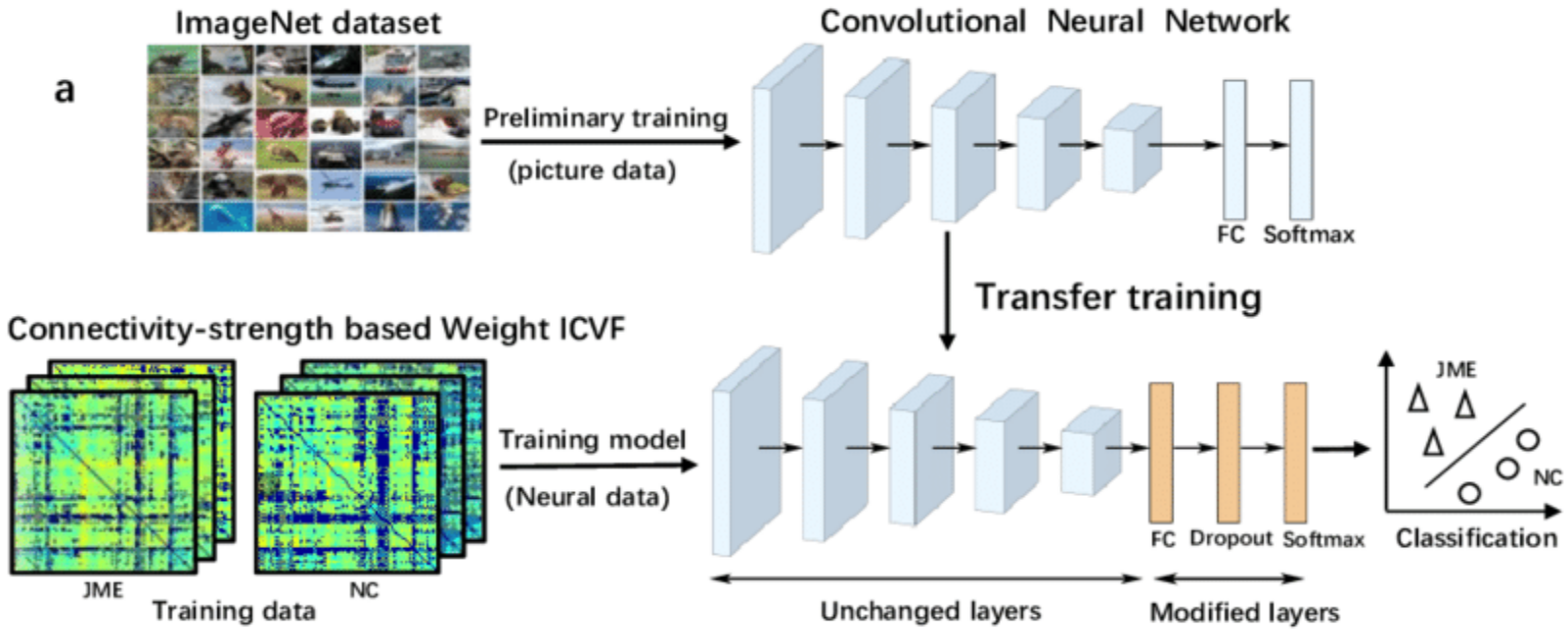
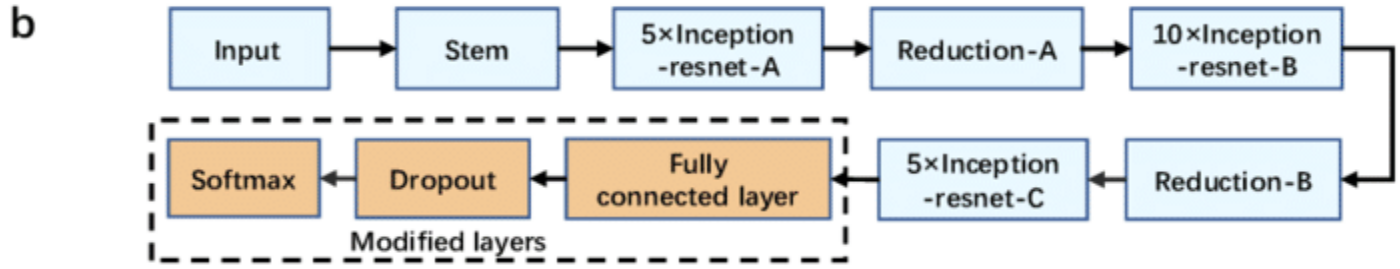
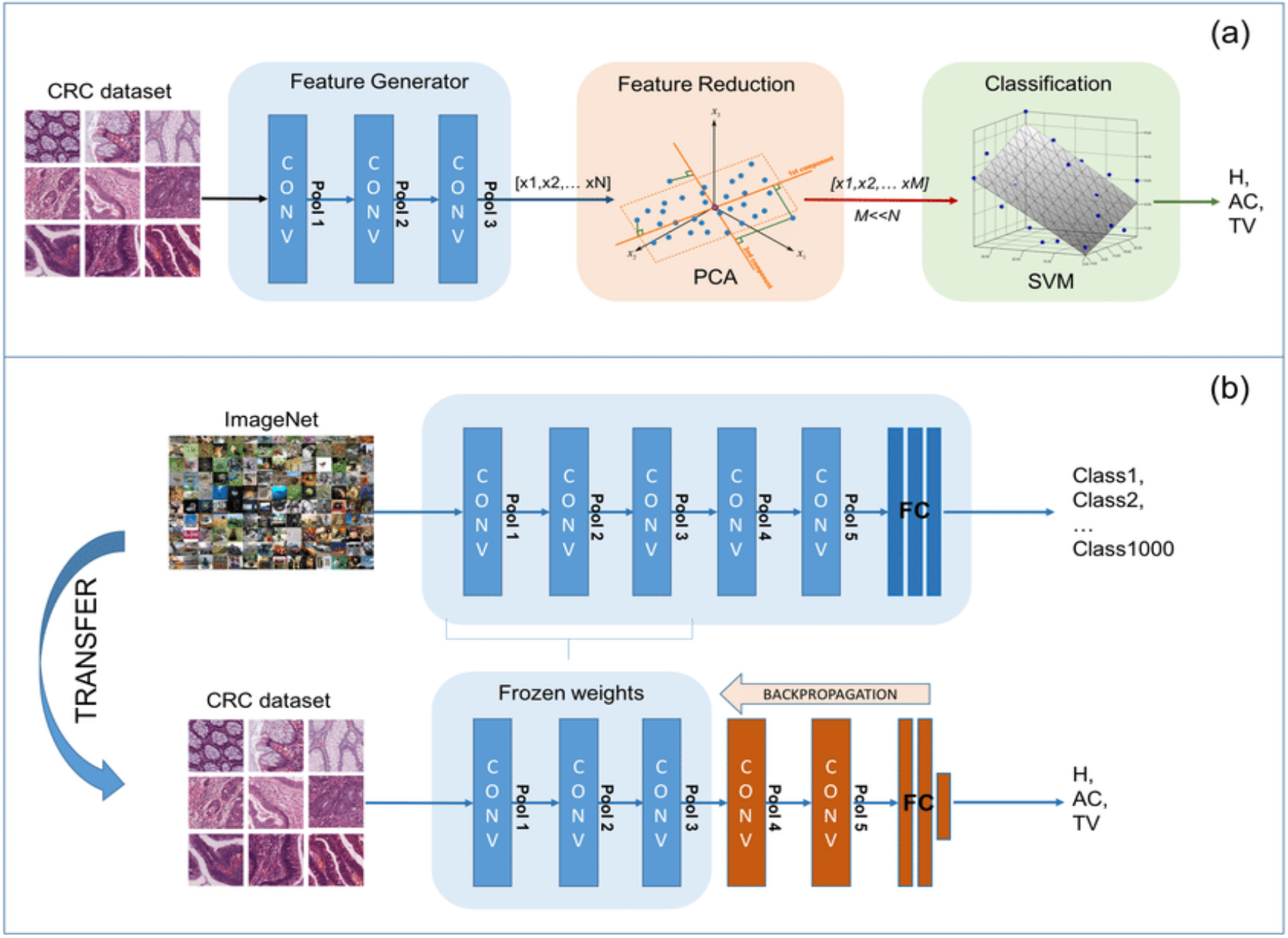
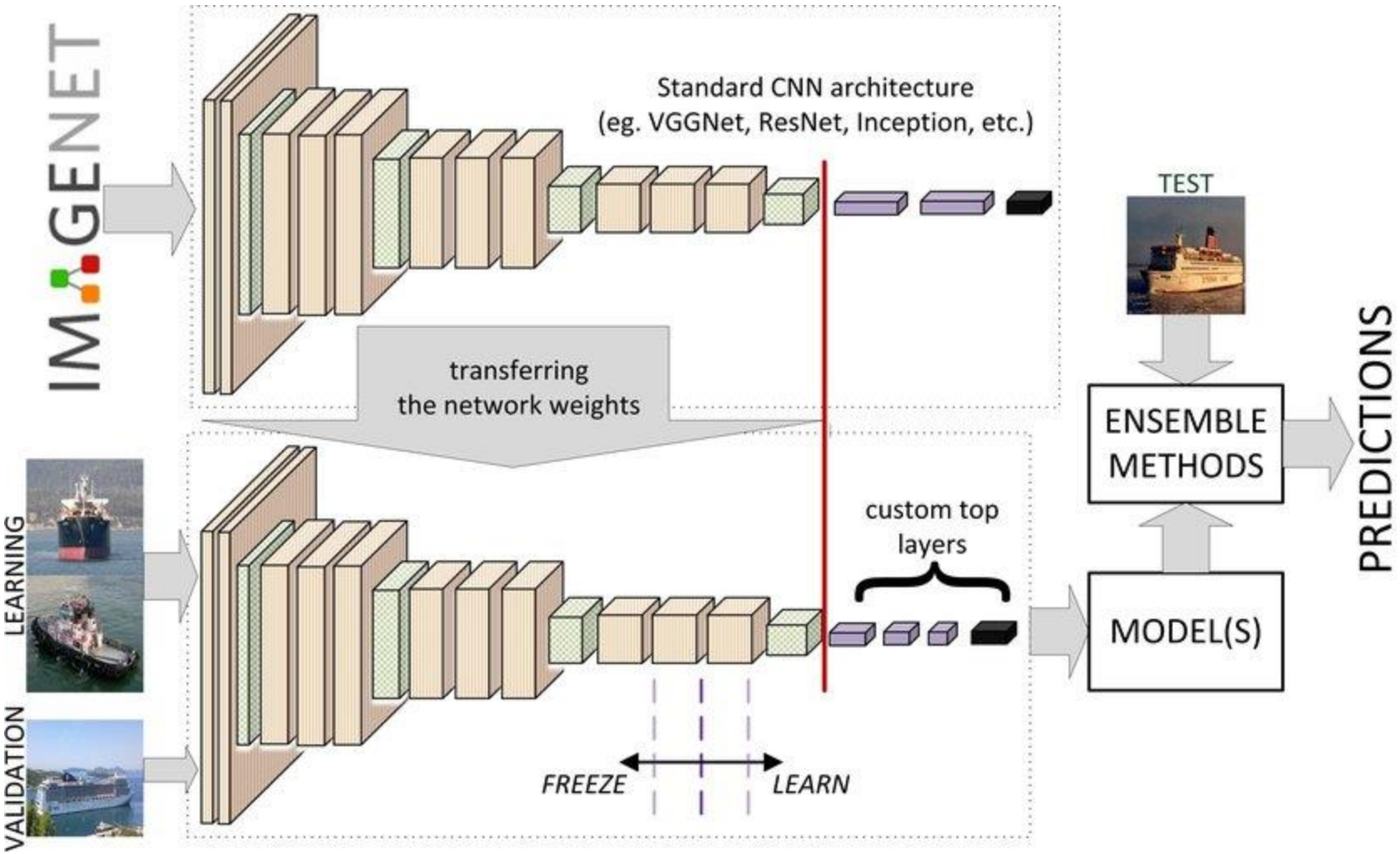
Transfer Learning



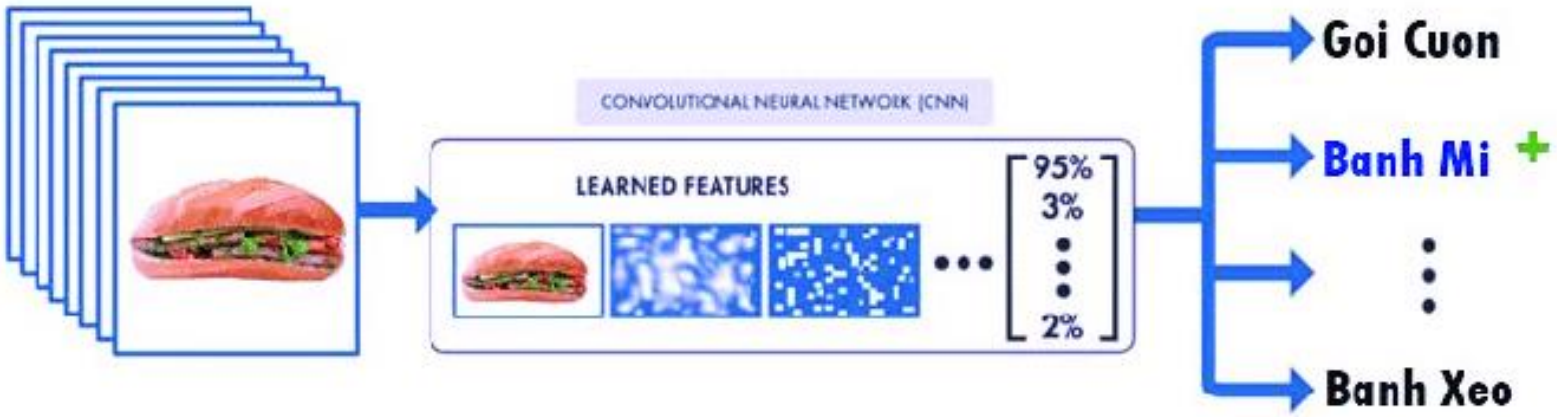



(a)
CRC dataset
Feature Generator
CONV
Pool 1
CONV
Pool 2
CONV
Pool 3
[x1,x2,… xN]
Feature Reduction
PCA
[x1,x2,… xM]
M<<N
Classification
SVM
H,
AC,
TV
(b)
ImageNet
CONV
Pool 1
CONV
Pool 2
CONV
Pool 3
CONV
Pool 4
CONV
Pool 5
FC
Class1,
Class2,
…
Class1000
TRANSFER
CRC dataset
Frozen weights
BACKPROPAGATION
CONV
Pool 1
CONV
Pool 2
CONV
Pool 3
CONV
Pool 4
CONV
Pool 5
FC
H,
AC,
TV

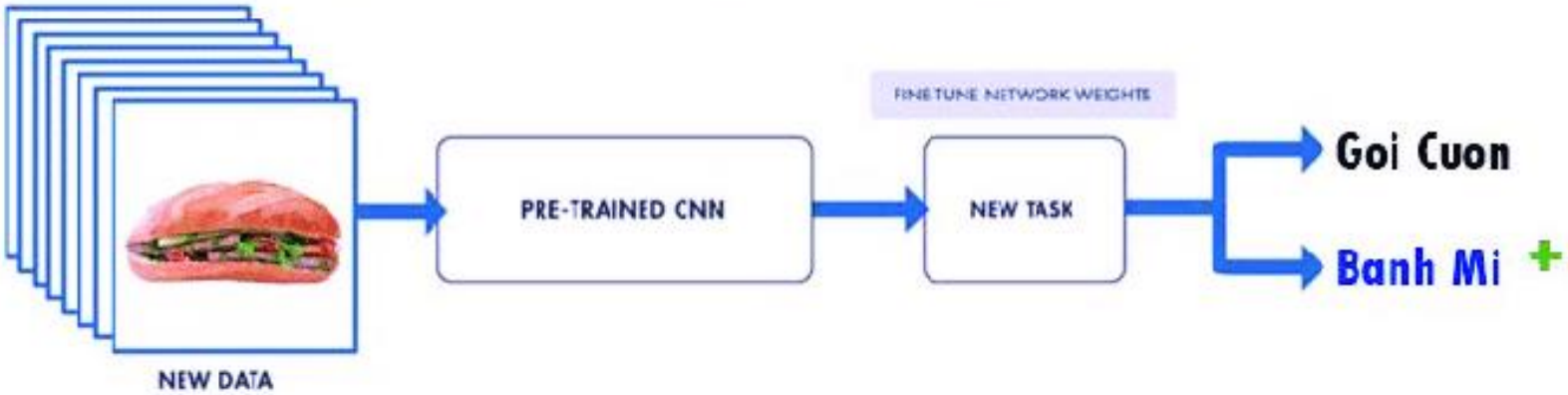


Deep Learning Step by Step
https://t.me/AIMLDeepThaught/712

without Transfer Learning



with Transfer Learning



Transfer learning in a CNN

Transfer learning in a CNN refers to using a pre-trained model on a similar task as a starting point for training a new model on a different task. Transfer learning involves using models trained on one problem as a starting point on a related problem. It is flexible, allowing the use of pre-trained models directly, as feature extraction preprocessing, and integrated into entirely new models

Observe results before and after applying Transfer Learning.

Transfer learning is a research problem in machine learning & deep learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks.So, in transfer learning your previous learning helps you to understand the new concept or learning. In transfer learning we use pre-trained model & we make some modification on that to make a new model.

So, here we will create a hand writing classifier by MNIST data then we will modify this model to predict a given number is odd or even by the help of transfer learning. Then we will compare them not using transfer learning.

Now first creating a Model which can classify MNIST handwriting:

In [ ]:
```
# Importing necessary libraries
import os
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
plt.style.use("fivethirtyeight")
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

In [ ]:
```
# Loading the data of MNIST handwritten
(X_train_full, y_train_full), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
X_train_full = X_train_full / 255.0
X_test = X_test / 255.0
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

In [ ]:
```
# Creating layer of model
tf.random.set_seed(42)  #For getting similar output (optional)
np.random.seed(42)  #For getting similar output (optional)

LAYERS = [ tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(10, activation="softmax")]


model = tf.keras.models.Sequential(LAYERS)
```

In [ ]:
```
# Compiling the model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(learning_rate=1e-3),
              metrics=["accuracy"])
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 300)               0
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Lets train the model
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid), verbose=2)
```

```
Epoch 1/10
1719/1719 - 3s - loss: 1.5275 - accuracy: 0.5970 - val_loss: 0.9444 - val_accuracy: 0.7
980
Epoch 2/10
1719/1719 - 3s - loss: 0.7465 - accuracy: 0.8287 - val_loss: 0.5868 - val_accuracy: 0.8
596
Epoch 3/10
1719/1719 - 3s - loss: 0.5412 - accuracy: 0.8624 - val_loss: 0.4685 - val_accuracy: 0.8
834
Epoch 4/10
1719/1719 - 3s - loss: 0.4591 - accuracy: 0.8771 - val_loss: 0.4104 - val_accuracy: 0.8
940
Epoch 5/10
1719/1719 - 3s - loss: 0.4142 - accuracy: 0.8869 - val_loss: 0.3758 - val_accuracy: 0.9
006
Epoch 6/10
1719/1719 - 3s - loss: 0.3852 - accuracy: 0.8938 - val_loss: 0.3525 - val_accuracy: 0.9
052
Epoch 7/10
1719/1719 - 3s - loss: 0.3644 - accuracy: 0.8980 - val_loss: 0.3348 - val_accuracy: 0.9
102
Epoch 8/10
1719/1719 - 3s - loss: 0.3485 - accuracy: 0.9021 - val_loss: 0.3209 - val_accuracy: 0.9
138
Epoch 9/10
1719/1719 - 3s - loss: 0.3356 - accuracy: 0.9053 - val_loss: 0.3111 - val_accuracy: 0.9
152
Epoch 10/10
1719/1719 - 3s - loss: 0.3251 - accuracy: 0.9077 - val_loss: 0.3016 - val_accuracy: 0.9
170
```

In [ ]:
```
# Saving the model
model.save("pretrained_mnist_model.h5")
```

Now Lets create a model which can predict a given number is odd or even without having Transfer learning technique.

In [ ]:
```
# Making the label as an even or odd category from numbers where even is 1 and odd is 0

def update_even_odd_labels(labels):
  for idx, label in enumerate(labels):
    labels[idx] = np.where(label % 2 == 0, 1, 0)
  return labels
```

In [ ]:
```
y_train_bin, y_test_bin, y_valid_bin = update_even_odd_labels([y_train, y_test, y_valid]
```

In [ ]:
```
# Creating layer of model
tf.random.set_seed(42)  #For getting similar output (optional)
np.random.seed(42)  #For getting similar output (optional)

LAYERS = [ tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(2, activation="softmax")]  # Here I have just used 2 output la


model_1 = tf.keras.models.Sequential(LAYERS)
```

In [ ]:
```
model_1.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_10 (Dense)             (None, 300)               235500
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 300)               0
_________________________________________________________________
dense_11 (Dense)             (None, 100)               30100
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_12 (Dense)             (None, 2)                 202
=================================================================
Total params: 265,802
Trainable params: 265,802
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Compiling new model
model_1.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/optimizer_v2/optimizer_v
2.py:375: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```

In [ ]:
```
# now training & calculating the training time.

# starting time
start = time.time()

history = model_1.fit(X_train, y_train_bin, epochs=10,
                      validation_data=(X_valid, y_valid_bin), verbose=2)

#ending time
end = time.time()

# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Epoch 1/10
1719/1719 - 3s - loss: 0.4357 - accuracy: 0.8088 - val_loss: 0.3279 - val_accuracy: 0.8
664
Epoch 2/10
1719/1719 - 3s - loss: 0.3124 - accuracy: 0.8699 - val_loss: 0.2763 - val_accuracy: 0.8
900
Epoch 3/10
1719/1719 - 3s - loss: 0.2767 - accuracy: 0.8885 - val_loss: 0.2474 - val_accuracy: 0.9
046
Epoch 4/10
1719/1719 - 3s - loss: 0.2530 - accuracy: 0.9005 - val_loss: 0.2262 - val_accuracy: 0.9
160
Epoch 5/10
1719/1719 - 3s - loss: 0.2333 - accuracy: 0.9101 - val_loss: 0.2083 - val_accuracy: 0.9
228
Epoch 6/10
1719/1719 - 3s - loss: 0.2167 - accuracy: 0.9183 - val_loss: 0.1927 - val_accuracy: 0.9
300
Epoch 7/10
1719/1719 - 3s - loss: 0.2018 - accuracy: 0.9252 - val_loss: 0.1795 - val_accuracy: 0.9
364
Epoch 8/10
1719/1719 - 3s - loss: 0.1888 - accuracy: 0.9319 - val_loss: 0.1676 - val_accuracy: 0.9
434
Epoch 9/10
1719/1719 - 3s - loss: 0.1774 - accuracy: 0.9375 - val_loss: 0.1581 - val_accuracy: 0.9
458
Epoch 10/10
1719/1719 - 3s - loss: 0.1675 - accuracy: 0.9409 - val_loss: 0.1494 - val_accuracy: 0.9
506
Runtime of the program is 41.243441104888916
```

Conclusion:

Runtime of the program is 41.24 sec & val_accuracy: 0.9506

Now Let's create the same model which can predict a given number is odd or even with having Transfer learning technique.

In [ ]:
```
# Loading pre-trained model
pretrained_mnist_model = tf.keras.models.load_model("pretrained_mnist_model.h5")
```

In [ ]:
```
pretrained_mnist_model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 300)               0
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Checking layers are trainable or not
for layer in pretrained_mnist_model.layers:
  print(f"{layer.name}: {layer.trainable}")
```

```
flatten_2: True
dense_7: True
leaky_re_lu_4: True
dense_8: True
leaky_re_lu_5: True
dense_9: True
```

In [ ]:
```
# Lets make them false or non trainable except last one
for layer in pretrained_mnist_model.layers[:-1]:
  layer.trainable = False
  print(f"{layer.name}: {layer.trainable}")
```

```
flatten_2: False
dense_7: False
leaky_re_lu_4: False
dense_8: False
leaky_re_lu_5: False
```

In [ ]:
```
for layer in pretrained_mnist_model.layers:
  print(f"{layer.name}: {layer.trainable}")
```

```
flatten_2: False
dense_7: False
leaky_re_lu_4: False
dense_8: False
leaky_re_lu_5: False
dense_9: True
```

In [ ]:
```
# Now make a model using that one
lower_pretrained_layers = pretrained_mnist_model.layers[:-1]

new_model = tf.keras.models.Sequential(lower_pretrained_layers)
new_model.add(
    tf.keras.layers.Dense(2, activation="softmax")
)
```

In [ ]:
```
new_model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 300)               0
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_13 (Dense)             (None, 2)                 202
=================================================================
Total params: 265,802
Trainable params: 202
Non-trainable params: 265,600
_________________________________________________________________
```

In [ ]:
```
# Making the label as an even or odd category from numbers where even is 1 and odd is 0

def update_even_odd_labels(labels):
  for idx, label in enumerate(labels):
    labels[idx] = np.where(label % 2 == 0, 1, 0)
  return labels
```

In [ ]:
```
y_train_bin, y_test_bin, y_valid_bin = update_even_odd_labels([y_train, y_test, y_valid]
```

In [ ]:
```
# Compiling new model
new_model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/optimizer_v2/optimizer_v
2.py:375: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```

In [ ]:
```
# now train and calculating the time

# starting time
start = time.time()

history = new_model.fit(X_train, y_train_bin, epochs=10,
                    validation_data=(X_valid, y_valid_bin), verbose=2)

#ending time
end = time.time()
print(f"Runtime of the program is {end - start}")
```

```
Epoch 1/10
1719/1719 - 3s - loss: 0.3898 - accuracy: 0.8288 - val_loss: 0.3247 - val_accuracy: 0.8
676
Epoch 2/10
1719/1719 - 3s - loss: 0.3300 - accuracy: 0.8602 - val_loss: 0.3049 - val_accuracy: 0.8
752
Epoch 3/10
1719/1719 - 3s - loss: 0.3163 - accuracy: 0.8660 - val_loss: 0.2948 - val_accuracy: 0.8
796
Epoch 4/10
1719/1719 - 3s - loss: 0.3083 - accuracy: 0.8701 - val_loss: 0.2884 - val_accuracy: 0.8
832
Epoch 5/10
1719/1719 - 2s - loss: 0.3023 - accuracy: 0.8725 - val_loss: 0.2834 - val_accuracy: 0.8
846
Epoch 6/10
1719/1719 - 2s - loss: 0.2978 - accuracy: 0.8752 - val_loss: 0.2792 - val_accuracy: 0.8
874
Epoch 7/10
1719/1719 - 3s - loss: 0.2939 - accuracy: 0.8772 - val_loss: 0.2758 - val_accuracy: 0.8
872
Epoch 8/10
1719/1719 - 3s - loss: 0.2907 - accuracy: 0.8788 - val_loss: 0.2728 - val_accuracy: 0.8
890
Epoch 9/10
1719/1719 - 3s - loss: 0.2876 - accuracy: 0.8797 - val_loss: 0.2708 - val_accuracy: 0.8
906
Epoch 10/10
1719/1719 - 3s - loss: 0.2851 - accuracy: 0.8817 - val_loss: 0.2678 - val_accuracy: 0.8
930
Runtime of the program is 41.2496874332428
```

Conclusion:

Runtime of the program is 41.24 sec & val_accuracy: 0.8930

Comparison:

Without Transfer learning:

- Runtime of the program is 41.24 sec
- val_accuracy: 0.9506

With Transfer Learning:

- Runtime of the program is 41.24 sec
- val_accuracy: 0.8930

Here we can we have transfer learning output is pretty close to actual accuracy, although we are just training 202 parameters. So, if we increase the epochs then the accuracy would be high. Now it is taking same time but in big problem it may take less time with compare to Without Transfer learning.

Q1:Explain the importance of weight initialization in artificial neural networks. Why is it necessary to initialize the weights carefully?

Answer: Weight initialization is crucial in artificial neural networks because it sets the initial values for the model's parameters (weights). Proper weight initialization ensures that the neural network starts with reasonable initial values, which significantly impacts its training and convergence. If the weights are initialized poorly, the model may struggle to learn effectively, leading to slow convergence or even complete failure to learn.

Q2:Describe the challenges associated with improper weight initialization. How do these issues affect model training and convergence?

Answer: Improper weight initialization can lead to several challenges during model training. If weights are initialized too large or small, the gradients during backpropagation can become vanishingly small or exploding, respectively, resulting in slow convergence or instability. Additionally, improper weight initialization may cause the model to get stuck in local minima, leading to suboptimal solutions. It can also result in slow learning or oscillating behaviors during training, making it difficult to reach a global minimum.

Q3: Discuss the concept of variance and how it relates to weight initialization. Why is it crucial to consider the variance of weights during initialization?

Answer: Variance is a measure of the spread or dispersion of data points. In the context of weight initialization, it refers to the range of values that weights can take. A very high or very low variance can cause gradients to explode or vanish during backpropagation, respectively. Controlling the variance during weight initialization is essential to ensure stable and effective learning. Properly balanced variance helps prevent training issues, allowing the model to learn more efficiently and converge faster.

Part 2: Weight Initialization Techniques

Q1: Explain the concept of zero initialization. Discuss its potential limitations and when it can be appropriate to use.

Answer: Zero initialization involves setting all weights to zero at the beginning of training. However, this approach has limitations as all neurons will have the same output, leading to symmetrical gradients during backpropagation. This means that each neuron will learn the same features, making the learning process ineffective. Zero initialization is generally not recommended for most neural network architectures.

Q2: Describe the process of random initialization. How can random initialization be adjusted to mitigate potential issues like saturation or vanishing/exploding gradients?

Answer: Random initialization sets the weights to random values, often drawn from a normal or uniform distribution. This introduces diversity among neurons, allowing them to learn different features. To mitigate saturation or vanishing/exploding gradients, it is crucial to adjust the scale of random initialization based on the activation function. For example, Xavier/Glorot initialization scales the random weights based on the number of input and output neurons for a layer, which helps balance the variance and improve convergence.

