



Computer Vision Applications

BY QUADEER SHAIKH

About me



Work Experience

- Risk Analyst
 - Morgan Stanley (Jan 2023 – Present)
- Data Science Intern
 - AkzoNobel Coatings International B.V. Netherlands (Feb 2022 – Dec 2022)
- Data Science Intern
 - EzeRx Health Tech Pvt. Ltd. (Jan 2022 – July 2022)
- Associate Engineer
 - Tata Communications Ltd. (July 2019 – Aug 2020)
- Network Automation and Analysis Engineer Intern
 - Cisco (June 2018 – July 2018)

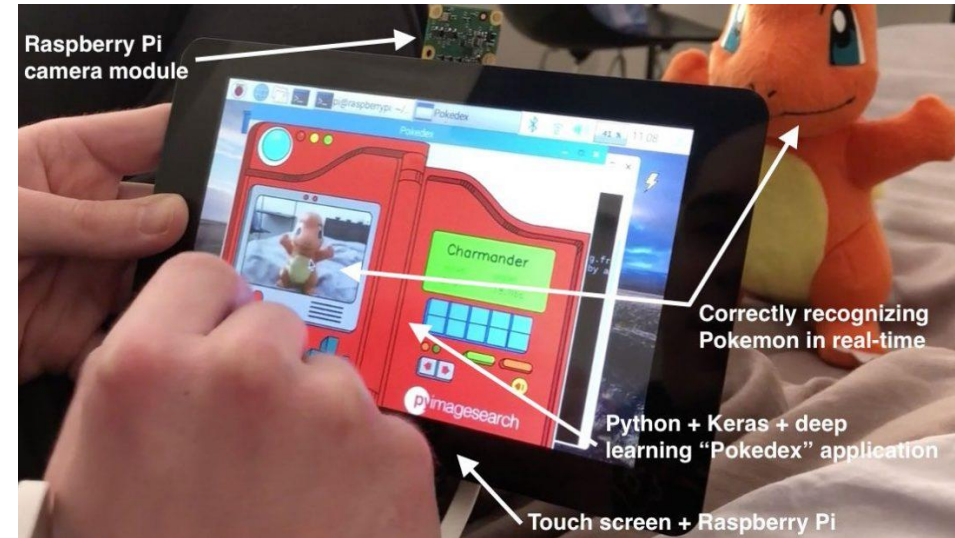
Education

- M.Tech – Artificial Intelligence
 - NMIMS (2021 - 2023, currently pursuing)
- B.E. – Computer Engineering
 - Mumbai University (2015 - 2019)

The Deep Learning Era

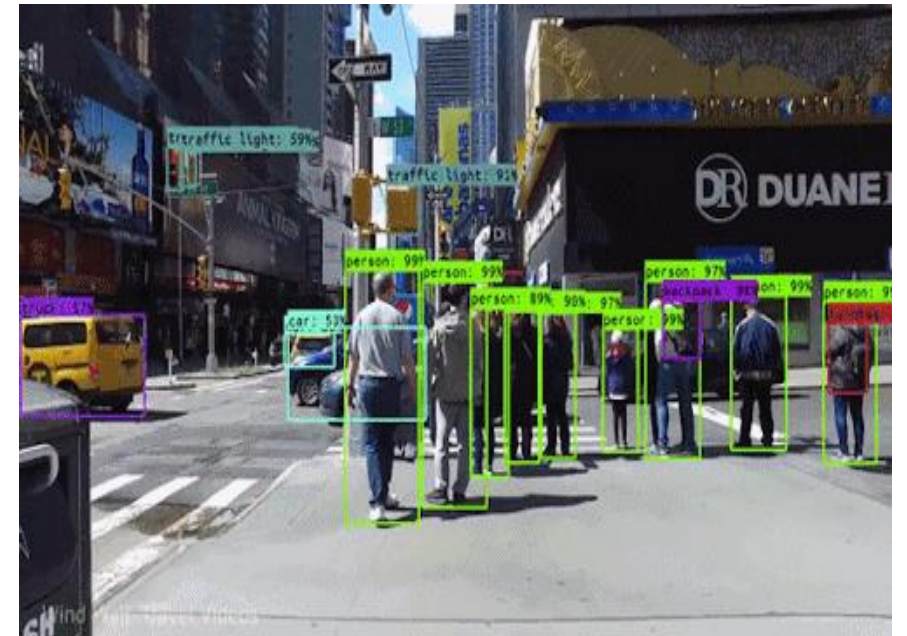
Deep Learning Phase Outline

1. A revisit to an old friend: Neural Networks
2. A visit to a new friend: Convolutional Neural Networks (CNNs)
3. Image Augmentation
4. Optimization and Improvement of CNNs
5. Diagnosis of CNNs
6. CNNs Architectures and Transfer Learning
 1. AlexNet
 2. VGG16/19
 3. InceptionNet/GoogleNet
 4. ResNet
 5. MobileNet
 6. EfficientNet



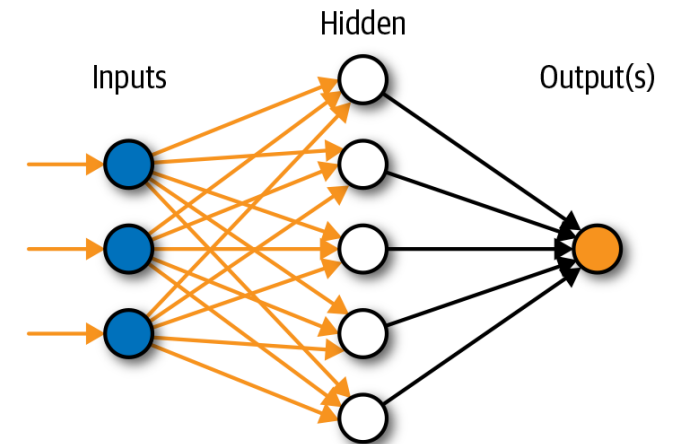
Deep Learning Phase Outline

1. What do CNNs Learn ?
2. One Shot Learning (Face Recognition)
 1. Siamese Network
3. Image Segmentation
 1. FCN
 2. U-Net (Optional)
4. Object Detection (Architecture details and training framework)
 1. RCNN
 2. Fast RCNN
 3. Faster RCNN
 4. YOLO



A revisit to an old friend: NNs

1. An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain
2. Used for non linear pattern recognition
3. Always count the hidden layers + output layers
 - This is a 2 layers neural network
4. Hyperparameters need to be configured:-
 1. No. of layers
 2. No. of neurons in each layer
 3. No. of epochs
 4. Learning rate
 5. Loss function, Optimizer
 6. Batch size
5. How do you count the total parameters in your neural network ?



A revisit to an old friend: NNs

1. How do you calculate the total parameters in the neural network ?

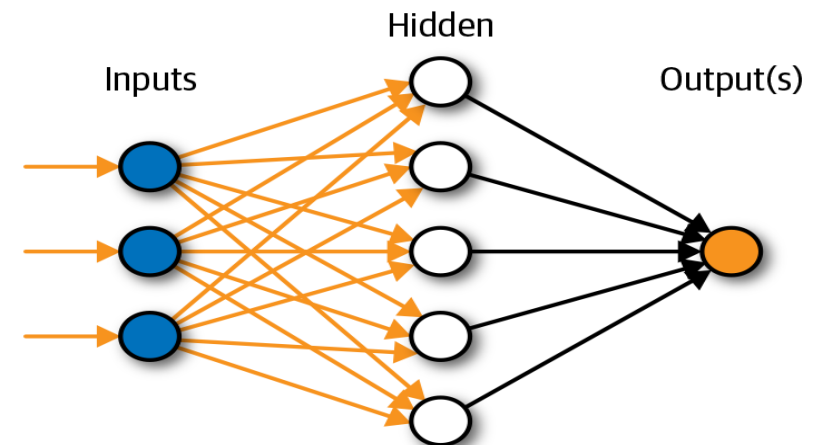
Ans. A neural network has 2 kind of params: weights and biases

Params for a given layer l

Weights = No. of units in current layer (l) x No. of units in prev layer ($l-1$)

Biases = No. of units in current layer (l) x 1

Params = no. of weights + no. of biases



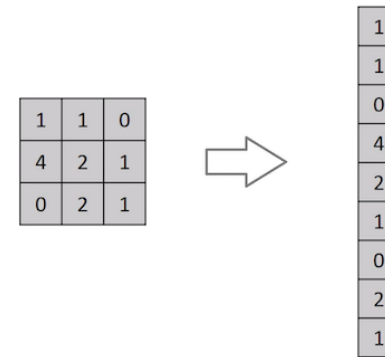
A revisit to an old friend: NNs

How to perform image classification using ANNs ?

1. Flatten the image
2. Pass the flattened image to the neural network

Problems with this for image related tasks:-

1. Images have a 2D spatial structure, therefore considering only the flattened pixel values are not very useful features for tasks like classification
 - Pixels that spatially separated are treated the same way as pixels that are adjacent
2. No way for network to learn features at different places in an input image
3. Imagine flattening an image of resolution $224 \times 224 \times 3$, every image will be represented using a vector of size 150,528.
4. Now imagine using a neural network which has 2 hidden layers of 512 and 128 neurons each and one output layer of 1 neuron. Can you calculate the number of parameters required ?



