A Gentle Introduction to Convolutional Neural Networks

- Shashank Mujumdar, Abhinav Jain
IBM Research, India

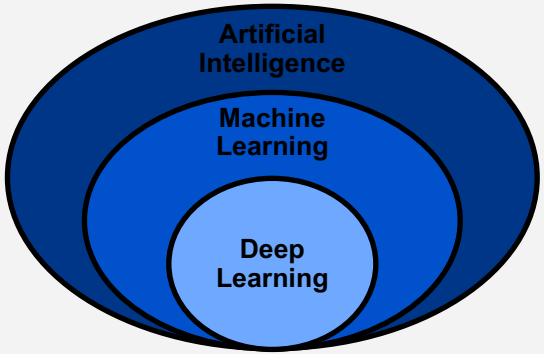
Outline

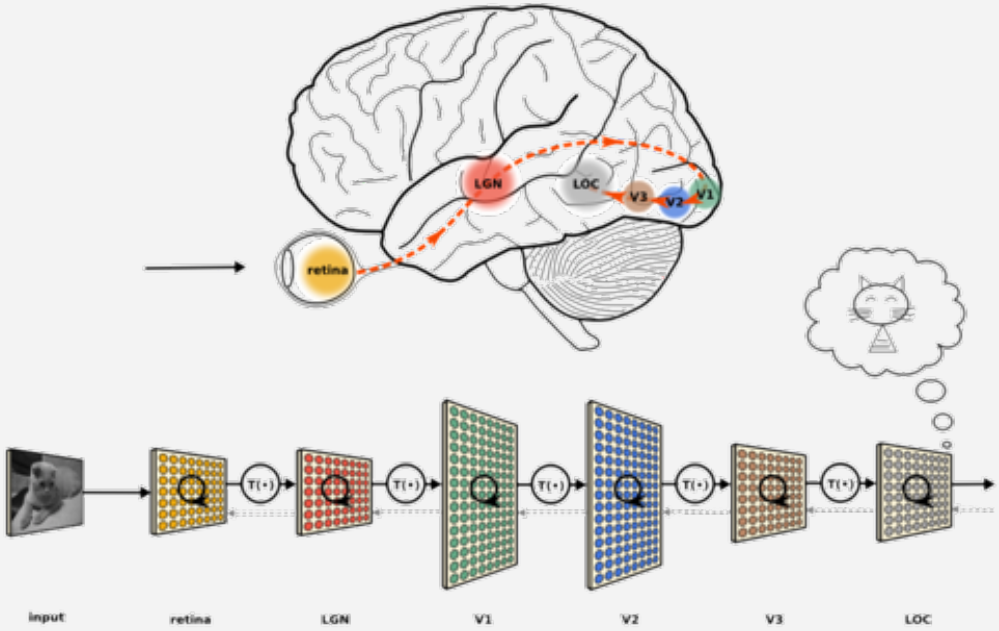
Code Repository: https://github.com/jabhinav/Talk-IGDTUW

What is Deep Learning?

Deep Learning is the set of machine learning algorithms inspired by the structure and function of the brain which learns useful representations, aka features directly from the data.

- **Artificial Intelligence:** Study of all the techniques that enable machines to mimic human behavior
- **Machine Learning:** Subset of AI techniques that utilize statistical methods to train machines
- **Deep Learning:** Subset of ML techniques that utilize multi-layered neural networks to train machines





The human brain has **deep architecture**, in which the cortex seems to have a **generic learning approach**. A given input is perceived at **multiple levels of abstraction**. Each level corresponds to a **different area of the cortex**. We process information in **hierarchical ways**, with multi-level transformation and representation. Therefore, we learn **simple concepts first** then **compose them together**.

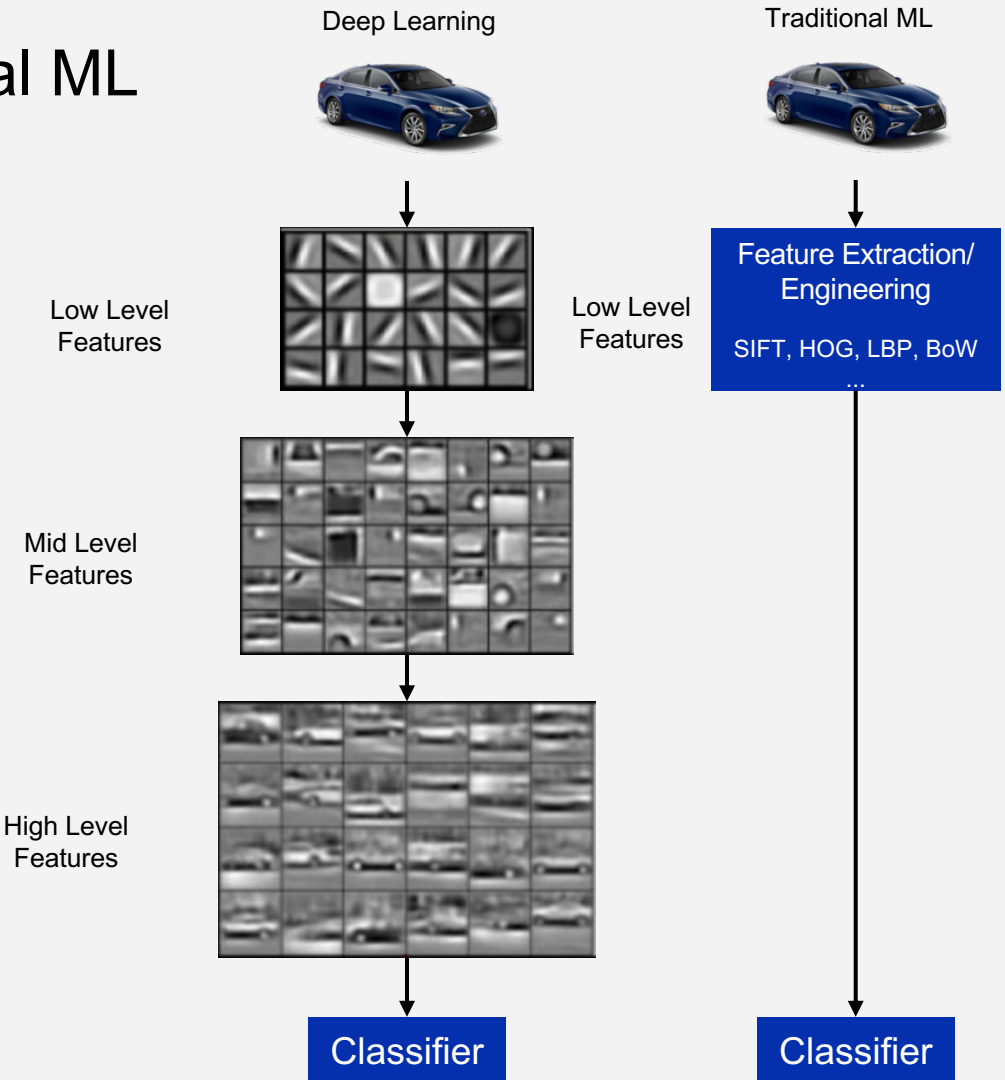
Image Credits: https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781785880360/1/ch01lvl1sec13/the-motivation-of-deep-architecture

Deep Learning vs Traditional ML

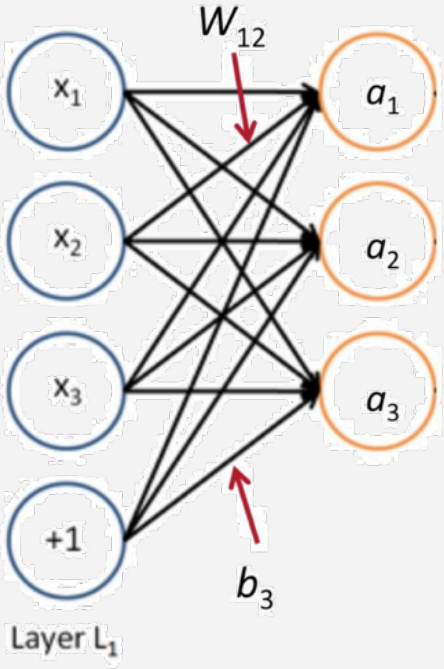
Deep learning refers to the set of Machine Learning algorithms that involve the use of Multi-layered Neural Networks which learn abstract feature representations from large amounts of training data.

Deep Learning:

- Cascade of non-linear transformations
- End to end learning
- General framework (any hierarchical model is deep)



Neural Networks



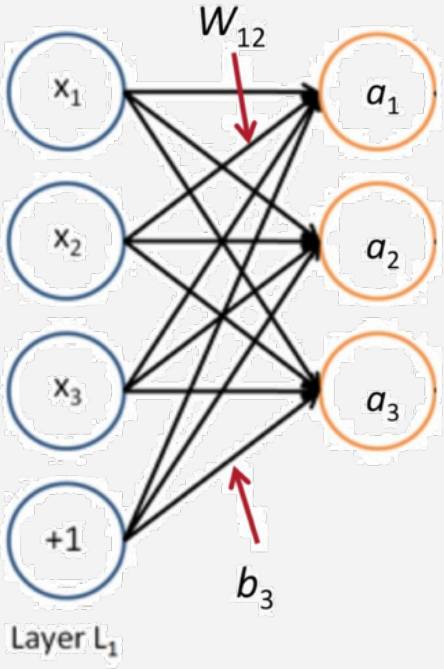
$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
$$a_3 = f(W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3)$$

$$z = Wx + b$$
$$a = f(z)$$

- Consists of linear combination of inputs through a non-linear function
- W is the weight parameter to be learned
- x is the output of the previous layer
- f is a non-linear function, popular choice is ReLU (Rectified Linear Units)

$$ReLU(x) = max(x, 0)$$

Neural Networks



Q: Why can't the mapping between layers be linear?
A: Because composition of linear functions is a linear function. Neural network would reduce to (1 layer) logistic regression.

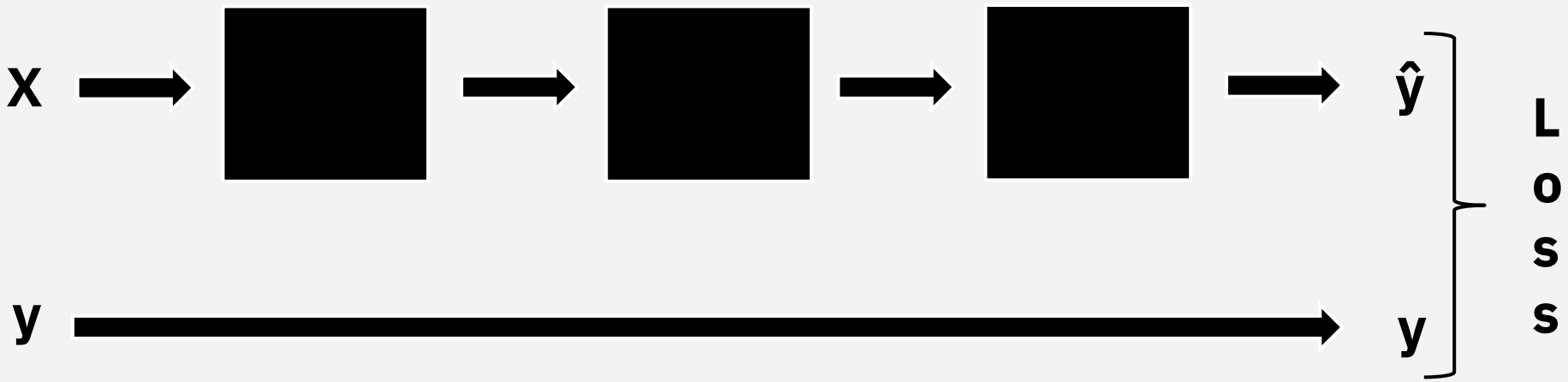
Q: Why do we need many layers?
A: When input has hierarchical structure, the use of a hierarchical architecture is potentially more efficient because intermediate computations can be re-used. DL architectures are efficient also because they use distributed representations which are shared across classes.