Q3: Discuss the concept of Xavier/Glorot initialization. Explain how it addresses the challenges of improper weight initialization and the underlying theory behind it.

Answer: Xavier/Glorot initialization is a weight initialization technique that sets the initial weights based on the size of the layer's input and output. The goal is to ensure that the variance of the activations and gradients remains consistent across layers. By balancing the variance, it mitigates the vanishing and

exploding gradient problems, promoting stable training and faster convergence. The initialization formula takes into account the number of input and output neurons and follows a specific distribution based on the activation function.

Q4: Explain the concept of He initialization. How does it differ from Xavier initialization, and when is it preferred?

Answer: He initialization is similar to Xavier/Glorot initialization, but it scales the weights based only on the number of input neurons. It is specifically designed for activation functions like ReLU and its variants, which introduce non-linearity and may cause the vanishing gradient problem. He initialization allows ReLU-based activations to maintain a balanced variance during training, making it a preferred choice when using ReLU or similar activation functions.

In [1]:

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.initializers import Zeros, RandomNormal, GlorotUniform, HeNormal

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1, 784) / 255.0
x_test = x_test.reshape(-1, 784) / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Define the neural network architecture
def create_model(initializer):
    model = Sequential()
    model.add(Dense(256, input_shape=(784,), kernel_initializer=initializer))
    model.add(Activation('relu'))
    model.add(Dense(128, kernel_initializer=initializer))
    model.add(Activation('relu'))
    model.add(Dense(10, kernel_initializer=initializer))
    model.add(Activation('softmax'))
    return model

# Define the weight initialization techniques
zero_initializer = Zeros()
random_initializer = RandomNormal(mean=0, stddev=0.01)
xavier_initializer = GlorotUniform()
he_initializer = HeNormal()

# Train and evaluate the models with different initializers
initializers = [zero_initializer, random_initializer, xavier_initializer, he_initializer
for initializer in initializers:
    model = create_model(initializer)
    model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate=0.01), me
    history = model.fit(x_train, y_train, batch_size=32, epochs=10, validation_split=0.1
    test_loss, test_accuracy = model.evaluate(x_test, y_test)
    print(f"\nModel with {initializer} initialization:")
    print(f"Test Loss: {test_loss}, Test Accuracy: {test_accuracy}")
```

```
0.8436 - val_loss: 0.2527 - val_accuracy: 0.9283
Epoch 2/10
1688/1688 [==============================] - 8s 5ms/step - loss: 0.2773 - accuracy:
0.9202 - val_loss: 0.2001 - val_accuracy: 0.9427
Epoch 3/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.2253 - accuracy:
0.9356 - val_loss: 0.1750 - val_accuracy: 0.9505
Epoch 4/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.1925 - accuracy:
0.9451 - val_loss: 0.1538 - val_accuracy: 0.9578
Epoch 5/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.1675 - accuracy:
0.9515 - val_loss: 0.1349 - val_accuracy: 0.9628
Epoch 6/10
1688/1688 [==============================] - 8s 5ms/step - loss: 0.1490 - accuracy:
0.9570 - val_loss: 0.1232 - val_accuracy: 0.9662
Epoch 7/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.1335 - accuracy:
0.9621 - val_loss: 0.1137 - val_accuracy: 0.9693
Epoch 8/10
```

So , HE Normalization Perform better because we use Relu activation function

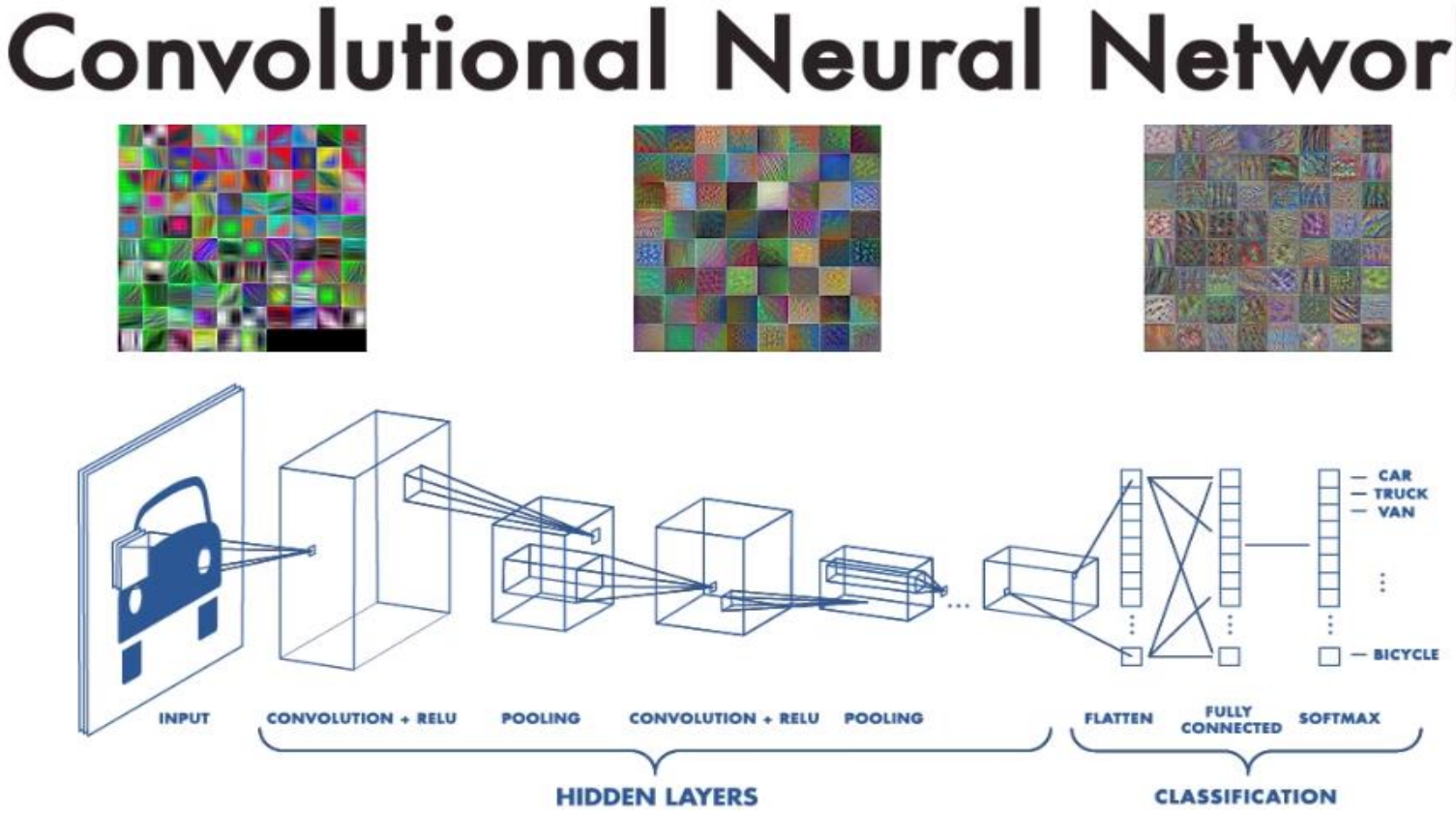
In this code, we define a neural network with three layers (two hidden layers and one output layer) and use different initializers for each layer. We then train the model with each initializer and evaluate their performance.

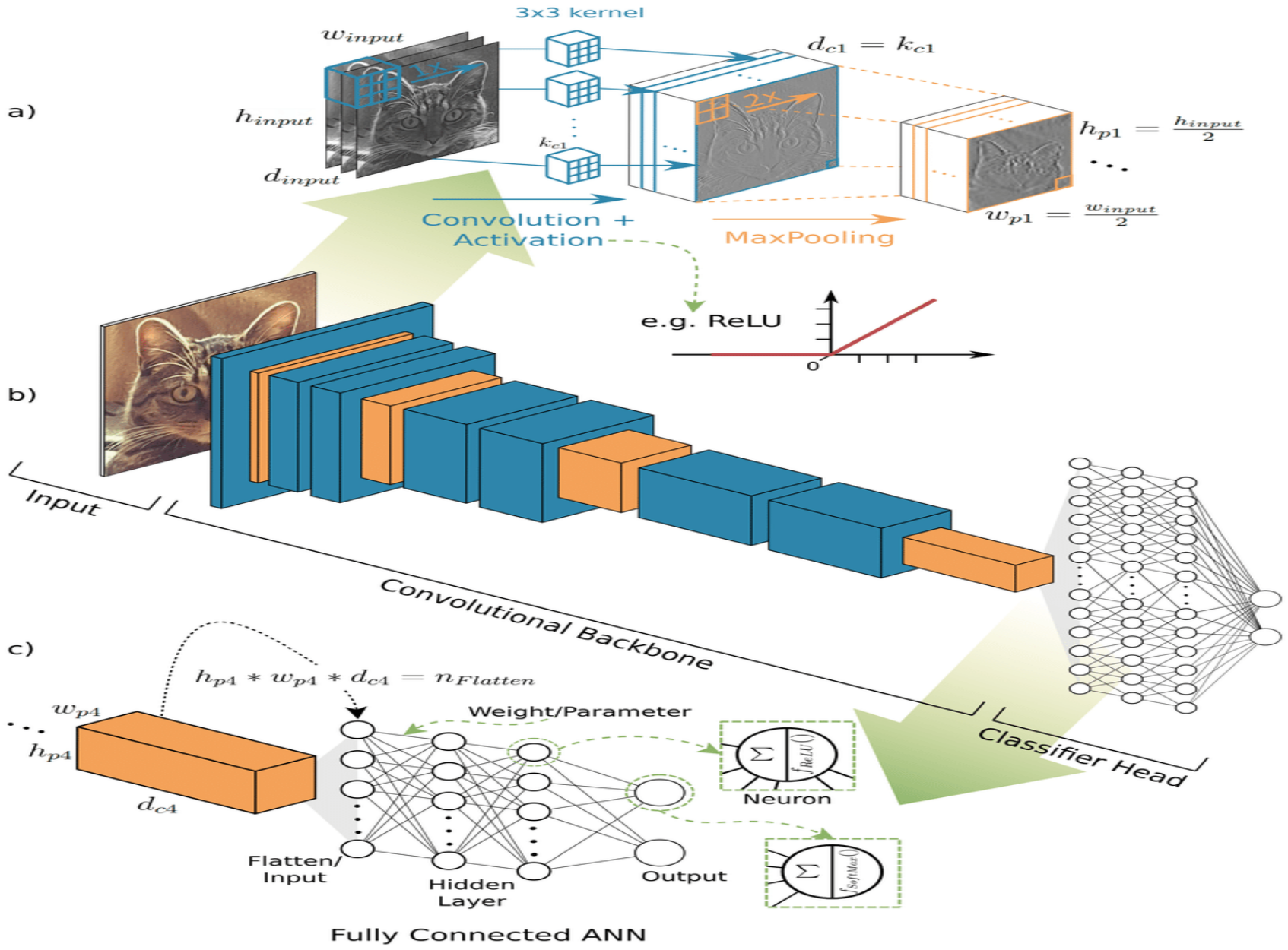
Considerations and tradeoffs when choosing weight initialization techniques:

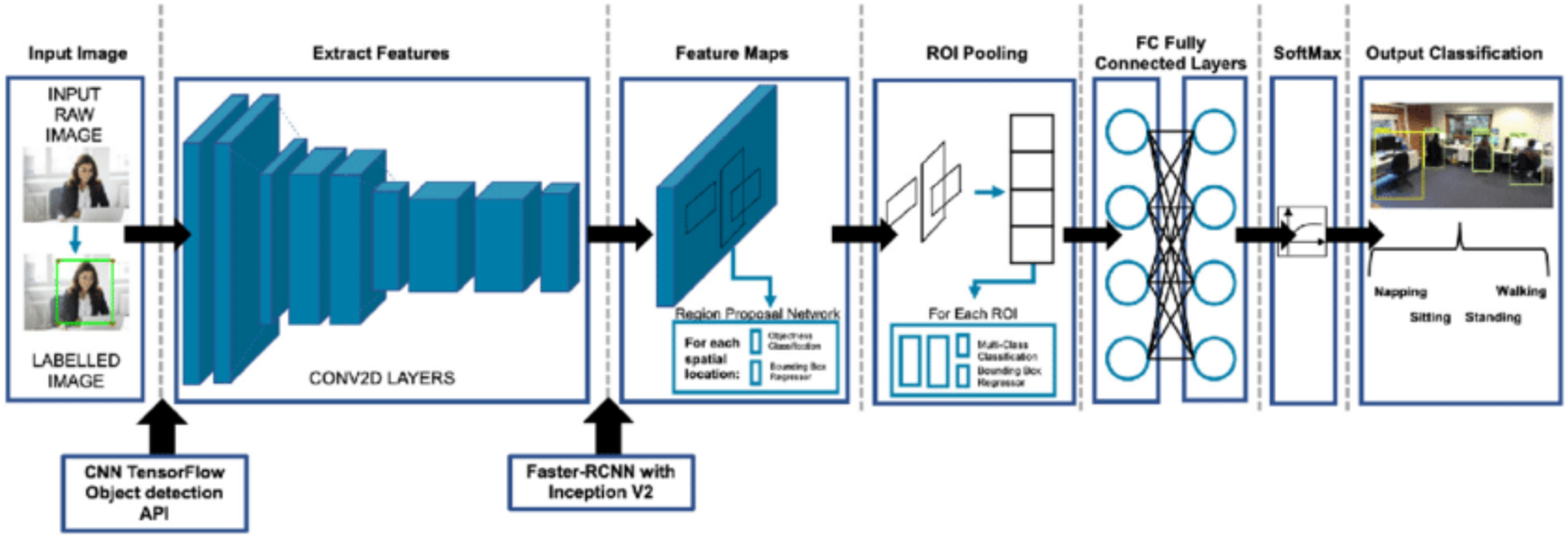
- Activation Function: Different weight initializers work better with specific activation functions. For example, He initialization is suitable for ReLU-based activations, while Xavier initialization works well with tanh or sigmoid activations.
- Layer Size: The number of neurons in each layer can impact the choice of weight initialization. Smaller layers might work well with zero or random initialization, while larger layers might benefit from techniques like Xavier or He initialization.
- Convergence Speed: Proper weight initialization can lead to faster convergence during training. Techniques like Xavier and He initialization are known to speed up convergence.
- Avoiding Vanishing/Exploding Gradients: Weight initialization can help prevent vanishing or exploding gradients, which can affect the stability of training.
- Model Performance: It's essential to evaluate the model's performance on the validation set and test set to choose the best weight initialization technique for a specific task.
- Experimentation: Experimenting with different weight initialization techniques is crucial to find the one that works best for a given neural network architecture and task.

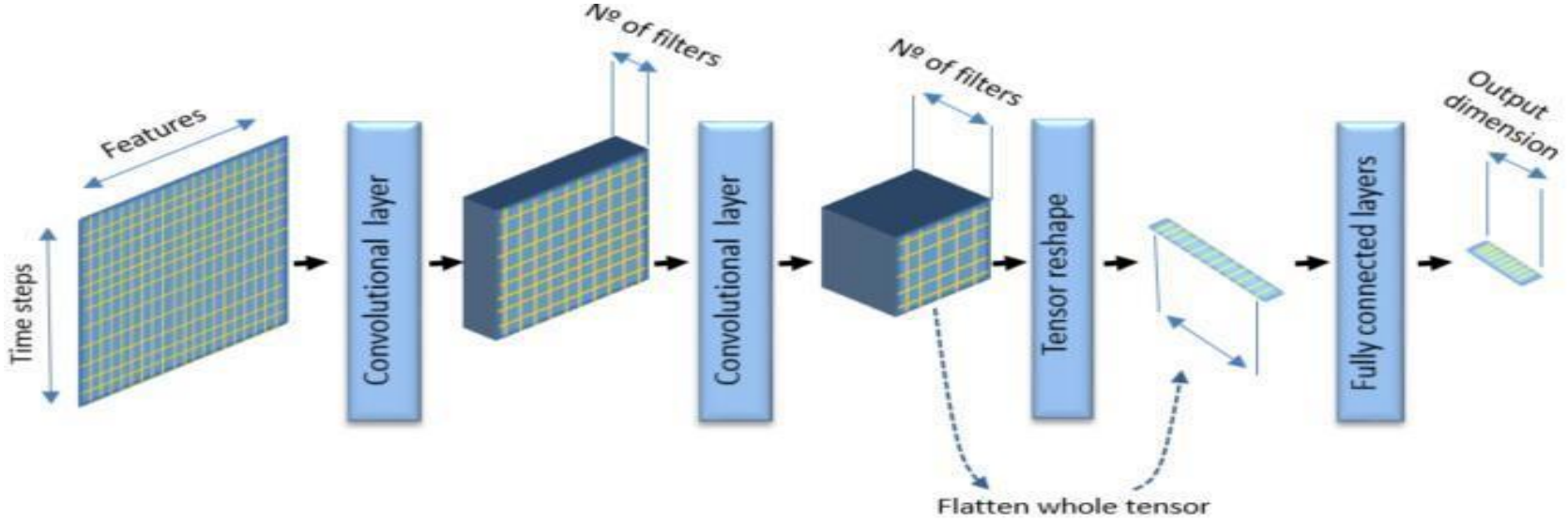
In [ ]:

CNN Net


Convolutional Neural Network
CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
HIDDEN LAYERS
CLASSIFICATION


a)
3x3 kernel
w_{input}
h_{input}
d_{input}
k_{c1}
$d_{c1} = k_{c1}$
$h_{p1} = \frac{h_{input}}{2}$
$w_{p1} = \frac{w_{input}}{2}$
Convolution + Activation
MaxPooling
e.g. ReLU
0
b)
Input
Convolutional Backbone
Classifier Head
c)
$h_{p4} * w_{p4} * d_{c4} = n_{Flatten}$
w_{p4}
h_{p4}
d_{c4}
Weight/Parameter
Neuron
Flatten/ Input
Hidden Layer
Output
Fully Connected ANN


Input Image
INPUT RAW IMAGE
LABELLED IMAGE
Extract Features
CONV2D LAYERS
Feature Maps
Region Proposal Network
For each spatial location:
ROI Pooling
For Each ROI
FC Fully Connected Layers
SoftMax
Output Classification
Napping
Walking
Sitting
Standing
CNN TensorFlow Object detection API
Faster-RCNN with Inception V2


Features
Time steps
Convolutional layer
№ of filters
Convolutional layer
№ of filters
Tensor reshape
Flatten whole tensor
Fully connected layers
Output dimension

Till now in MLP

In [1]:
```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
# Imports a Flatten layer to convert the image matrix into a vector
# Defines the neural network architecture
model.add( Flatten(input_shape = (28,28) ))
model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 10)                5130

=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

- Params after the flatten layer = 0, because this layer only flattens the image to a vector for feeding into the input layer. The weights haven't been added yet.
- Params after layer 1 = (784 nodes in input layer) × (512 in hidden layer 1) + (512 connections to biases) = 401,920.
- Params after layer 2 = (512 nodes in hidden layer 1) × (512 in hidden layer 2) + (512 connections to biases) = 262,656.
- Params after layer 3= (512 nodes in hidden layer 2) × (10 in output layer) + (10 connections to biases) = 5,130.
- Total params in the network = 401,920 + 262,656 + 5,130 = 669,706.

Why Convolutions?

SPATIAL INVARIANCE or LOSS IN FEATURES

The spatial features of a 2D image are lost when it is flattened to a 1D vector input. Before feeding an image to the hidden layers of an MLP, we must flatten the image matrix to a 1D vector, as we saw in the mini project. This implies that all of the image's 2D information is discarded.

In [ ]:
```
### To Install the library on your system
```

In [4]:
```
!pip install opencv-python
```

```
Collecting opencv-python
  Downloading opencv_python-4.8.0.74-cp37-abi3-win_amd64.whl (38.1 MB)
     ---------------------------------------- 38.1/38.1 MB 1.3 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17.0 in d:\anaconda setup\lib\site-packages (fr
om opencv-python) (1.23.5)
Installing collected packages: opencv-python
Successfully installed opencv-python-4.8.0.74
```

In [9]:
```
import os

# Check the current working directory
print(os.getcwd())

# Check if the file exists
file_path = 'input.jfif'
print(os.path.exists(file_path))
```

```
D:\JupyterNotebook\PW_Skills_Data_Science\Deep Learning\CV-Content-20230803T131230Z-001
\CV-Content\01. CNN Foundation
True
```

In [10]:
```
import os, json, cv2, random
import matplotlib.pyplot as plt

# Your code...

img = cv2.imread('input.jfif')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
plt.axis('off')
plt.show()
```

Image Matrix

In [12]:
```
img
```

Out[12]:
```
array([[[201, 194, 201],
        [197, 190, 197],
        [192, 185, 192],
        ...,
        [207, 198, 203],
        [207, 198, 203],
        [207, 198, 203]],

       [[199, 192, 199],
        [197, 190, 197],
        [194, 187, 194],
        ...,
        [206, 197, 202],
        [206, 197, 202],
        [206, 197, 202]],

       [[201, 194, 201],
        [201, 194, 201],
        [201, 194, 201],
        ...,
        [206, 197, 202],
        [206, 197, 202],
        [206, 197, 202]],

       ...,

       [[177, 150, 141],
        [190, 163, 154],
        [178, 149, 141],
        ...,
        [230, 201, 187],
        [227, 198, 184],
        [224, 195, 181]],

       [[204, 175, 167],
        [212, 183, 175],
        [224, 194, 186],
        ...,
        [239, 211, 199],
        [237, 209, 197],
        [234, 206, 194]],

       [[248, 219, 211],
        [228, 199, 191],
        [234, 204, 196],
        ...,
        [244, 216, 204],
        [243, 215, 203],
        [241, 213, 201]]], dtype=uint8)
```

Sample Image

0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

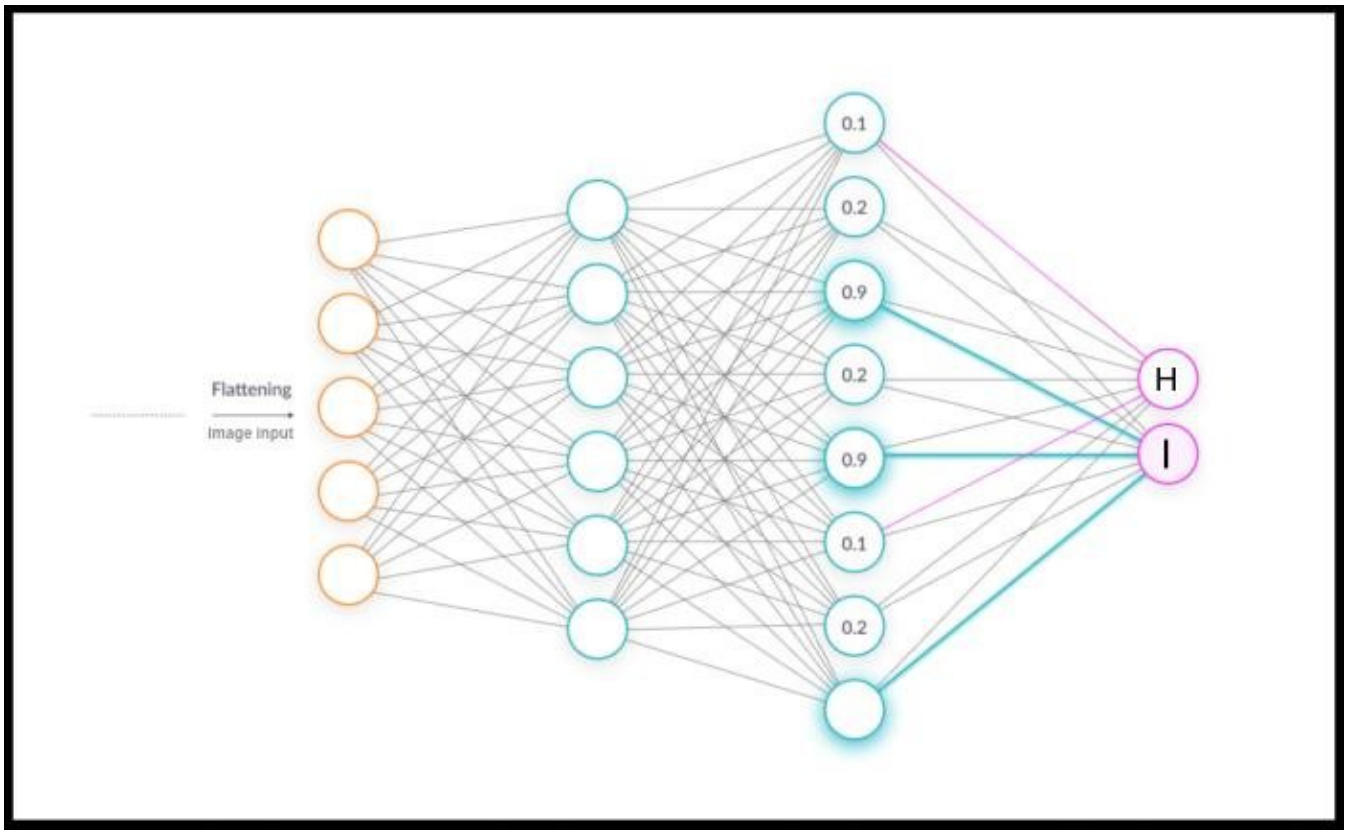
Increase in Parameter Issue

While increase in Parameter Issue is not a big problem for the MNIST dataset because the images are really small in size (28 × 28), what happens when we try to process larger images?

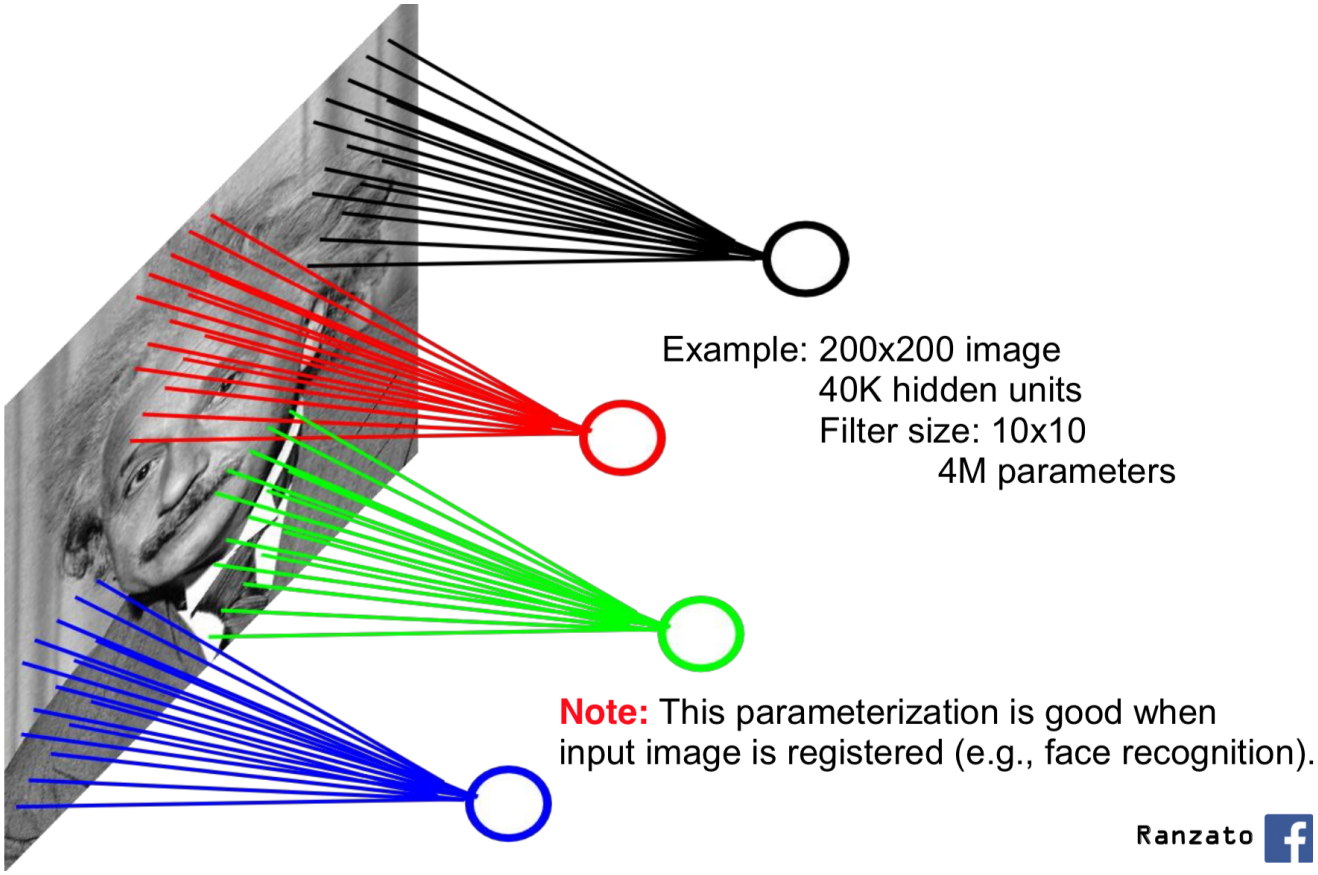
For example, if we have an image with dimensions 1,000 × 1,000, it will yield 1 million parameters for each node in the first hidden layer.

- So if the first hidden layer has 1,000 neurons, this will yield 1 billion parameters even in such a small network. You can imagine the computational complexity of optimizing 1 billion parameters after only the first layer.