Params

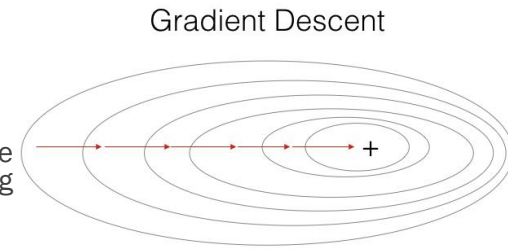
1. $512 \times 150,528 + 512 \times 1 = 77,070,848$
 2. $128 \times 512 + 128 \times 1 = 65,664$
 3. $1 \times 128 + 1 \times 1 = 129$
- Total = 77,136,641**

A revisit to an old friend: NNs

Fantastic optimization algorithms and how to use them

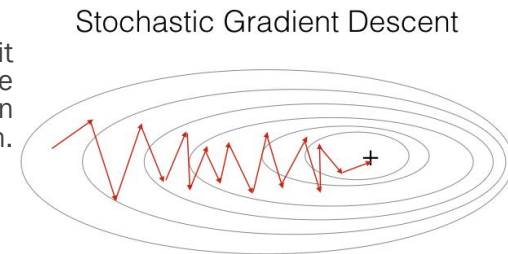
1. Batch Gradient Descent

- A misleading name given by the ML community, the batch gradient descent updates the weight of the neural network by performing forward & backward propagation on the entire dataset i.e. all the training samples/records in a single epoch/iteration



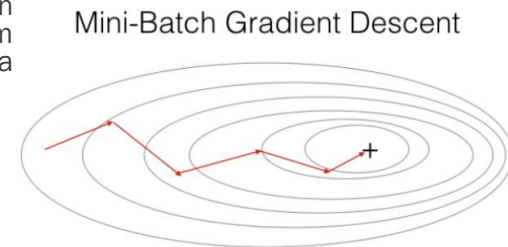
2. Stochastic Gradient Descent

- In batch gd we consider all the training records, but what if the training dataset is huge. You cannot fit huge amount of data into your memory (RAM) for the training process. We take one example each time from the records to update the weights using forward-backward prop (done for all records one by one in an entire epoch). Cost fluctuates a lot during the minimization and starts decreasing in the long run. Has a problem with convergence to minima.



3. Mini Batch Gradient Descent

- SGD has a minima convergence problem and also cannot be implemented with vectorized computation (due to one sample taken each time for a forward-backward prop. Therefore we take mini batches from the dataset and update the weights repeatedly batch wise for an entire epoch. Convergence to minima is smoother than SGD.



Other optimizers like RMSProp, AdaGrad, Adam, etc also use mini batch implementations

A visit to a new friend: CNNs

Remember Convolution operation ?

Kernels/Filters were used to detect features in an image by convolving them on the image

How to decide weights of kernels ?

Make the kernels learnable

Treat the kernels as trainable weights

What about nonlinear patterns in images ?

Add activation functions to the kernels

The diagram illustrates a 2D convolution operation. It shows a 6x6 input grid, a 3x3 kernel, and a 4x4 output grid. The top-left element of the output is -5, calculated from the top-left 3x3 region of the input and the kernel.

Input Grid (6x6):

3 ₁	0 ₀	1 ₋₁	2	7	4
1 ₁	5 ₀	8 ₁	9	3	1
2 ₁	7 ₀	2 ₋₁	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

Kernel (3x3):

1	0	-1
1	0	-1
1	0	-1

Output Grid (4x4):

-5			

Calculation:

$$3 \times 1 + 1 \times 1 + 2 \times 1 + 0 \times 0 + 5 \times 0 + 7 \times 0 + 1 \times -1 + 8 \times -1 + 2 \times -1 = -5$$

CNNs: Understanding Dimensions

Lets just consider the W1 and H1 for now. Suppose we have an image of size 5x5 (W1xH1) and kernel of size 3x3.

The result of convolution is 3x3

$$W2: W1 - F + 1 \rightarrow 5 - 3 + 1 = 3$$

$$H2: H1 - F + 1 \rightarrow 5 - 3 + 1 = 3$$

Lets say we have a 7x7 image and kernel of size 3x3

$$7 - 3 + 1 = 5$$

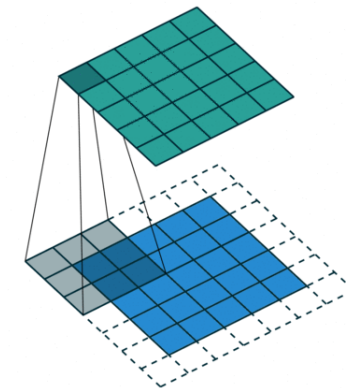
Note: The output result dimensions are smaller than input dimensions

7	2	3	3	8
4	5	3	8	4
3	3	2	8	4
2	8	7	2	7
5	4	4	5	4

1	0	-1
1	0	-1
1	0	-1

$$\begin{matrix} 7 \times 1 + 4 \times 1 + 3 \times 1 + \\ 2 \times 0 + 5 \times 0 + 3 \times 0 + \\ 3 \times -1 + 3 \times -1 + 2 \times -1 \\ = 6 \end{matrix}$$

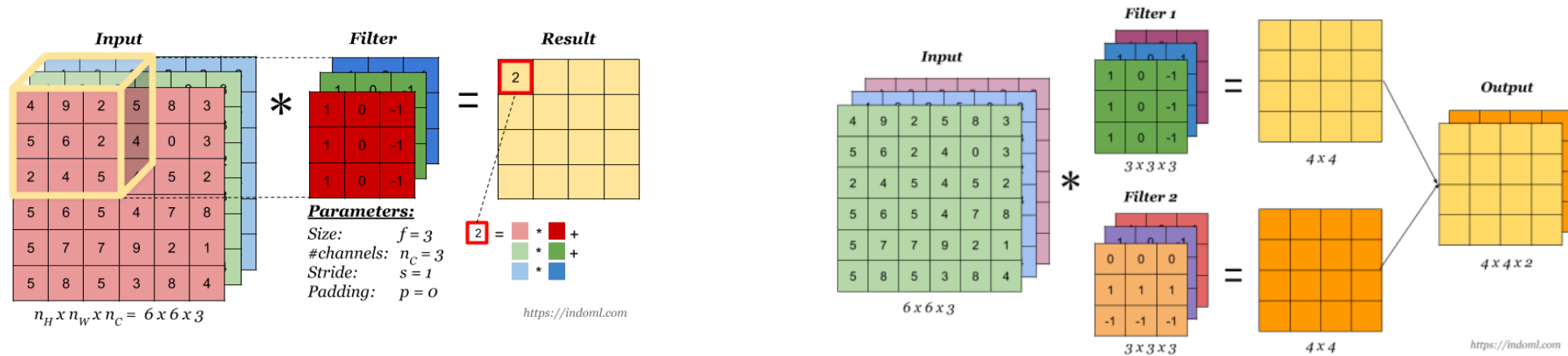
6		



Convolutions over Volumes

Convolutions on RGB image/over volumes

Note: Filter/Kernel depth should be the same as input image depth



CNNs: Understanding Dimensions

Input Dimensions: Width (W_1) x Height (H_1) x Depth (D_1) [e.g. 224x224x3]

Spatial Extent of Filter/Kernel: F (depth of the filter is same as of input)

Output Dimensions: W_2 x H_2 x D_2

Strides: S

Kernels: K

Padding: P

CNNs: Understanding Dimensions

What if we want the output of convolution to be of same size as input ?

Pad input image with appropriate number of inputs so that you can now apply the kernel on input image corners as well. There are different methods for padding but the most commonly used one is zero padding. No. of Padding units (P)

If P is set to 1 then it will pad rows and columns of 0's to the top, bottom, left and right of the image

W2: $W1 - F + 2P + 1$, **H2:** $H1 - F + 2P + 1$

$$5 - 3 + 2 \times 1 + 1 = 5$$

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel		
0	-1	0
-1	5	-1
0	-1	0

114				

CNNs: Understanding Dimensions

What does stride do ?

Defines the intervals at which the kernel is applied to the image

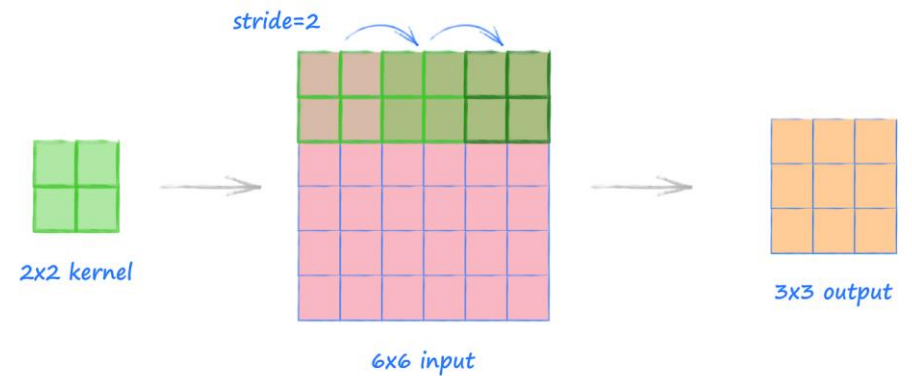
Results in an output of smaller dimensions

$$W2: [(W1 - F + 2P)/S] + 1$$

$$H2: [(H1 - F + 2P)/S] + 1$$

6x6 image, 2x2 kernel and stride 2

$$[(6 - 2 + 2 \times 0)/2] + 1 = 3$$



CNNs: Understanding Dimensions

Lets talk about the depth of the output now.