Q: What does a hidden unit do?
A: It can be thought of as a classifier or feature detector.

Q: How many layers? How many hidden units?
A: Cross-validation or hyper-parameter search methods are the answer. In general, the wider and the deeper the network the more complicated the mapping.

Q: How do I set the weight matrices?
A: Weight matrices and biases are learned. First, we need to define a measure of quality of the current mapping. Then, we need to define a procedure to adjust the parameters.

Neural Networks Training



Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

Forward Propagation:

- Compute network activations on minibatch of samples
- Compute Loss on minibatch of samples

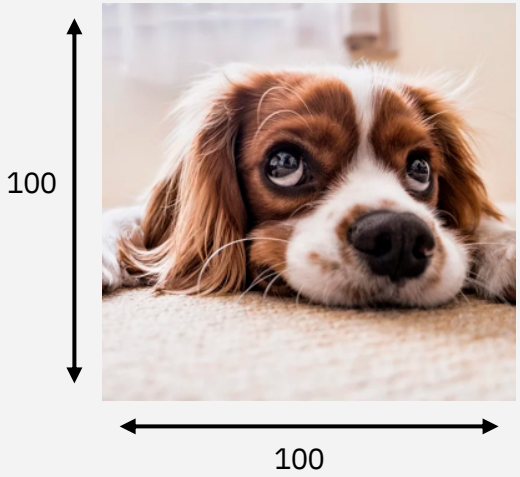
Backword Propagation:

- Compute gradient w.r.t. parameters
- Use gradients to update network parameters

Convolutional Neural Networks

- Computer vision is the design of computers that can process visual data and accomplish some given task

- We can design neural networks that are specifically adapted for such problems



High dimensional input
100 x 100 pixels = 10000 inputs (30000 inputs for RGB image)

For a fully connected layer with 1000 Hidden Units =>
10000*1000 = **10^7 parameters!**

Regular Neural Networks do not scale well to images. Multiple fully connected layers would lead to huge number of parameters and quickly result in overfitting while training.

Convolutional Neural Networks

- How do we design a deep network that can scale to images?
- Can we exploit the 2D topology of pixels?
- Can we build invariance to certain variations such as translation, illumination, skew etc?

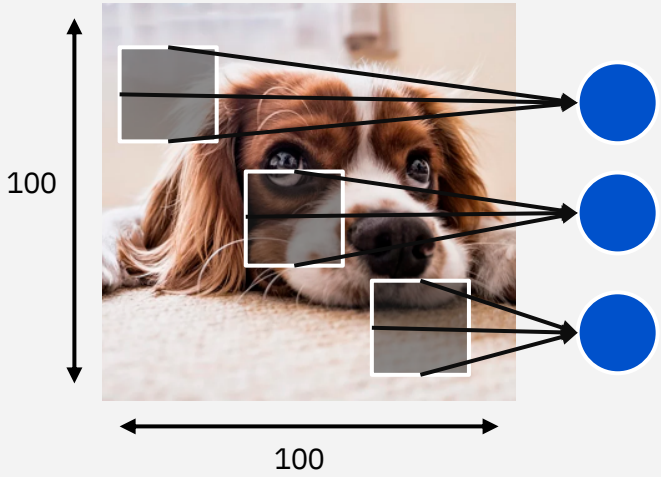
Convolutional Neural Networks are the solution!

Characteristic Properties:
- Local Connectivity
- Parameter Sharing
- Pooling/Subsampling Hidden Units

Convolutional Neural Networks

Local Connectivity

- Each hidden unit is only connected to small patch of the input
- Each hidden unit is connected to all the channels of the input



High dimensional input
100 x 100 pixels = 10000 inputs (30000 inputs for RGB image)
Total Hidden Units = 1000
Receptive Field (Filter size) = 5 x 5
After local connectivity => 25*1000 = 25000 parameters

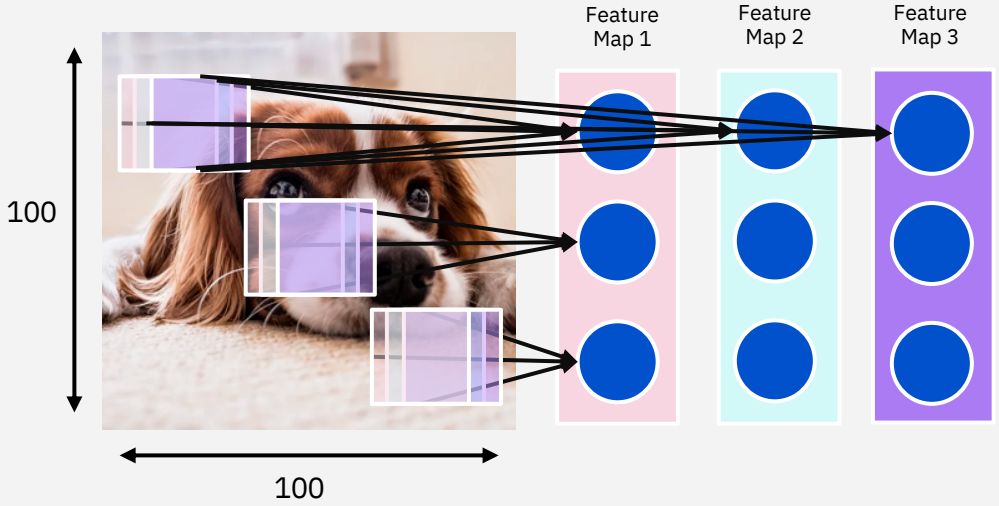
For RGB image, every hidden unit =>
5 x 5 x 3 = 75 parameters
Total Parameters = 75 * 1000 = 75000 parameters

The connections are local in space (along width and height), but always full along the entire depth of the input volume.

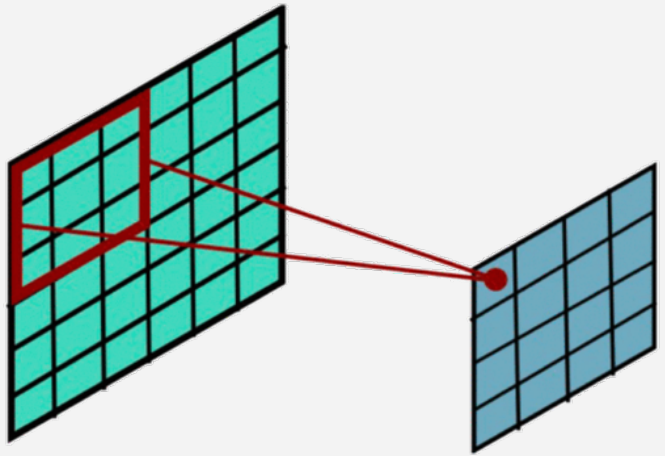
Convolutional Neural Networks

Parameter Sharing

- Share matrix of parameters across certain units
- Units organized into same “feature map” share parameters
- Hidden units within a feature map cover different positions in the input
- The same feature is extracted at every location



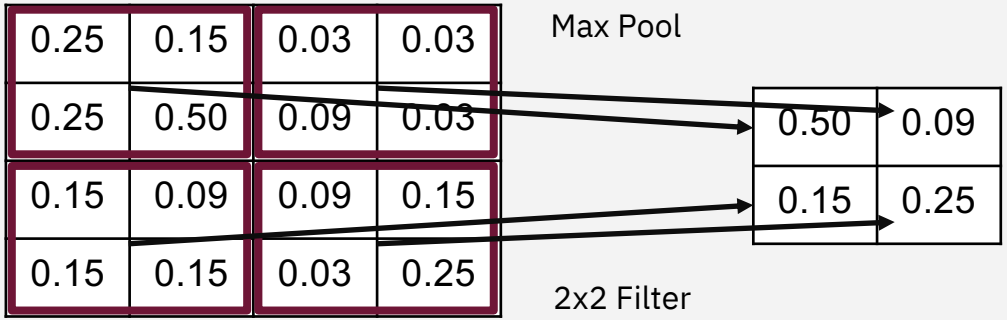
Same color => Same matrix of connections



Convolutional Neural Networks

Pooling/Subsampling

- Pool hidden units in same neighbourhood
- Pooling is performed in non-overlapping neighbourhoods (subsampling)
- Introduces invariance to local translations
- Reduces the number of hidden units in hidden layer



By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Convolutional Neural Networks

Convolution Operation

- The convolution of an image x with a kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} \; k_{r-p,r-q}$$

80	0	50
70	20	10
0	40	0

x

*

0	1
0.5	-1

k

= ?

Can you work out this example?

Convolutional Neural Networks

Convolution Operation

- The convolution of an image x with a kernel k is computed as follows:

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q}\ k_{r-p,r-q}$$

80	0	50
70	20	10
0	40	0

x

*

0	1
0.5	-1

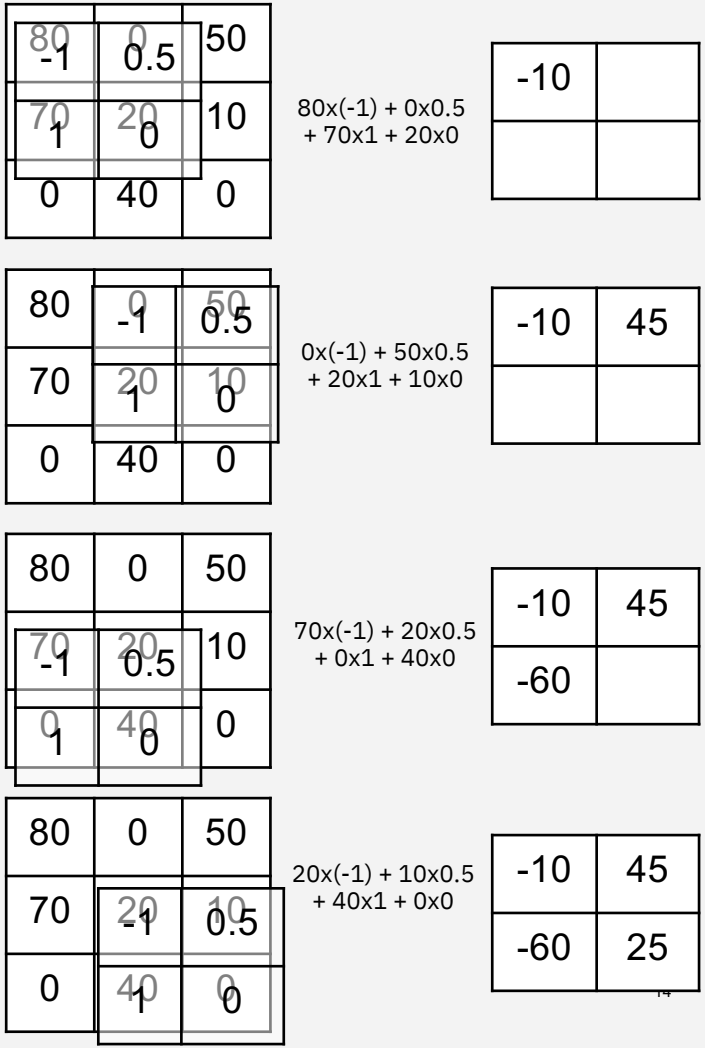
k

Flip ~

-1	0.5
1	0

Activations from layer ***i*** are mapped to layer ***j*** as follows

- Convolution kernel $k_{ij} = \widetilde{W}_{ij}$ from the connection matrix W_{ij}
- Apply the convolution $x_i * k_{ij}$
- Equivalent to computing discrete correlation of x_i with W_{ij}