Fully Connected Neural Net



Local Connected Neural Net



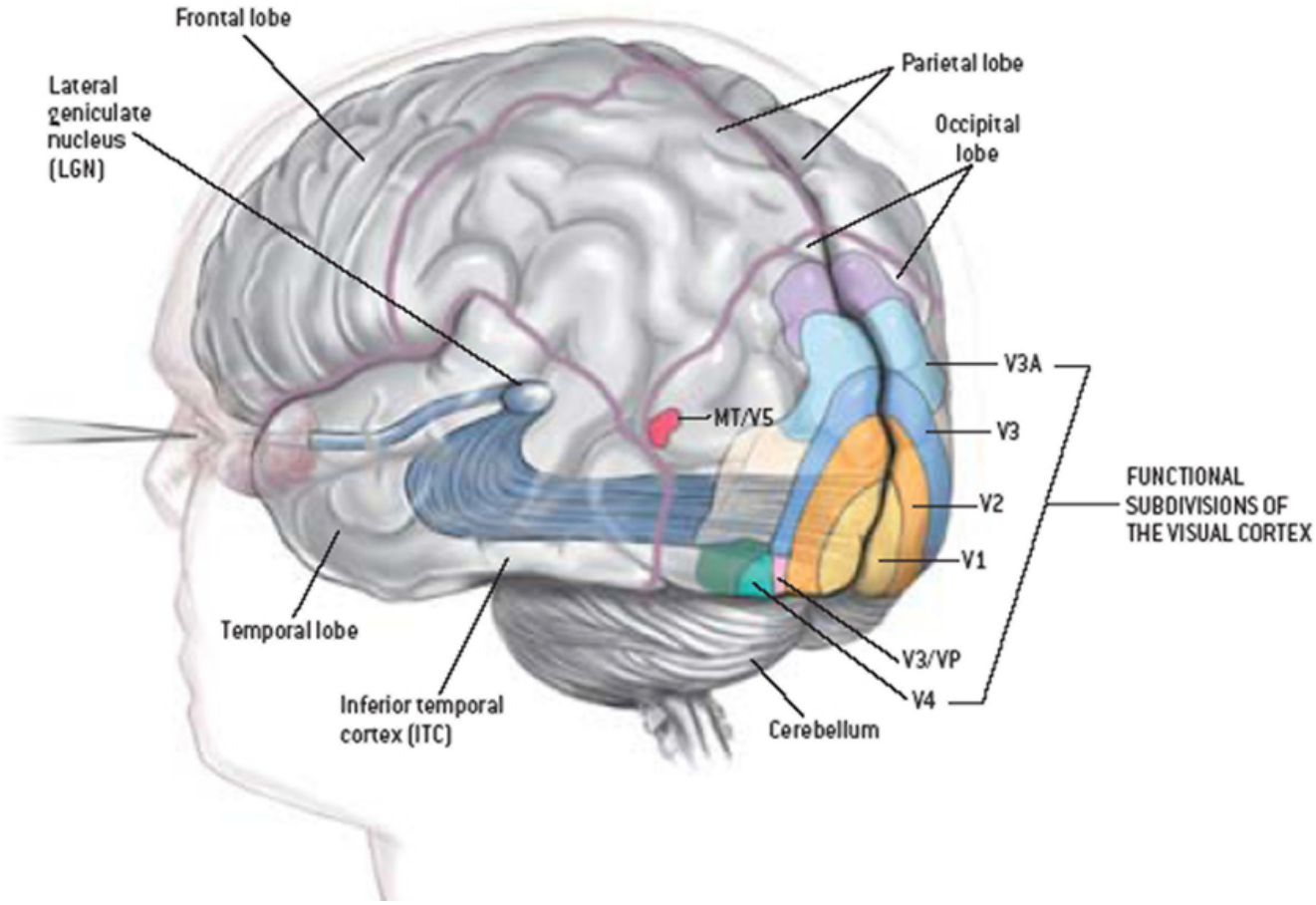
Source (https://www.cs.toronto.edu)

Guide for design of a neural network architecture suitable for computer vision

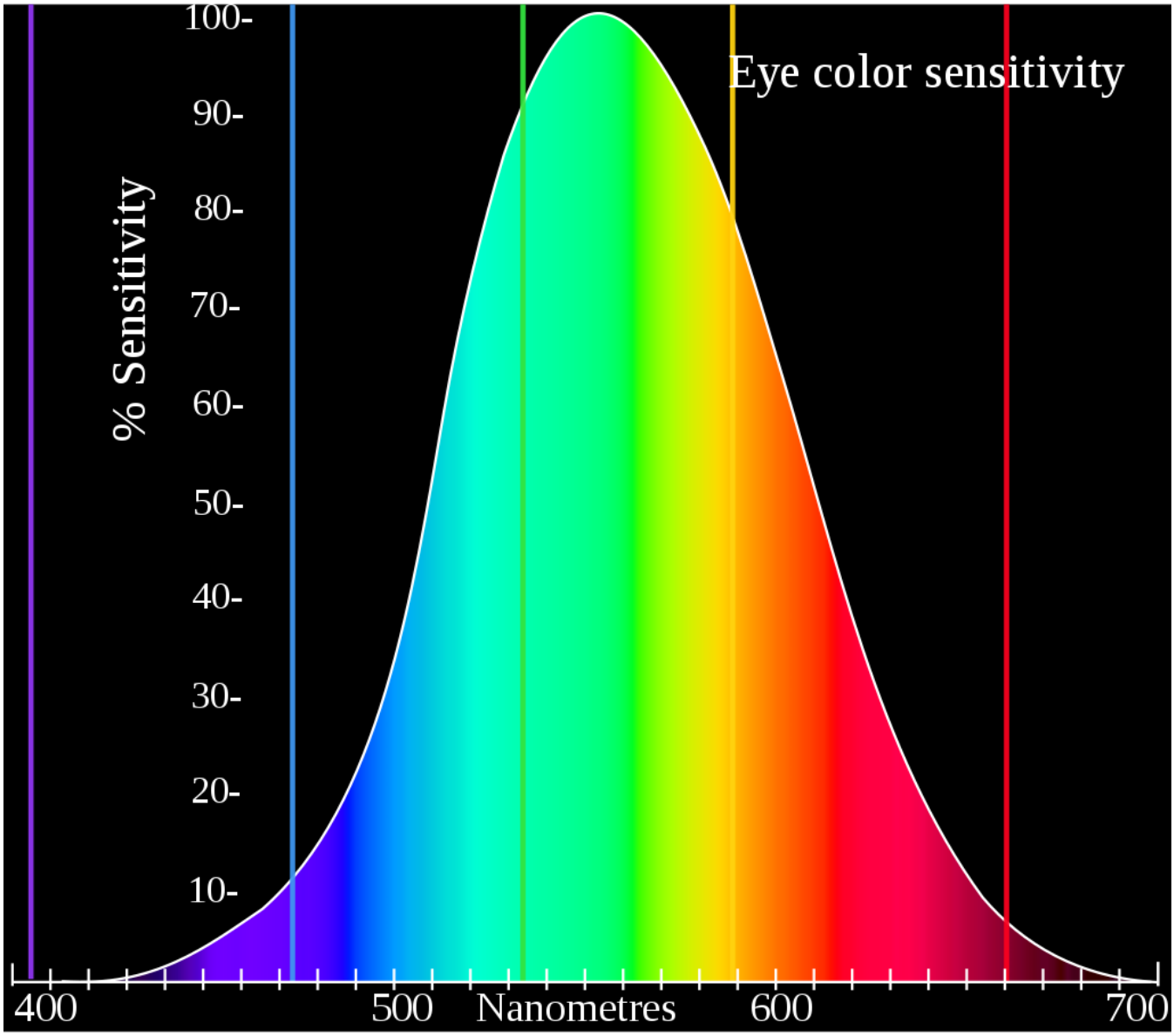
- In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called translation invariance.
- The earliest layers of the network should focus on local regions, without regard for the contents of the

Type *Markdown* and LaTeX: α^2

Human Brain Visual Cortex processing



Human Eye Colour Sensitivity



Source (https://en.wikipedia.org/wiki/Color_vision)

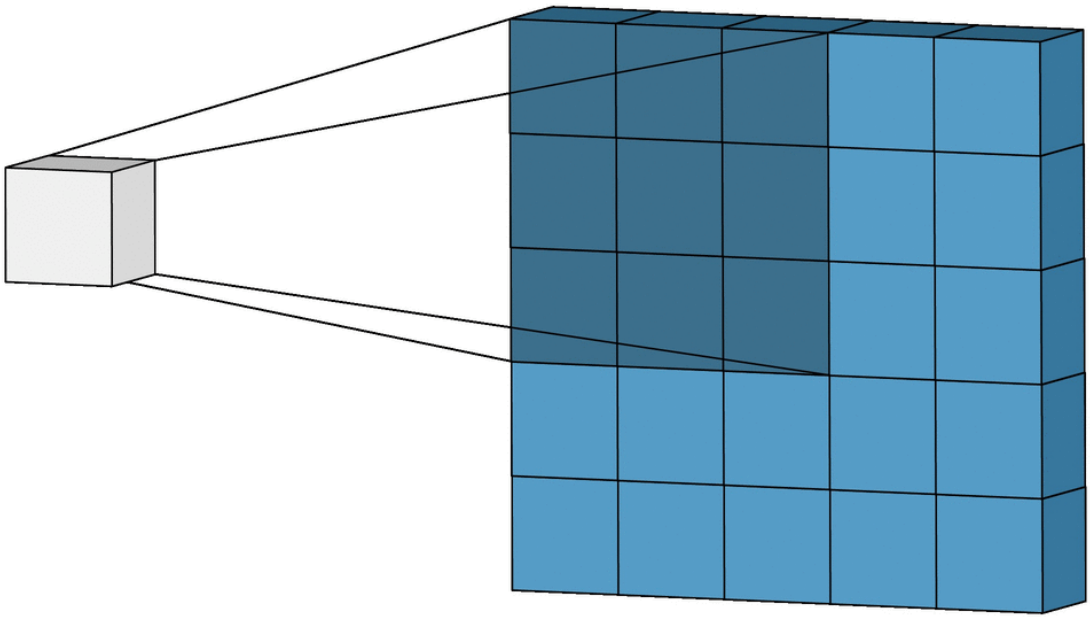
What are Convolutional Neural Networks?

Convolutional Neural Networks (ConvNets or CNNs) are a category of Neural Networks that have proven very effective in areas such as image recognition and classification. ConvNets have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self driving cars.

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same

Visualizing the Process

Simple Convolution

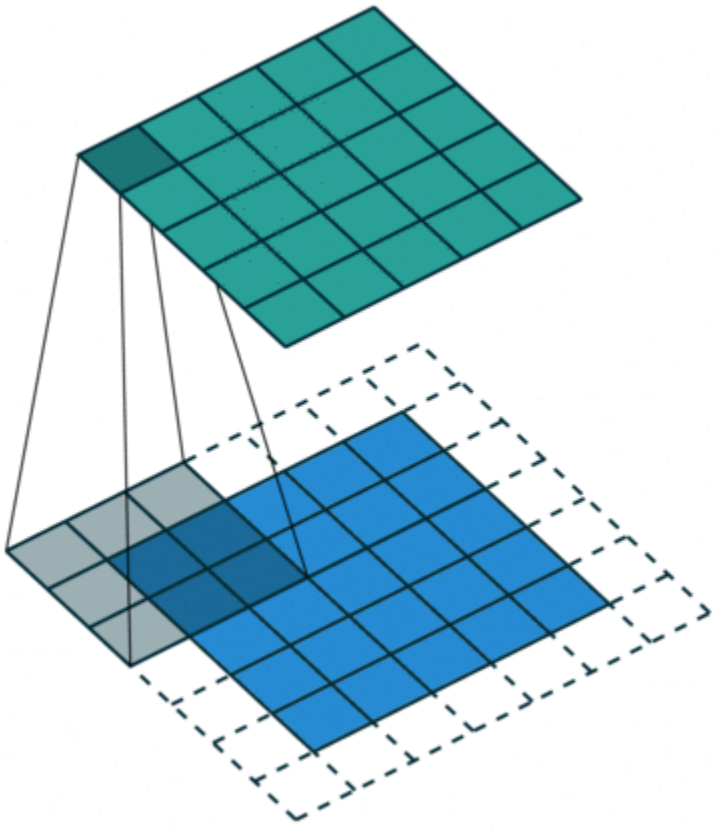


Matrix Calculation

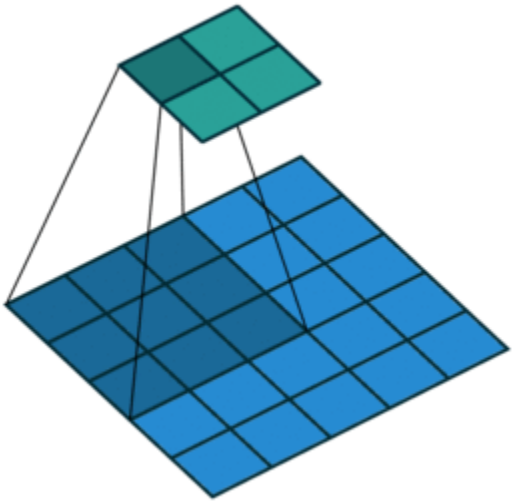
3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

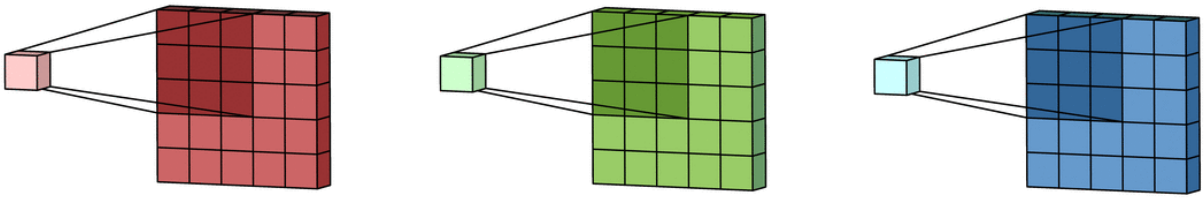
Padding Concept



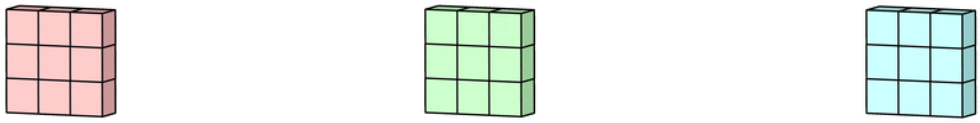
Stride Concept



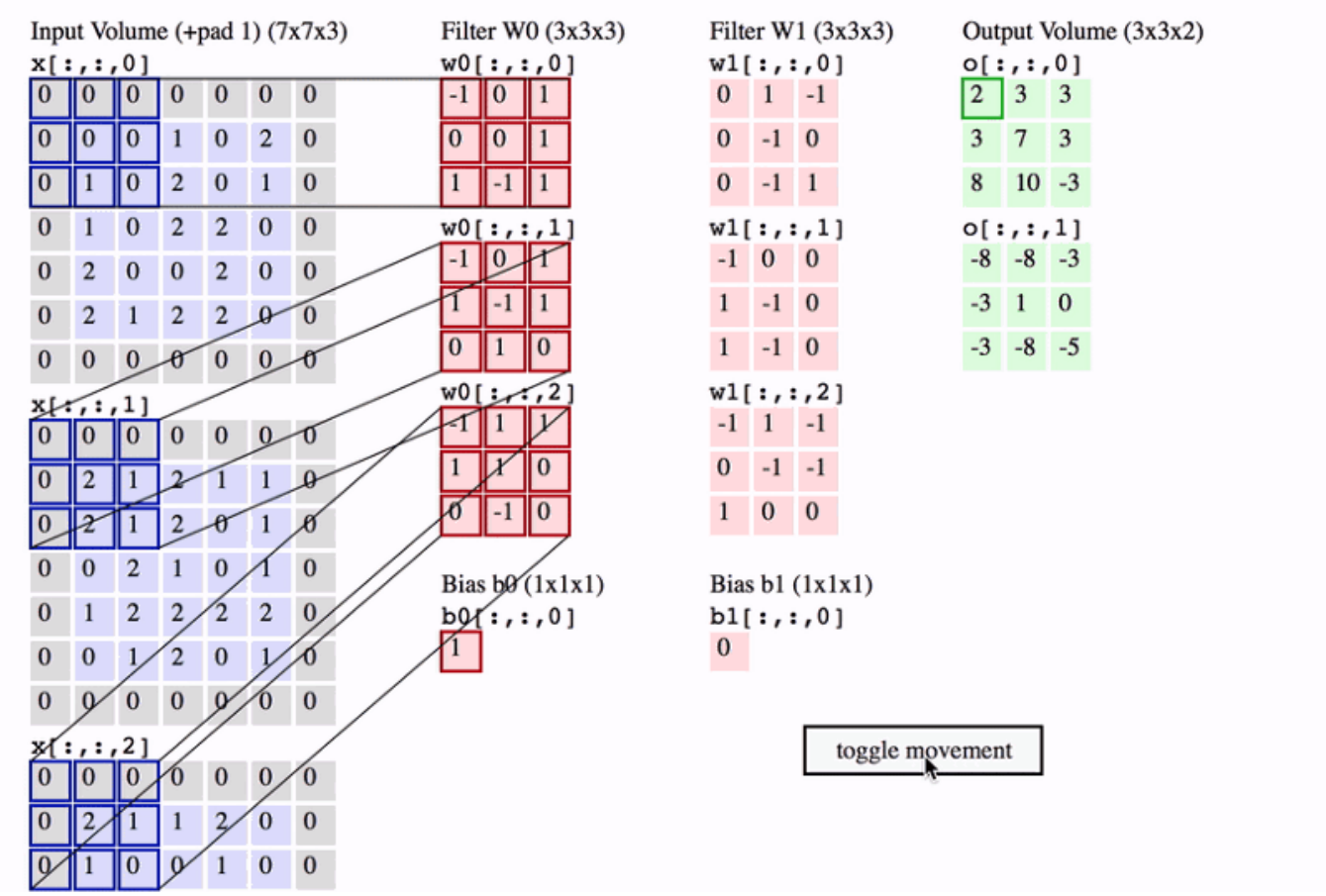
Feature Accumulation



Feature Aggregation



Convolution Operation



Type *Markdown* and LaTeX: α^2

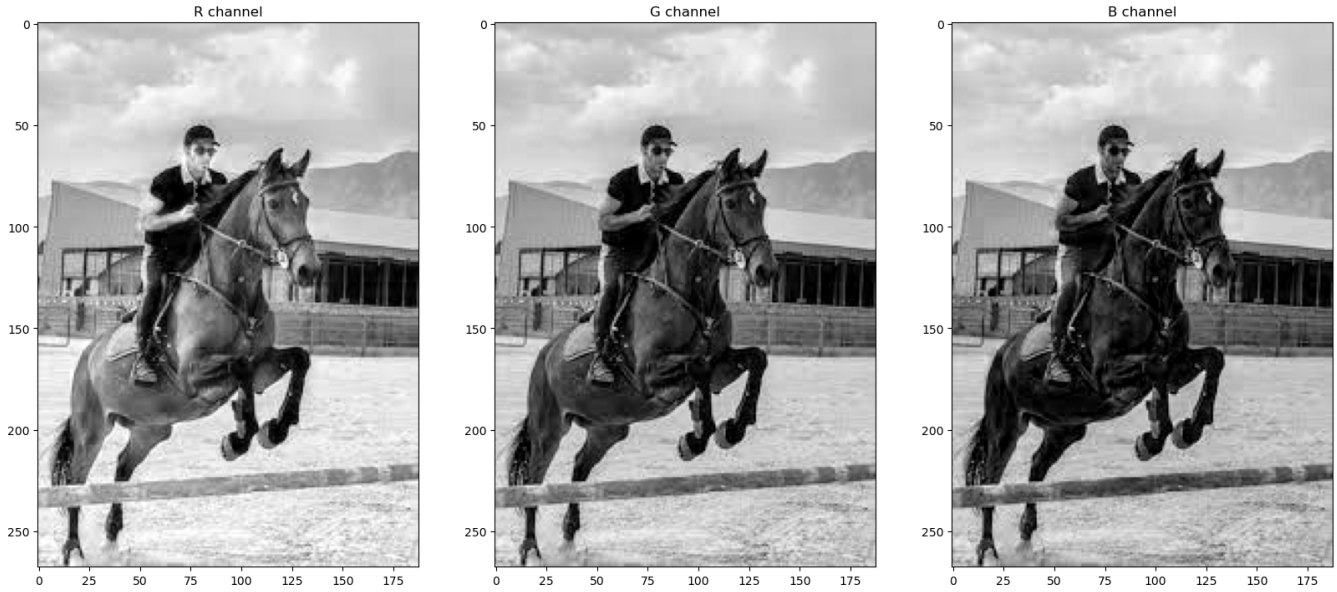
In [13]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

%matplotlib inline
```

In [14]:
```
# Read in the image
image = mpimg.imread('input.jfif')

plt.imshow(image)
```

Out[14]:
```
<matplotlib.image.AxesImage at 0x19597f05b10>
```

In [15]:
```
# Isolate RGB channels
r = image[:,:,0]
g = image[:,:,1]
b = image[:,:,2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.set_title('R channel')
ax1.imshow(r, cmap='gray')
ax2.set_title('G channel')
ax2.imshow(g, cmap='gray')
ax3.set_title('B channel')
ax3.imshow(b, cmap='gray')
```

Out[15]: <matplotlib.image.AxesImage at 0x195992c0520>



In [16]:
```
# Isolate RGB channels
r = image[:,:,0]
g = image[:,:,1]
b = image[:,:,2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.set_title('R channel')
ax1.imshow(r)
ax2.set_title('G channel')
ax2.imshow(g)
ax3.set_title('B channel')
ax3.imshow(b)
```

Out[16]: <matplotlib.image.AxesImage at 0x19599538a60>

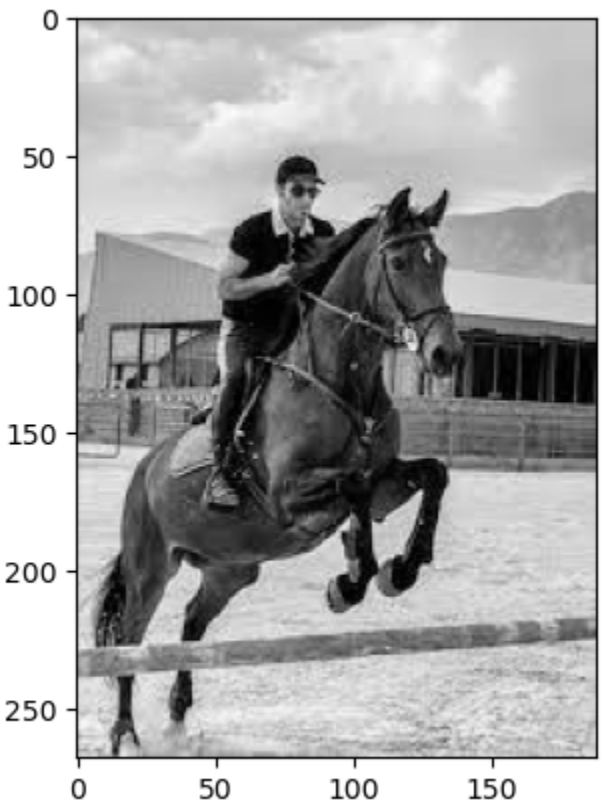
Focusing on Filters

In [17]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import cv2
import numpy as np

%matplotlib inline

# Read in the image
image = mpimg.imread('input.jfif')

plt.imshow(image)
```

Out[17]:
```
<matplotlib.image.AxesImage at 0x195997f71f0>
```

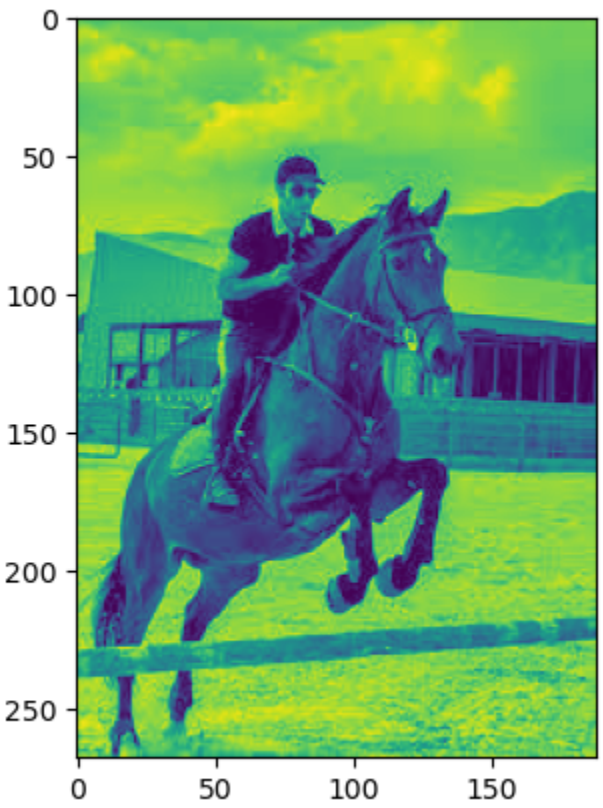
In [18]:
```
# Convert to grayscale for filtering
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

plt.imshow(gray, cmap='gray')
```

Out[18]: <matplotlib.image.AxesImage at 0x19599f7a890>



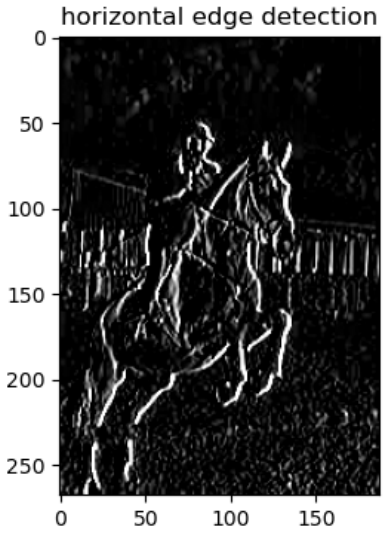
In [19]:
```
# Convert to grayscale for filtering
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

plt.imshow(gray)
```

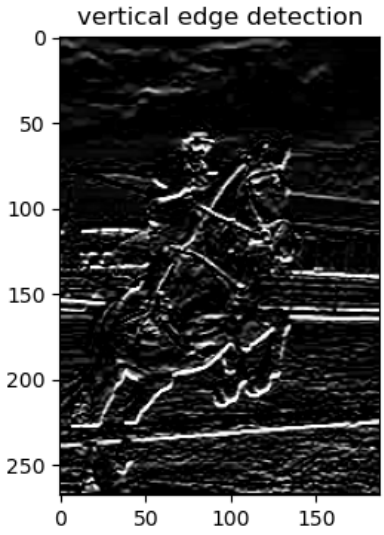
Out[19]: <matplotlib.image.AxesImage at 0x1959a015900>



In [20]:
```
#Horizontal edge detection
sobel_y = np.array([[ -1, -2, -1],
                    [ 0, 0, 0],
                    [ 1, 2, 1]])
# vertical edge detection
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])
# filter the image using filter2D(grayscale image, bit-depth, kernel)
filtered_image1 = cv2.filter2D(gray, -1, sobel_x)
filtered_image2 = cv2.filter2D(gray, -1, sobel_y)
f, ax = plt.subplots(1, 2, figsize=(15, 4))
ax[0].set_title('horizontal edge detection')
ax[0].imshow(filtered_image1, cmap='gray')
ax[1].set_title('vertical edge detection')
ax[1].imshow(filtered_image2, cmap='gray')
```

Out[20]: <matplotlib.image.AxesImage at 0x19599f98940>





In [21]:
```
#Sharpen the Image
sharpen = np.array([[ 0, -1, 0],
                    [ -1, 5, -1],
                    [ 0, -1, 0]])
# Outline
outline = np.array([[-1, -1, -1],
                    [-1, 8, -1],
                    [-1, -1, -1]])
# filter the image using filter2D(grayscale image, bit-depth, kernel)
filtered_image1 = cv2.filter2D(gray, -1, sharpen)
filtered_image2 = cv2.filter2D(gray, -1, outline)
f, ax = plt.subplots(1, 2, figsize=(15, 4))
ax[0].set_title('Checking Sharpness of Image')
ax[0].imshow(filtered_image1, cmap='gray')
ax[1].set_title('Checking Outline of Image')
ax[1].imshow(filtered_image2, cmap='gray')
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x1959b18f3d0>
```

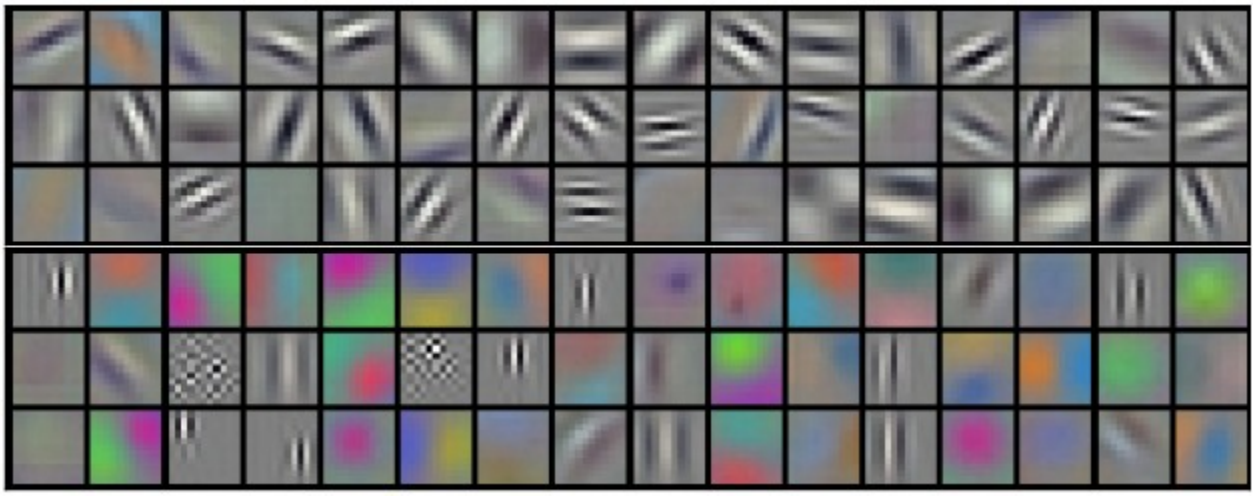
As youre thinking from Where I get Matrixes for sharpness or to check outline , I used below link

Best Place to Explore Kernels

Kernels (https://setosa.io/ev/image-kernels/)

Kernels as Edge Detector (https://aishack.in/tutorials/image-convolution-examples/)

Features extracted by Kernels



Features > Patterns > Parts of Object

Source (https://cs231n.github.io/convolutional-networks/)

Intuition

Let's develop better intuition for how Convolutional Neural Networks (CNN) work. We'll examine how humans classify images, and then see how CNNs use similar approaches.