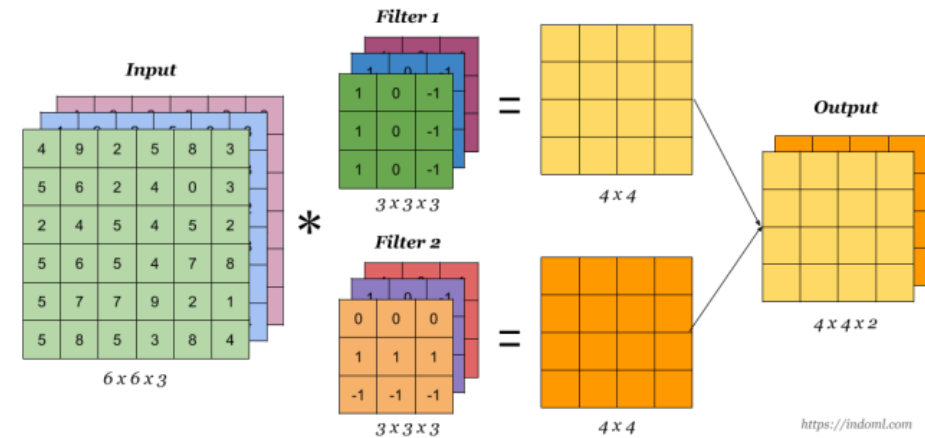
Each kernel/filter gives us a 2D output

If we use K filters then we will get K such 2D outputs

The resulting dimensions would then be $W_2 \times H_2 \times K$

Thus, $D_2 = K$

1



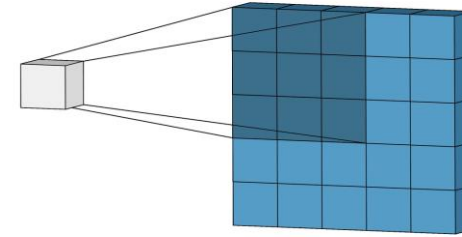
What were the challenges faced by ANNs aka FNNs in image tasks ?

Problems with this for image related tasks:-

1. Images have a 2D spatial structure, therefore considering only the flattened pixel values are not very useful features for tasks like classification
 - Pixels that spatially separated are treated the same way as pixels that are adjacent
2. No obvious way for network to learn features at different places in an input image (similar edges, corners or blobs)
3. Can get computationally expensive for large images

How do CNNs solve these challenges ?

1. Local Receptive Fields: Hidden units are connected to the local patches from the previous layer. This serves 2 main purposes:-
 - Captures local spatial relationships in pixels
 - Reduces the number of parameters significantly
2. Weight Sharing: Also serves 2 purposes
 - Enables rotation, scale and translational invariance to objects in images (no more usage of the painful SIFT algo)
 - Reduces number of parameters in the model
3. Pooling: condenses info from previous layer
 - Aggregates info, especially the minor variations
 - Reduces size of output from previous layer, which reduces the number of computations in the previous layer



0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel		
0	-1	0
-1	5	-1
0	-1	0

114				

Max Pooling			
29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2
pool size

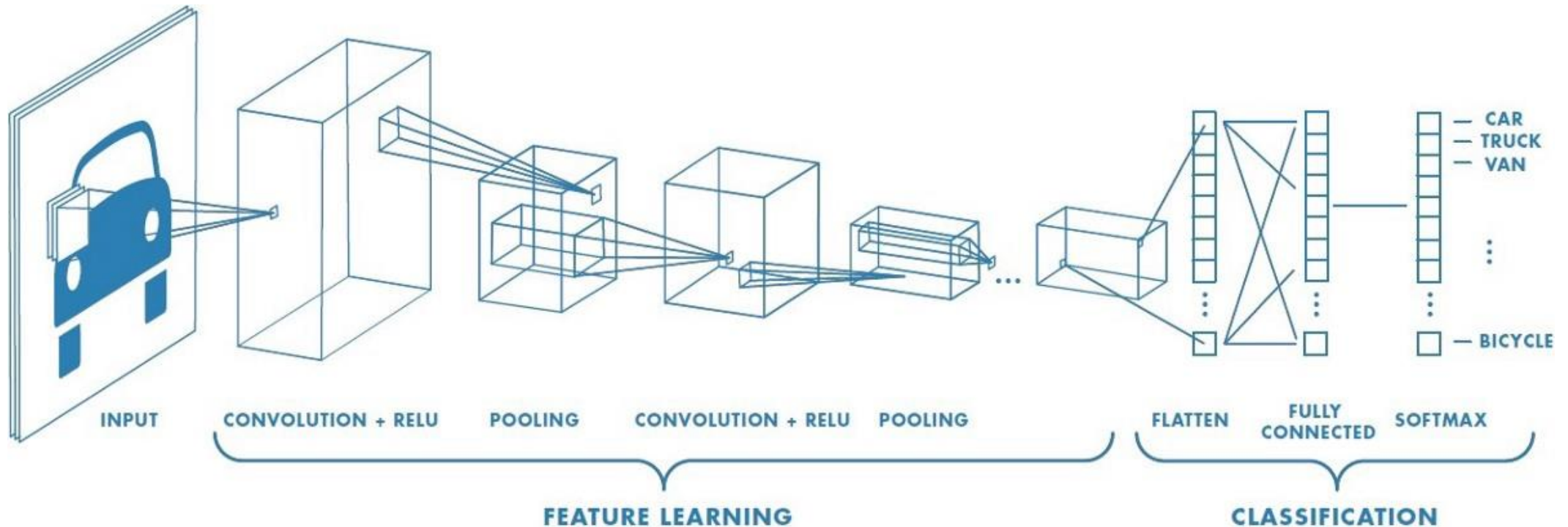
100	184
12	45

Average Pooling			
31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2
pool size

36	80
12	15

What a typical CNN architecture looks like ?



How do CNNs go about learning patterns ?

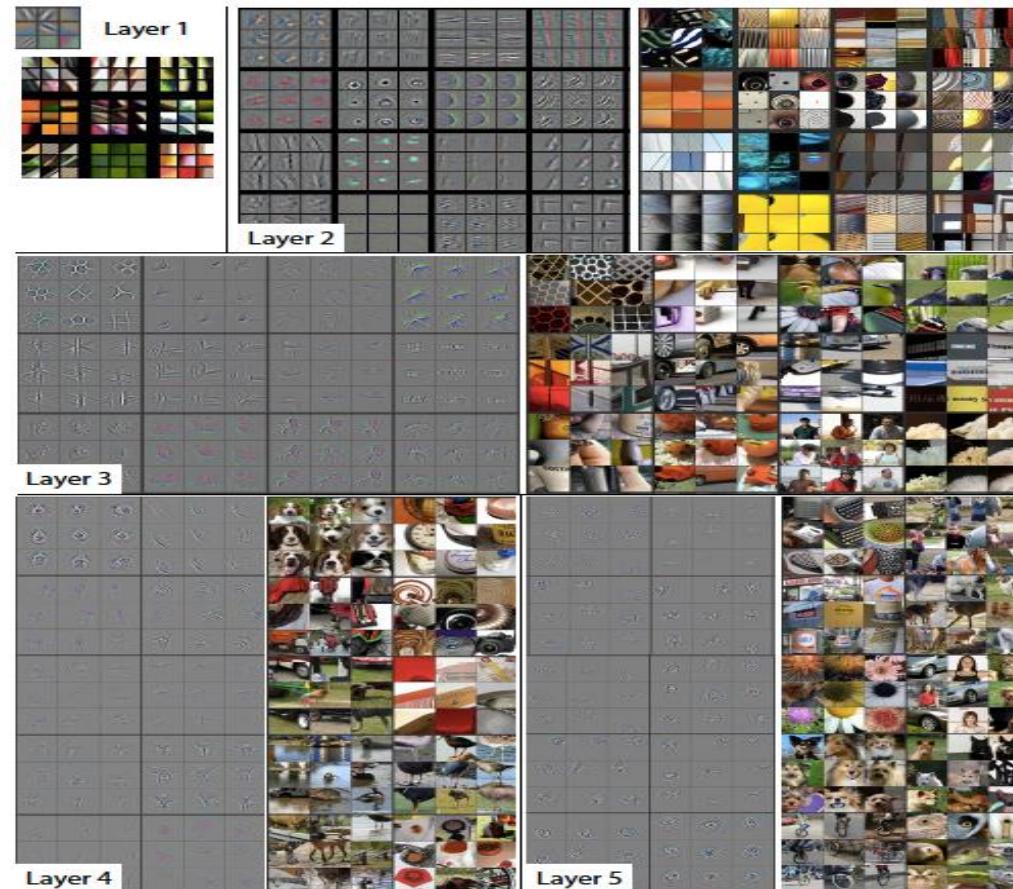
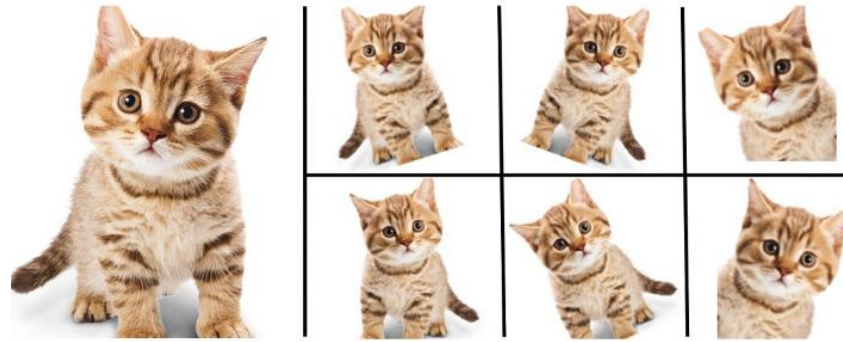


Image Augmentation

Image augmentation is **a technique that is used to artificially expand the data-set**. This is helpful when we are given a data-set with very few data samples. In case of Deep Learning, this situation is bad as the model tends to over-fit when we train it on limited number of data samples.

We use both image filtering and image transformations (affine transformations) to expand the dataset.

Since we know that the weight sharing property of the CNNs make the network rotation, scale and translational invariant we can easily leverage these transformation methods.