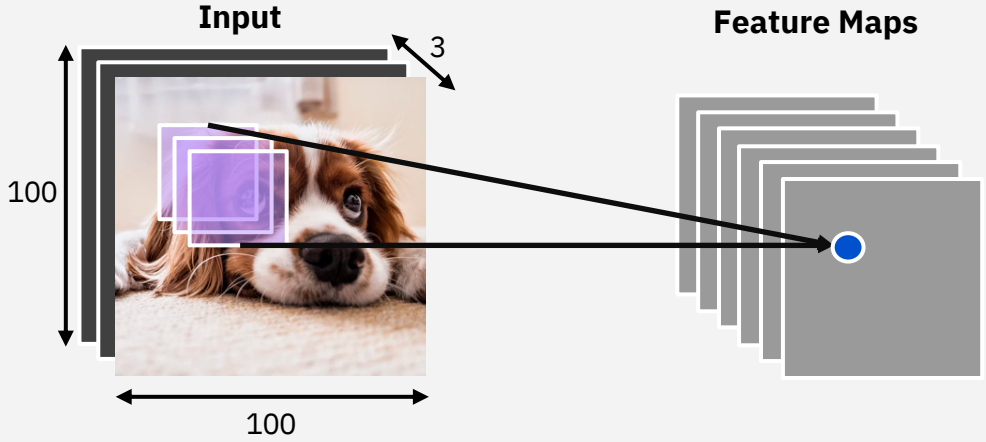


Convolutional Neural Networks

Spatial Arrangement

- How many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume - the **depth, stride** and **zero-padding**.

- **Depth** of the output volume corresponds to the number of filters we would like to use, each potentially learning to look for something different in the input.



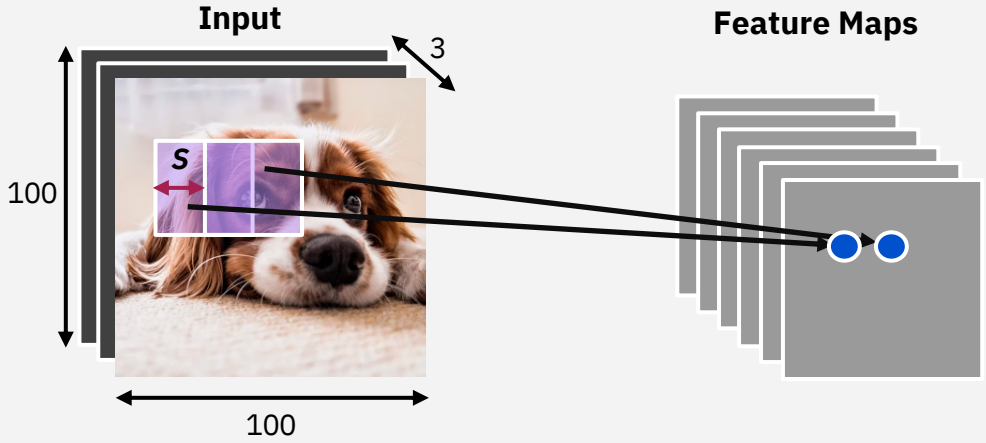
- Filter Size = FxFxC
- Each Filter after convolution operation generates a corresponding feature map
- Depth of output volume is the total number of filters used in that layer.
- Different filters may activate in presence of various oriented edges or blobs of color etc.

Convolutional Neural Networks

Spatial Arrangement

- How many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume - the **depth, stride** and **zero-padding**.

- **Stride** - For a convolutional operation, the stride ***S*** denotes the number of pixels by which the filter moves after each operation.



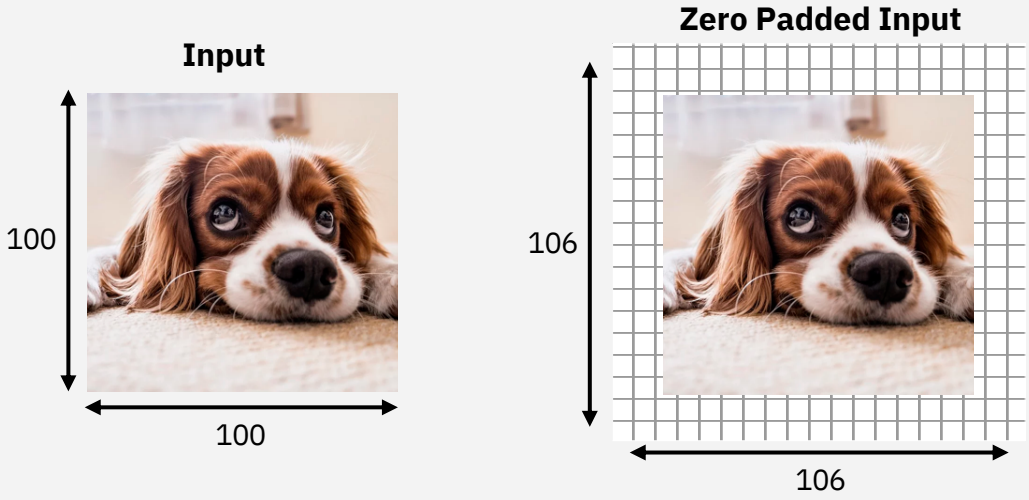
- When the stride is 1 then we move the filters one pixel at a time.
- When the stride is 2 (or uncommonly 3 or more) then the filters jump 2 pixels at a time as we slide them around.
- This will produce smaller output volumes spatially.

Convolutional Neural Networks

Spatial Arrangement

- How many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume - the **depth, stride** and **zero-padding**.

- **Zero Padding** - Zero-padding denotes the process of adding zeroes to each side of the boundaries of the input.



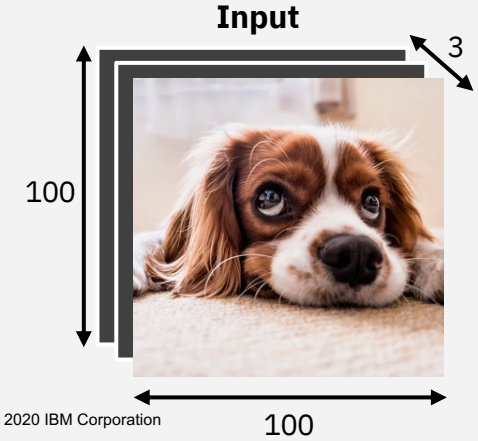
- Zero Padding ***P = 3***
- Input size 100x100
- Output size 106x106

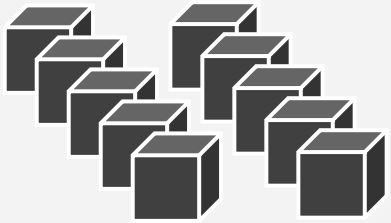
- **Valid Padding:** Implies no padding on the input. Only uses valid input data. Might drop rightmost columns or bottommost rows.
- **Same Padding:** Tries to pad evenly on all sides. With a stride of 1, the output size is same as the input size.

Convolutional Neural Networks

Spatial Arrangement

- **How to Decide Valid Stride and Zero-Padding?**
- We can compute the spatial size of the output volume as a function of the:
- input volume size ***(WxW)***
- the receptive field size of the Conv Layer neurons ***(FxF)***, or alternatively the filter size
- the stride with which the filters are applied ***(S)***
- the amount of zero padding used ***(P)*** on the border.
- Formula for calculating the output size => $\frac{W-F+2P}{S}+1$





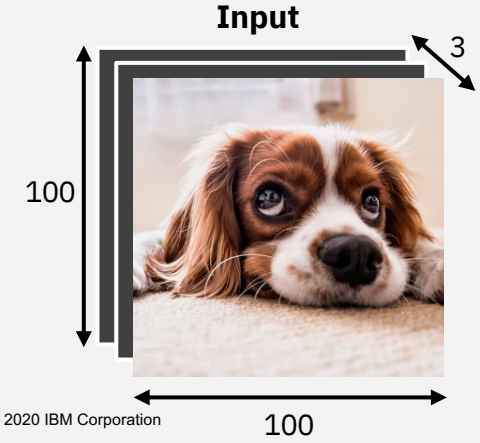
- 5x5x3 filters (10 filters)
- stride 1, pad 2

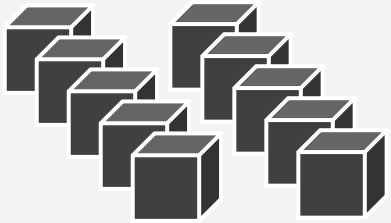
Output Volume = ?

Convolutional Neural Networks

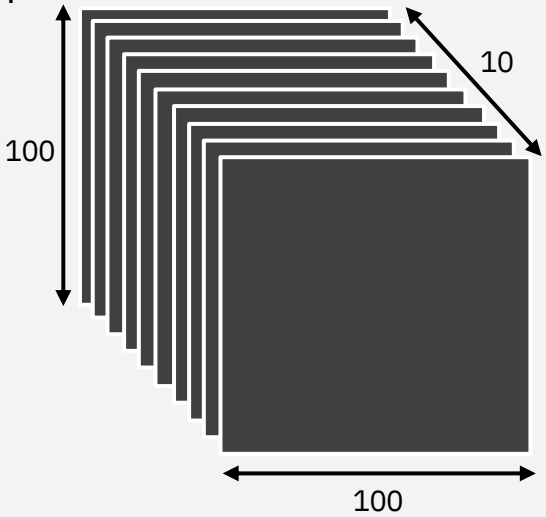
Spatial Arrangement

- **How to Decide Valid Stride and Zero-Padding?**
- We can compute the spatial size of the output volume as a function of the:
- input volume size ***(WxW)***
- the receptive field size of the Conv Layer neurons ***(FxF)***, or alternatively the filter size
- the stride with which the filters are applied ***(S)***
- the amount of zero padding used ***(P)*** on the border.
- Formula for calculating the output size => $\frac{W-F+2P}{S}+1$





- 5x5x3 filters (10 filters)
- stride 1, pad 2



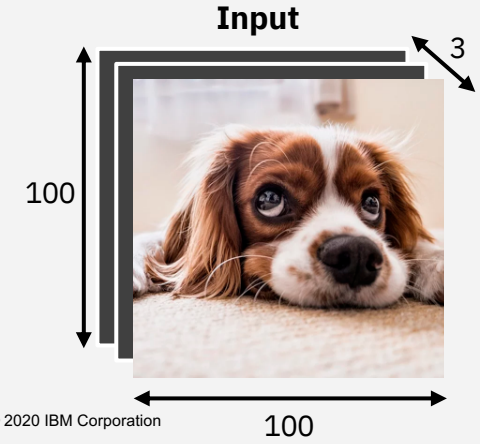
$\frac{100-5+2x2}{1}+1$

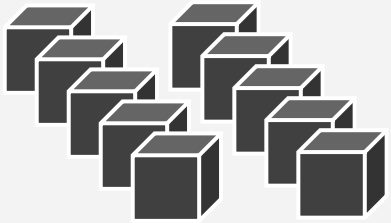
Output Volume = 100x100x10

Convolutional Neural Networks

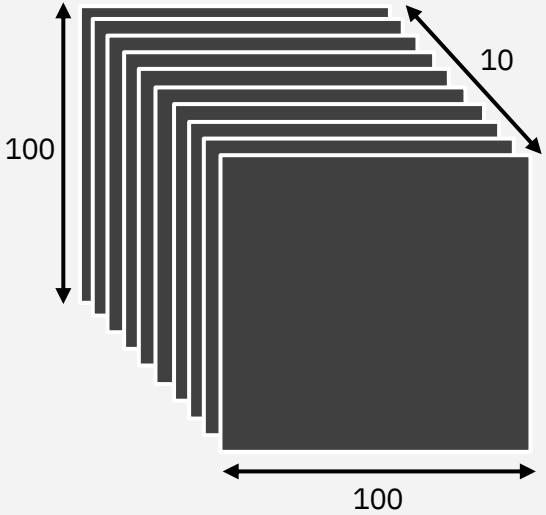
To Summarize, the Conv Layer:

- Accepts a volume of size $W_{inp} \times H_{inp} \times D_{inp}$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P
- Produces a volume of size $W_{out} \times H_{out} \times D_{out}$
  - $W_{out} = (W_{inp} - F + 2P)/S + 1$
  - $H_{out} = (H_{inp} - F + 2P)/S + 1$
  - $D_{out} = K$
  - Weights per filter = $FxFxD_{inp} + 1$ (for bias)





- 5x5x3 filters (10 filters)
- stride 1, pad 2

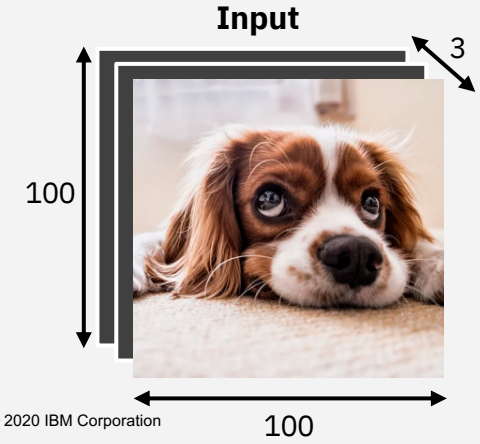


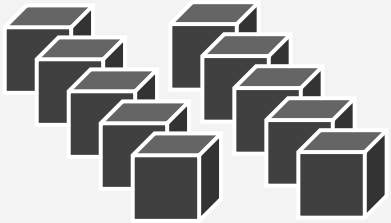
Total Parameters = ?

Convolutional Neural Networks

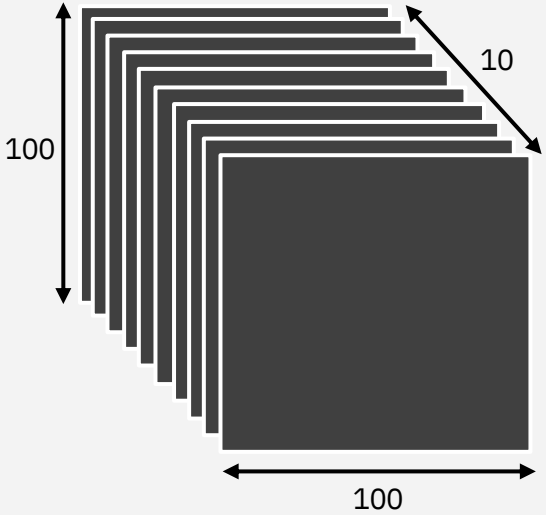
To Summarize, the Conv Layer:

- Accepts a volume of size $W_{inp} \times H_{inp} \times D_{inp}$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P
- Produces a volume of size $W_{out} \times H_{out} \times D_{out}$
  - $W_{out} = (W_{inp} - F + 2P)/S + 1$
  - $H_{out} = (H_{inp} - F + 2P)/S + 1$
  - $D_{out} = K$
  - Weights per filter = $FxFxD_{inp} + 1$ (for bias)





- 5x5x3 filters (10 filters)
- stride 1, pad 2

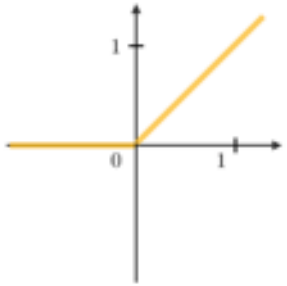
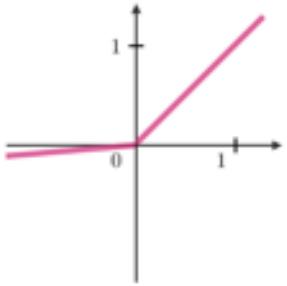
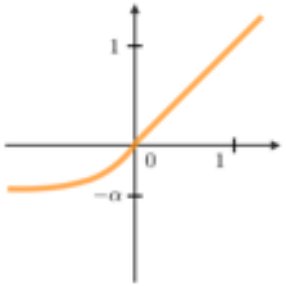


Total Parameters = (5x5x3 + 1)x10 = 760

Convolutional Neural Networks

Activation Layer

- Used to increase non-linearity of the network without affecting receptive fields of conv layers
- Most commonly used in training are Rectified Linear Units
- Others include Sigmoid, Tanh, SoftMax etc.

ReLU	Leaky ReLU	ELU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
		
• Non-linearity complexities biologically interpretable	• Addresses dying ReLU issue for negative values	• Differentiable everywhere

ReLU: Results in faster training

LeakyReLU: Addresses the vanishing gradient problem

Exponential Linear Units ELU: Differentiable Everywhere

Graph Credits: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks#activation-function

Convolutional Neural Networks

Activation Layer

- Used to increase non-linearity of the network without affecting receptive fields of conv layers

- **SoftMax:**
- A special kind of activation layer, usually at the end of FC layer outputs
- Can be viewed as a fancy normalizer (a.k.a. Normalized exponential function)
- Produce a discrete probability distribution vector

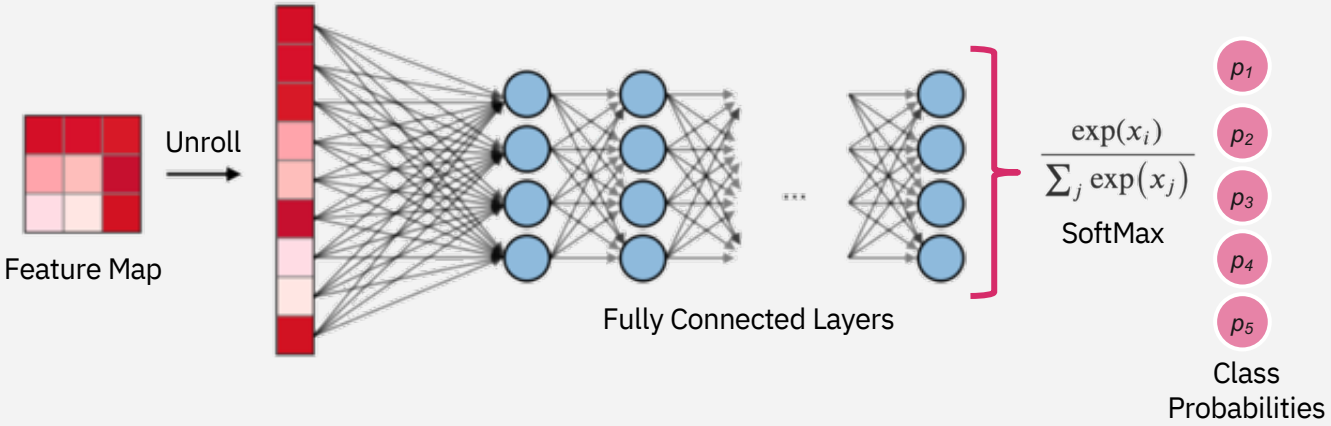
The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $x \in R^n$ and outputs a vector of output probability $p \in R^n$ through a softmax function at the end of the architecture.

$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Convolutional Neural Networks

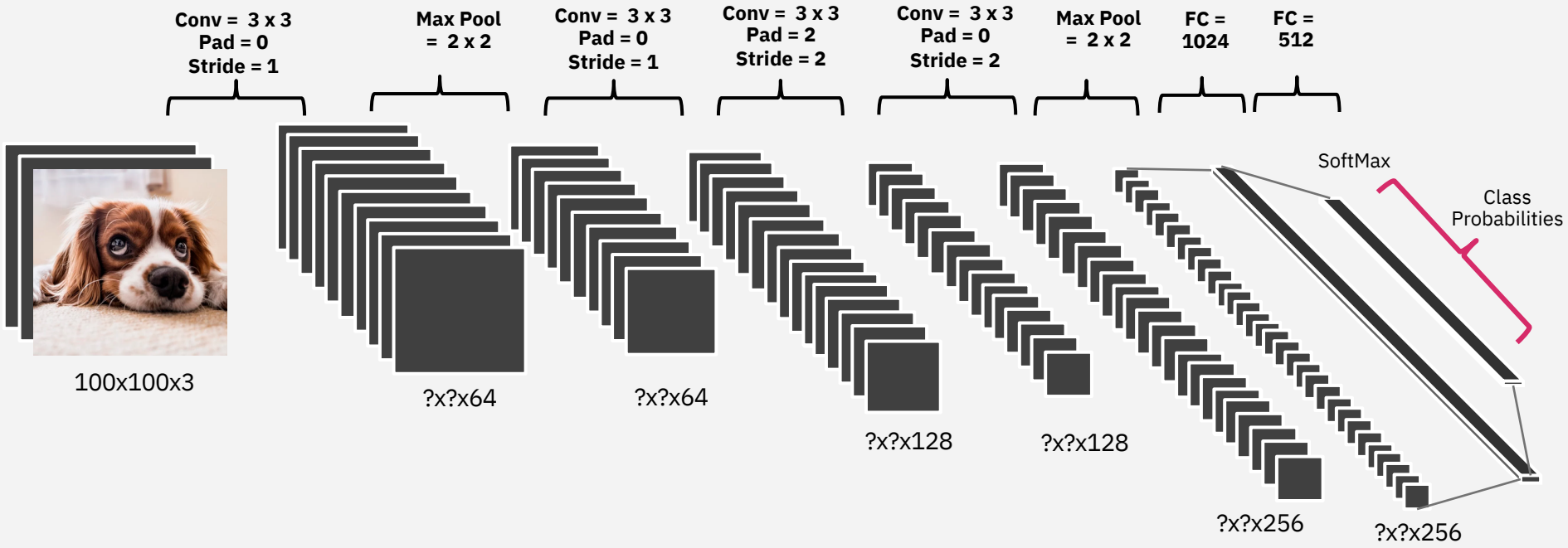
Fully Connected Layer

- **Fully Connected (FC)** - The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons.
- If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.
- Can view as the final learning phase, which maps extracted visual features to desired outputs.
- Common output is a vector, which is then passed through softmax to represent confidence of classification