Let’s say we wanted to classify the following image of a dog as a Golden Retriever:

As humans, how do we do this?

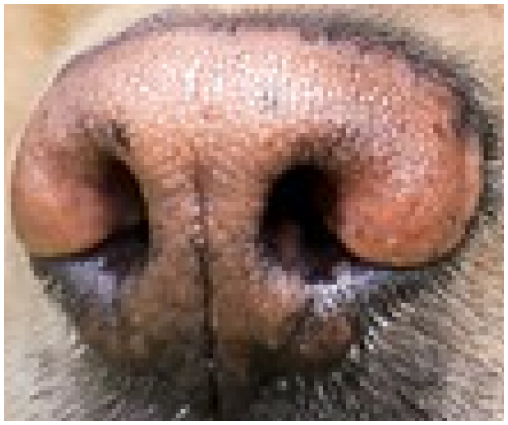
One thing we do is that we identify certain parts of the dog, such as the nose, the eyes, and the fur. We essentially break up the image into smaller pieces, recognize the smaller pieces, and then combine those pieces to get an idea of the overall dog.

In this case, we might break down the image into a combination of the following:

- A nose
- Two eyes
- Golden fur

These pieces can be seen below:

The eye of the dog.



The nose of the dog.

Going One Step Further

But let's take this one step further. How do we determine what exactly a nose is? A Golden Retriever nose can be seen as an oval with two black holes inside it. Thus, one way of classifying a Retriever's nose is to to break it up into smaller pieces and look for black holes (nostrils) and curves that define an oval as shown below:

A curve that we can use to determine a nose

A nostril that we can use to classify the nose of the dog

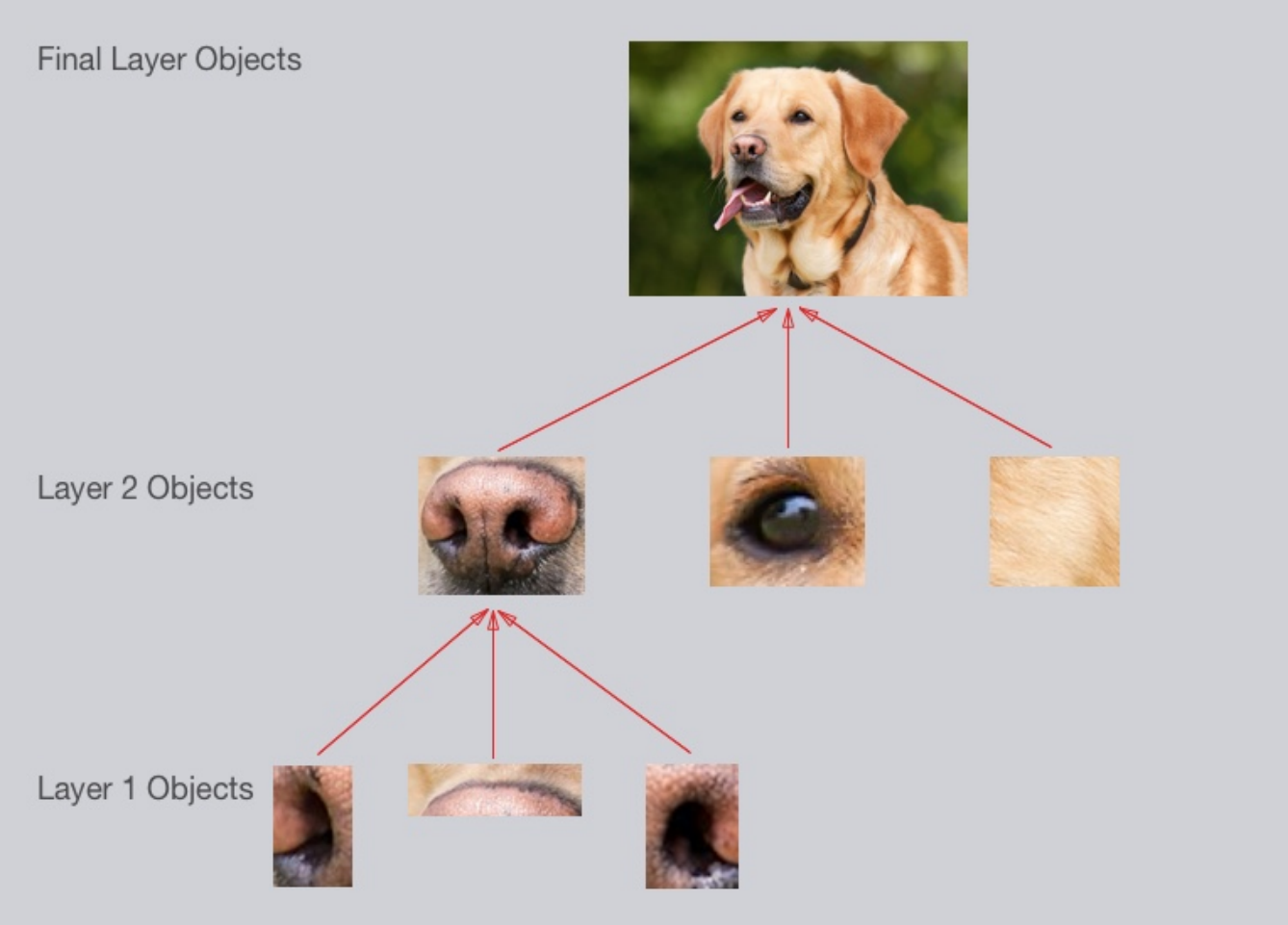
Broadly speaking, this is what a CNN learns to do. It learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. Finally, the CNN classifies the image by combining the larger, more complex objects.

In our case, the levels in the hierarchy are:

- Simple shapes, like ovals and dark circles
- Complex objects (combinations of simple shapes), like eyes, nose, and fur
- The dog as a whole (a combination of complex objects)

With deep learning, we don't actually program the CNN to recognize these specific features. Rather, the CNN learns on its own to recognize such objects through forward propagation and backpropagation!

It's amazing how well a CNN can learn to classify images, even though we never program the CNN with



An example of what each layer in a CNN might recognize when classifying a picture of a dog

A CNN might have several layers, and each layer might capture a different level in the hierarchy of objects. The first layer is the lowest level in the hierarchy, where the CNN generally classifies small parts of the image into simple shapes like horizontal and vertical lines and simple blobs of colors. The subsequent layers tend to be higher levels in the hierarchy and generally classify more complex ideas like shapes (combinations of lines), and eventually full objects like dogs.

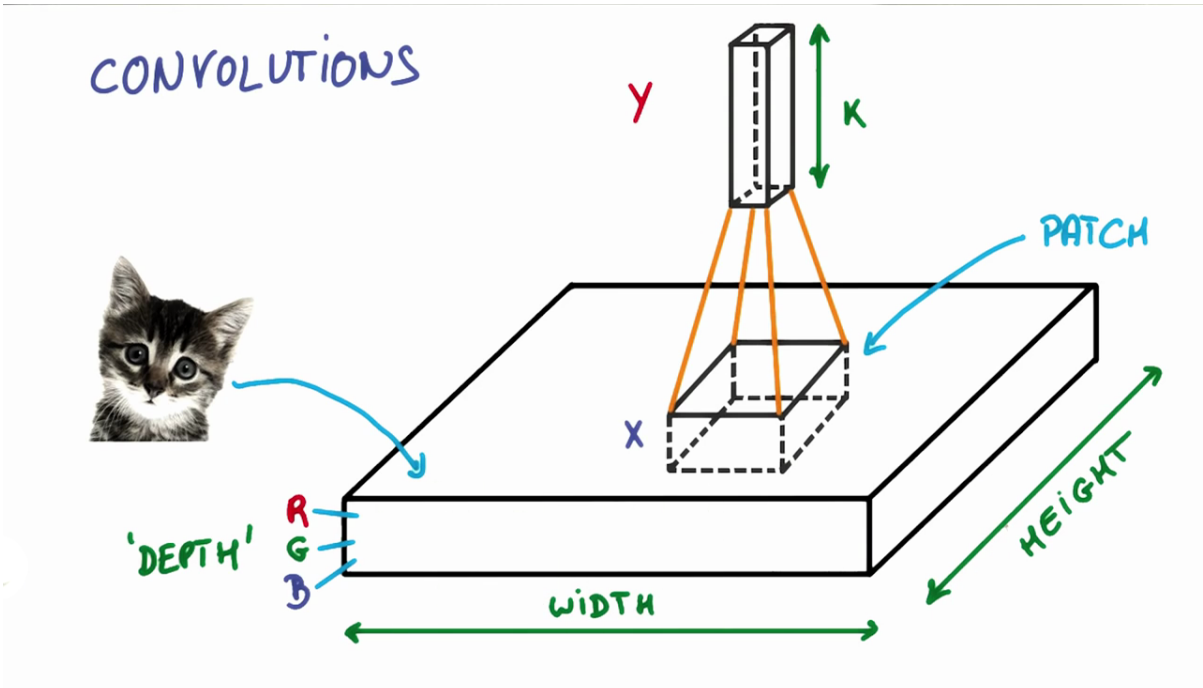
Once again, the CNN **learns all of this on its own**. We don't ever have to tell the CNN to go looking for lines or curves or noses or fur. The CNN just learns from the training set and discovers which characteristics of a Golden Retriever are worth looking for.

Filters

Breaking up an Image

The first step for a CNN is to break up the image into smaller pieces. We do this by selecting a width and height that defines a filter.

The filter looks at small pieces, or patches, of the image. These patches are the same size as the filter.

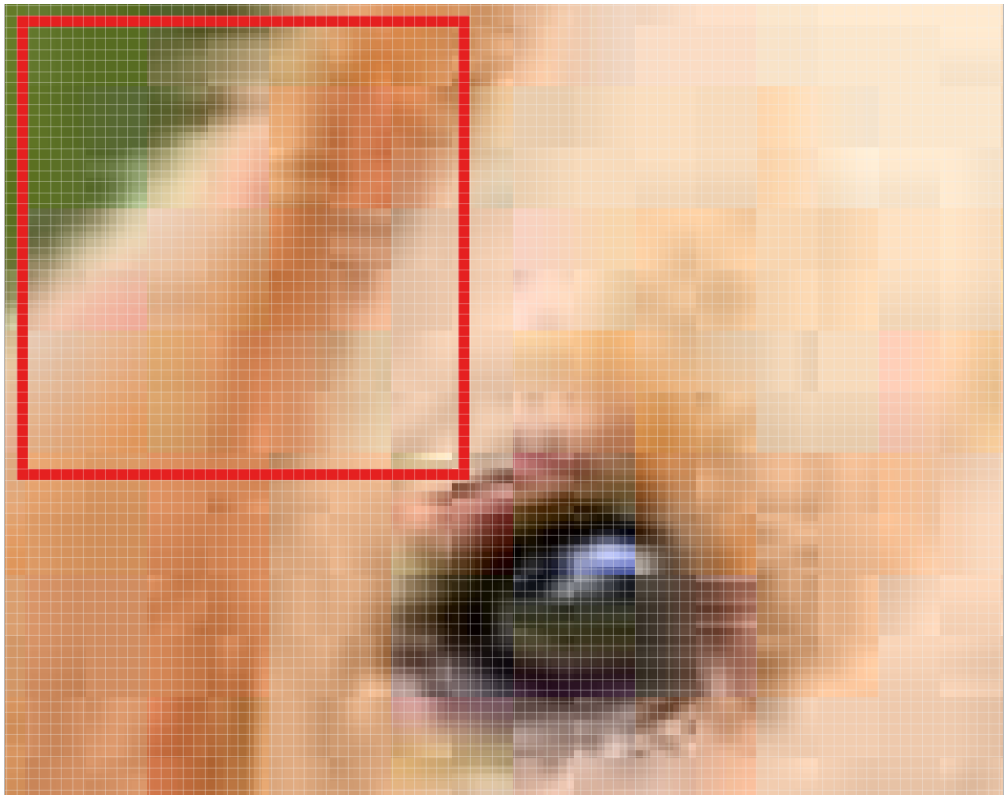


A CNN uses filters to split an image into smaller patches. The size of these patches matches the filter size.

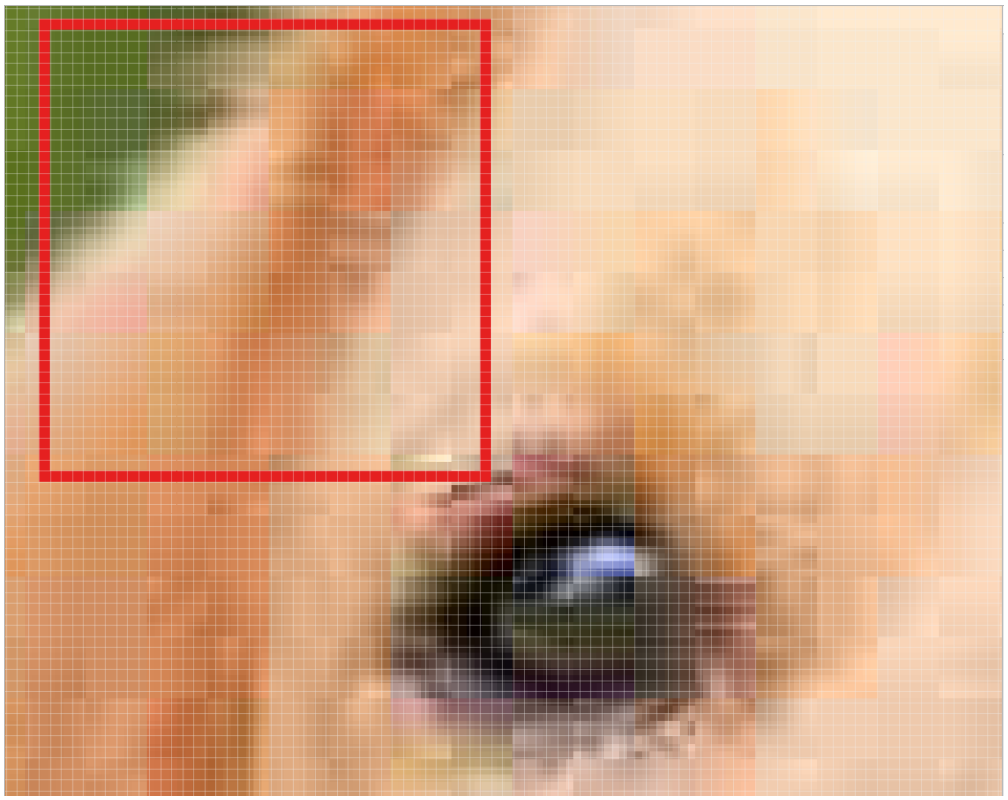
We then simply slide this filter horizontally or vertically to focus on a different piece of the image.

The amount by which the filter slides is referred to as the **'stride'**. The stride is a hyperparameter which the engineer can tune. Increasing the stride reduces the size of your model by reducing the number of total patches each layer observes. However, this usually comes with a reduction in accuracy.

Let's look at an example. In this zoomed in image of the dog, we first start with the patch outlined in red. The width and height of our filter define the size of this square.



We then move the square over to the right by a given stride (2 in this case) to get another patch.



We move our square to the right by two pixels to create another patch.

What's important here is that we are **grouping together adjacent pixels** and treating them as a collective.

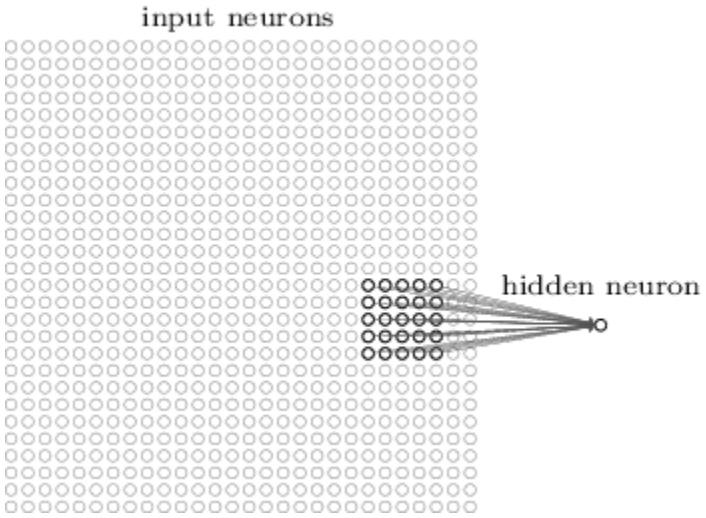
In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason and have

special meaning.

By taking advantage of this local structure, our CNN learns to classify local patterns, like shapes and objects, in an image.

Filter Depth

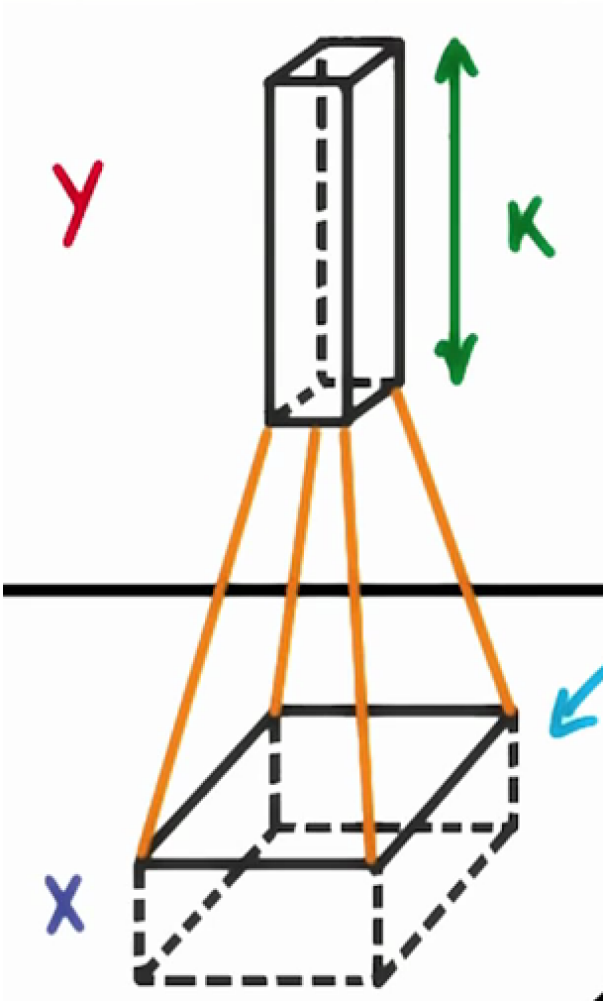
It's common to have more than one filter. Different filters pick up different qualities of a patch. For example, one filter might look for a particular color, while another might look for a kind of object of a specific shape. The amount of filters in a convolutional layer is called the **filter depth**.



In the above example, a patch is connected to a neuron in the next layer. Source: MIchael Neilsen.

How many neurons does each patch connect to?

That’s dependent on our filter depth. If we have a depth of k, we connect each patch of pixels to k neurons in the next layer. This gives us the height of k in the next layer, as shown below. In practice, k is a hyperparameter we tune, and most CNNs tend to pick the same starting values.



Choosing a filter depth of k connects each path to k neurons in the next layer

But why connect a single patch to multiple neurons in the next layer? Isn't one neuron good enough?

Multiple neurons can be useful because a patch can have multiple interesting characteristics that we want to capture.

For example, one patch might include some white teeth, some blonde whiskers, and part of a red tongue. In that case, we might want a filter depth of at least three - one for each of teeth, whiskers, and tongue.

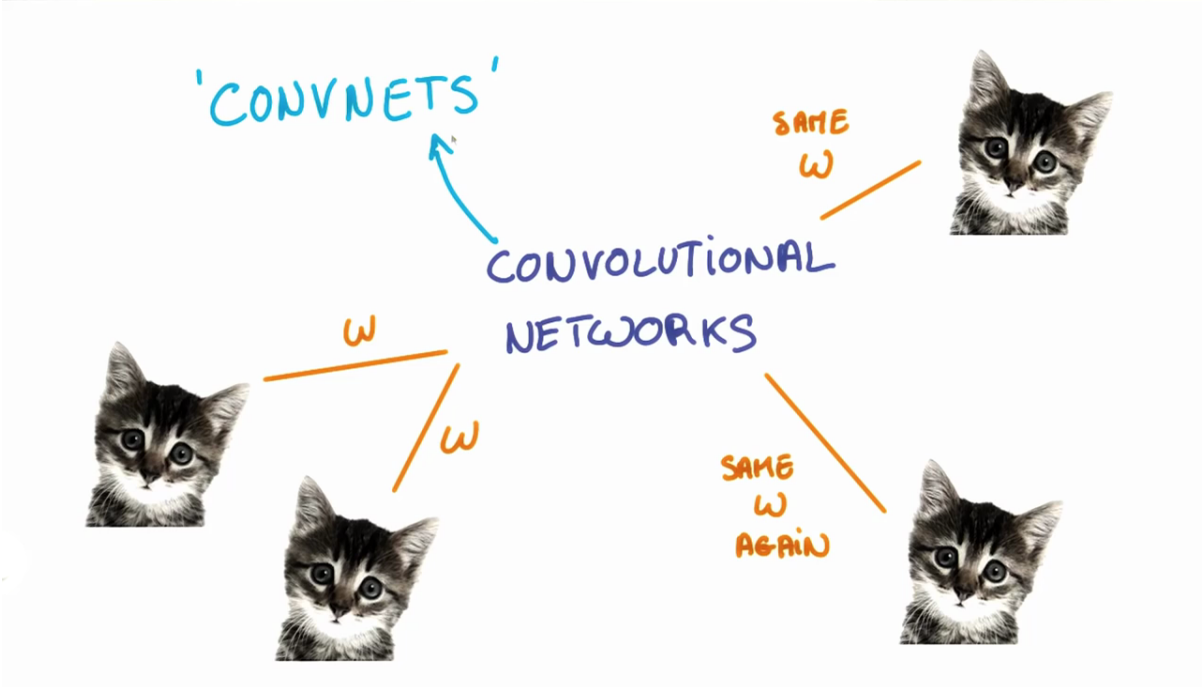
This patch of the dog has many interesting features we may want to capture. These include the presence of teeth, the presence of whiskers, and the pink color of the tongue.

Having multiple neurons for a given patch ensures that our CNN can learn to capture whatever characteristics the CNN learns are important.

Remember that the CNN isn't "programmed" to look for certain characteristics. Rather, it **learns on its own** which characteristics to notice.

Parameters

Parameter Sharing



The weights, w, are shared across patches for a given layer in a CNN to detect the cat above regardless of where in the image it is located.

When we are trying to classify a picture of a cat, we don't care where in the image a cat is. If it's in the top left or the bottom right, it's still a cat in our eyes. We would like our CNNs to also possess this ability known as translation invariance. How can we achieve this?

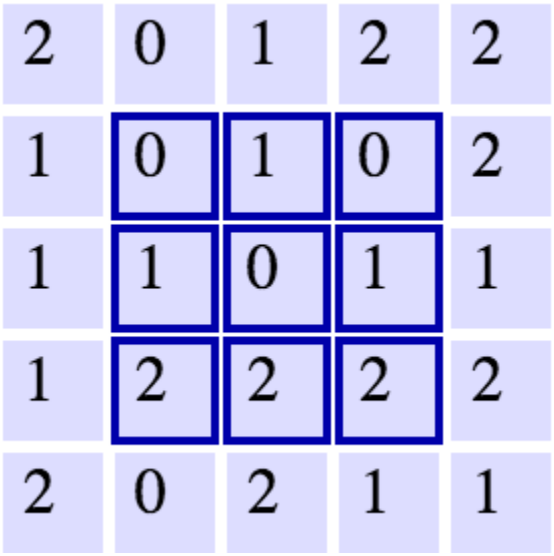
As we saw earlier, the classification of a given patch in an image is determined by the weights and biases corresponding to that patch.

If we want a cat that's in the top left patch to be classified in the same way as a cat in the bottom right patch, we need the weights and biases corresponding to those patches to be the same, so that they are classified the same way.

This is exactly what we do in CNNs. The weights and biases we learn for a given output layer are shared across all patches in a given input layer. Note that as we increase the depth of our filter, the number of weights and biases we have to learn still increases, as the weights aren't shared across the output channels.

There's an additional benefit to sharing our parameters. If we did not reuse the same weights across all patches, we would have to learn new parameters for every single patch and hidden layer neuron pair. This does not scale well, especially for higher fidelity images. Thus, sharing parameters not only helps us with translation invariance, but also gives us a smaller, more scalable model.

Padding



A 5x5 grid with a 3x3 filter. Source: Andrej Karpathy.

Let's say we have a `5x5` grid (as shown above) and a filter of size `3x3` with a stride of `1`. What's the width and height of the next layer? We see that we can fit at most three patches in each direction, giving us a dimension of `3x3` in our next layer. As we can see, the width and height of each subsequent layer decreases in such a scheme.

In an ideal world, we'd be able to maintain the same width and height across layers so that we can continue to add layers without worrying about the dimensionality shrinking and so that we have consistency. How might we achieve this? One way is to simple add a border of `0`s to our original `5x5` image. You can see what this looks like in the below image:

0	0	0	0	0	0	0
0	2	0	1	2	2	0
0	1	0	1	0	2	0
0	1	1	0	1	1	0
0	1	2	2	2	2	0
0	2	0	2	1	1	0
0	0	0	0	0	0	0

The same grid with 0 padding. Source: Andrej Karpathy.

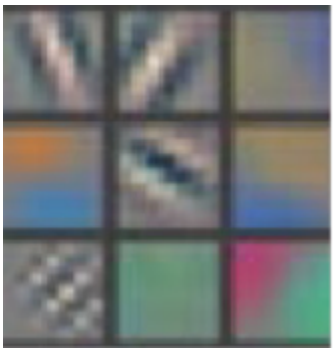
This would expand our original image to a `7x7`. With this, we now see how our next layer's size is again a `5x5`, keeping our dimensionality consistent.

Visualizing CNNs

Let's look at an example CNN to see how it works in action.

The CNN we will look at is trained on ImageNet as described in this paper (http://www.matthewzeiler.com/pubs/arxive2013/eccv2014.pdf) by Zeiler and Fergus. In the images below (from the same paper), we'll see what each layer in this network detects and see *how* each layer detects more and more complex ideas.

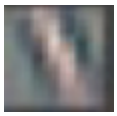
Layer 1



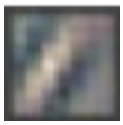
Example patterns that cause activations in the first layer of the network. These range from simple diagonal lines (top left) to green blobs (bottom middle).

The images above are from Matthew Zeiler and Rob Fergus' deep visualization toolbox (https://www.youtube.com/watch?v=ghEmQSxT6tw), which lets us visualize what each layer in a CNN focuses on.

Each image in the above grid represents a pattern that causes the neurons in the first layer to activate - in other words, they are patterns that the first layer recognizes. The top left image shows a -45 degree line, while the middle top square shows a +45 degree line. These squares are shown below again for reference:

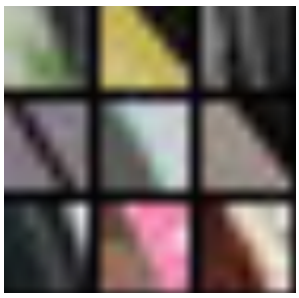


As visualized here, the first layer of the CNN can recognize -45 degree lines.



The first layer of the CNN is also able to recognize +45 degree lines, like the one above.

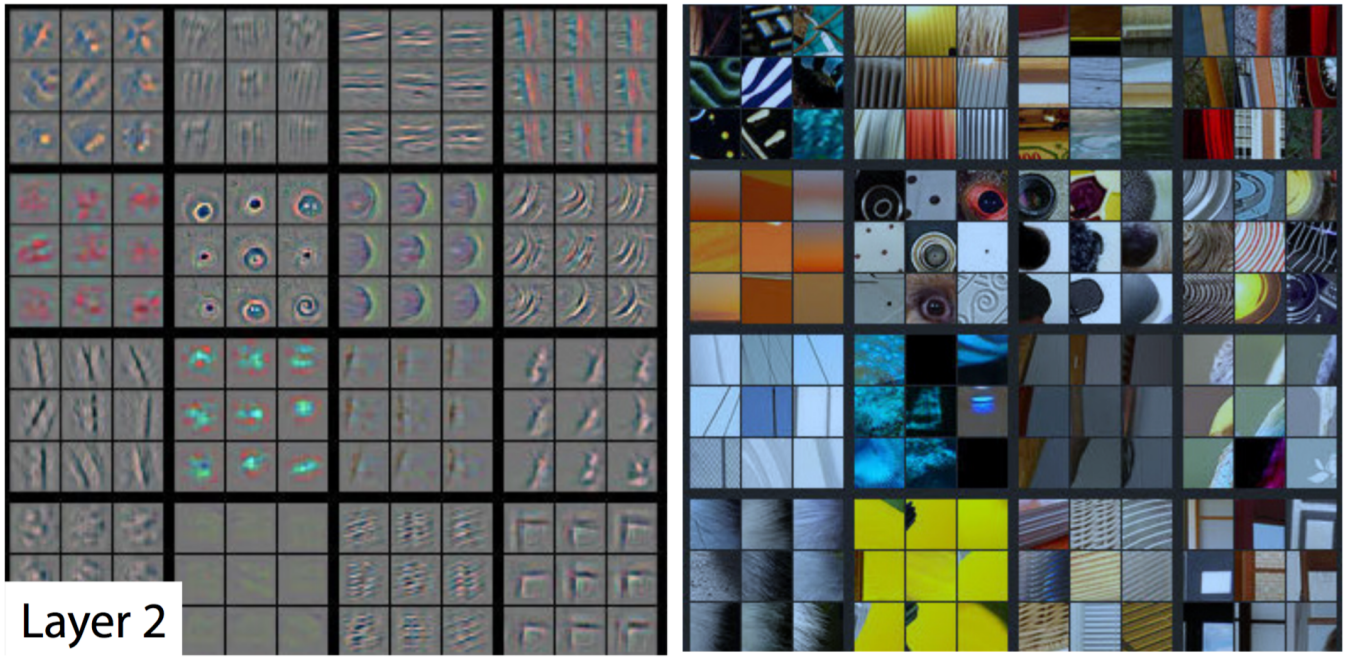
Let's now see some example images that cause such activations. The below grid of images all activated the -45 degree line. Notice how they are all selected despite the fact that they have different colors, gradients, and patterns.