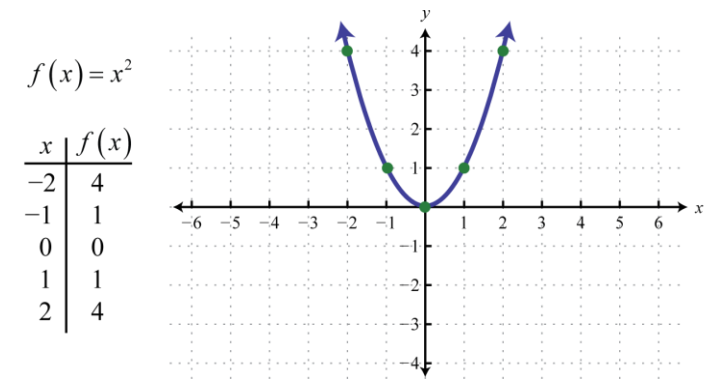
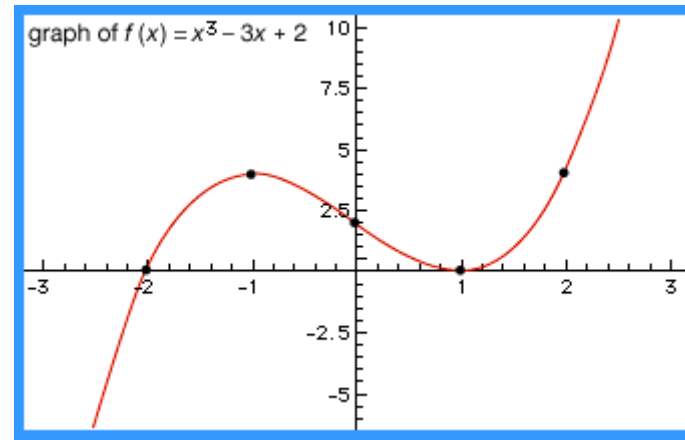
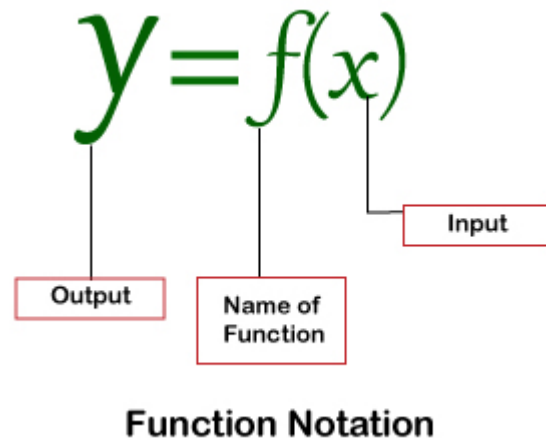


Enlarge your Dataset

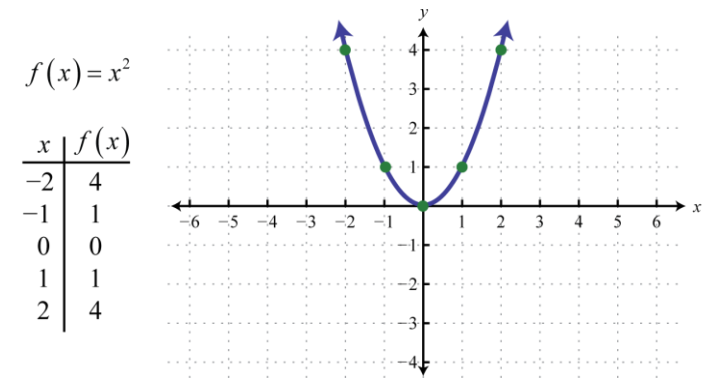
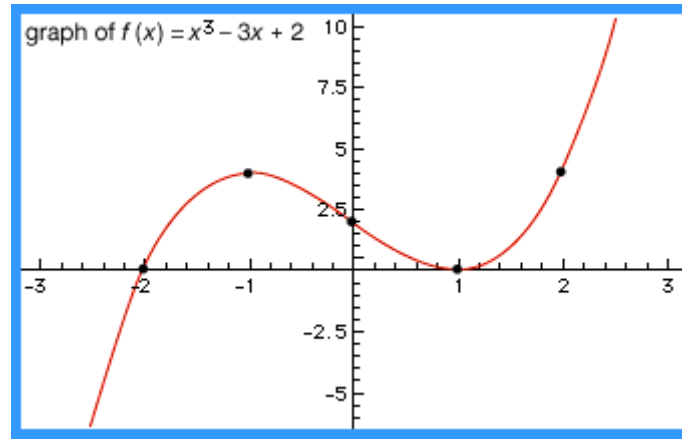
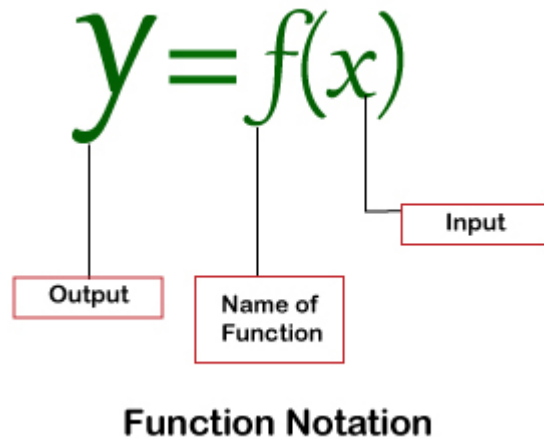
Diagnosis of CNNs

1. Use Classification Metrics
2. Observe the high and low probability scores
3. Observe the feature maps
4. Extract the features from the second last layer
5. Perform 2D or 3D plots to visualize them in the x-y(z) coordinate space
 1. Requires dimensionality reduction
6. Advanced visualization methods
 1. Maximal activated image patches
 2. Gradient based class activation maps

Neural Networks as Functions



Neural Networks as Functions



Note: Neural Networks are nothing but trainable parametric non linear functions. The complexity of the function depends on the number of layers of the neural network

Overfitting in Neural Nets

What happens when a function is overtrained on same instances of data ? It overfits on those values and fails to generalize on new instances of data.

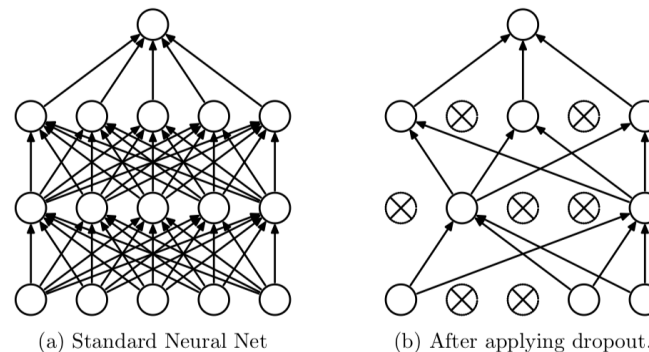
Solution: -

1. Regularization
 1. L1 and L2, ElasticNet
 2. Dropout
2. Addressing the difference in data distributions
3. Adding more data (diverse kind not the repeated instances)
4. Stop being greedy and stop the training a little earlier

Dropout

The term “dropout” refers to dropping out the nodes (input and hidden layer) in a neural network. All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p . E.g. if a hidden layer has 8 neurons and the dropout rate p is 0.5, then 4 neurons will be randomly dropped out from that hidden layer.

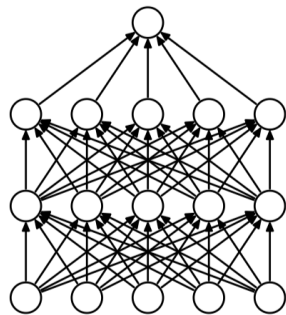
Another way to think of it is to realize that a unique neural net is generated at each training step. The final neural network during the inference time will be nothing but the average ensemble of these smaller reduced neural nets. It has the same intuition as Random Forest.



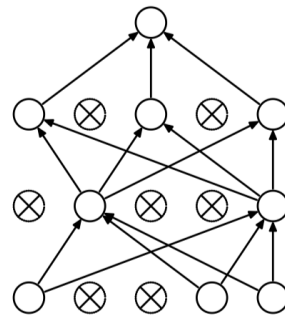
Dropout

Dropout solves the problem of overfitting which is prominently caused by the problem of co-adaptation.

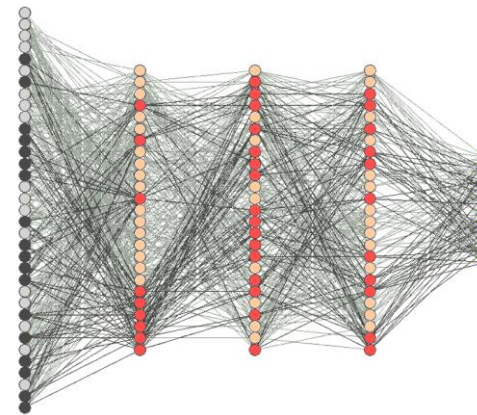
In neural network, co-adaptation means that some neurons are highly dependent on others. If those independent neurons receive “bad” inputs, then the dependent neurons can be affected as well, and ultimately it can significantly alter the model performance, which is what might happen with overfitting.



(a) Standard Neural Net

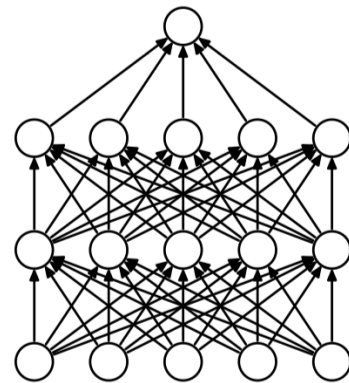


(b) After applying dropout.

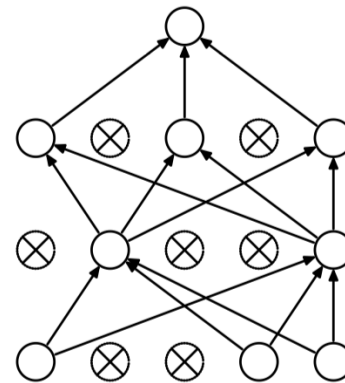


Dropout During Inference (Test/Val)

During inference (testing/validation) dropout layer is not used. This means that we consider all the neurons of the original/parent neural network. But due to this the final weights of the neural network will be larger than expected. Therefore the weights are scaled by multiplying p the dropout rate with the weights for each unique neurons dropped out neural network and then summed together to get the final weights of the neural network at inference time.