Convolutional Neural Networks

Adding it all together



Convolutional Neural Networks

Adding it all together

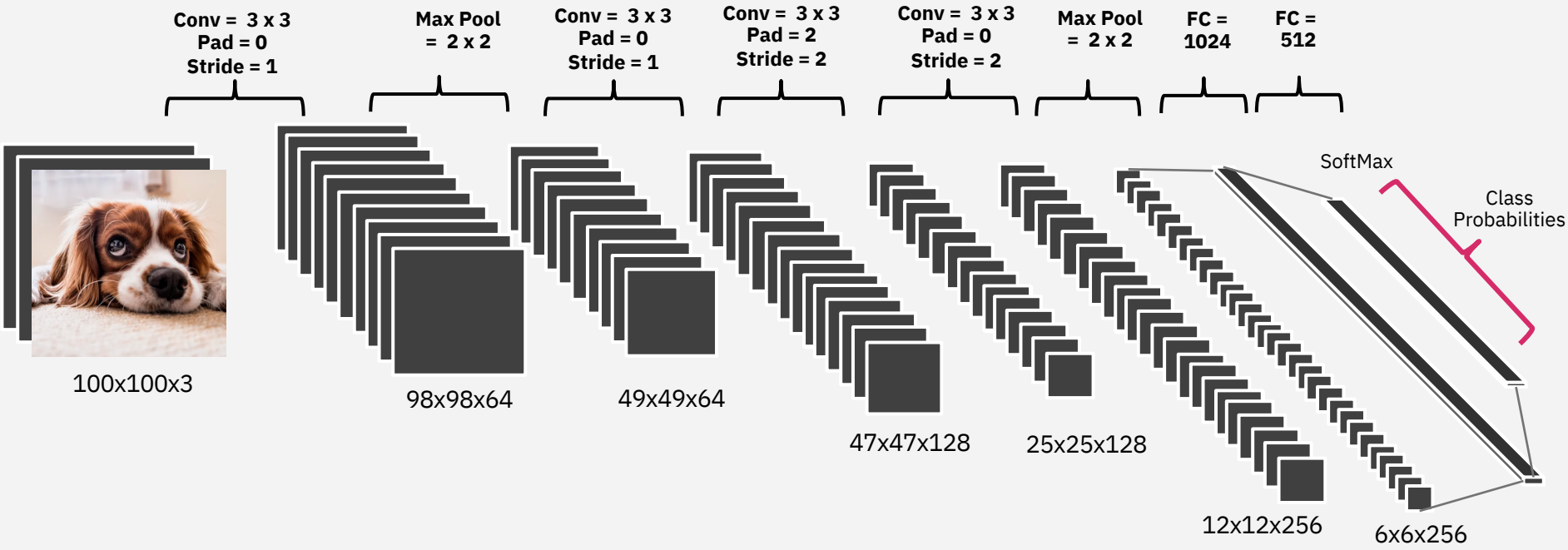


Image Classification

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- ImageNet is a large dataset of annotated photographs intended for computer vision research.
- More than 14 million images, more than 21 thousand classes
- 1 million images that have bounding box annotations
- Image Classification Challenge:
  - The dataset comprised approximately 1 million images and 1,000 object classes.
  - Additional 50,000 images for a validation dataset and 150,000 for a test set

- AlexNet (ILSVRC 2012)
  - Alex Krizhevsky, et al. Propose the first deep CNN architecture that achieves top results
- ZFNet (ILSVRC 2013)
  - Matthew Zeiler and Rob Fergus develop DeconvNet for visualizing what the CNN is learning. Their modification to Alexnet result in better performance.
- GoogLeNet (ILSVRC 2014)
  - Christian Szegedy, et al. from Google propose the Inception module that utilizes 1x1, 3x3, 5x5 and Max Pool operations at every layer and combines them before passing to the next layer.
- VGG (ILSVRC 2014)
  - Karen Simonyan and Andrew Zisserman from the Oxford Vision Geometry Group (VGG) study the effects of performance on very deep CNN architectures and propose a 16 layer and 19 layer deep CNN that achieves top results.
- ResNet (ILSVRC 2015)
  - Kaiming He, et al. from Microsoft Research propose residual learning framework that utilize skip connections to reliably train very deep CNN architecture (152 layers!)

Image Classification

AlexNet Architecture

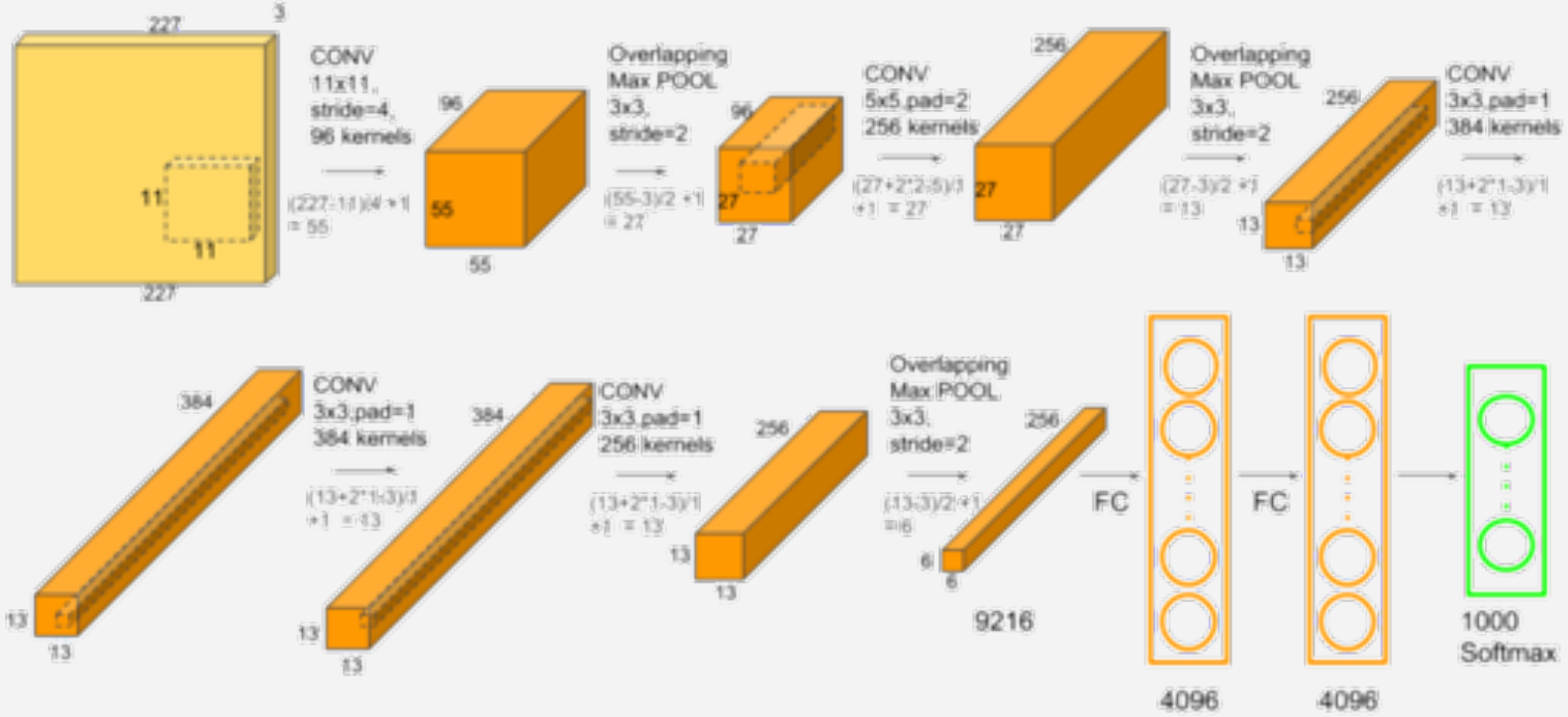
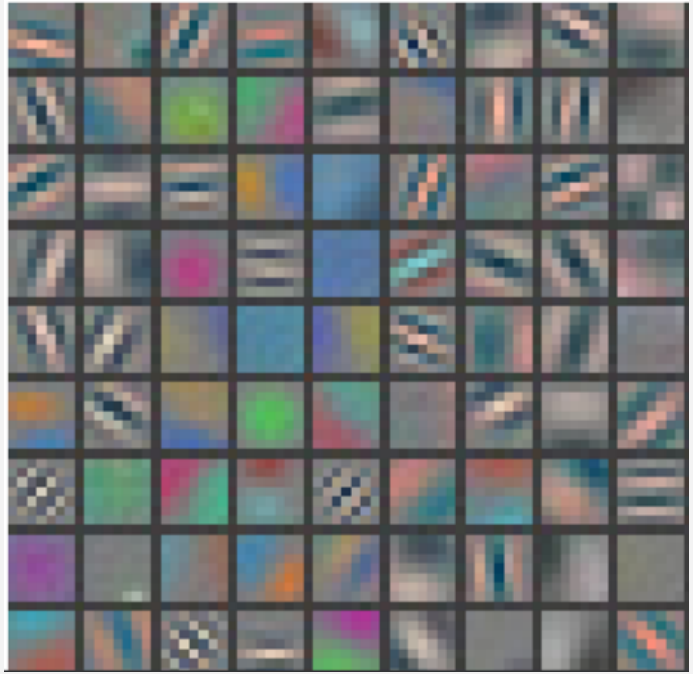


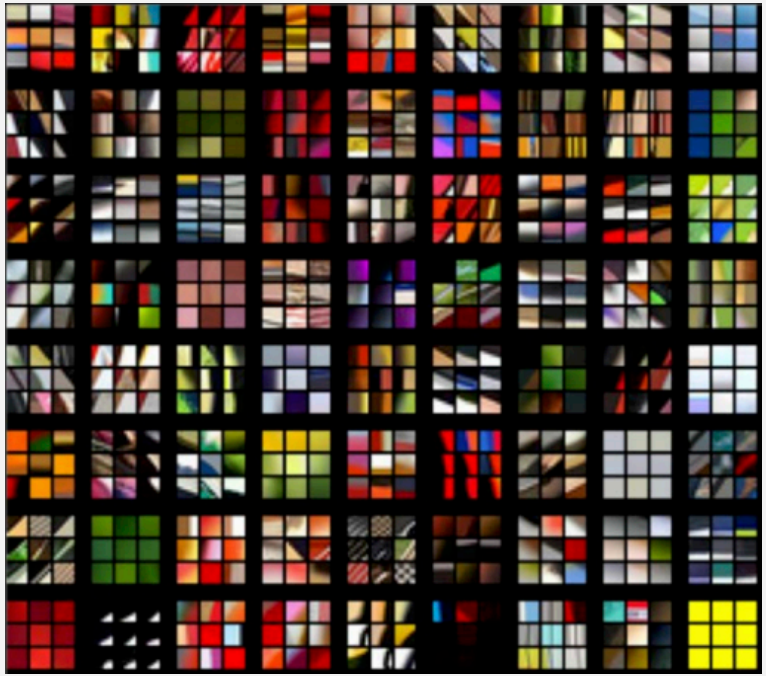
Image Classification

Filter Visualizations of AlexNet Network

- First Layer Filters
- Showing 81 filters of 11x11x3
- Capture low-level features like oriented edges, blobs