Example patches that activate the -45 degree line detector in the first layer.

So, the first layer of our CNN clearly picks out very simple shapes and patterns like lines and blobs.

Layer 2



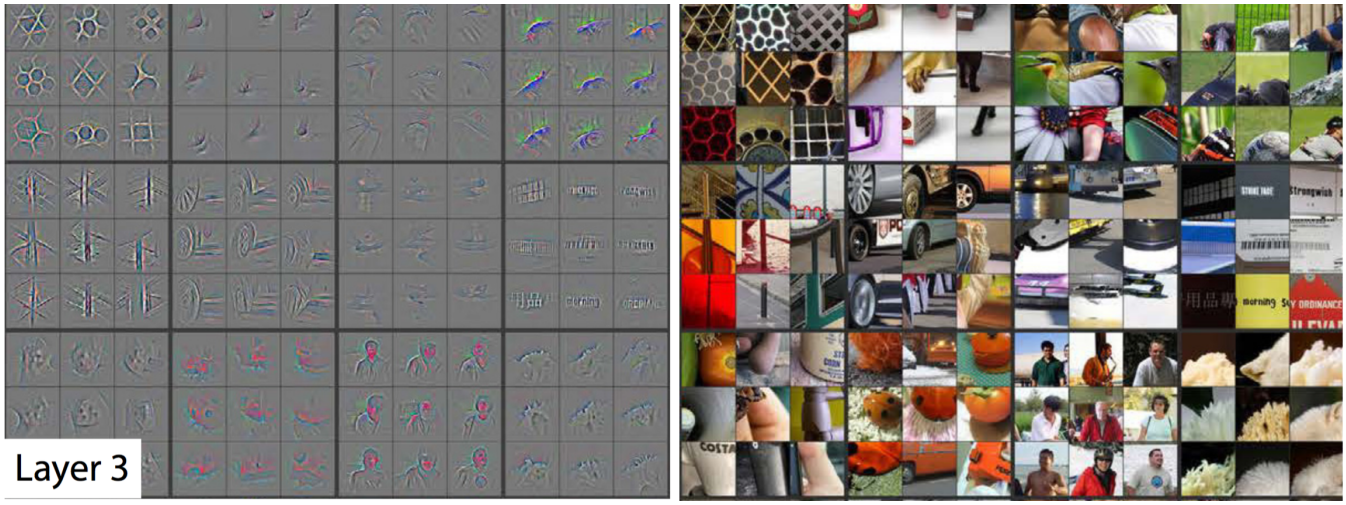
A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The second layer of the CNN captures complex ideas.

As you see in the image above, the second layer of the CNN recognizes circles (second row, second column), stripes (first row, second column), and rectangles (bottom right).

The CNN learns to do this on its own. There is no special instruction for the CNN to focus on more complex objects in deeper layers. That's just how it normally works out when you feed training data into a CNN.

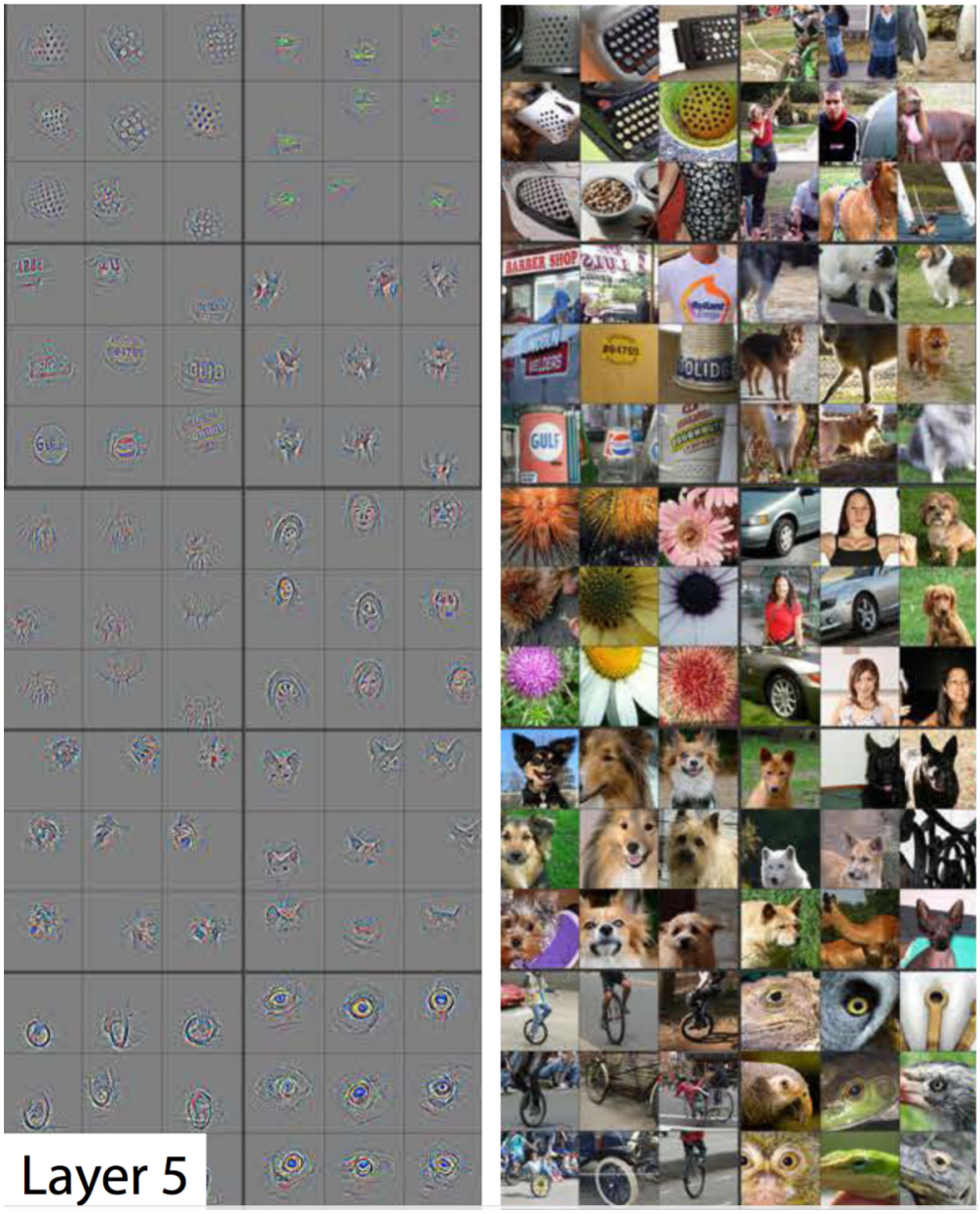
Layer 3



A visualization of the third layer in the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The third layer picks out complex combinations of features from the second layer. These include things like grids, and honeycombs (top left), wheels (second row, second column), and even faces (third row, third column).

Layer 5

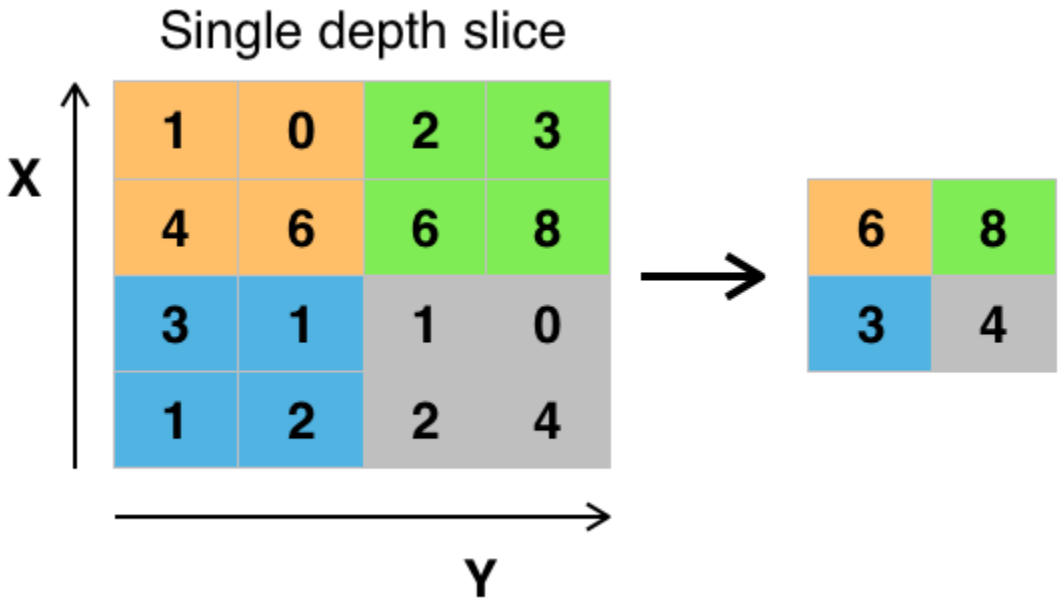


A visualization of the fifth and final layer of the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

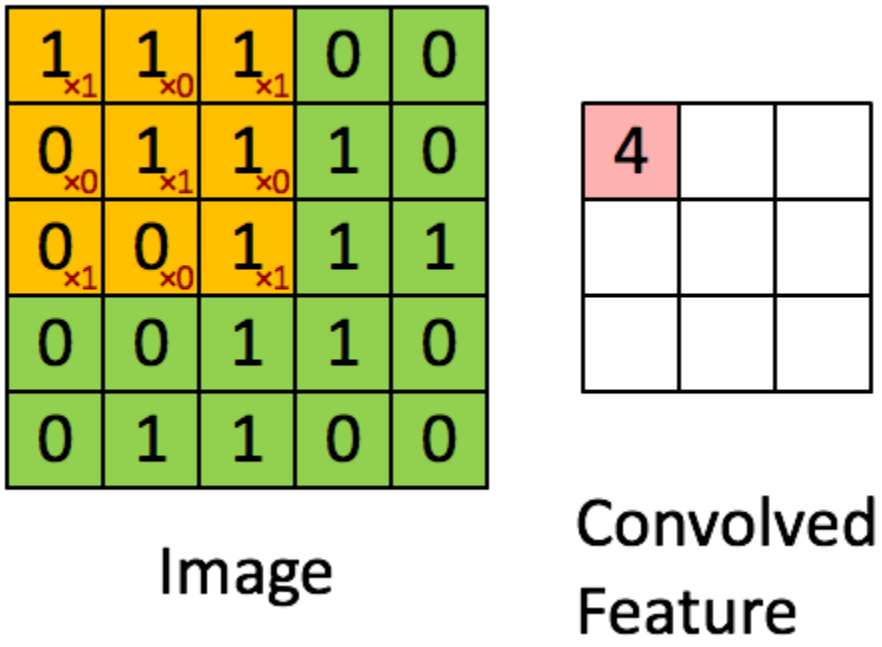
We'll skip layer 4, which continues this progression, and jump right to the fifth and final layer of this CNN.

The last layer picks out the highest order ideas that we care about for classification, like dog faces, bird faces, and bicycles.

Max Pooling



Convolutions

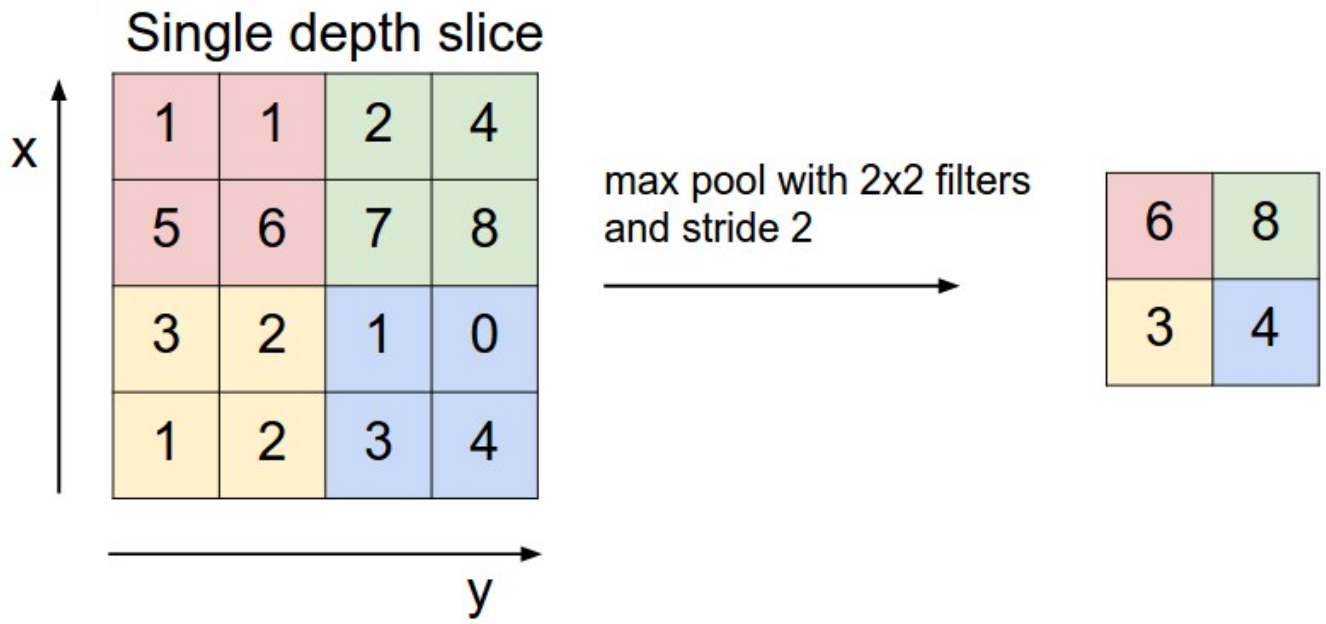


Convolution with 3×3 Filter. Source: http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution (http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution)

The above is an example of a convolution with a 3x3 filter and a stride of 1 being applied to data with a range of 0 to 1. The convolution for each 3x3 section is calculated against the weight, `[[1, 0, 1], [0, 1, 0], [1, 0, 1]]`, then a bias is added to create the convolved feature on the right. In this case, the bias is zero.

Stride is an array of 4 elements; the first element in the stride array indicates the stride for batch and last element indicates stride for features. It's good practice to remove the batches or features you want to skip from the data set rather than use stride to skip them. You can always set the first and last element to 1 in stride in order to use all batches and features.

Max Pooling



Max Pooling with 2x2 filter and stride of 2. Source: http://cs231n.github.io/convolutional-networks/ (http://cs231n.github.io/convolutional-networks/)

The above is an example of max pooling with a 2x2 filter and stride of 2. The left square is the input and the right square is the output. The four 2x2 colors in input represents each time the filter was applied to create the max on the right side. For example, `[[1, 1], [5, 6]]` becomes `6` and `[[3, 2], [1, 2]]` becomes `3`.

Practical Intuition with Code Explanation

In this notebook, we train a Convolutional Neural Network to classify images from the MNIST database.

1. Load MNIST Database

MNIST is one of the most famous datasets in the field of machine learning.

- It has 70,000 images of hand-written digits
- Very straight forward to download
- Images dimensions are 28x28

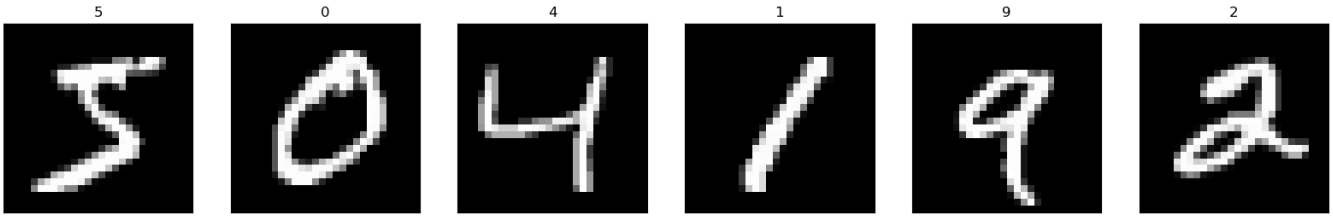
- Grayscale images

In [22]:
```
from tensorflow.keras.datasets import mnist

# use Keras to import pre-shuffled MNIST database
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("The MNIST database has a training set of %d examples." % len(X_train))
print("The MNIST database has a test set of %d examples." % len(X_test))
```

```
The MNIST database has a training set of 60000 examples.
The MNIST database has a test set of 10000 examples.
```

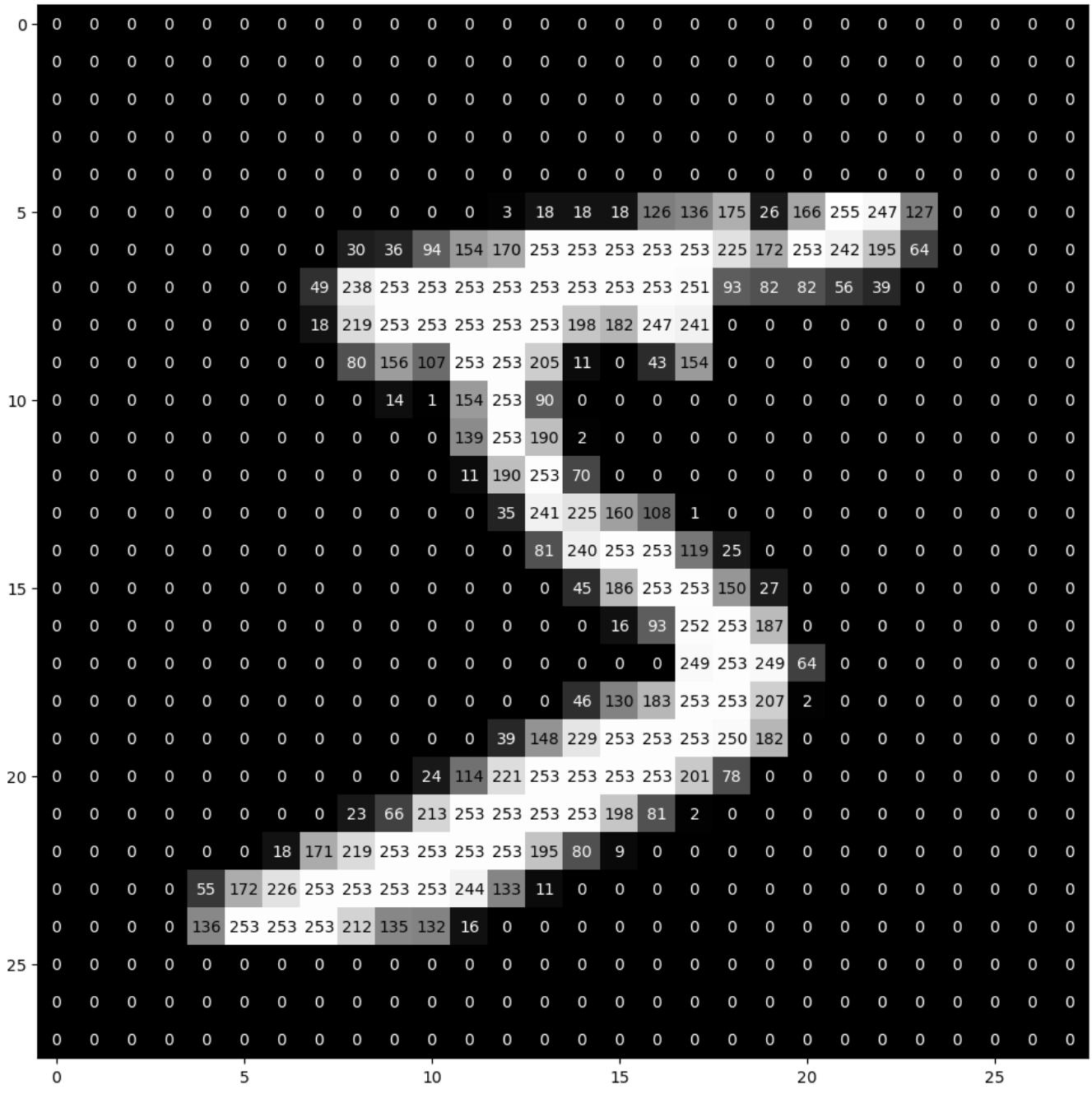
2. Visualize the First Six Training Images

In [23]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.cm as cm
import numpy as np

# plot first six training images
fig = plt.figure(figsize=(20,20))
for i in range(6):
    ax = fig.add_subplot(1, 6, i+1, xticks=[], yticks=[])
    ax.imshow(X_train[i], cmap='gray')
    ax.set_title(str(y_train[i]))
```



3. View an Image in More Detail

In [24]:
```
def visualize_input(img, ax):
    ax.imshow(img, cmap='gray')
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            ax.annotate(str(round(img[x][y],2)), xy=(y,x),
                        horizontalalignment='center',
                        verticalalignment='center',
                        color='white' if img[x][y]<thresh else 'black')

fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
visualize_input(X_train[0], ax)
```



4. Preprocess input images: Rescale the Images by Dividing Every Pixel in Every Image by 255

In [25]:
```
# rescale to have values within 0 - 1 range [0,255] --> [0,1]
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255

print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
X_train shape: (60000, 28, 28)
60000 train samples
10000 test samples
```

5. Preprocess the labels: Encode Categorical Integer Labels Using a One-Hot Scheme

In [26]:
```
from keras.utils import np_utils

num_classes = 10
# print first ten (integer-valued) training labels
print('Integer-valued labels:')
print(y_train[:10])

# one-hot encode the labels
# convert class vectors to binary class matrices
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

# print first ten (one-hot) training labels
print('One-hot labels:')
print(y_train[:10])
```

```
Integer-valued labels:
[5 0 4 1 9 2 1 3 1 4]
One-hot labels:
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

6. Reshape data to fit our CNN (and input_shape)

In [27]:
```
# input image dimensions 28x28 pixel images.
img_rows, img_cols = 28, 28

X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

print('input_shape: ', input_shape)
print('x_train shape:', X_train.shape)
```

```
input_shape:  (28, 28, 1)
x_train shape: (60000, 28, 28, 1)
```

7. Define the Model Architecture

You must pass the following arguments:

- filters - The number of filters.
- kernel_size - Number specifying both the height and width of the (square) convolution window.

There are some additional, optional arguments that you might like to tune:

- strides - The stride of the convolution. If you don't specify anything, strides is set to 1.
- padding - One of 'valid' or 'same'. If you don't specify anything, padding is set to 'valid'.
- activation - Typically 'relu'. If you don't specify anything, no activation is applied. You are strongly encouraged to add a ReLU activation function to every convolutional layer in your networks.

** Things to remember **

- Always add a ReLU activation function to the **Conv2D** layers in your CNN. With the exception of the final layer in the network, Dense layers should also have a ReLU activation function.
- When constructing a network for classification, the final layer in the network should be a **Dense** layer with a softmax activation function. The number of nodes in the final layer should equal the total number of classes in the dataset.

In [28]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Globa

# build the model object
model = Sequential()

# CONV_1: add CONV layer with RELU activation and depth = 32 kernels
model.add(Conv2D(32, kernel_size=(3, 3), padding='same',activation='relu',input_shape=(2
# POOL_1: downsample the image to choose the best features
model.add(MaxPooling2D(pool_size=(2, 2)))

# CONV_2: here we increase the depth to 64
model.add(Conv2D(64, (3, 3),padding='same', activation='relu'))
# POOL_2: more downsampling
model.add(MaxPooling2D(pool_size=(2, 2)))

# flatten since too many dimensions, we only want a classification output
model.add(Flatten())

# FC_1: fully connected to get all relevant data
model.add(Dense(64, activation='relu'))

# FC_2: output a softmax to squash the matrix into output probabilities for the 10 class
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten_1 (Flatten)         (None, 3136)              0

 dense_3 (Dense)             (None, 64)                200768

 dense_4 (Dense)             (None, 10)                650

=================================================================
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
_________________________________________________________________
```

Things to notice:

- The network begins with a sequence of two convolutional layers, followed by max pooling layers.

- The final layer has one entry for each object class in the dataset, and has a softmax activation function, so that it returns probabilities.
- The Conv2D depth increases from the input layer of 1 to 32 to 64.
- We also want to decrease the height and width - This is where maxpooling comes in. Notice that the image dimensions decrease from 28 to 14 after the pooling layer.
- You can see that every output shape has **None** in place of the batch-size. This is so as to facilitate changing of batch size at runtime.
- Finally, we add one or more fully connected layers to determine what object is contained in the image. For instance, if wheels were found in the last max pooling layer, this FC layer will transform that information to predict that a car is present in the image with higher probability. If there were eyes, legs

8. Compile the Model

In [29]:
```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
              metrics=['accuracy'])
```

9. Train the Model

In [30]:
```
from tensorflow.keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
                               save_best_only=True)
hist = model.fit(X_train, y_train, batch_size=64, epochs=10,
          validation_data=(X_test, y_test), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

```
Epoch 1/10

Epoch 1: val_loss improved from inf to 0.04668, saving model to model.weights.best.hdf5
938/938 - 43s - loss: 0.1545 - accuracy: 0.9517 - val_loss: 0.0467 - val_accuracy: 0.98
51 - 43s/epoch - 46ms/step
Epoch 2/10

Epoch 2: val_loss improved from 0.04668 to 0.03473, saving model to model.weights.best.
hdf5
938/938 - 35s - loss: 0.0456 - accuracy: 0.9855 - val_loss: 0.0347 - val_accuracy: 0.98
84 - 35s/epoch - 37ms/step
Epoch 3/10

Epoch 3: val_loss improved from 0.03473 to 0.03256, saving model to model.weights.best.
hdf5
938/938 - 46s - loss: 0.0325 - accuracy: 0.9898 - val_loss: 0.0326 - val_accuracy: 0.98
88 - 46s/epoch - 49ms/step
Epoch 4/10

Epoch 4: val_loss improved from 0.03256 to 0.02414, saving model to model.weights.best.
hdf5
938/938 - 45s - loss: 0.0237 - accuracy: 0.9929 - val_loss: 0.0241 - val_accuracy: 0.99
19 - 45s/epoch - 48ms/step
Epoch 5/10

Epoch 5: val_loss did not improve from 0.02414
938/938 - 53s - loss: 0.0192 - accuracy: 0.9942 - val_loss: 0.0265 - val_accuracy: 0.99
16 - 53s/epoch - 56ms/step
Epoch 6/10

Epoch 6: val_loss did not improve from 0.02414
938/938 - 49s - loss: 0.0145 - accuracy: 0.9955 - val_loss: 0.0288 - val_accuracy: 0.99
11 - 49s/epoch - 53ms/step
Epoch 7/10

Epoch 7: val_loss did not improve from 0.02414
938/938 - 51s - loss: 0.0118 - accuracy: 0.9967 - val_loss: 0.0358 - val_accuracy: 0.99
06 - 51s/epoch - 55ms/step
Epoch 8/10

Epoch 8: val_loss did not improve from 0.02414
938/938 - 50s - loss: 0.0090 - accuracy: 0.9974 - val_loss: 0.0406 - val_accuracy: 0.99
09 - 50s/epoch - 53ms/step
Epoch 9/10

Epoch 9: val_loss did not improve from 0.02414
938/938 - 51s - loss: 0.0079 - accuracy: 0.9977 - val_loss: 0.0328 - val_accuracy: 0.99
15 - 51s/epoch - 54ms/step
Epoch 10/10

Epoch 10: val_loss did not improve from 0.02414
938/938 - 51s - loss: 0.0059 - accuracy: 0.9983 - val_loss: 0.0360 - val_accuracy: 0.99
25 - 51s/epoch - 54ms/step
```

10. Load the Model with the Best Classification Accuracy on the Validation Set

In [31]:
```
# load the weights that yielded the best validation accuracy
model.load_weights('model.weights.best.hdf5')
```

11. Calculate the Classification Accuracy on the Test Set

In [33]:
```
# evaluate test accuracy
score = model.evaluate(X_test, y_test, verbose=0)
accuracy = 100*score[1]

# print test accuracy
print('Test accuracy: %.4f%%' % accuracy)
```

```
Test accuracy: 99.1900%
```

TO get Visualization the CNN Go to this website : Source (https://poloclub.github.io/cnn-explainer/)

CNNs will truly OUTPERFORM MLPs.

Lets Work with RGB Images

Here we will train a CNN to classify images from the CIFAR-10 dataset.