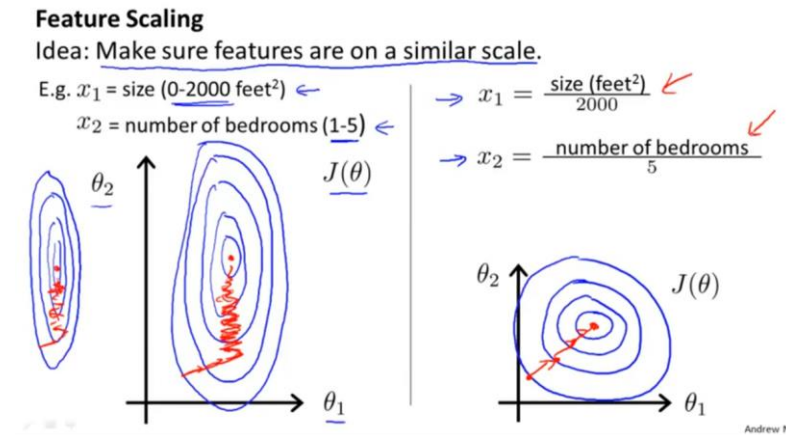
(a) Standard Neural Net



(b) After applying dropout.

Why Feature Scaling is Important ?

We need **input feature scaling** in parametric methods like neural networks because the parameters descend quickly on smaller homogeneous scales and slowly on larger heterogeneous scales.



But what about the scales of features of the hidden layers of the neural network ?

Internal Covariate Shift














- “We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training.” – Authors of **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift**.
- In neural networks, the output of the first layer feeds into the second layer, the output of the second layer feeds into the third, and so on. When the parameters of a layer change, so does the distribution of inputs to subsequent layers.
- These shifts in input distributions known as **Internal Covariate Shifts** can be problematic for neural networks, especially deep neural networks that could have a large number of layers.

Example of Internal Covariate Shift

Lets say we are solving a binary classification problem of Dogs vs..... Not Dogs ?

Let's say we have the images of white dogs only in a certain batch, these images will have certain distribution as well. Using these images model will update its parameters.

Later when we introduce a new set of images consisting of not white dogs, they will have a different distribution from the previous images.

<div>Dog $Y = 1$</div>  <div>Non-Dog $Y = 0$</div> 	<p>Hope: *hemk I is part of distribn*</p> 	<table border="0"><tr><td data-bbox="1345 829 1513 1253"><div>Dog $Y = 1$</div></td><td data-bbox="1538 829 1704 1253"><div>Non-Dog $Y = 0$</div></td><td data-bbox="1717 1053 1849 1110"><p>Internal Covariate Shift</p></td><td data-bbox="1862 829 2040 1253"><div>Dog $Y = 1$</div></td></tr></table>	<div>Dog $Y = 1$</div> 	<div>Non-Dog $Y = 0$</div> 	<p>Internal Covariate Shift</p>	<div>Dog $Y = 1$</div> 	<p>Coco: *wut the floof, demn you internal covariate shift*</p> 
<div>Dog $Y = 1$</div> 	<div>Non-Dog $Y = 0$</div> 	<p>Internal Covariate Shift</p>	<div>Dog $Y = 1$</div> 				

Batch Normalization

This technique adds an operation in the model before or after the activation function of each hidden layer.

This operation simply zero centers and normalizes each input then scales and shifts the result using two new parameter vectors per layer

- One for scaling
- The other for shifting

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

Batch Normalization during Testing/Validation

During Training Batch Normalization standardizes intermediate inputs, then rescales and offsets them.

What about Test/Validation Time ?

We may be passing just one instance of image at test time, or maybe a smaller batch of images so we may have no way to compute mean and SD or we may have to compute the mean and SD over a smaller batch/sample.

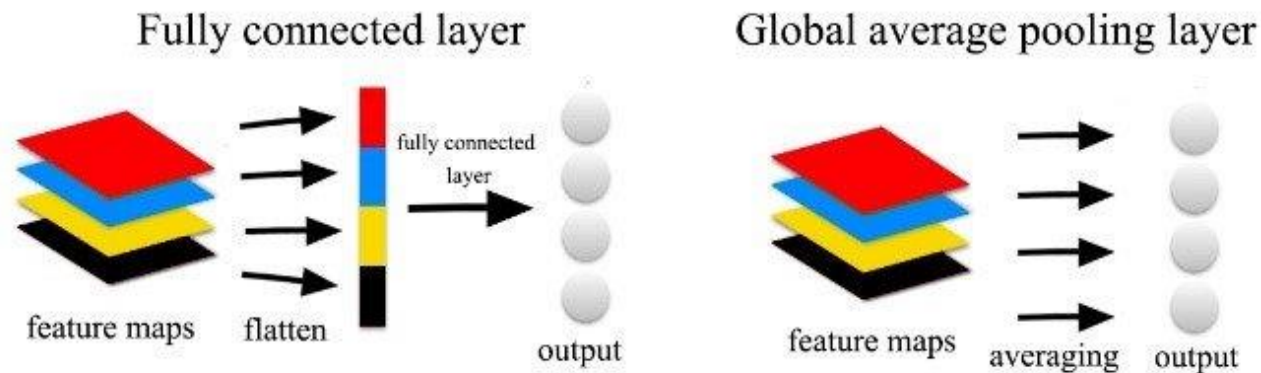
Most DL frameworks estimate the mean and SD for test and validation by performing a moving average of layer's input means and SD then uses this mean and SD for BatchNormalization during the test/validation

Global Average Pooling

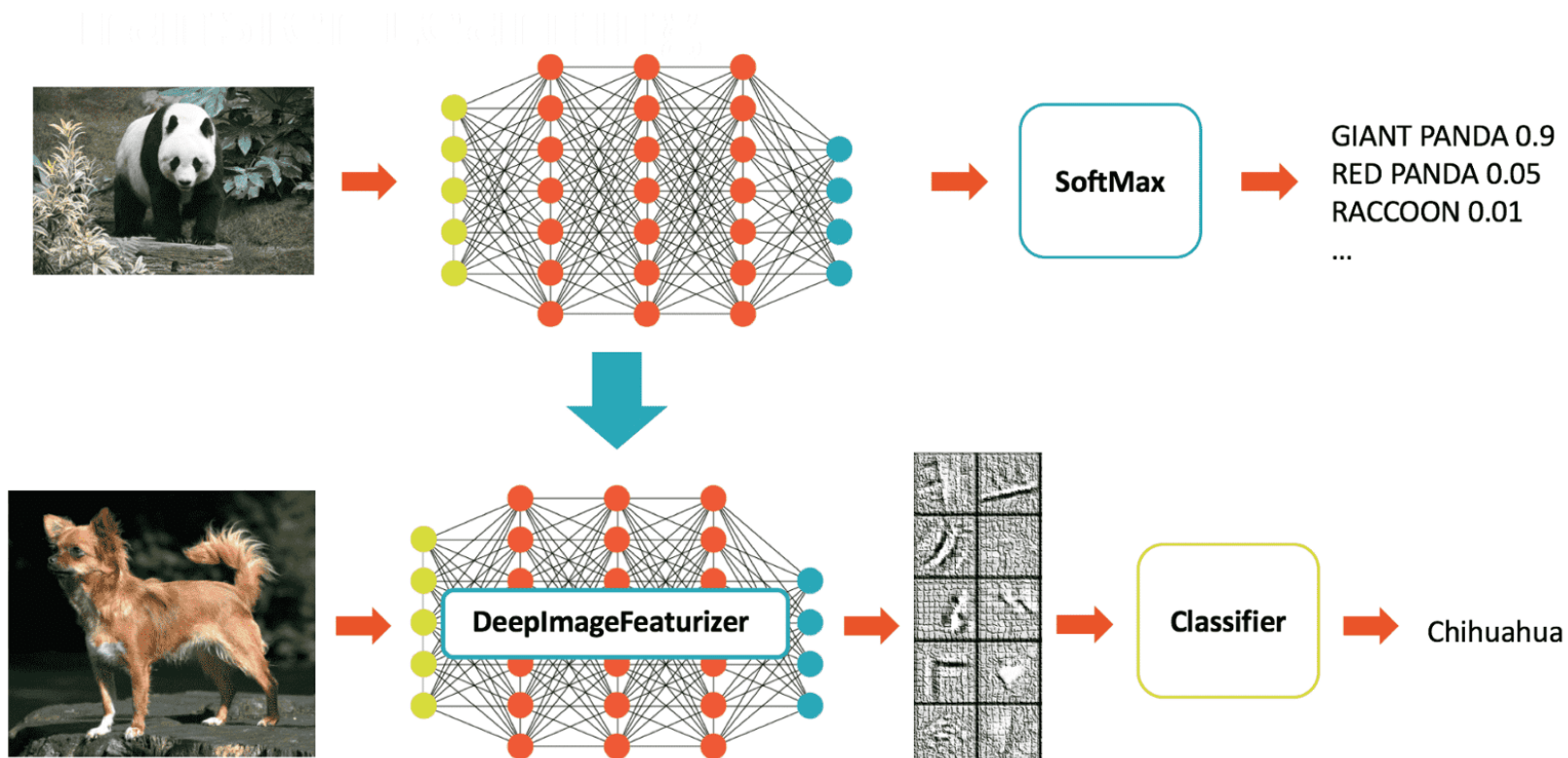
Reduces the increase in number of params caused by flattening layer

Takes an average of all the feature maps of the layer

E.g. if the last conv layer in the network has 512 filters thus producing 512 feature maps, the global average pooling will produce an output vector of 512