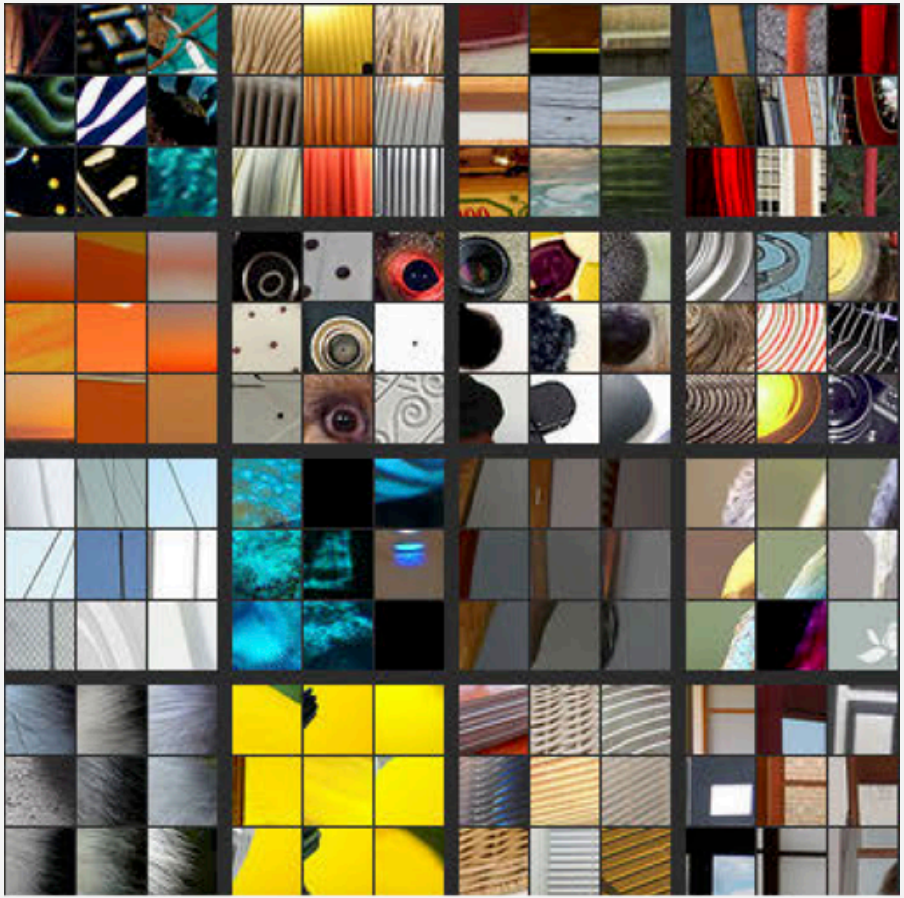
- Top 9 patches that activate each filter in layer 1
- Each 3x3 block shows the top 9 patches for one filter



Visualizing and Understanding Convolutional Networks

Image Classification

Filter Visualizations of AlexNet Network

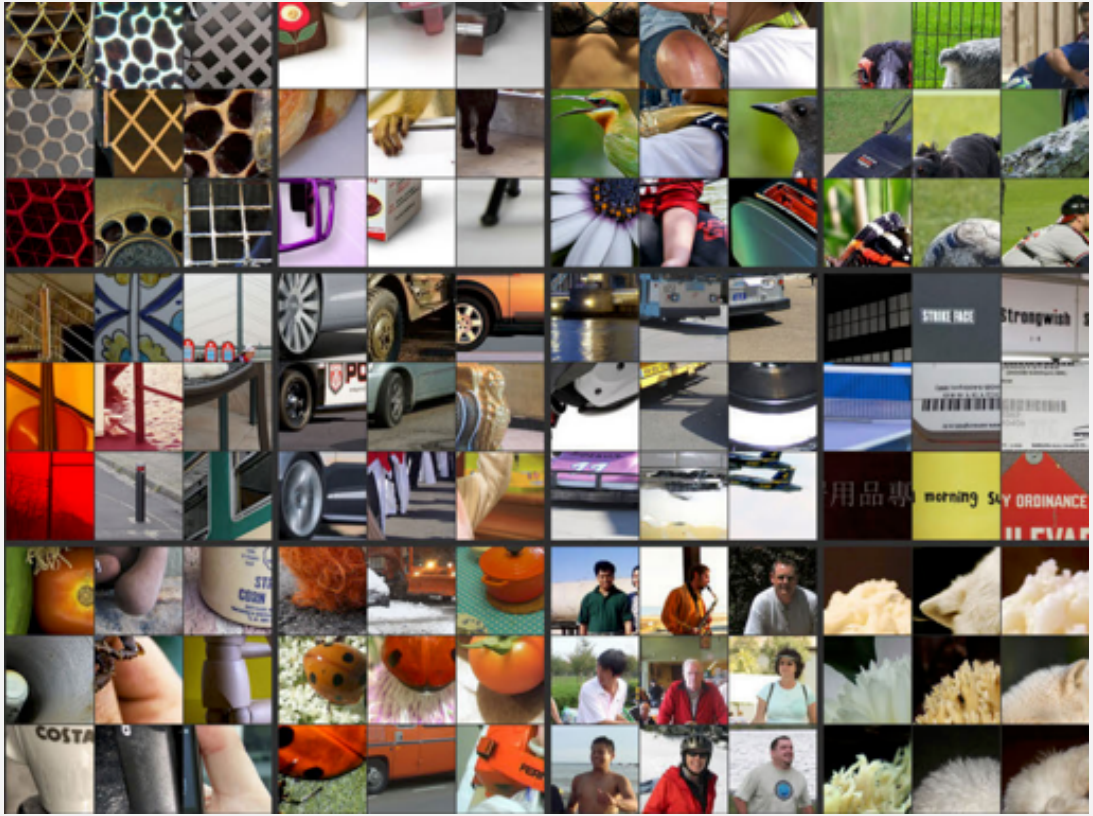


- Top 9 patches that activate each filter in layer 2
- Each 3x3 block shows the top 9 patches for one filter
- Note how the previous low-level features are combined to detect a little more abstract features like textures

Visualizing and Understanding Convolutional Networks

Image Classification

Filter Visualizations of AlexNet Network

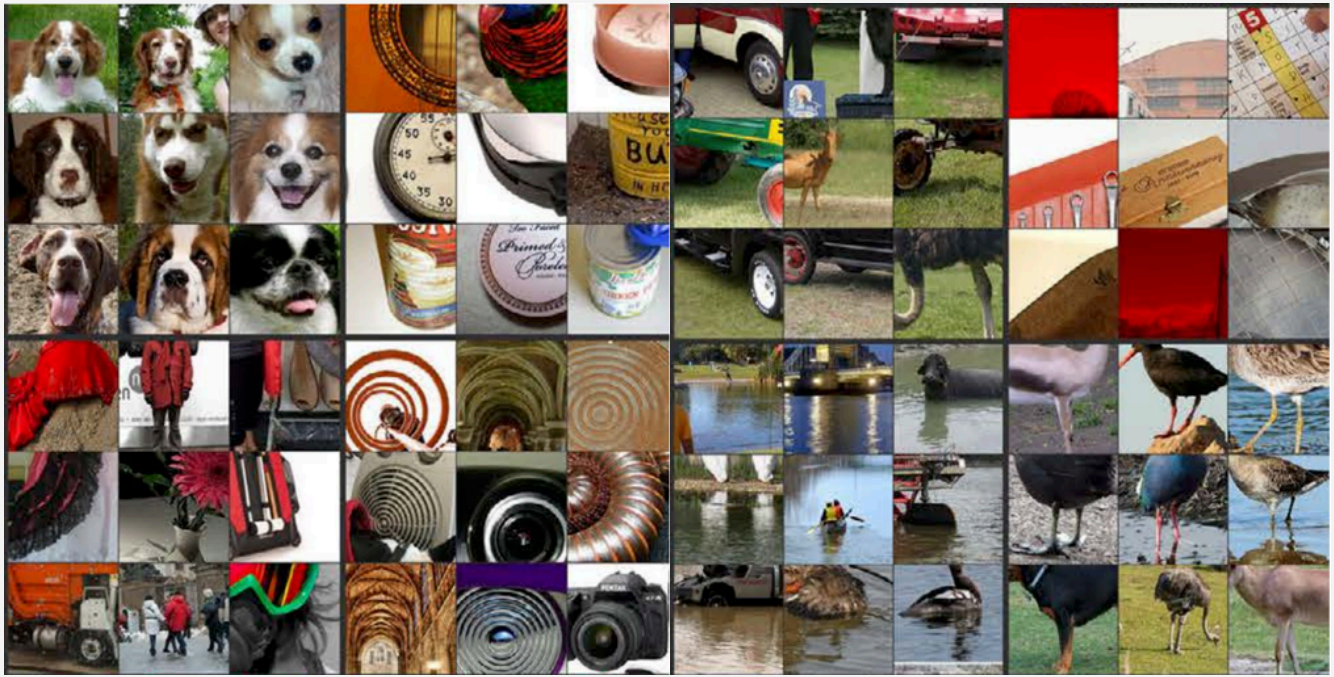


- Top 9 patches that activate each filter in layer 3
- Each 3x3 block shows the top 9 patches for one filter
- Every layer learns a feature detector by combining the output of the layer before
- More and more abstract features are learned as we stack layers

Visualizing and Understanding Convolutional Networks

Image Classification

Filter Visualizations of AlexNet Network



- Top 9 patches that activate each filter in layer 4
- Each 3x3 block shows the top 9 patches for one filter
- Every layer learns a feature detector by combining the output of the layer before
- More and more abstract features are learned as we stack layers

Visualizing and Understanding Convolutional Networks

Image Classification

Filter Visualizations of AlexNet Network

- Top 9 patches that activate each filter in layer 5
- Each 3x3 block shows the top 9 patches for one filter
- Every layer learns a feature detector by combining the output of the layer before
- More and more abstract features are learned as we stack layers

Visualizing and Understanding Convolutional Networks

Building a Deep Neural Network for Image Classification with Keras

Implementing a Neural Network

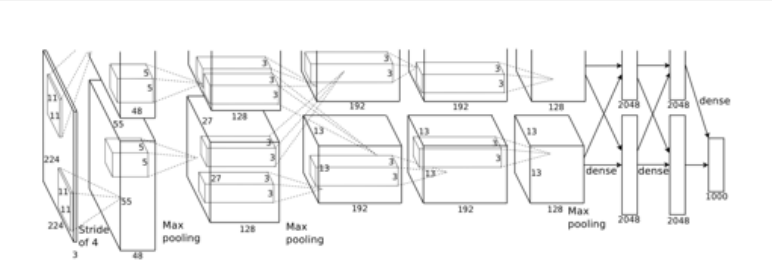


Fig. A sample Convolutional Neural Network for Classification

1. Data Preparation

- Image Resizing
- Image Normalization
- Random data transformation or pre-processing required by the model for specific purposes

2. Model Architecture

- Define layers – convolutional, max-pooling, fully-connected etc.
- Configure each layer – specify parameters like num of filters, stride, activation function etc.

Implementing a Neural Network

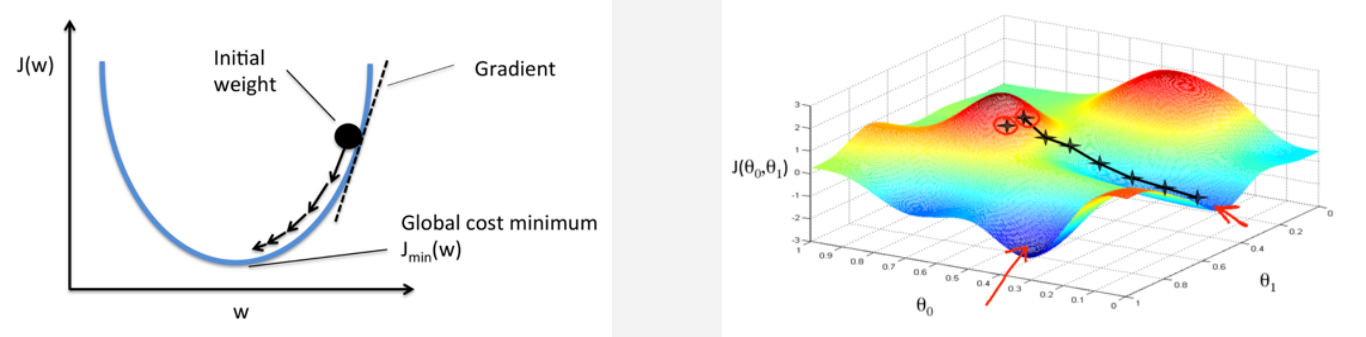


Fig. Minimizing loss function using Gradient Descent at training time. (Illustration) 2-Dimensional (left) v/s 3-Dimensional (right)

3. Model Configuration

- Specify the optimizer – SGD, Adam, RMSProp
- Configure the learning process with batch size, iterations, number of epochs, learning rate etc.
- Specify the Loss function – Cross Entropy, MSE, Hinge etc.