1. Load CIFAR-10 Database

In [55]:
```
import keras

# Get the list of available datasets in Keras
available_datasets = [name for name in dir(keras.datasets) if not name.startswith('_')]
print("Available datasets in Keras:", available_datasets)
```

```
Available datasets in Keras: ['boston_housing', 'cifar', 'cifar10', 'cifar100', 'fashio
n_mnist', 'imdb', 'mnist', 'reuters']
```

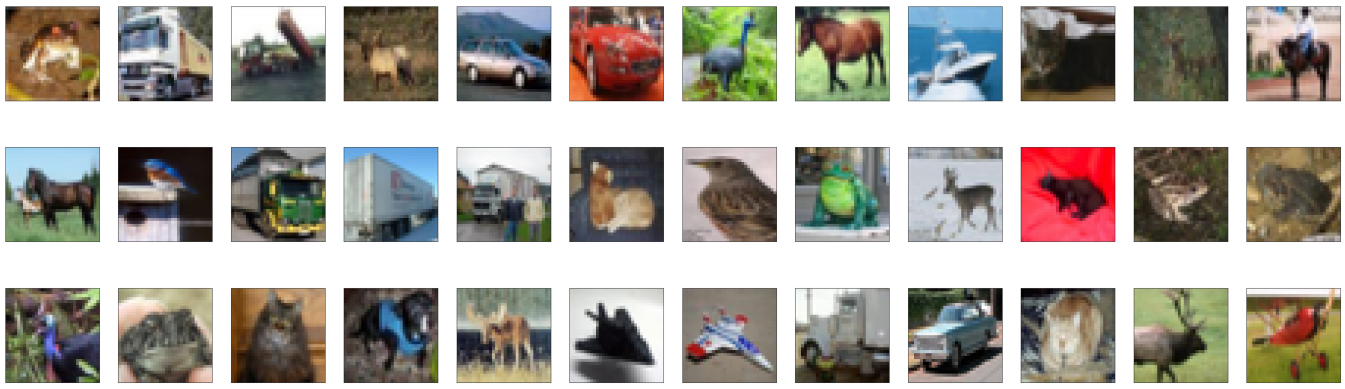
In [ ]:
```
from keras.datasets import reuters

# Load the Reuters dataset
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000

# Define the labels for Reuters dataset
# Note: There are 46 topics (0 to 45)
reuters_labels = [
    "cocoa", "grain", "veg-oil", "earn", "acq", "wheat", "copper", "housing",
    "money-supply", "coffee", "sugar", "trade", "reserves", "ship", "cotton",
    "carcass", "crude", "nat-gas", "cpi", "money-fx", "interest", "gnp", "meal-feed",
    "alum", "oilseed", "gold", "tin", "strategic-metal", "livestock", "retail",
    "ipi", "iron-steel", "rubber", "heat", "jobs", "lei", "bop", "zinc", "orange",
    "pet-chem", "dlr", "gas", "silver", "wpi", "hog", "lead"
]

# Print the labels for the Reuters test dataset
print("Labels of Reuters dataset:")
for label_index in test_labels:
    print(reuters_labels[label_index])
```

In [34]:
```
import keras
from keras.datasets import cifar10

# load the pre-shuffled train and test data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://
www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)
170498071/170498071 [==============================] - 192s 1us/step
```

2. Visualize the First 24 Training Images

In [37]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(50,15))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```



3. Rescale the Images by Dividing Every Pixel in Every Image by 255

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure below shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

** Tip: ** When using Gradient Descent, you should ensure that all features have a similar scale to speed up training or else it will take much longer to converge.

In [38]:
```
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

4. Break Dataset into Training, Testing, and Validation Sets

In [47]:
```
from keras.utils import np_utils
from tensorflow import keras

# one-hot encode the labels
num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

# print shape of training set
print('x_train shape:', x_train.shape)

# print number of training, validation, and test images
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
print(num_classes)
```

```
x_train shape: (40000, 32, 32, 3)
40000 train samples
10000 test samples
5000 validation samples
2
```

In [48]:
```
# print first ten (integer-valued) training labels
print('Integer-valued labels:')
print(y_train[:10])
```

```
Integer-valued labels:
[[[1. 0.]
  [0. 1.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]]

 [[1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [1. 0.]
  [0. 1.]
```

5. Define the Model Architecture

In [40]:
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=3, padding='same', activation='relu',
                 input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)            (None, 32, 32, 16)        448

 max_pooling2d_2 (MaxPooling  (None, 16, 16, 16)       0
 2D)

 conv2d_3 (Conv2D)            (None, 16, 16, 32)        4640

 max_pooling2d_3 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_4 (Conv2D)            (None, 8, 8, 64)          18496

 max_pooling2d_4 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 dropout (Dropout)            (None, 4, 4, 64)          0

 flatten_2 (Flatten)          (None, 1024)              0

 dense_5 (Dense)              (None, 500)               512500

 dropout_1 (Dropout)          (None, 500)               0

 dense_6 (Dense)              (None, 10)                5010

=================================================================
Total params: 541,094
Trainable params: 541,094
Non-trainable params: 0
_________________________________________________________________
```

6. Compile the Model

In [41]:
```
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
```

7. Train the Model

In [42]:
```
from keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1, save_best_

hist = model.fit(x_train, y_train, batch_size=32, epochs=5,
          validation_data=(x_valid, y_valid), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

```
Epoch 1/5

Epoch 1: val_loss improved from inf to 1.19884, saving model to model.weights.best.hdf5
1407/1407 - 31s - loss: 1.5251 - accuracy: 0.4464 - val_loss: 1.1988 - val_accuracy: 0.
5774 - 31s/epoch - 22ms/step
Epoch 2/5

Epoch 2: val_loss improved from 1.19884 to 0.99467, saving model to model.weights.best.
hdf5
1407/1407 - 27s - loss: 1.1671 - accuracy: 0.5870 - val_loss: 0.9947 - val_accuracy: 0.
6496 - 27s/epoch - 19ms/step
Epoch 3/5

Epoch 3: val_loss improved from 0.99467 to 0.92308, saving model to model.weights.best.
hdf5
1407/1407 - 28s - loss: 1.0343 - accuracy: 0.6336 - val_loss: 0.9231 - val_accuracy: 0.
6810 - 28s/epoch - 20ms/step
Epoch 4/5

Epoch 4: val_loss improved from 0.92308 to 0.85512, saving model to model.weights.best.
hdf5
1407/1407 - 33s - loss: 0.9418 - accuracy: 0.6686 - val_loss: 0.8551 - val_accuracy: 0.
6994 - 33s/epoch - 23ms/step
Epoch 5/5

Epoch 5: val_loss improved from 0.85512 to 0.79477, saving model to model.weights.best.
hdf5
1407/1407 - 38s - loss: 0.8747 - accuracy: 0.6910 - val_loss: 0.7948 - val_accuracy: 0.
7194 - 38s/epoch - 27ms/step
```

8. Load the Model with the Best Validation Accuracy

In [50]:
```
model.load_weights('model.weights.best.hdf5')
```

In [51]:
```
model
```

Out[51]:
```
<keras.engine.sequential.Sequential at 0x1959f1f8f70>
```

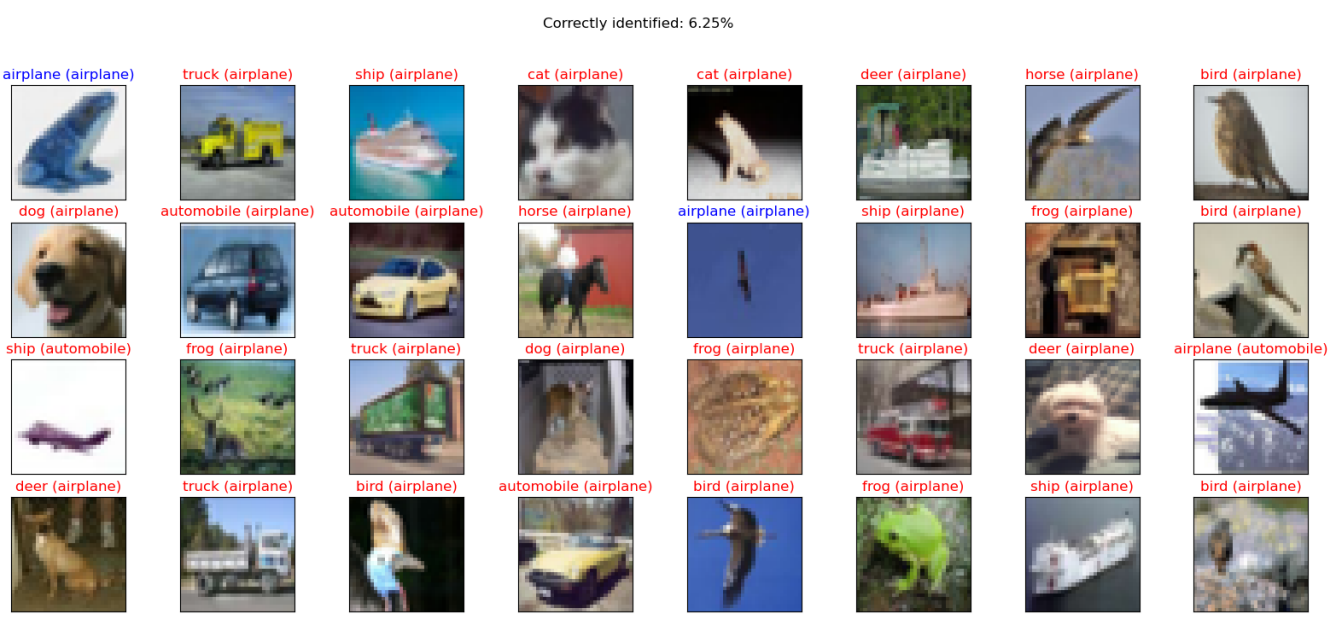
10. Visualize Some Predictions

In [52]:
```
# get predictions on the test set
y_hat = model.predict(x_test)

# define text labels (source: https://www.cs.toronto.edu/~kriz/cifar.html)
cifar10_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse
```

```
313/313 [==============================] - 2s 7ms/step
```

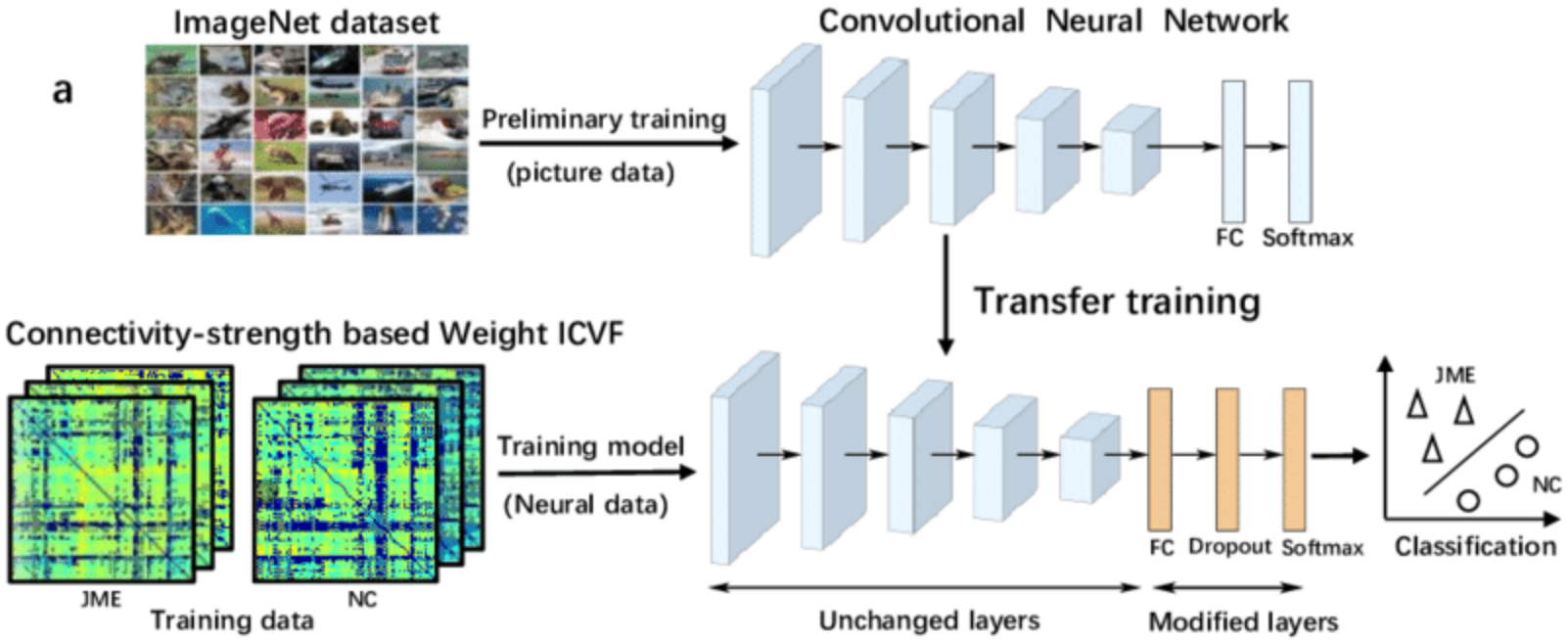
In [46]:
```
# plot a random sample of test images, their predicted labels, and ground truth
fig = plt.figure(figsize=(20, 8))
for i, idx in enumerate(np.random.choice(x_test.shape[0], size=32, replace=False)):
    ax = fig.add_subplot(4, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_test[idx]))
    pred_idx = np.argmax(y_hat[idx])
    true_idx = np.argmax(y_test[idx])
    ax.set_title("{} ({})".format(cifar10_labels[pred_idx], cifar10_labels[true_idx]),
                 color=("blue" if pred_idx == true_idx else "red"))
```

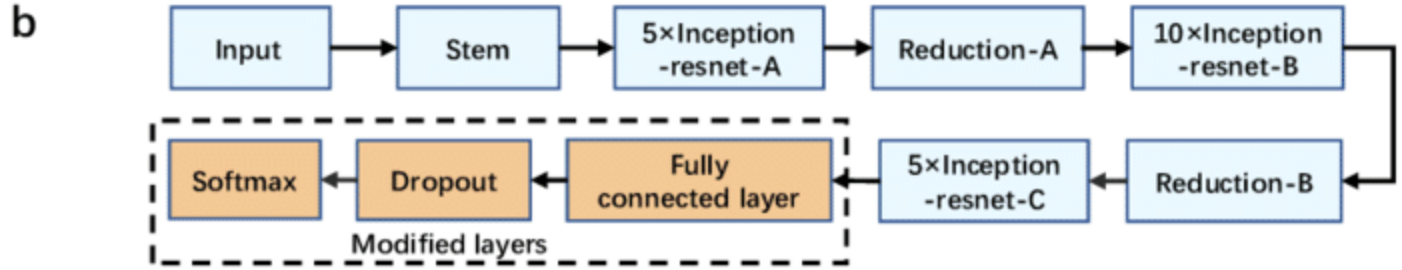
In [53]:
```
# plot a random sample of test images, their predicted labels, and ground truth
fig = plt.figure(figsize=(20, 8))
correct_predictions = 0  # Initialize a counter for correct predictions

for i, idx in enumerate(np.random.choice(x_test.shape[0], size=32, replace=False)):
    ax = fig.add_subplot(4, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_test[idx]))
    pred_idx = np.argmax(y_hat[idx])
    true_idx = np.argmax(y_test[idx])
    ax.set_title("{} ({})".format(cifar10_labels[pred_idx], cifar10_labels[true_idx]),
                 color=("blue" if pred_idx == true_idx else "red"))

    if pred_idx == true_idx:
        correct_predictions += 1

accuracy_ratio = correct_predictions / 32  # Calculate the accuracy ratio
plt.suptitle("Correctly identified: {:.2f}%".format(accuracy_ratio * 100))
plt.show()
```



In [ ]:
```
```

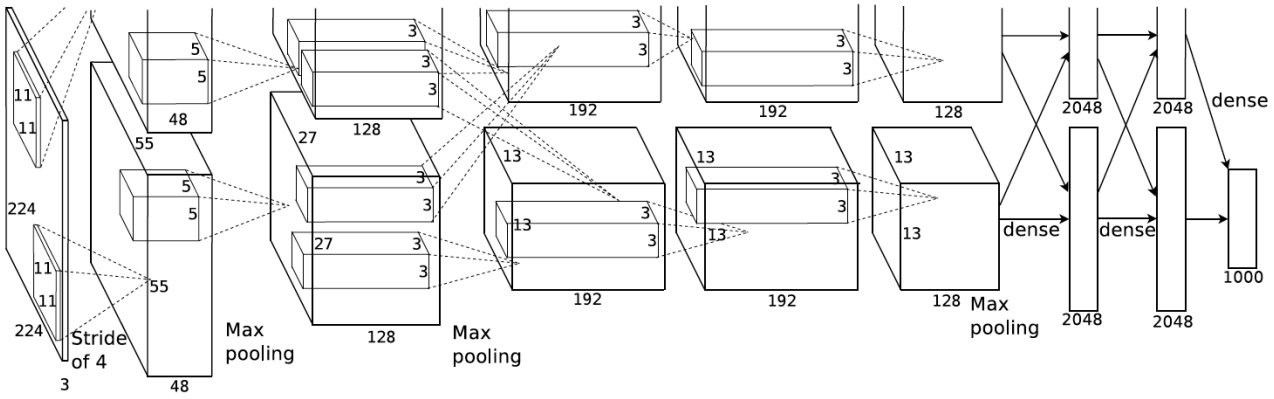
ALexa Net CNN

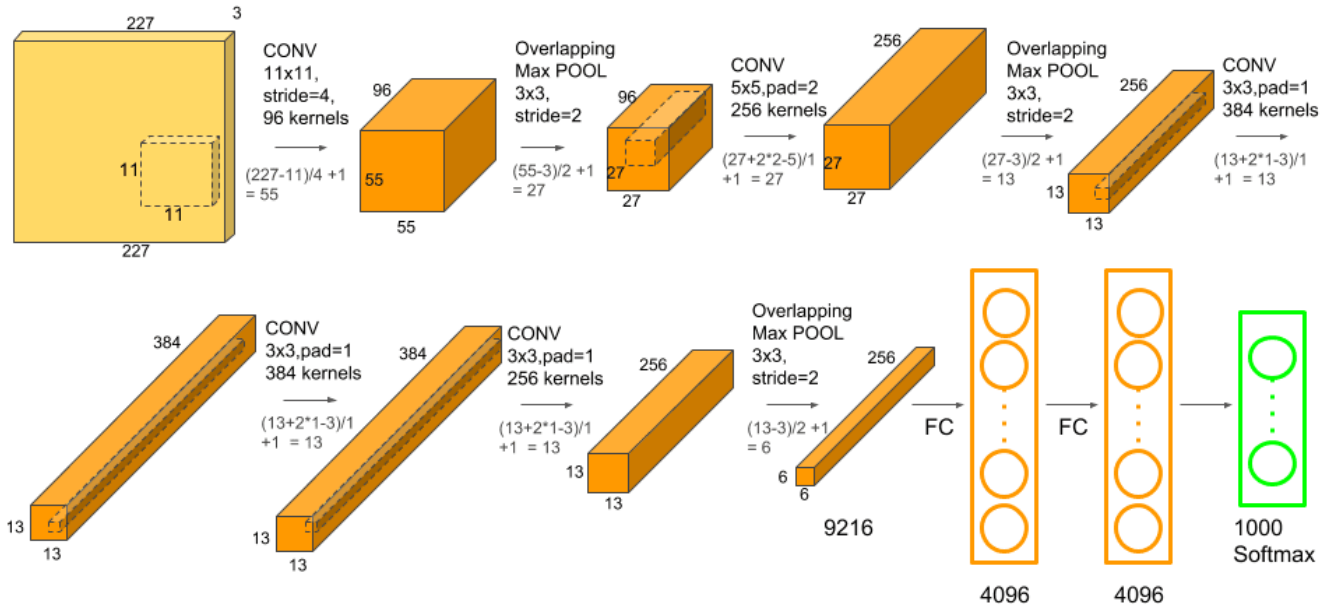




Introduction

AlexNet was designed by Hinton, winner of the 2012 ImageNet competition, and his student Alex Krizhevsky. It was also after that year that more and deeper neural networks were proposed, such as the excellent vgg, GoogleLeNet. Its official data model has an accuracy rate of 57.1% and top 1-5 reaches 80.2%. This is already quite outstanding for traditional machine learning classification algorithms.





The following table below explains the network structure of AlexNet:

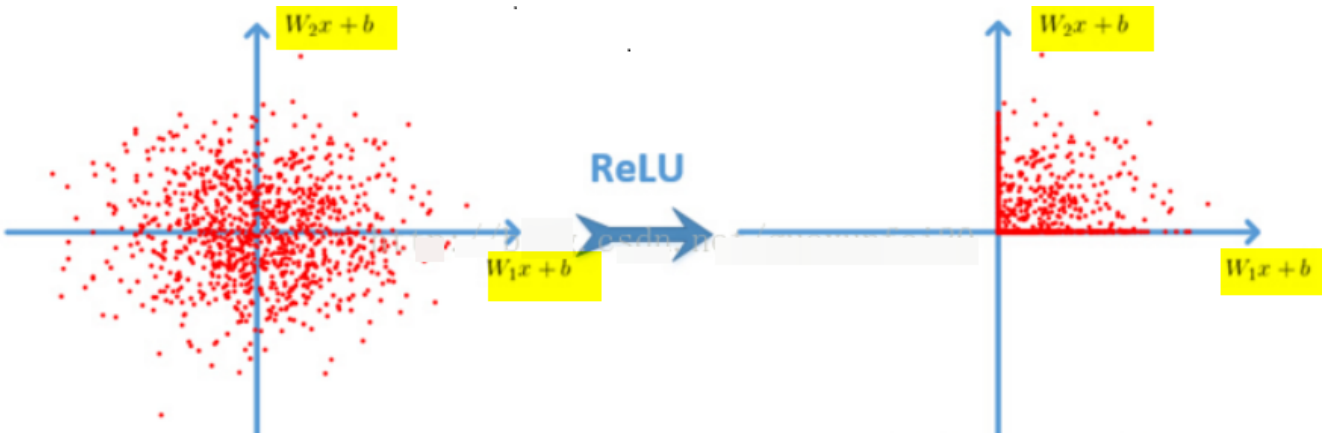
Size / Operation	Filter	Depth	Stride	Padding	Number of Parameters	Forward Computation
3* 227 * 227						
Conv1 + Relu	11 * 11	96	4		(11*11*3 + 1) * 96=34944	(11*11*3 + 1) * 96 * 55 * 55=105705600

Size / Operation	Filter	Depth	Stride	Padding	Number of Parameters	Forward Computation
96 * 55 * 55						
Max Pooling	3 * 3		2			
96 * 27 * 27						
Norm						
Conv2 + Relu	5 * 5	256	1	2	(5 * 5 * 96 + 1) * 256=614656	(5 * 5 * 96 + 1) * 256 * 27 * 27=448084224
256 * 27 * 27						
Max Pooling	3 * 3		2			
256 * 13 * 13						
Norm						
Conv3 + Relu	3 * 3	384	1	1	(3 * 3 * 256 + 1) * 384=885120	(3 * 3 * 256 + 1) * 384 * 13 * 13=149585280
384 * 13 * 13						
Conv4 + Relu	3 * 3	384	1	1	(3 * 3 * 384 + 1) * 384=1327488	(3 * 3 * 384 + 1) * 384 * 13 * 13=224345472
384 * 13 * 13						
Conv5 + Relu	3 * 3	256	1	1	(3 * 3 * 384 + 1) * 256=884992	(3 * 3 * 384 + 1) * 256 * 13 * 13=149563648
256 * 13 * 13						
Max Pooling	3 * 3		2			
256 * 6 * 6						
Dropout (rate 0.5)						
FC6 + Relu					256 * 6 * 6 * 4096=37748736	256 * 6 * 6 * 4096=37748736
4096						
Dropout (rate 0.5)						
FC7 + Relu					4096 * 4096=16777216	4096 * 4096=16777216
4096						
FC8 + Relu					4096 * 1000=4096000	4096 * 1000=4096000
1000 classes						
Overall					62369152=62.3 million	1135906176=1.1 billion
Conv VS FC					Conv:3.7million (6%) , FC: 58.6 million (94%)	Conv: 1.08 billion (95%) , FC: 58.6 million (5%)

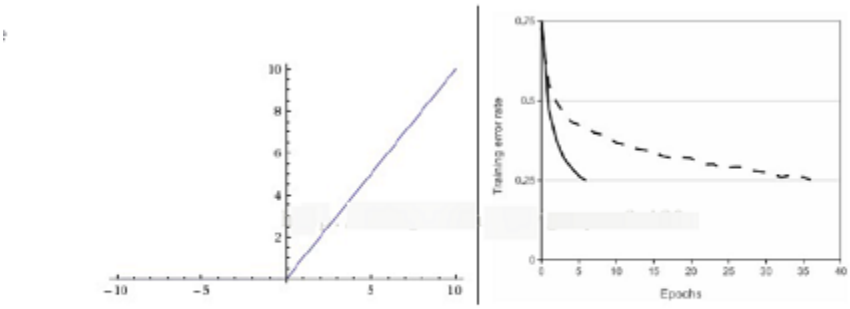
Why does AlexNet achieve better results?

1. **Relu activation function is used.**

Relu function: f (x) = max (0, x)



ReLU-based deep convolutional networks are trained several times faster than tanh and sigmoid- based networks. The following figure shows the number of iterations for a four-layer convolutional network based on CIFAR-10 that reached 25% training error in tanh and ReLU:



Left: Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. **Right:** A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

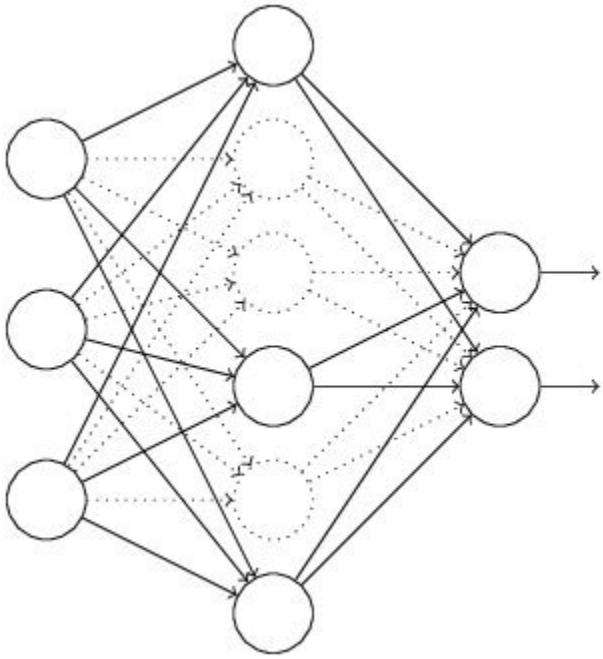
2. **Standardization (Local Response Normalization)**

After using ReLU f (x) = max (0, x), you will find that the value after the activation function has no range like the tanh and sigmoid functions, so a normalization will usually be done after ReLU, and the LRU is a steady proposal (Not sure here, it should be proposed?) One method in neuroscience is called "Lateral inhibition", which talks about the effect of active neurons on its surrounding neurons.

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^\beta$$

3. **Dropout**

Dropout is also a concept often said, which can effectively prevent overfitting of neural networks. Compared to the general linear model, a regular method is used to prevent the model from overfitting. In the neural network, Dropout is implemented by modifying the structure of the neural network itself. For a certain layer of neurons, randomly delete some neurons with a defined probability, while keeping the individuals of the input layer and output layer neurons unchanged, and then update the parameters according to the learning method of the neural network. In the next iteration, rerandom Remove some neurons until the end of training.



4. **Enhanced Data (Data Augmentation)**

In deep learning, when the amount of data is not large enough, there are generally 4 solutions:

> Data augmentation- artificially increase the size of the training set-create a batch of "new" data from existing data by means of translation, flipping, noise

> Regularization——The relatively small amount of data will cause the model to overfit, making the training error small and the test error particularly large. By adding a regular term after the Loss Function , the overfitting can be suppressed. The disadvantage is that a need is introduced Manually adjusted hyper-parameter.

Code Implementation

In [1]:
```
!pip install tflearn
```

```
Collecting tflearn
  Downloading tflearn-0.5.0.tar.gz (107 kB)
     ------------------------------------ 107.3/107.3 kB 1.6 MB/s eta 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Requirement already satisfied: numpy in d:\anaconda setup\lib\site-packages (from tflea
rn) (1.23.5)
Requirement already satisfied: six in d:\anaconda setup\lib\site-packages (from tflear
n) (1.16.0)
Requirement already satisfied: Pillow in d:\anaconda setup\lib\site-packages (from tfle
arn) (9.4.0)
Building wheels for collected packages: tflearn
  Building wheel for tflearn (setup.py): started
  Building wheel for tflearn (setup.py): finished with status 'done'
  Created wheel for tflearn: filename=tflearn-0.5.0-py3-none-any.whl size=127290 sha256
=35d3e450583535c181c918100373f68882a1b7fc62f4c54e0149a8b043bb17db
  Stored in directory: c:\users\shehryar gondal\appdata\local\pip\cache\wheels\5d\83\f7
\63e33ac9c0560f1dddb2ecff627b8ab6cb076d4b1996416be1
Successfully built tflearn
Installing collected packages: tflearn
Successfully installed tflearn-0.5.0
```

In [2]:
```
import tensorflow as tf
from tensorflow import keras
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
```

In [8]:
```
# Get Data
import tflearn.datasets.oxflower17 as oxflower17
from keras.utils import to_categorical

x, y = oxflower17.load_data()

x_train = x.astype('float32') / 255.0
y_train = to_categorical(y, num_classes=17)
```

In [10]:
```
print(x_train.shape)
print(y_train.shape)
```

```
(1360, 224, 224, 3)
(1360, 17)
```

In [5]:
```
# Create a sequential model
model = Sequential()

# 1st Convolutional Layer
model.add(Conv2D(filters=96, input_shape=(224,224,3), kernel_size=(11,11), strides=(4,4)
model.add(Activation('relu'))

# Pooling
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))
# Batch Normalisation before passing it to the next layer
model.add(BatchNormalization())

# 2nd Convolutional Layer
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# Pooling
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))
# Batch Normalisation
model.add(BatchNormalization())



# 3rd Convolutional Layer
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='valid'))
model.add(Activation('relu'))
# Batch Normalisation
model.add(BatchNormalization())

# 4th Convolutional Layer
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='valid'))
model.add(Activation('relu'))
# Batch Normalisation
model.add(BatchNormalization())


# 5th Convolutional Layer
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='valid'))
model.add(Activation('relu'))


# Pooling
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))
# Batch Normalisation
model.add(BatchNormalization())


# Passing it to a dense layer
model.add(Flatten())

# 1st Dense Layer
model.add(Dense(4096, input_shape=(224*224*3,)))
model.add(Activation('relu'))
# Add Dropout to prevent overfitting
model.add(Dropout(0.4))
# Batch Normalisation
model.add(BatchNormalization())

# 2nd Dense Layer
```

```
model.add(Dense(4096))
model.add(Activation('relu'))
# Add Dropout
model.add(Dropout(0.4))
# Batch Normalisation
model.add(BatchNormalization())