Transfer Learning

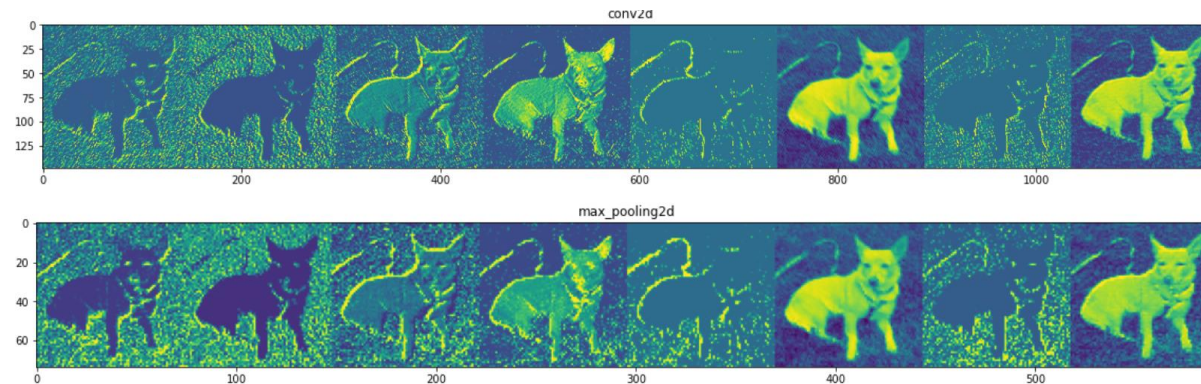


Transfer Learning

1. Transfer Learning is the transfer of the knowledge (feature maps) that the network has acquired from one task, where we have large amount of data, to a new task where the data is not abundantly available.
2. Generally used where a neural network model is trained on a problem similar to the problem that is being solved.
3. The intuition behind transfer learning is that if a model is trained on a large and general enough dataset, the model will effectively serve as a generic function for that problem.
4. We use the feature maps that this model has learnt, and without having to train a new model on a large dataset we can transfer the learning of the trained model's knowledge to our model and use it as a base starting model for our own task

How Transfer Learning works ?

1. What is really being learned by a CNN during training ? Feature Maps
2. How are these features learned ? During backpropagation process the weights are updated until we get to the optimized weights that minimize the error function
3. What is the relationship between features and weights ? Feature maps are a result of passing the weights filter/kernel on the input image during the convolution process
4. What is really being transferred from one network to another ? To transfer features we use the optimized weights of the pretrained network and reuse them as the starting point for the training process to adapt to our new problem



One Problem to Rule them all: ImageNet

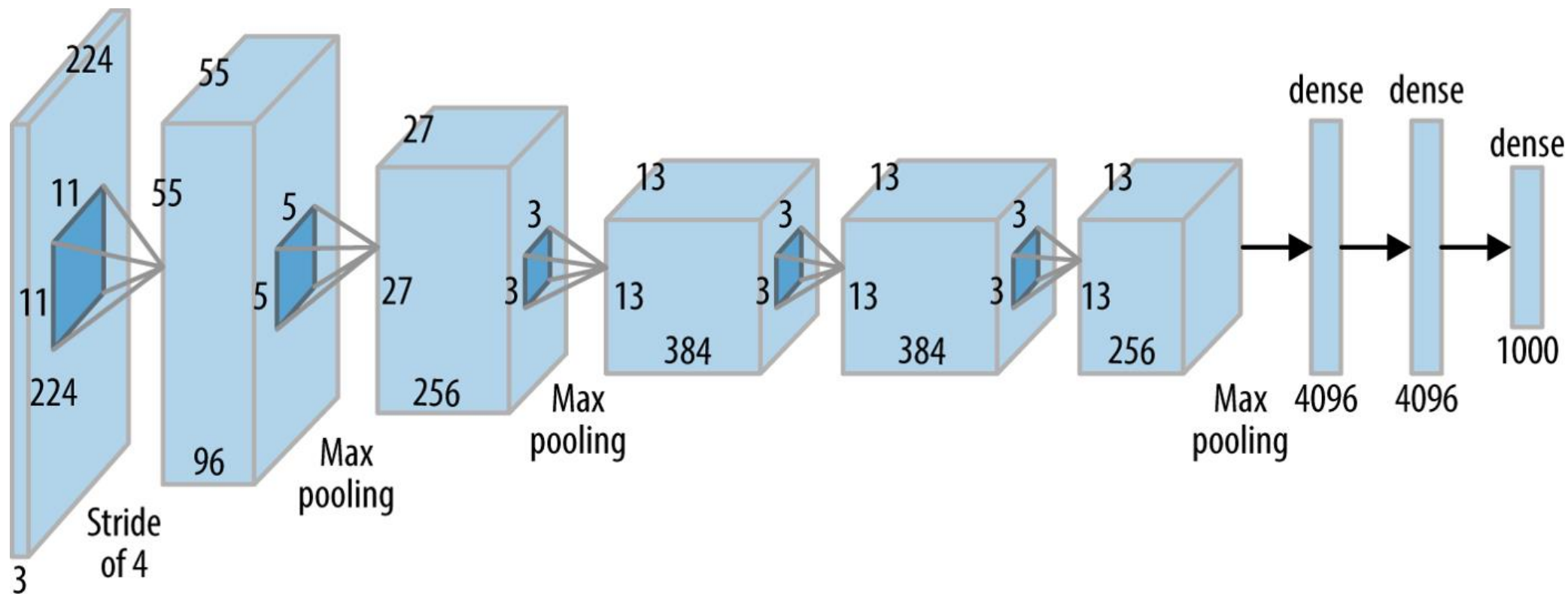
ImageNet Large Scale Visual Recognition Challenge

1. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale. One high level motivation is to allow researchers to compare progress in detection across a wider variety of objects -- taking advantage of the quite expensive labeling effort. Another motivation is to measure the progress of computer vision for large scale image indexing for retrieval and annotation.