4. Model training and testing

- Train the model using training data
- Optimize for model parameters using validation data
- Evaluate the model using test data

USE-CASE

CLASSIFYING DOG-CAT IMAGES FROM KAGGLE

Building an Image Classifier using Conv Nets

System Requirements

- ✓ Python 3.0+
- ✓ Keras 2.0.0+
- ✓ Numpy
- ✓ Scipy
- ✓ PIL

Outline

We will go over the following training regimes:

- Training a neural network from scratch (for baseline)
- Using the bottleneck features of pre-trained VGG16
- Fine-tuning the top layers of pre-trained VGG16

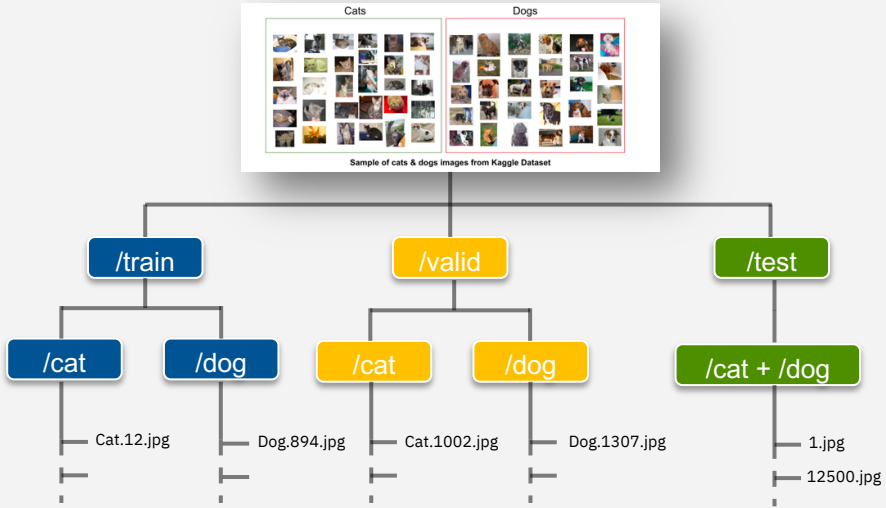
We will learn about following features of Keras

- Sequential and Functional API for model definition
- Fit_generator and fir methods for model training
- ImageDataGenerator for real-time data augmentation
- Layer-freezing and model fine-tuning

Code Repository: https://github.com/jabhinav/Talk-IGDTUW

Training the neural network from scratch – Data Preprocessing

Dataset



- Kaggle Cats and Dogs Dataset for Image Classification
- Number of Samples per class (original) are 12,500
- For our use-case: 1000 samples for training and 200 for validation per class

Data Augmentation and Generation

- For a small dataset, we augment images with random transformations so that model does not see an image twice. This helps the model generalize better
- We will use Keras provided method for data generation (Simple and easy to use)

Fig. Data augmentation example: random transformations of a sample image

Training the neural network from scratch – Data Preprocessing (Code)

Keras module for image preprocessing

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
```

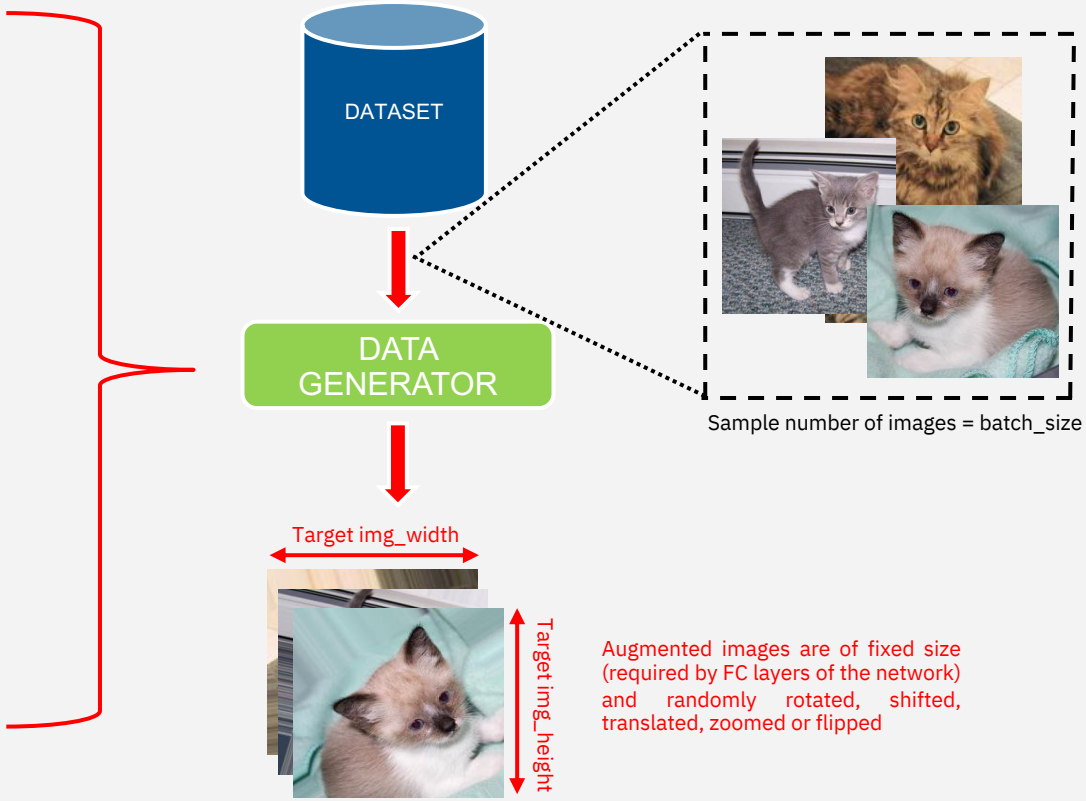
```
datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

Configure random transformations

Src: https://keras.io/preprocessing/image/

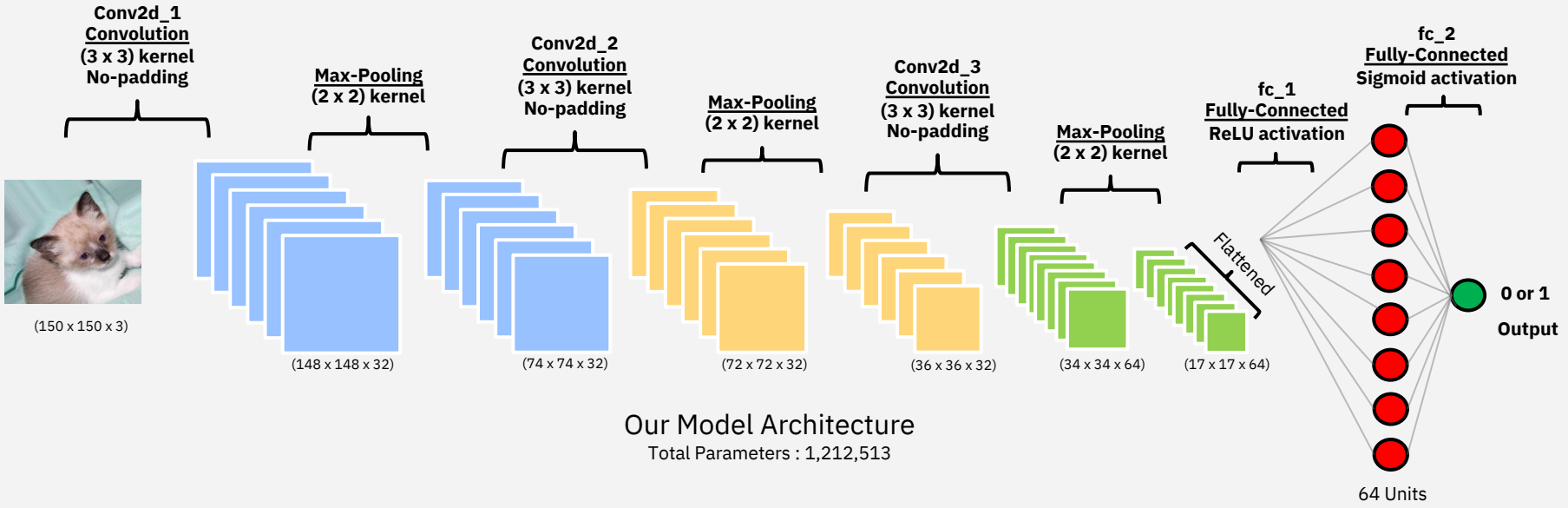
```
train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode='binary')
```

Use 'flow_from_directory' method of 'ImageDataGenerator' to instantiate a data generator that reads pictures from sub-folders of 'data/train' and generates batches of augmented image data



Augmented images are of fixed size (required by FC layers of the network) and randomly rotated, shifted, translated, zoomed or flipped

Training the neural network from scratch – Defining Model



Our Model Architecture

Total Parameters : 1,212,513

Key features of our model:

- It is a small convNet consisting of only 3 convolutional layers.
- **Overfitting**: Occurs when model is exposed to too few examples and and learns patterns that do not generalize to new data
- Data augmentation as one of the strategies to prevent overfitting but it might not work if augmented samples are highly correlated.
- Use of dropout to prevent overfitting by preventing a layer (in our case, fc_1) from seeing twice the exact same pattern

Other ways of controlling our model's entropic capacity:

- Other way is to optimize for number of layers and size of each layer in our network
- We can also use L1 or L2 regularization to force model weights to take smaller values

Training the neural network from scratch – Defining Model (Code)

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

1

2

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

3

For more arguments that Conv Layers can take like padding, initializers, regularizers etc.: https://keras.io/layers/convolutional/

1. We import the Sequential API of Keras to construct our model
2. We then import the following layers:-
   - Conv2D(num_of_filters, kernel_size): Convolutional Layer
   - MaxPooling2D(pool_size): Max Pooling Layer
   - Activation('act_fn'): To define activation function for the preceding layer
   - Flatten(): To collapse the preceding 17 x 17 x 64 feature map into one-dimensional 18496 x 1 feature vector required by the subsequent fully-connected layers.
   - Dense(num_units): Fully-connected layer
   - Dropout(drop_prob): To randomly drop activations of the preceding layer
3. We use sigmoid function which is perfect for binary classification. It evaluates to a probabilistic value that determines how close model prediction is to the ground truth label.