# Output Layer
model.add(Dense(17))
model.add(Activation('softmax'))

model.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/keras/layers/normalizat
ion/batch_normalization.py:581: _colocate_with (from tensorflow.python.framework.ops) i
s deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 54, 54, 96)        34944

 activation (Activation)     (None, 54, 54, 96)        0

 max_pooling2d (MaxPooling2D  (None, 26, 26, 96)       0
 )

 batch_normalization (BatchN  (None, 26, 26, 96)       384
 ormalization)

 conv2d_1 (Conv2D)           (None, 26, 26, 256)       614656

 activation_1 (Activation)   (None, 26, 26, 256)       0

 max_pooling2d_1 (MaxPooling  (None, 12, 12, 256)      0
 2D)

 batch_normalization_1 (Batc  (None, 12, 12, 256)      1024
 hNormalization)

 conv2d_2 (Conv2D)           (None, 10, 10, 384)       885120

 activation_2 (Activation)   (None, 10, 10, 384)       0

 batch_normalization_2 (Batc  (None, 10, 10, 384)      1536
 hNormalization)

 conv2d_3 (Conv2D)           (None, 8, 8, 384)         1327488

 activation_3 (Activation)   (None, 8, 8, 384)         0

 batch_normalization_3 (Batc  (None, 8, 8, 384)        1536
 hNormalization)

 conv2d_4 (Conv2D)           (None, 6, 6, 256)         884992

 activation_4 (Activation)   (None, 6, 6, 256)         0

 max_pooling2d_2 (MaxPooling  (None, 2, 2, 256)        0
 2D)

 batch_normalization_4 (Batc  (None, 2, 2, 256)        1024
 hNormalization)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 4096)              4198400

 activation_5 (Activation)   (None, 4096)              0

 dropout (Dropout)           (None, 4096)              0

 batch_normalization_5 (Batc  (None, 4096)             16384
 hNormalization)
```

```
dense_1 (Dense)              (None, 4096)              16781312

activation_6 (Activation)    (None, 4096)              0

dropout_1 (Dropout)          (None, 4096)              0

batch_normalization_6 (Batc  (None, 4096)              16384
hNormalization)

dense_2 (Dense)              (None, 17)                69649

activation_7 (Activation)    (None, 17)                0

=================================================================
Total params: 24,834,833
Trainable params: 24,815,697
Non-trainable params: 19,136
_________________________________________________________________
```

In [11]:
```
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

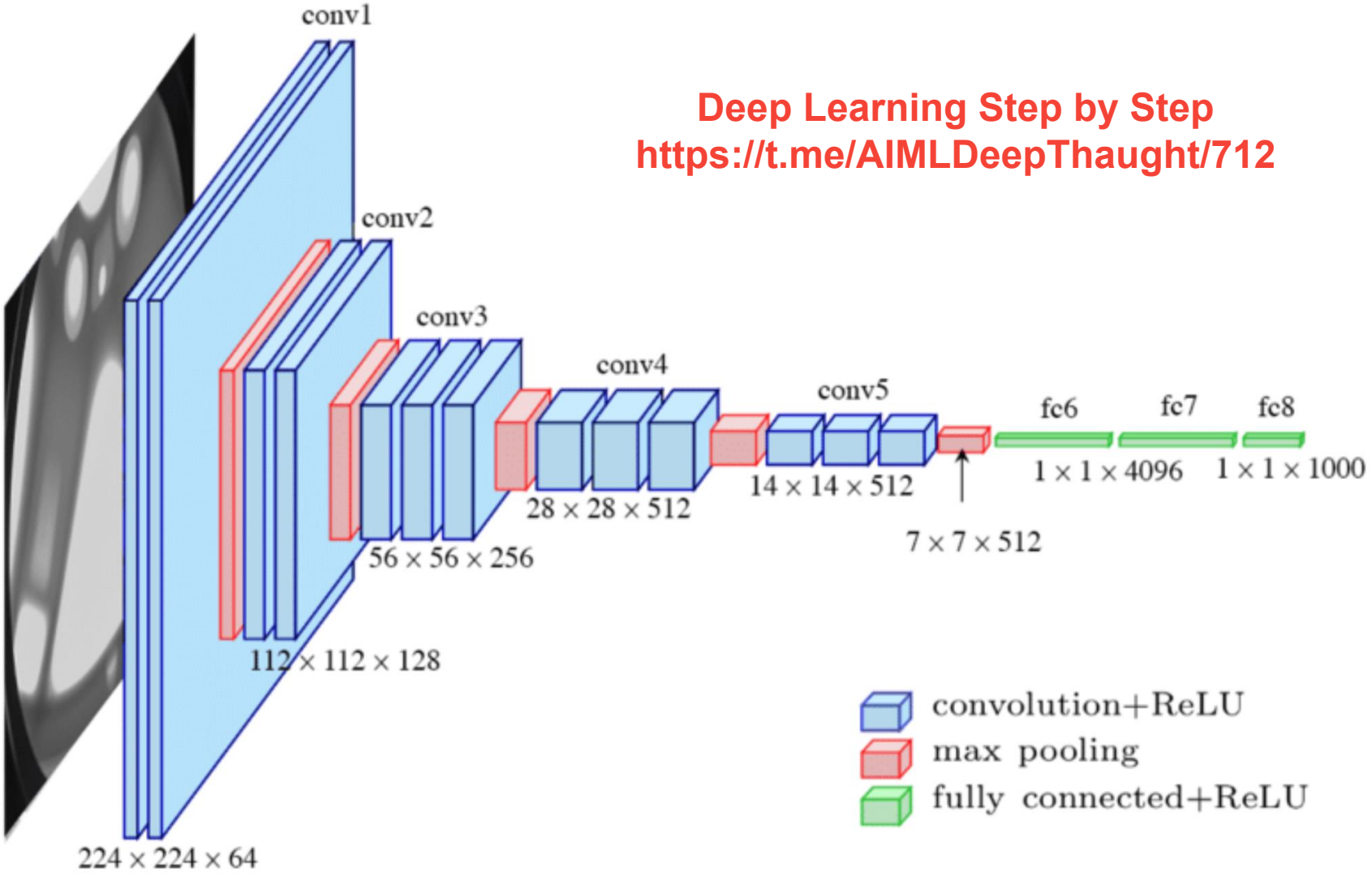
In [12]:
```
# Train
model.fit(x_train, y_train, batch_size=64, epochs=5, verbose=1,validation_split=0.2, shu
```

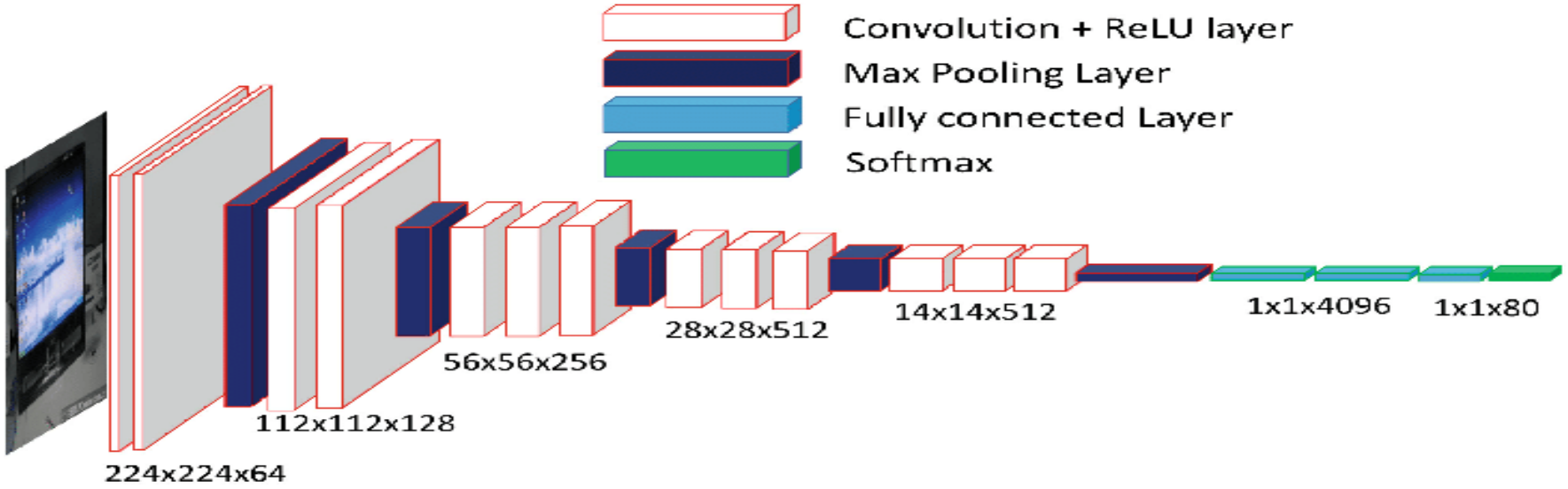
```
Train on 1088 samples, validate on 272 samples
Epoch 1/5
1088/1088 [==============================] - ETA: 0s - loss: 3.6493 - acc: 0.2858

/usr/local/lib/python3.10/dist-packages/keras/engine/training_v1.py:2335: UserWarning:
`Model.state_updates` will be removed in a future version. This property should not be
used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates

1088/1088 [==============================] - 11s 10ms/sample - loss: 3.6493 - acc: 0.28
58 - val_loss: 3.1504 - val_acc: 0.0699
Epoch 2/5
1088/1088 [==============================] - 2s 2ms/sample - loss: 2.0367 - acc: 0.4357
- val_loss: 5.0040 - val_acc: 0.0699
Epoch 3/5
1088/1088 [==============================] - 2s 2ms/sample - loss: 1.7029 - acc: 0.5055
- val_loss: 7.8723 - val_acc: 0.0551
Epoch 4/5
1088/1088 [==============================] - 2s 2ms/sample - loss: 1.5586 - acc: 0.5441
- val_loss: 5.6248 - val_acc: 0.0699
Epoch 5/5
1088/1088 [==============================] - 2s 2ms/sample - loss: 1.3488 - acc: 0.6094
- val_loss: 7.2296 - val_acc: 0.0699
```

Out[12]:
```
<keras.callbacks.History at 0x7fe7577e01f0>
```

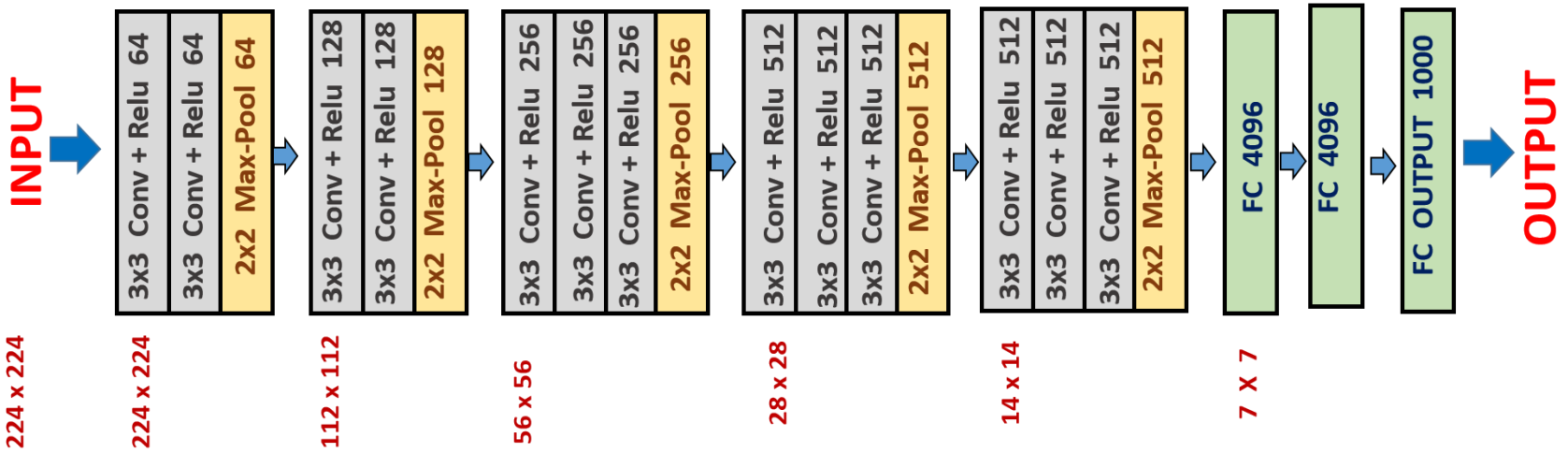
In [ ]:
```

```

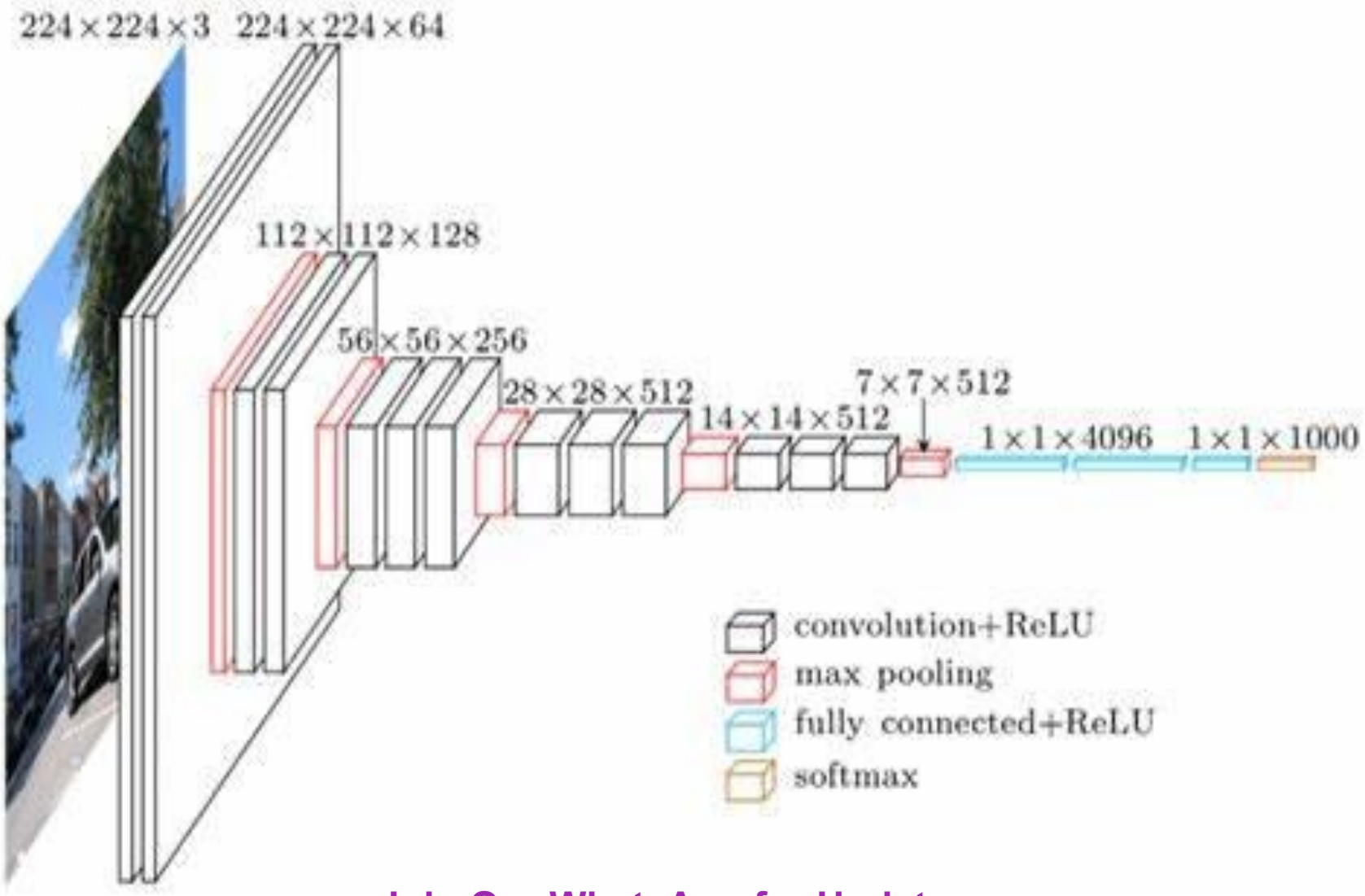
VGG-Net




Convolution + ReLU layer
Max Pooling Layer
Fully connected Layer
Softmax
224x224x64
112x112x128
56x56x256
28x28x512
14x14x512
1x1x4096
1x1x80

VGG-16


INPUT
3x3 Conv + Relu 64
3x3 Conv + Relu 64
2x2 Max-Pool 64
3x3 Conv + Relu 128
3x3 Conv + Relu 128
2x2 Max-Pool 128
3x3 Conv + Relu 256
3x3 Conv + Relu 256
3x3 Conv + Relu 256
2x2 Max-Pool 256
3x3 Conv + Relu 512
3x3 Conv + Relu 512
3x3 Conv + Relu 512
2x2 Max-Pool 512
3x3 Conv + Relu 512
3x3 Conv + Relu 512
3x3 Conv + Relu 512
2x2 Max-Pool 512
FC 4096
FC 4096
FC OUTPUT 1000
OUTPUT
224 x 224
224 x 224
112 x 112
56 x 56
28 x 28
14 x 14
7 X 7



VGG-Net

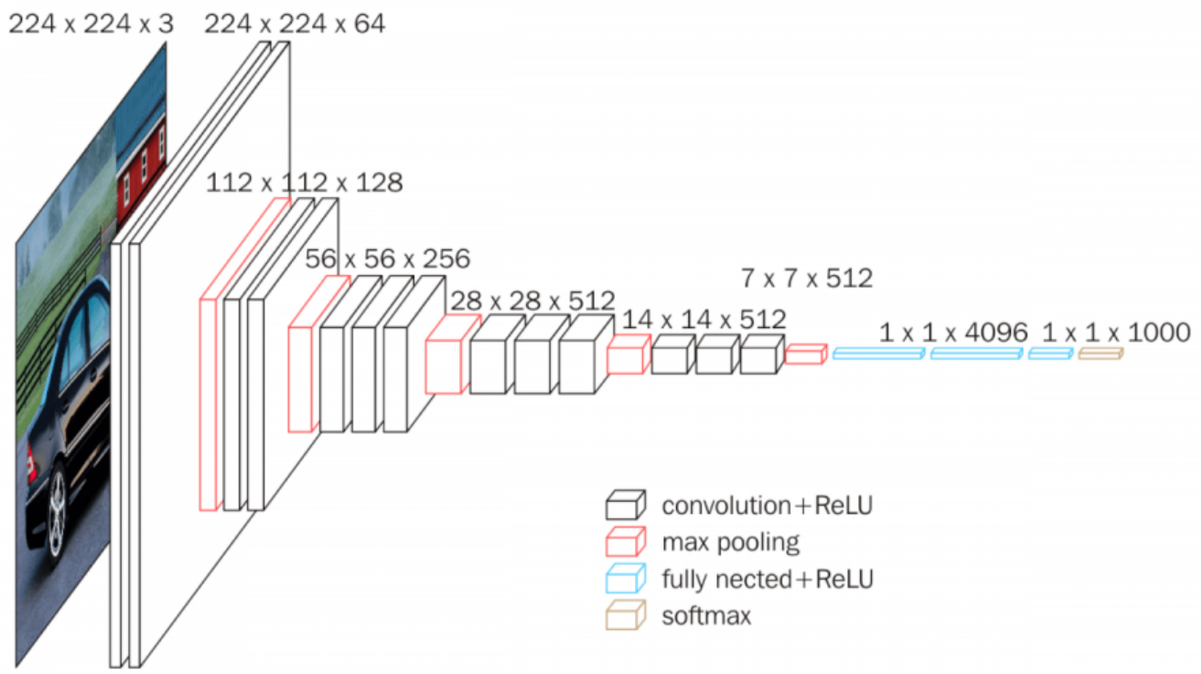
Introduction

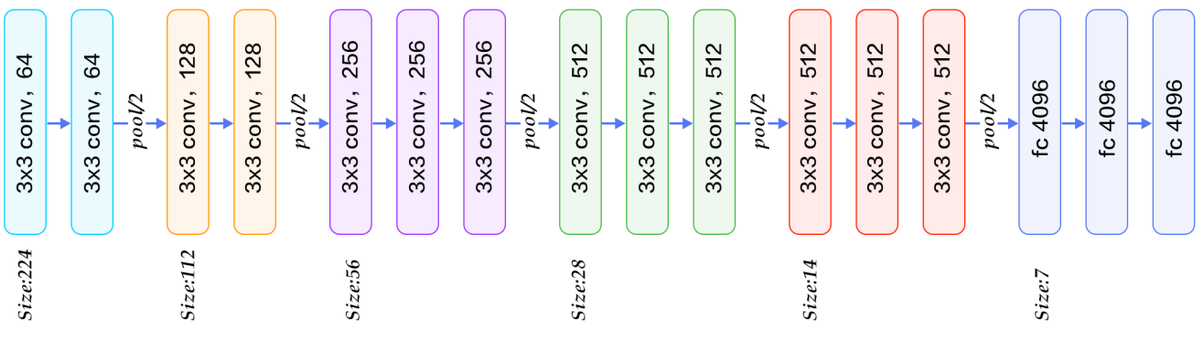
> The full name of VGG is the Visual Geometry Group, which belongs to the Department of Science and Engineering of Oxford University. It has released a series of convolutional network models beginning with VGG, which can be applied to face recognition and image classification, from VGG16 to VGG19. The original purpose of VGG's research on the depth of convolutional networks is to understand how the depth of convolutional networks affects the accuracy and accuracy of large-scale image classification and recognition. -Deep-16 CNN), in order to deepen the number of network layers and to avoid too many parameters, a small 3x3 convolution kernel is used in all layers.

[Network Structure of VGG19 (http://ethereon.github.io/netscope/#/gist/dc5003de6943ea5a6b8b)](http://ethereon.github.io/netscope/#/gist/dc5003de6943ea5a6b8b)

The network structure

> The input of VGG is set to an RGB image of 224x244 size. The average RGB value is calculated for all images on the training set image, and then the image is input as an input to the VGG convolution network. A 3x3 or 1x1 filter is used, and the convolution step is fixed. . There are 3 VGG fully connected layers, which can vary from VGG11 to VGG19 according to the total number of convolutional layers + fully connected layers. The minimum VGG11 has 8 convolutional layers and 3 fully connected layers. The maximum VGG19 has 16 convolutional layers. +3 fully connected layers. In addition, the VGG network is not followed by a pooling layer behind each convolutional layer, or a total of 5 pooling layers distributed under different convolutional layers. The following figure is VGG Structure diagram:

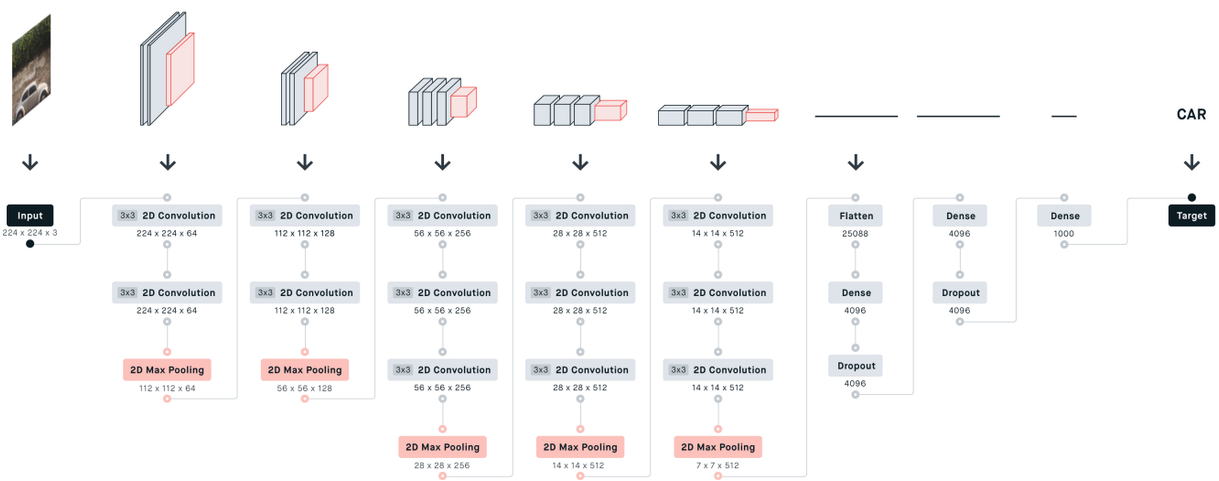




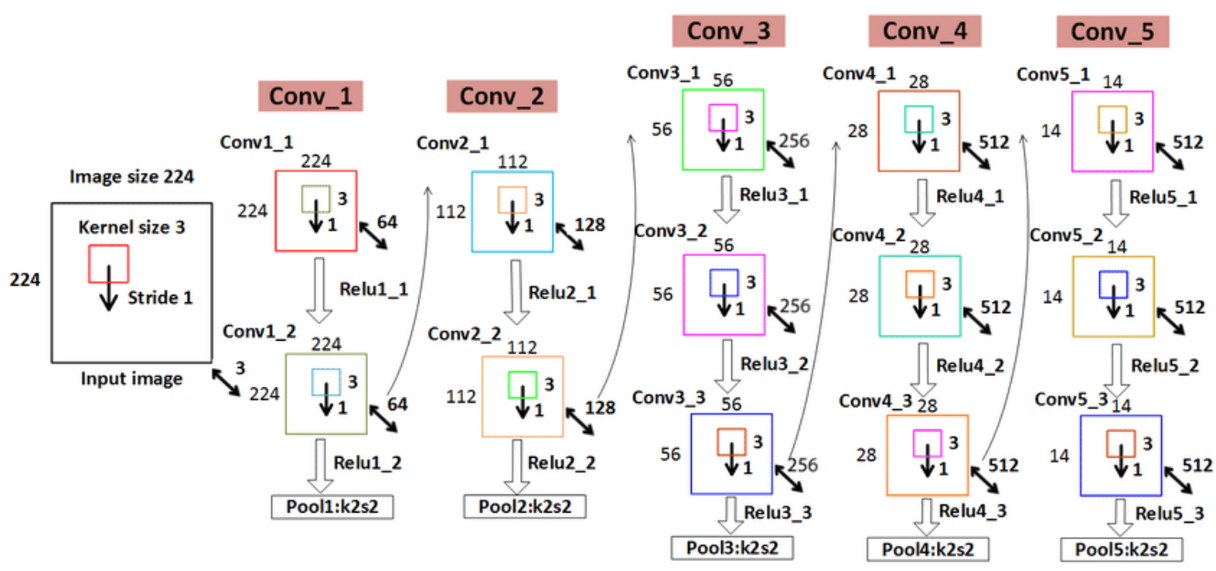
> VGG16 contains 16 layers and VGG19 contains 19 layers. A series of VGGs are exactly the same in the last three fully connected layers. The overall structure includes 5 sets of convolutional layers, followed by a MaxPool. The difference is that more and more cascaded convolutional layers are included in the five sets of convolutional layers .

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	224 x 224 x 3	-	-	-
1	2 X Convolution	64	224 x 224 x 64	3x3	1	relu
	Max Pooling	64	112 x 112 x 64	3x3	2	relu
3	2 X Convolution	128	112 x 112 x 128	3x3	1	relu
	Max Pooling	128	56 x 56 x 128	3x3	2	relu
5	2 X Convolution	256	56 x 56 x 256	3x3	1	relu
	Max Pooling	256	28 x 28 x 256	3x3	2	relu
7	3 X Convolution	512	28 x 28 x 512	3x3	1	relu
	Max Pooling	512	14 x 14 x 512	3x3	2	relu
10	3 X Convolution	512	14 x 14 x 512	3x3	1	relu
	Max Pooling	512	7 x 7 x 512	3x3	2	relu
13	FC	-	25088	-	-	relu
14	FC	-	4096	-	-	relu
15	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

Each convolutional layer in AlexNet contains only one convolution, and the size of the convolution kernel is 7 * 7 ,. In VGGNet, each convolution layer contains 2 to 4 convolution operations. The size of the convolution kernel is 3 * 3, the convolution step size is 1, the pooling kernel is 2 * 2, and the step size is 2. The most obvious improvement of VGGNet is to reduce the size of the convolution kernel and increase the number of convolution layers.



Using multiple convolution layers with smaller convolution kernels instead of a larger convolution layer with convolution kernels can reduce parameters on the one hand, and the author believes that it is equivalent to more non-linear mapping, which increases the Fit expression ability.



> Two consecutive 3 * 3 convolutions are equivalent to a 5 * 5 receptive field, and three are equivalent to 7 * 7. The advantages of using three 3 * 3 convolutions instead of one 7 * 7 convolution are twofold : one, including three ReLu layers instead of one , makes the decision function more discriminative; and two, reducing parameters . For example, the input and output are all C channels. 3 convolutional layers using 3 * 3 require 3 (3 * 3 * C * C) = 27 * C * C, and 1 convolutional layer using 7 * 7 requires 7 * 7 * C * C = 49C * C. This can be seen as applying a kind of regularization to the 7 * 7 convolution, so that it is decomposed into three 3 * 3 convolutions.

> The 1 * 1 convolution layer is mainly to increase the non-linearity of the decision function without affecting the receptive field of the convolution layer. Although the 1 * 1 convolution operation is linear, ReLu adds non-linearity.

Network Configuration

> Table 1 shows all network configurations. These networks follow the same design principles, but differ in depth.

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

> This picture is definitely used when introducing VGG16. This picture contains a lot of information. My interpretation here may be limited. If you have any supplements, please leave a message.

- **Number 1** : This is a comparison chart of 6 networks. From A to E, the network is getting deeper. Several layers have been added to verify the effect.
- **Number 2** : Each column explains the structure of each network in detail.
- **Number 3**: This is a correct way to do experiments, that is, use the simplest method to solve the problem , and then gradually optimize for the problems that occur.

Network A: First mention a shallow network, this network can easily converge on ImageNet. And then?