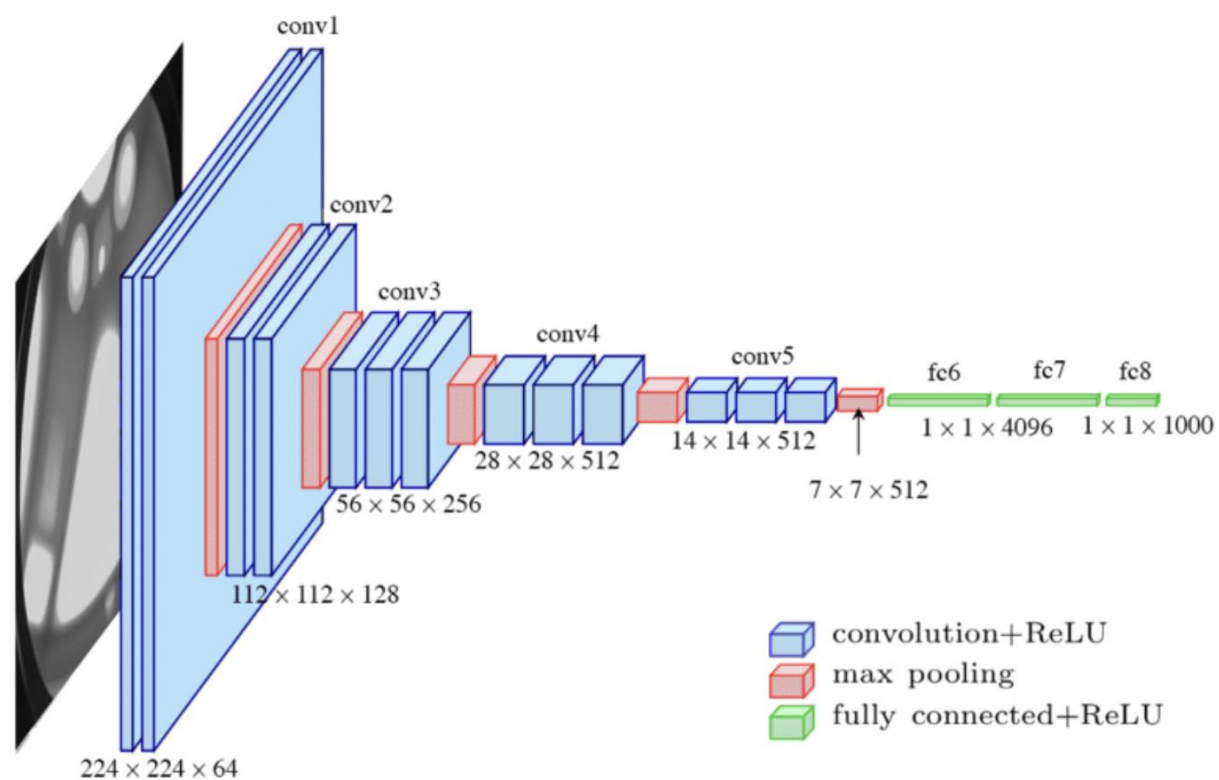
Transfer Learning Approaches

1. Using a pretrained network as a classifier
2. Using a pretrained network as a feature extractor
3. Finetuning

AlexNet

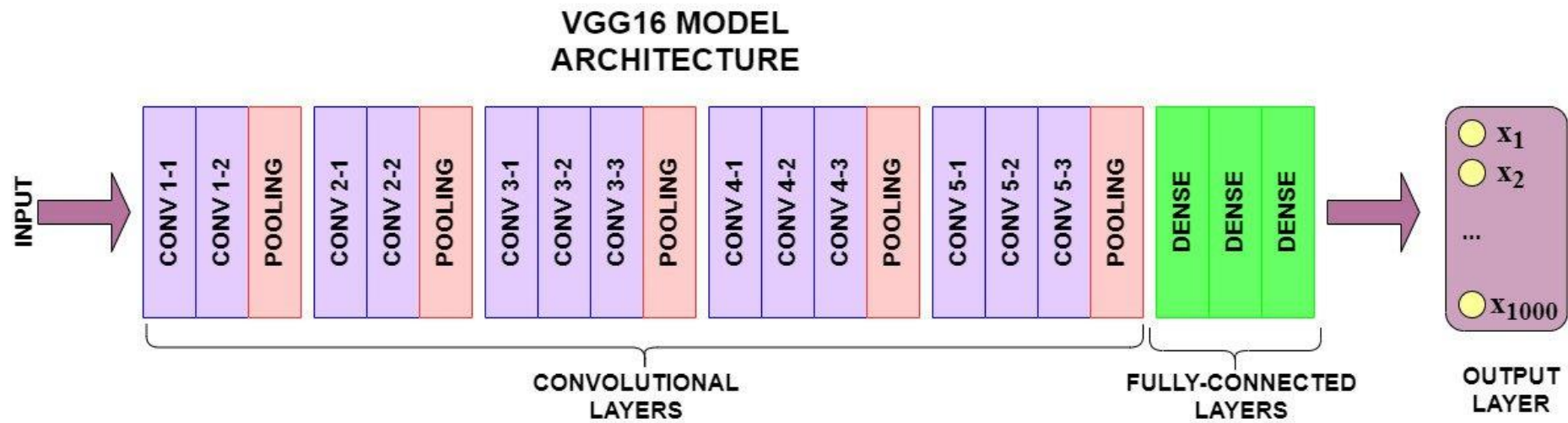


VGG-16



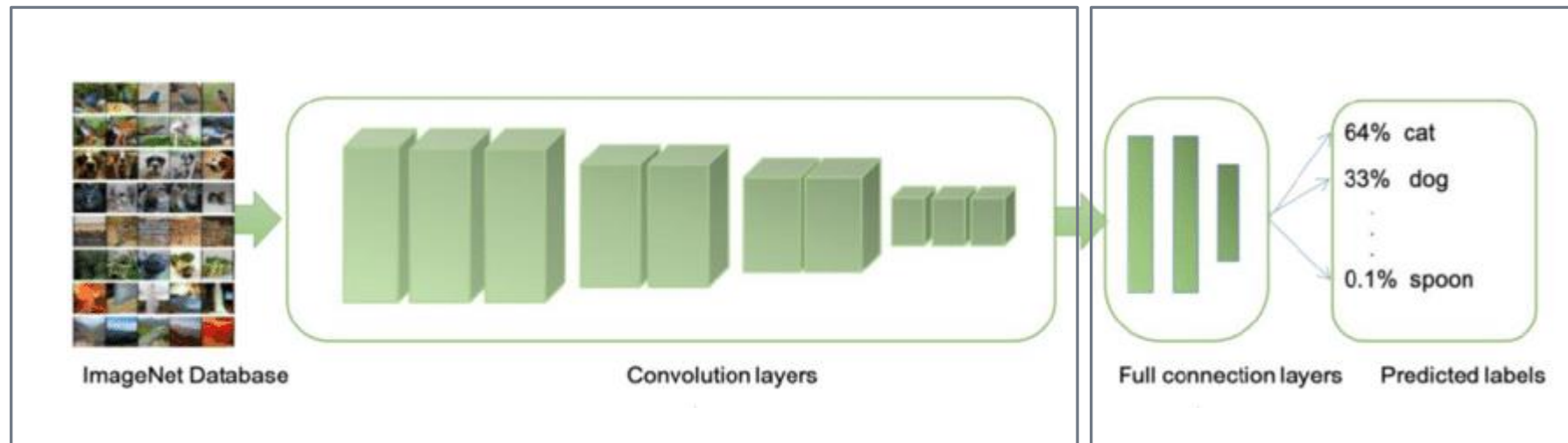
Pretrained Network as a Classifier

Produces the class output from the data labels it has been trained on



Pretrained Network as a Feature Extractor

1. Use the convolutional layers of the trained model to extract features
2. Remove the classifier of the pretrained network and add your own classifier

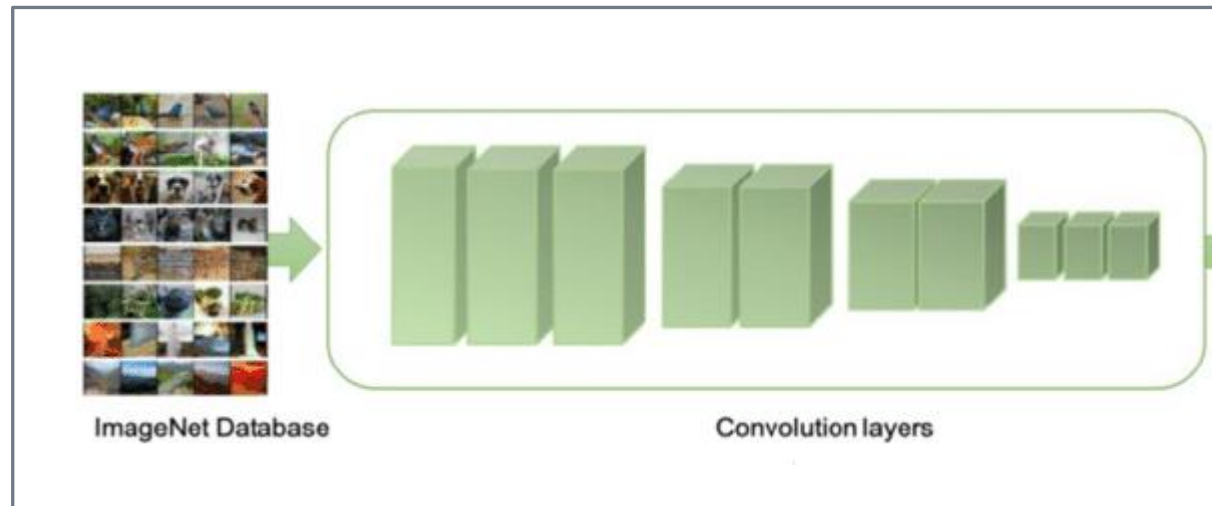


Freeze the layers and use
pretrained weights

Add a classifier on top, fully
connected layers + decision
neuron

Pretrained Network as a Feature Extractor

Use the convolutional layers of the trained model to extract features. What if you could use some other classifier instead of a fully connected layers + decision neuron (which is basically a NN) ? What if you could use a Naïve Bayes, SVM, RandomForest or a XGBoost Classifier



Freeze the layers and use pretrained weights

Add a classifier on top

Pretrained Network as a Feature Extractor

1. Use the image embeddings extracted from the pretrained network and build a tabular dataset
2. Use the target labels and these extracted features to train the classifier model like you usually do

```
In [76]: feature_extractor = Model(model.inputs,model.layers[-2].output)
```

```
In [77]: features = feature_extractor.predict(validation_generator)
```

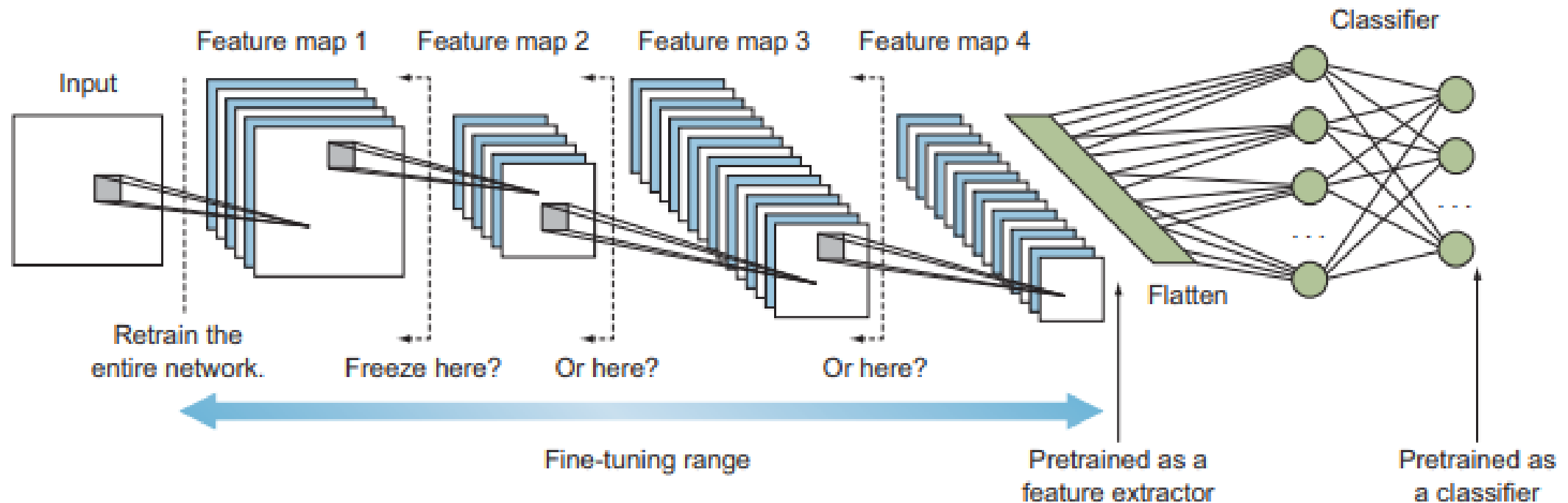
```
In [90]: features.shape
```

```
Out[90]: (2000, 512)
```

Fine Tuning

1. So far we have seen how to use a pretrained network directly for classification or as a feature extractor. We generally use these approaches when the domain of the target dataset (problem we are interested in solving) is somewhat similar to the source domain (imagenet problem)
2. But what if the target domain is different from the source domain or if its very different ?
3. We just extract the correct feature maps from the source domain pretrained network and fine tune them to the target domain. This is known as fine tuning.
4. Fine tuning can be formally defined as freezing a few of the network layers that are used for feature extraction and transferring all of its high level feature maps (outputs of later convolutional layers) to your domain.

So how many layers should we freeze ?