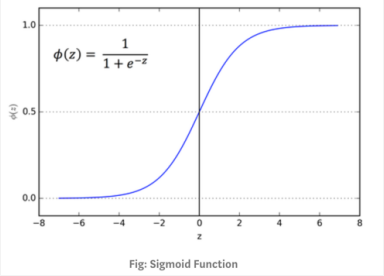


Fig: Sigmoid Function

Training the neural network from scratch – Model Optimization and Training

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

1

```
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

2

$$-(y \log(p) + (1 - y) \log(1 - p))$$

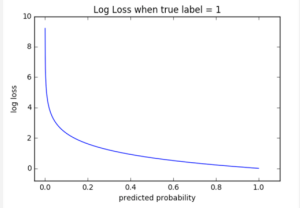


Fig. Cross-entropy Loss equation (left) and plot (right)

```
model.fit_generator(
        train_generator,
        steps_per_epoch=nb_train_samples // batch_size,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=nb_validation_samples // batch_size)
```

3

1. We use Keras in-built optimizers like 'Stochastic Gradient Descent, *sgd*', 'Root Mean-Square Propagation, *rmsprop*', *'adagrad'* etc.
2. Before training the model, we have to compile (or compile) the model.
   - *Loss*: Objective function that needs to be optimizer. Here, we minimize the binary cross entropy loss. It measures the performance of a classification model whose output is a probability between 0 and 1
   - *Optimizer*: Each optimizer can be declared separately along with its parameters. But for now, we use default values and pass the optimizer as an argument to *compile* method.
   - *Metrics*: List of metrics to be evaluated by the model during training and testing. In our case, we want to measure the *accuracy* of the model while predicting labels (*0 for cat or 1 for dog*).
3. Finally, we use *fit_generator* method to train the model on data generated batch-by-batch. It accepts following arguments.
   - *Training data generator:* The generator instance that we created using *'ImageDataGenerator'* and *'flow_from_directory'*
   - *Num of epochs:* Number of iterations where each iteration runs over entire data.
   - *Steps per epoch:* In how many batches do we want to cover the entire training data. This is set to number of samples divided by batch size.
   - *Validation data:* Generator instance for validation data
   - *Validation Steps:* In how many batches do we want to cover the entire validation data.

Training the neural network from scratch – Model Optimization and Training

1
```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

2
```
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

$$-(y \log(p) + (1 - y) \log(1 - p))$$

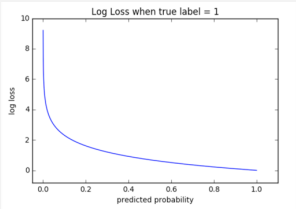


Fig. Cross-entropy Loss equation (left) and plot (right)

3
```
model.fit_generator(
        train_generator,
        steps_per_epoch=nb_train_samples // batch_size,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=nb_validation_samples // batch_size)
```

Baseline Results:-
Validation Accuracy of 74-78% after 50 epochs

Can we achieve better accuracy in shorter time?

Using the bottleneck Features of a Pre-Trained Network - Overview

- In computer vision applications, we can achieve much ***better accuracy*** by leveraging a pre-trained network on a larger dataset.
- In the following part, we will use the VGG16, a network pre-trained on ***1000 classes*** of ***ImageNet Dataset*** (includes multiple 'cat' and 'dog' classes)

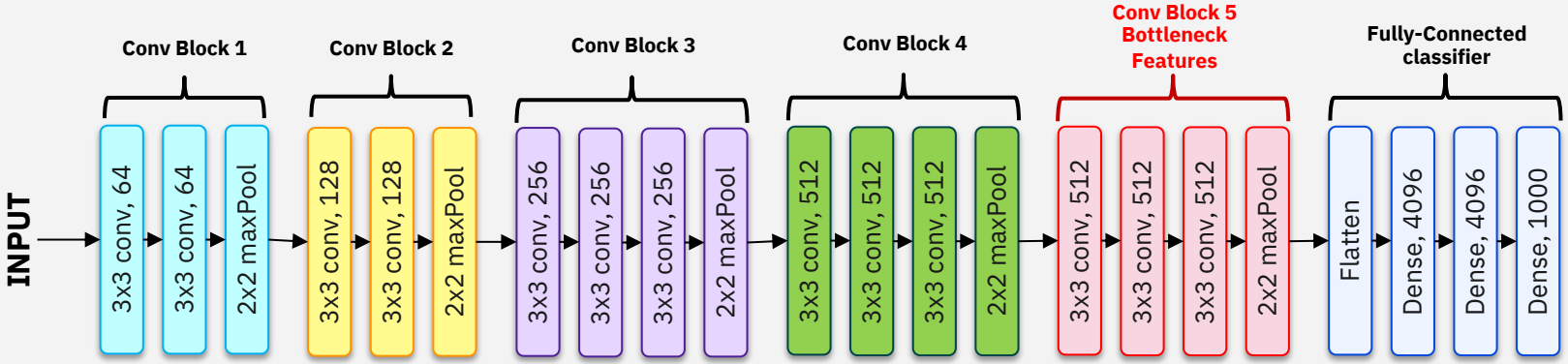


Fig. VGG16 architecture for 1000 class image classification

How do we adapt a pre-trained network for our use case?

- We will use the bottleneck features, which are features from the last activation maps (Conv Block 5) before the fully-connected layers
- Record the features for both training and validation data
- Train a fully-connected model on top of the stored features

Using the bottleneck Features of a Pre-Trained Network - Code

1
```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

2
```
# build the VGG16 network
model = applications.VGG16(include_top=False, weights='imagenet')
```

3
```
generator = datagen.flow_from_directory(
train_data_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode=None,
shuffle=False)

bottleneck_features_train = model.predict_generator(generator, nb_train_samples // batch_size)

np.save('bottleneck_features_train.npy', bottleneck_features_train)
```

4
```
train_data = np.load('bottleneck_features_train.npy')
train_labels = np.array([0] * (nb_train_samples // 2) + [1] * (nb_train_samples // 2))
```

5
```
model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy', metrics=['accuracy'])

model.fit(train_data, train_labels,
          epochs=epochs,
          batch_size=batch_size,
          validation_data=(validation_data, validation_labels))
```

1. Import Keras pre-defined models using '*applications*' module
2. Import in particular, the VGG16 model trained on *'ImageNet'* weights. Make sure to import the model without its top fully-connected layers, set '*include_top*' as **False.** (Note: Weights are downloaded automatically)
3. Instantiate a generator for training and validation data. Use ***'predict_generator'*** to predict Conv_5 features for each sample. Lastly, save the features into a NumPy array
4. Load the bottleneck features back into a NumPy array and set the class labels for each sample.
5. Define the fully-connected classifier (Using Sequential Model API) consisting of two dense layers (256 units and 1 unit respectively). Compile the model and train it (Conv Layers : frozen, FC Layers : trainable) using *'fit'* method.

Note: The 'fit' method is different from 'fit_generator' method. Former requires complete dataset for training while later requires a generator that provides samples batch-by-batch.

Using the bottleneck Features of a Pre-Trained Network - Code

1

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

2

```
# build the VGG16 network
model = applications.VGG16(include_top=False, weights='imagenet')
```

3

```
generator = datagen.flow_from_directory(
train_data_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode=None,
shuffle=False)

bottleneck_features_train = model.predict_generator(generator, nb_train_samples // batch_size)

np.save('bottleneck_features_train.npy', bottleneck_features_train)
```

4

```
train_data = np.load('bottleneck_features_train.npy')
train_labels = np.array([0] * (nb_train_samples // 2) + [1] * (nb_train_samples // 2))
```

5

```
model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy', metrics=['accuracy'])

model.fit(train_data, train_labels,
          epochs=epochs,
          batch_size=batch_size,
          validation_data=(validation_data, validation_labels))
```

Validation Accuracy of ~90% after 50 epochs

Taking a step further – Fine-tuning (Using Functional API of Keras)

With fine-tuning, we can further adapt VGG16 model to the defined task. Fine-tuning retrains an already trained network on a new dataset using 'small' weight updates. For our use-case, we follow following steps:

- Instantiate the convolutional base of VGG16 and load its weights
- Add our previously-defined FC classifier and load its weights
- Freeze the layers of the model up to last conv block; we only fine-tune Conv5 Block of our model

Import the VGG16 model without its top layers and pre-set weights (1)

```
# build the VGG16 network
model_vgg16_conv = applications.VGG16(include_top=False, weights='imagenet', input_shape=input_shape)
```

Build the classifier model and load its weights. (2)

```
input_FC = Input(shape=model_vgg16_conv.output_shape[1:])
x = Flatten()(input_FC)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)
FC_model = Model(input=input_FC, output=x)
FC_model.load_weights(top_model_weights_path)
```

Note that architecture of the defined model should match with the model whose weights we are loading.

(3) Add the FC classifier on top of the convolutional base.

```
input_model = Input(shape=input_shape)
output_vgg16_conv = model_vgg16_conv(input_model)
prediction = FC_model(output_vgg16_conv)
model = Model(input=input_model, output=prediction)
```

(4) Fine-tuning: Freeze the first 25 layers i.e. set them as untrainable

```
for layer in model.layers[:25]:
    layer.trainable = False
```

(5) Compile the model using a non-adaptive optimizer like SGD with a small learning rate.

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

Taking a step further – Fine-tuning (Using Functional API of Keras)

Validation Accuracy of ~95% after 50 epochs at the cost of higher computation time

Import the VGG16 model without its top layers and pre-set weights 1

```
# build the VGG16 network
model_vgg16_conv = applications.VGG16(include_top=False, weights='imagenet', input_shape=input_shape)
```

Build the classifier model and load its weights. 2

```
input_FC = Input(shape=model_vgg16_conv.output_shape[1:])
x = Flatten()(input_FC)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)
FC_model = Model(input=input_FC, output=x)
FC_model.load_weights(top_model_weights_path)
```

Note that architecture of the defined model should match with the model whose weights we are loading.

3 Add the FC classifier on top of the convolutional base.

```
input_model = Input(shape=input_shape)
output_vgg16_conv = model_vgg16_conv(input_model)
prediction = FC_model(output_vgg16_conv)
model = Model(input=input_model, output=prediction)
```

4 Fine-tuning: Freeze the first 25 layers i.e. set them as untrainable

```
for layer in model.layers[:25]:
    layer.trainable = False
```

5 Compile the model using a non-adaptive optimizer like SGD with a small learning rate.

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

Taking a step further – Fine-tuning (Precautions)

In order to not wreck the previously learned features of a pre-trained model, here are some of the common practices followed during fine-tuning:

- **Fine-tuning is to be done for the network with properly trained weights**. We cannot put randomly initialized FC layers on top of a trained Convolutional base because large weight updates for the FC layers will spoil the weights of the Conv Base. That's why we first trained the FC layers using bottleneck features of the Conv base and then only fine-tuned it.

- We keep the first few Conv blocks frozen (learning general features) and only fine-tune the last ones (learning specialized features) in a deep network. This is done to prevent overfitting since a large network has high model entropic capacity and thus a strong tendency to overfit.

- Fine-tuning should be done with a very slow learning rate so as to keep the weight updates small. We use non-adaptive learning rate optimizer like SGD instead of RMSProp.

Exercise

Try getting accuracy above 95% (HINTS!!):

- Try more aggressive dropout. (Try increasing dropout probability)
- Use L1 or L2 regularization. (Regularization in Keras can be done per-layer basis. For details, visit https://keras.io/regularizers/)
- Fine-tune one more convolutional block.

Summary

- Deep Learning = learning hierarchical models.
- Design of Convolution Neural Networks
- Visualization of AlexNet Network
- Implementation of Convolutional Neural Networks in Keras

References

- https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks#activation-function
- http://www.cs.umd.edu/~djacobs/CMSC733/CNN.pdf
- http://cs231n.github.io/convolutional-networks/
- https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
- Visualizing and Understanding Convolutional Networks
- Very Deep Convolutional Networks for Large-Scale Image Recognition
- ImageNet Classification with Deep Convolutional Neural Networks

Thank You