by unfreezing the base model or part of it.
- Re-train the entire model again on the whole dataset at a very low learning rate.
- The low learning rate will increase the performance of the model on the new dataset while preventing overfitting.
- Which layer to re-train?
  - Depends on the domain
  - Start by re-training the last layers (last full-connected and last convolutional)
    - work backwards if performance is not satisfactory



5.4 Freezing and Fine Tuning.

1. Start with pre-trained **base network**.
   - Base model is initiated using popular architecture such as ResNet/VGG etc.
   - Generally Trained on Imagenet Dataset.
2. Downloads the pre-trained weights.
3. **Redesign** a base model, i.e. add a final layer that is compatible with problem in hand.
4. Train the model:
   - **Freeze the weights** and re-train (linear) classifier with new data.
   - Unfreeze Weights and **fine tune** whole network with small learning rate.



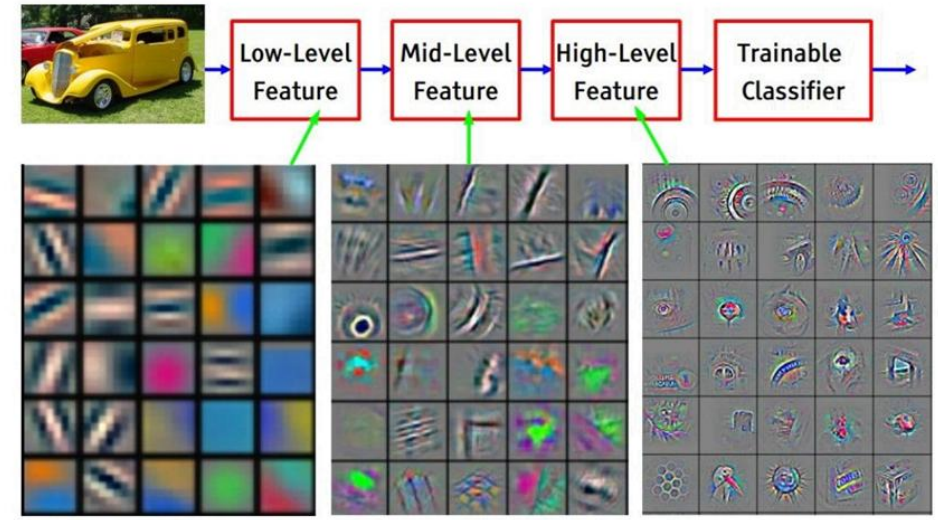
5.5 Where to find Pre-trained models?

- Keras pre-trained models:
- Classify with base classes (imagenet class) with ResNet.

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np

model = ResNet50(weights='imagenet')

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265', u'tusker', 0.1122357
```

5.6 Why it Works?

- Intuitively, deep networks learn multiple levels of abstraction i.e. Different layer of convolution learns/extract different features
  - Generally low level to high level features
  - Low level features such as lines, spot and edges will remain the same for all kind of images



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Closing Remarks.

- Setting up a problem:
  - Obtain training data
    - use appropriate representations for inputs and outputs.
- Choose network architecture
  - More neurons need more data
  - Deep is better, but training may be unstable
- Choose appropriate loss function
- Choose regularization and other heuristics like batch norm, dropout etc.
- Choose optimization algorithm.
  - E.g. Adam {Discuss this in Workshop}
- **Read a lot**.
- Happy Training!!!!!

Thank You