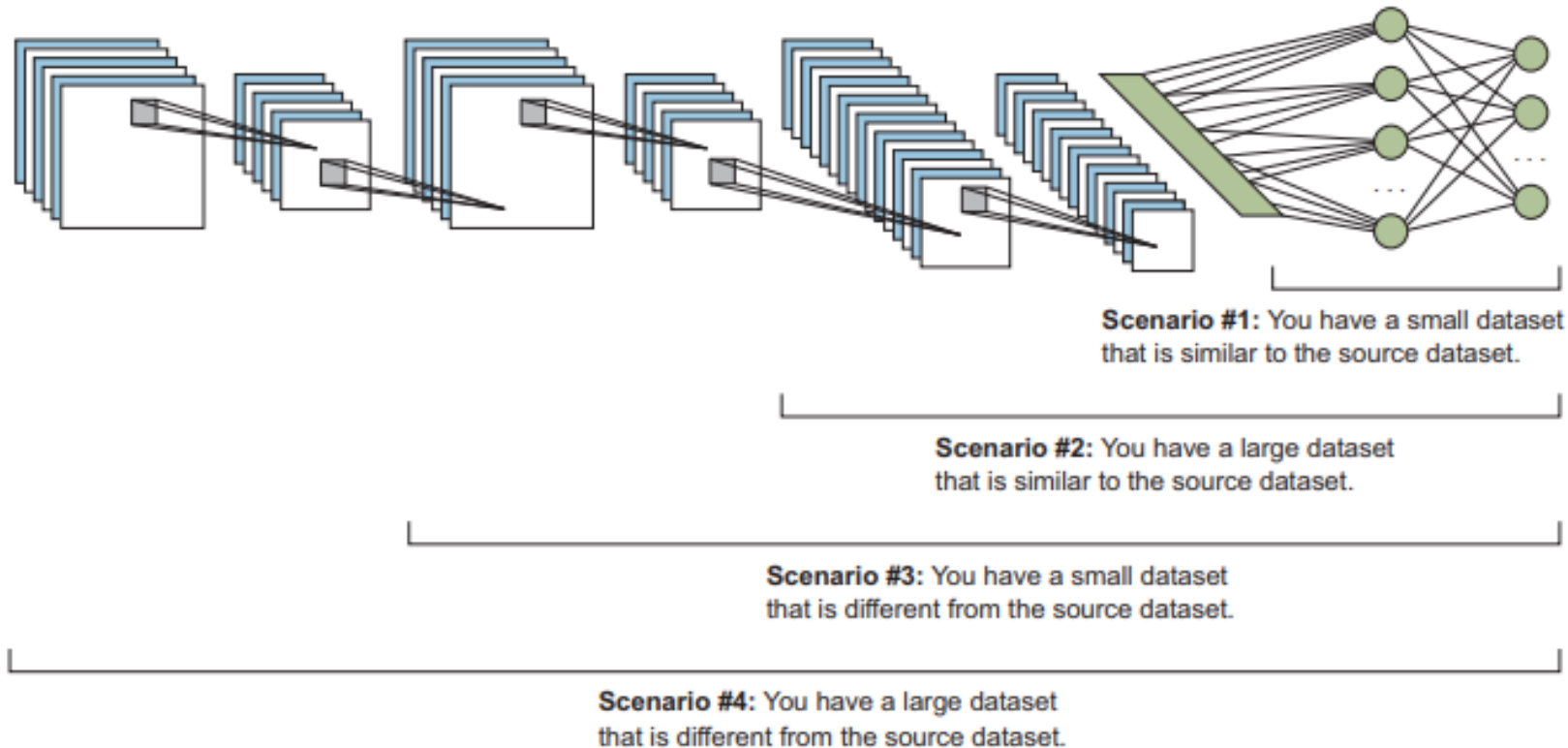
Choosing the appropriate level of Transfer Learning

Choosing the level of transfer learning depends on two important factors:-

1. Size of the target dataset (small or large): When we have a small dataset, the network probably won't learn much from training on larger number of layers, so it will tend to overfit the new data. In this case we want to rely more on the pretrained weights of the source dataset
2. Domain similarity of the source and target datasets: Is the target problem similar or different than the source problem ?

These two factors lead to 4 different scenarios

Choosing the appropriate level of Transfer Learning



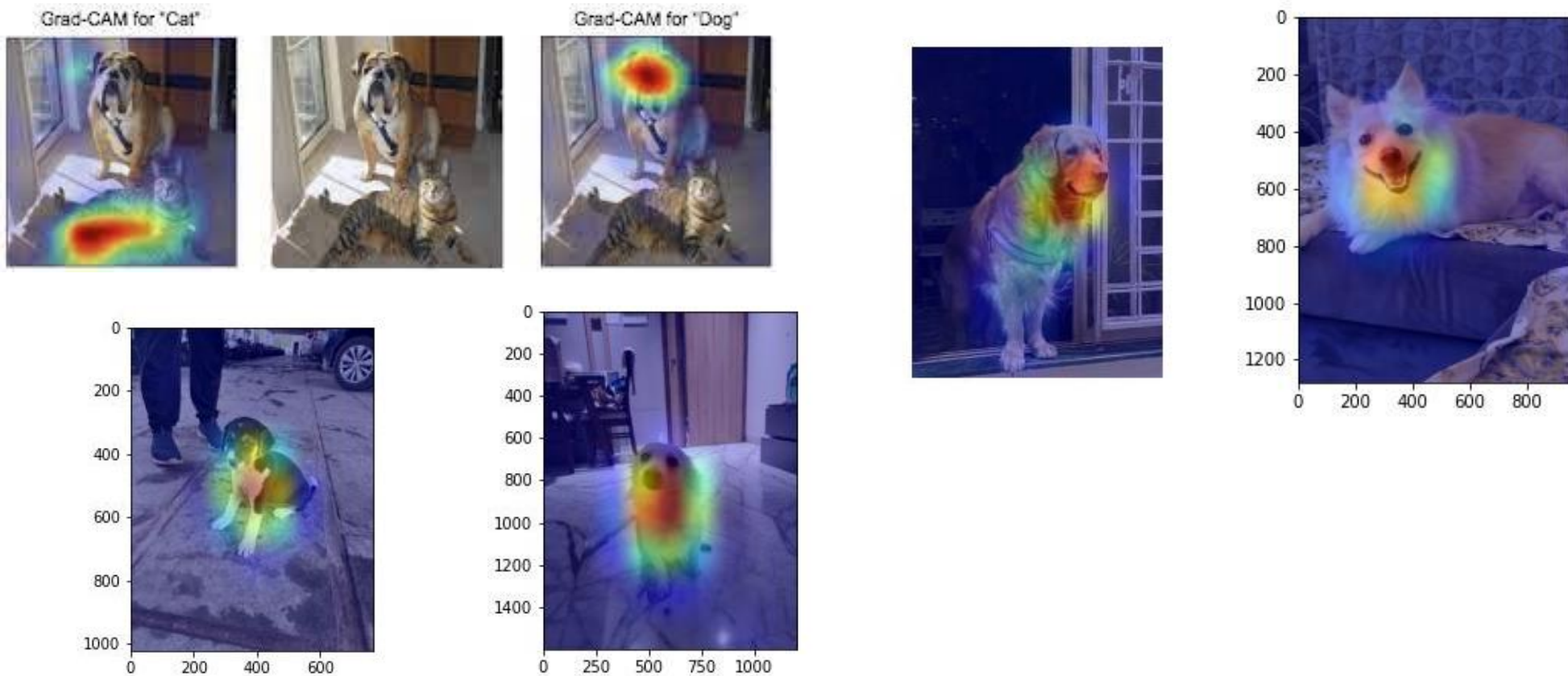
Choosing the appropriate level of Transfer Learning

Scenario	Size of the target data	Similarity of the original and new datasets	Approach
1	Small	Similar	Use pretrained network as a feature extractor
2	Large	Similar	Freeze 60-80% of the network and fine tune the rest of the network
3	Small	Very Different	Since target dataset is different it might not be the best to freeze the higher level features of the pretrained network because they contain source dataset specific features. It will be better to retrain layers from somewhere early in the network. Freeze 30-50% of the network.
4	Large	Very Different	Fine tune the entire network/Train the entire network by using the existing weights of the pretrained network

Transfer Learning and Learning Rates

Note: Learning rates should be kept small when transfer learning is applied, reason being you do not want the trainable layers' weights to change drastically during your training process.

Grad-CAM: What do CNNs learn?



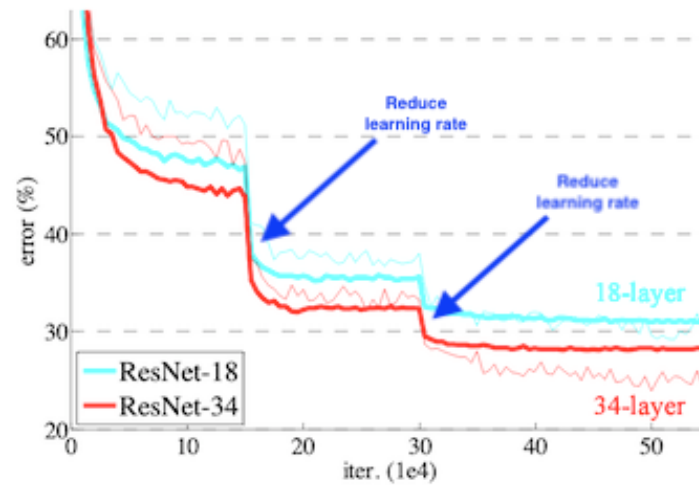
Miscellaneous

Model Callbacks

EarlyStopping

Checkpointing

ReduceLROnPlateau



ROC and AUC

<https://towardsdatascience.com/demystifying-roc-curves-df809474529a>

<https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5>

Thank you

For any queries drop an email at: quadeershaikh15.8@gmail.com