Network A-LRN: Add something that someone else (AlexNet) has experimented to say is effective (LRN), but it seems useless. And then?

Network B: Then try adding 2 layers? Seems to be effective. And then?

Network C: Add two more layers of 1 * 1 convolution, and it will definitely converge. The effect seems to be better. A little excited. And then?

Network D: Change the 1 * 1 convolution kernel to 3 * 3. Try it. The effect has improved again. Seems to be the best (2014).

Training

The optimization method is a stochastic gradient descent SGD + momentum (0.9) with momentum. The batch size is 256.

Regularization : L2 regularization is used, and the weight decay is 5e-4. Dropout is after the first two fully connected layers, p = 0.5.

Although it is deeper and has more parameters than the AlexNet network, we speculate that VGGNet can converge in less cycles for two reasons: one, the greater depth and smaller convolutions bring implicit regularization ; Second, some layers of pre-training.

Parameter initialization : For a shallow A network, parameters are randomly initialized, the weight w is sampled from N (0, 0.01), and the bias is initialized to 0. Then, for deeper networks, first the first four convolutional layers and three fully connected layers are initialized with the parameters of the A network. However, it was later discovered that it is also possible to directly initialize it without using pre-trained parameters.

In order to obtain a 224 * 224 input image, each rescaled image is randomly cropped in each SGD iteration. In order to enhance the data set, the cropped image is also randomly flipped horizontally and RGB color shifted.

Summary of VGGNet improvement points

1. A smaller 3 * 3 convolution kernel and a deeper network are used . The stack of two 3 * 3 convolution kernels is relative to the field of view of a 5 * 5 convolution kernel, and the stack of three 3 * 3 convolution kernels is equivalent to the field of view of a 7 * 7 convolution kernel. In this way, there can be fewer parameters (3 stacked 3 * 3 structures have only 7 * 7 structural parameters (3 * 3 * 3) / (7 * 7) = 55%); on the other hand, they have more The non-linear transformation increases the ability of CNN to learn features.
2. In the convolutional structure of VGGNet, a 1 * 1 convolution kernel is introduced. Without affecting the input and output dimensions, non-linear transformation is introduced to increase the expressive power of the network and reduce the amount of calculation.
3. During training, first train a simple (low-level) VGGNet A-level network, and then use the weights of the A network to initialize the complex models that follow to speed up the convergence of training .

Some basic questions

Q1: Why can 3 3x3 convolutions replace 7x7 convolutions?

Answer 1

3 3x3 convolutions, using 3 non-linear activation functions, increasing non-linear expression capabilities, making the segmentation plane more separable Reduce the number of parameters. For the convolution kernel of C channels, 7x7 contains parameters , and the number of 3 3x3 parameters is greatly reduced.

Q2: The role of 1x1 convolution kernel

Answer 2

Increase the nonlinearity of the model without affecting the receptive field 1x1 winding machine is equivalent to linear transformation, and the non-linear activation function plays a non-linear role

Q3: The effect of network depth on results (in the same year, Google also independently released the network GoogleNet with a depth of 22 layers)

Answer 3

VGG and GoogleNet models are deep Small convolution VGG only uses 3x3, while GoogleNet uses 1x1, 3x3, 5x5, the model is more complicated (the model began to use a large convolution kernel to reduce the calculation of the subsequent machine layer)

Code Implementation

From Scratch

In [2]:
```
!pip install tflearn
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) https://us-pyth
on.pkg.dev/colab-wheels/public/simple/ (https://us-python.pkg.dev/colab-wheels/public/s
imple/)
Collecting tflearn
  Downloading tflearn-0.5.0.tar.gz (107 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 107.3/107.3 kB 11.9 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from t
flearn) (1.22.4)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from tfl
earn) (1.16.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from
tflearn) (8.4.0)
Building wheels for collected packages: tflearn
  Building wheel for tflearn (setup.py) ... done
  Created wheel for tflearn: filename=tflearn-0.5.0-py3-none-any.whl size=127283 sha256
=f270efc507b05ff73c8125856b43368ccef0ca18caf050fe770d806a4f8b359f
  Stored in directory: /root/.cache/pip/wheels/55/fb/7b/e06204a0ceefa45443930b9a250cb5e
be31def0e4e8245a465
Successfully built tflearn
Installing collected packages: tflearn
Successfully installed tflearn-0.5.0
```

In [1]:
```
from tensorflow import keras
import keras,os
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPool2D , Flatten
from keras.preprocessing.image import ImageDataGenerator
import numpy as np
```

In [3]:
```
# Get Data
import tflearn.datasets.oxflower17 as oxflower17
from keras.utils import to_categorical

x, y = oxflower17.load_data()

x_train = x.astype('float32') / 255.0
y_train = to_categorical(y, num_classes=17)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/compa
t/v2_compat.py:107: disable_resource_variables (from tensorflow.python.ops.variable_sco
pe) is deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term

Downloading Oxford 17 category Flower Dataset, Please wait...

100.0% 60276736 / 60270631

Succesfully downloaded 17flowers.tgz 60270631 bytes.
File Extracted
Starting to parse images...
Parsing Done!
```

In [4]:
```
print(x_train.shape)
print(y_train.shape)
```

```
(1360, 224, 224, 3)
(1360, 17)
```

In [8]:
```
model = Sequential()
model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3),padding="same", ac
model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Flatten())
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=17, activation="softmax"))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_13 (Conv2D)          (None, 224, 224, 64)      1792

 conv2d_14 (Conv2D)          (None, 224, 224, 64)      36928

 max_pooling2d_5 (MaxPooling  (None, 112, 112, 64)     0
 2D)

 conv2d_15 (Conv2D)          (None, 112, 112, 128)     73856

 conv2d_16 (Conv2D)          (None, 112, 112, 128)     147584

 max_pooling2d_6 (MaxPooling  (None, 56, 56, 128)      0
 2D)

 conv2d_17 (Conv2D)          (None, 56, 56, 256)       295168

 conv2d_18 (Conv2D)          (None, 56, 56, 256)       590080

 conv2d_19 (Conv2D)          (None, 56, 56, 256)       590080

 max_pooling2d_7 (MaxPooling  (None, 28, 28, 256)      0
 2D)

 conv2d_20 (Conv2D)          (None, 28, 28, 512)       1180160

 conv2d_21 (Conv2D)          (None, 28, 28, 512)       2359808

 conv2d_22 (Conv2D)          (None, 28, 28, 512)       2359808

 max_pooling2d_8 (MaxPooling  (None, 14, 14, 512)      0
 2D)

 conv2d_23 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_24 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_25 (Conv2D)          (None, 14, 14, 512)       2359808

 max_pooling2d_9 (MaxPooling  (None, 7, 7, 512)        0
 2D)

 flatten_1 (Flatten)         (None, 25088)             0

 dense_3 (Dense)             (None, 4096)              102764544

 dense_4 (Dense)             (None, 4096)              16781312

 dense_5 (Dense)             (None, 17)                69649

=================================================================
Total params: 134,330,193
Trainable params: 134,330,193
Non-trainable params: 0
_________________________________________________________________
```

In [9]:
```
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

In [10]:
```
# Train
model.fit(x_train, y_train, batch_size=64, epochs=5, verbose=1,validation_split=0.2, shu
```

```
Train on 1088 samples, validate on 272 samples
Epoch 1/5
1088/1088 [==============================] - ETA: 0s - loss: 2.8376 - acc: 0.0478

/usr/local/lib/python3.10/dist-packages/keras/engine/training_v1.py:2335: UserWarning:
`Model.state_updates` will be removed in a future version. This property should not be
used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates

1088/1088 [==============================] - 44s 41ms/sample - loss: 2.8376 - acc: 0.04
78 - val_loss: 2.8353 - val_acc: 0.0331
Epoch 2/5
1088/1088 [==============================] - 15s 13ms/sample - loss: 2.8342 - acc: 0.05
42 - val_loss: 2.8366 - val_acc: 0.0331
Epoch 3/5
1088/1088 [==============================] - 15s 14ms/sample - loss: 2.8335 - acc: 0.06
53 - val_loss: 2.8365 - val_acc: 0.0331
Epoch 4/5
1088/1088 [==============================] - 15s 14ms/sample - loss: 2.8332 - acc: 0.06
53 - val_loss: 2.8387 - val_acc: 0.0331
Epoch 5/5
1088/1088 [==============================] - 15s 14ms/sample - loss: 2.8330 - acc: 0.06
53 - val_loss: 2.8391 - val_acc: 0.0331
```

Out[10]:
```
<keras.callbacks.History at 0x7fe540c4b4f0>
```

In [ ]:
```

```

VGG Pretrained

In [11]:
```
# download the data from g drive

import gdown
url = "https://drive.google.com/file/d/12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP/view?usp=sharin
file_id = url.split("/")[-2]
print(file_id)
prefix = 'https://drive.google.com/uc?/export=download&id='
gdown.download(prefix+file_id, "catdog.zip")
```

```
12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP

Downloading...
From: https://drive.google.com/uc?/export=download&id=12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP
(https://drive.google.com/uc?/export=download&id=12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP)
To: /content/catdog.zip
100%|██████████| 9.09M/9.09M [00:00<00:00, 118MB/s]
```

Out[11]:
```
'catdog.zip'
```

In [12]:
```
!unzip catdog.zip
```

```
Archive:  catdog.zip
   creating: train/
   creating: train/Cat/
  inflating: train/Cat/0.jpg
  inflating: train/Cat/1.jpg
  inflating: train/Cat/2.jpg
  inflating: train/Cat/cat.2405.jpg
  inflating: train/Cat/cat.2406.jpg
  inflating: train/Cat/cat.2436.jpg
  inflating: train/Cat/cat.2437.jpg
  inflating: train/Cat/cat.2438.jpg
  inflating: train/Cat/cat.2439.jpg
  inflating: train/Cat/cat.2440.jpg
  inflating: train/Cat/cat.2441.jpg
  inflating: train/Cat/cat.2442.jpg
  inflating: train/Cat/cat.2443.jpg
  inflating: train/Cat/cat.2444.jpg
  inflating: train/Cat/cat.2445.jpg
  inflating: train/Cat/cat.2446.jpg
```

In [14]:

```
from tensorflow import keras
from keras.applications.vgg16 import VGG16, preprocess_input
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten

# Set the path to your training and validation data
train_data_dir = '/content/train'
validation_data_dir = '/content/validation'

# Set the number of training and validation samples
num_train_samples = 2000
num_validation_samples = 800

# Set the number of epochs and batch size
epochs = 5
batch_size = 16

# Load the VGG16 model without the top layer
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Create a new model
model = Sequential()

# Add the base model as a layer
model.add(base_model)

# Add custom layers on top of the base model
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Preprocess the training and validation data
train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
validation_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='binary')

validation_generator = validation_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='binary')

# Train the model
model.fit(
    train_generator,
    steps_per_epoch=num_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=num_validation_samples // batch_size)

# Save the trained model
model.save('dog_cat_classifier.h5')
```

```
Found 337 images belonging to 2 classes.
Found 59 images belonging to 2 classes.
Epoch 1/5
125/125 [==============================] - ETA: 0s - batch: 62.0000 - size: 15.2800 - l
oss: 2.4077 - acc: 0.9665
```

```
/usr/local/lib/python3.10/dist-packages/keras/engine/training_v1.py:2335: UserWarning:
`Model.state_updates` will be removed in a future version. This property should not be
used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates

125/125 [==============================] - 19s 138ms/step - batch: 62.0000 - size: 15.2
800 - loss: 2.4086 - acc: 0.9665 - val_loss: 10.4672 - val_acc: 0.9297
Epoch 2/5
125/125 [==============================] - 17s 139ms/step - batch: 62.0000 - size: 15.2
800 - loss: 0.1754 - acc: 0.9958 - val_loss: 3.5723 - val_acc: 0.9311
Epoch 3/5
125/125 [==============================] - 17s 135ms/step - batch: 62.0000 - size: 15.4
000 - loss: 0.0877 - acc: 0.9984 - val_loss: 4.5018 - val_acc: 0.9473
Epoch 4/5
125/125 [==============================] - 17s 134ms/step - batch: 62.0000 - size: 15.2
800 - loss: 0.0918 - acc: 0.9969 - val_loss: 3.2619 - val_acc: 0.9824
Epoch 5/5
125/125 [==============================] - 16s 129ms/step - batch: 62.0000 - size: 15.2
800 - loss: 4.0183e-07 - acc: 1.0000 - val_loss: 2.7819 - val_acc: 0.9824
```

In [ ]:

In [ ]:

Inception ¶

Also known as GoogLeNet , it is a 22-layer network that won the 2014 ILSVRC Championship.

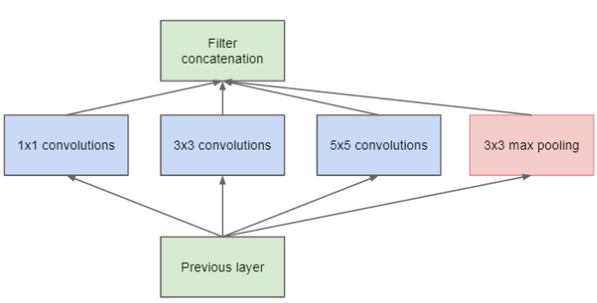
1. The original intention of the design is to expand the width and depth on its basis .
2. which is designed motives derived from improving the performance of the depth of the network generally can increase the size of the network and increase the size of the data set to increase, but at the same time cause the network parameters and easily fit through excessive , computing resources inefficient and The production of high-quality data sets is an expensive issue.
3. Its design philosophy is to change the full connection to a sparse architecture and try to change it to a sparse architecture inside the convolution.
4. The main idea is to design an inception module and increase the depth and width of the network by continuously copying these inception modules , but GooLeNet mainly extends these inception modules in depth.

There are four parallel channels in each inception module , and concat is performed at the end of the channel .

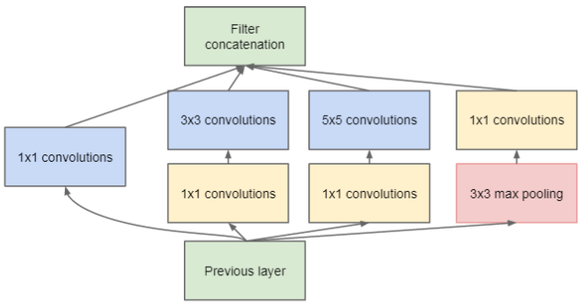
1x1 conv is mainly used to reduce the dimensions in the article to avoid calculation bottlenecks. It also adds additional softmax loss to some branches of the previous network layer to avoid the problem of gradient disappearance.

Four parallel channels:

- 1x1 conv: Borrowed from [Network in Network], the input feature map can be reduced in dimension and upgraded without too much loss of the input spatial information;
- 1x1conv followed by 3x3 conv: 3x3 conv increases the receptive field of the feature map, and changes the dimension through 1x1conv;
- 1x1 conv followed by 5x5 conv: 5x5 conv further increases the receptive field of the feature map, and changes the dimensions through 1x1 conv;
- 3x3 max pooling followed by 1x1 conv: The author believes that although the pooling layer will lose space information, it has been effectively applied in many fields, which proves its effectiveness, so a parallel channel is added, and it is changed by 1x1 conv Its output dimension.

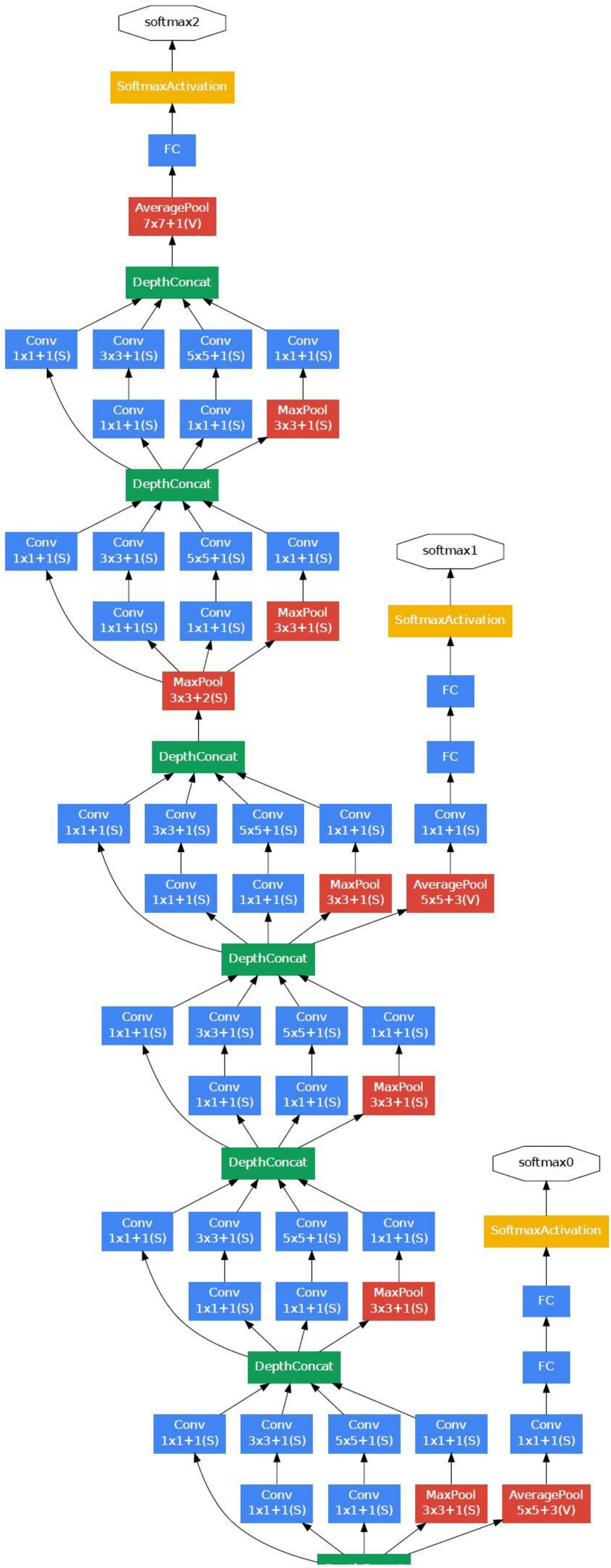


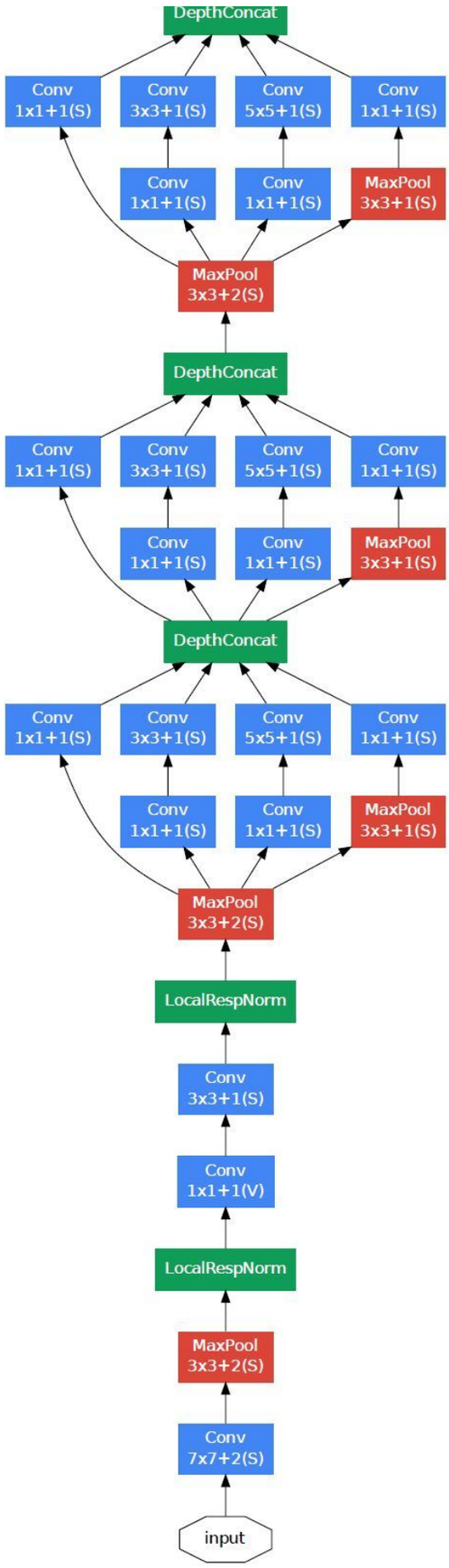
(a) Inception module, naïve version



(b) Inception module with dimension reductions

Complete network design : -


softmax2
SoftmaxActivation
FC
AveragePool
7x7+1(V)
DepthConcat
Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
DepthConcat
Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
softmax1
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
SoftmaxActivation
FC
MaxPool
3x3+2(S)
FC
DepthConcat
Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
AveragePool
5x5+3(V)
DepthConcat
Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
DepthConcat
softmax0
Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
SoftmaxActivation
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
FC
DepthConcat
FC
Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
AveragePool
5x5+3(V)



Two ways to improve network performance:

The most direct way to improve the performance of deep neural networks is to increase their size . This includes depth, the number of levels, and their width, the size of each level unit .

Another easy and safe way is to **increase the size of the training data**.

However, *both methods have two disadvantages* .

Larger models mean more parameters, which makes it easier for the network to overfit , especially when the number of label samples in the training data set is limited.

At the same time, because the production of high-quality training sets is tricky and expensive ,especially when some human experts do it , there is a large error rate . As shown below.

(a) Siberian husky

(b) Eskimo dog

Figure 1: Two distinct classes from the 1000 classes of the ILSVRC 2014 classification challenge.

Another shortcoming is that uniformly increasing the size of the network will increase the use of computing resources . For example, in a deep network, if two convolutions are chained, any unified improvement of their convolution kernels will cause demand for resources.

Power increase: If the increased capacity is inefficient, for example, if most of the weights end with 0 , then a lot of computing resources are wasted. But because the computing resources are always limited, an effective computational distribution always tends to increase the size of the model indiscriminately, and even the main objective goal is to improve the performance of the results.

The basic method to solve these two problems is to finally change the fully connected network to a sparse architecture, even inside the convolution.

The details of the GooLeNet network layer are shown in the following table:

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

To sum up:

- 128 1x1 convolution kernels are used to reduce dimensions and modify linear activation units
- A fully connected layer of 1024 units and a modified linear activation unit;
- A dropout layer that drops neuron connections with a 70% probability;
- A linear layer with softmax loss as classification Predict 1000 categories, but removed during the inference phase

Training Methodology

> The momentum is set to 0.9 and the learning rate is set to decrease by 4% every 8 epochs.

> Seven models were trained . To make the problem more detailed, some models were trained on small crops, and some were trained on large crops .

> The factors that make the model train well include : the sampling of patches of various sizes in the image , the size of which is evenly distributed between 8% and 100%, and the aspect ratio between 3/4 and 4/3.

> **Illumination changes have an effect on avoiding overfitting.**

> **Later, random interpolation is used to resize the image.**

Inception-v2(2015)

This architecture is a landmark in the development of deep network models . The most prominent contribution is to propose a normalized Batch Normalization layer to unify the output range of the network. It is fixed in a relatively uniform range. If the BN layer is not added, the value range of the network input and output of each layer is greatly different, so the size of the learning rate will be different. The BN layer avoids this situation This accelerates the training of the network and gives the network regular terms to a certain extent , reducing the degree of overfitting of the network. In the subsequent development of network models, most models have more or less added BN layers to the model.

In this paper, the BN layer is standardized before being input to the activation function. At the same time, VGG uses 2 3x3 convs instead of 5x5 convs in the inception module to reduce the amount of parameters and speed up the calculation.

Algorithm advantages:

1. **Improved learning rate** : In the BN model, a higher learning rate is used to accelerate training convergence, but it will not cause other effects. Because if the scale of each layer is different, then the learning rate required by each layer is different. The scale of the same layer dimension often also needs different learning rates. Usually, the minimum learning is required to ensure the loss function to decrease, but The BN layer keeps the scale of each layer and dimension consistent, so you can directly use a higher learning rate for optimization.
2. **Remove the dropout layer** : The BN layer makes full use of the goals of the dropout layer. Remove the dropout layer from the BN-Inception model, but no overfitting will occur.
3. **Decrease the attenuation coefficient of L2 weight** : Although the L2 loss controls the overfitting of the Inception model, the loss of weight has been reduced by five times in the BN-Inception model.
4. **Accelerate the decay of the learning rate** : When training the Inception model, we let the learning rate decrease exponentially. Because our network is faster than Inception, we will increase the speed of reducing the learning rate by 6 times.
5. **Remove the local response layer** : Although this layer has a certain role, but after the BN layer is added, this layer is not necessary.
6. **Scramble training samples more thoroughly** : We scramble training samples, which can prevent the same samples from appearing in a mini-batch. This can improve the accuracy of the validation set by 1%, which is the advantage of the BN layer as a regular term. In our method, random selection is more effective when the model sees different samples each time.
7. **To reduce image distortion**: Because BN network training is faster and observes each training sample less often, we want the model to see a more realistic image instead of a distorted image.

Inception-v3-2015

This architecture focuses, how to use the convolution kernel two or more smaller size of the convolution kernel to replace, but also the introduction of **asymmetrical layers i.e. a convolution dimensional convolution** has also been proposed for pooling layer Some remedies that can cause loss of spatial information; there are ideas such as **label-smoothing , BN-ahxiliary** .

Experiments were performed on inputs with different resolutions . The results show that although low-resolution inputs require more time to train, the accuracy and high-resolution achieved are not much different.

The computational cost is reduced while improving the accuracy of the network.

General Design Principles

We will describe some design principles that have been proposed through extensive experiments with different architectural designs for convolutional networks. At this point, full use of the following principles can be guessed, and some additional experiments in the future will be necessary to estimate their accuracy and effectiveness.

1. **Prevent bottlenecks in characterization** . The so-called bottleneck of feature description is that a large proportion of features are compressed in the middle layer (such as using a pooling operation). This operation will cause the loss of feature space information and the loss of features. Although the operation of pooling in CNN is important, there are some methods that can be used to avoid this loss as much as possible (I note: later hole convolution operations).
2. **The higher the dimensionality of the feature, the faster the training converges** . That is, the independence of features has a great relationship with the speed of model convergence. The more independent features, the more thoroughly the input feature information is decomposed. It is easier to converge if the correlation is strong. Hebbin principle : fire together, wire together.
3. **Reduce the amount of calculation through dimensionality reduction** . In v1, the feature is first reduced by 1x1 convolutional dimensionality reduction. There is a certain correlation between different dimensions. Dimension reduction can be understood as a lossless or low-loss compression. Even if the dimensions are reduced, the correlation can still be used to restore its original information.
4. **Balance the depth and width of the network** . Only by increasing the depth and width of the network in the same proportion can the performance of the model be maximized.

Factorizing Convolutions with Large Filter Size

GooLeNet uses many dimensionality reduction methods, which has achieved certain results. Consider the example of a 1x1 convolutional layer used to reduce dimensions before a 3x3 convolutional layer. In the network, we expect the network to be highly correlated between the output neighboring elements at the activation function. Therefore, we can reduce their activation values before aggregation , which should generate similar local expression descriptions.

This paper explores experiments to decompose the network layer into different factors under different settings in order to improve the computational efficiency of the method . Because the Inception network is fully convolutional, each weight value corresponds to a product operation each time it is activated.

Therefore, any reduction in computational cost will result in a reduction in parameters. This means that we can use some suitable decomposition factors to reduce the parameters and thus speed up the training.

3.1 **Factorizing Convolutions with Large Filter Size**

With the same number of convolution kernels, larger convolution kernels (such as 5x5 or 7x7) are more expensive to calculate than 3x3 convolution kernels , which is about a multiple of 25/9 = 2.78. Of course, the 5x5 convolution kernel can obtain more correlations between the information and activation units in the previous network, but under the premise of huge consumption of computing resources, a physical reduction in the size of the convolution kernel still appears.

However, we still want to know whether a 5x5 convolutional layer can be replaced by a multi-layer convolutional layer with fewer parameters when the input and output sizes are consistent . If we scale the calculation map of 5x5 convolution, we can see that each output is like a small fully connected network sliding on the input window with a size of 5x5. Refer to Figure 1.

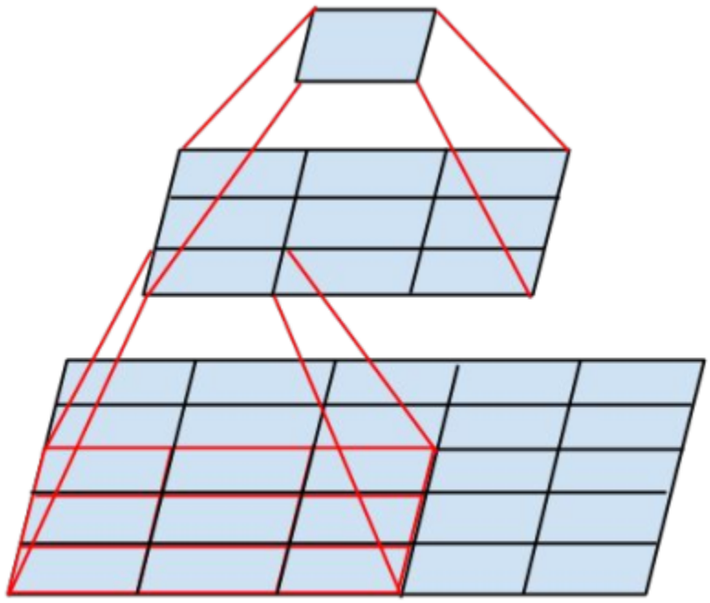


Figure 1. Mini-network replacing the 5×5 convolutions.

Therefore, we have developed a network that explores translation invariance and replaces one layer of convolution with two layers of convolution: the first layer is a 3x3 convolution layer and the second layer is a fully connected layer . Refer to Figure 1. We ended up replacing two 5x5 convolutional layers with two 3x3 convolutional layers. Refer to Figure 4 Figure 5. This operation can realize the weight sharing of neighboring layers. It is about (9 + 9) / 25 times reduction in computational consumption.

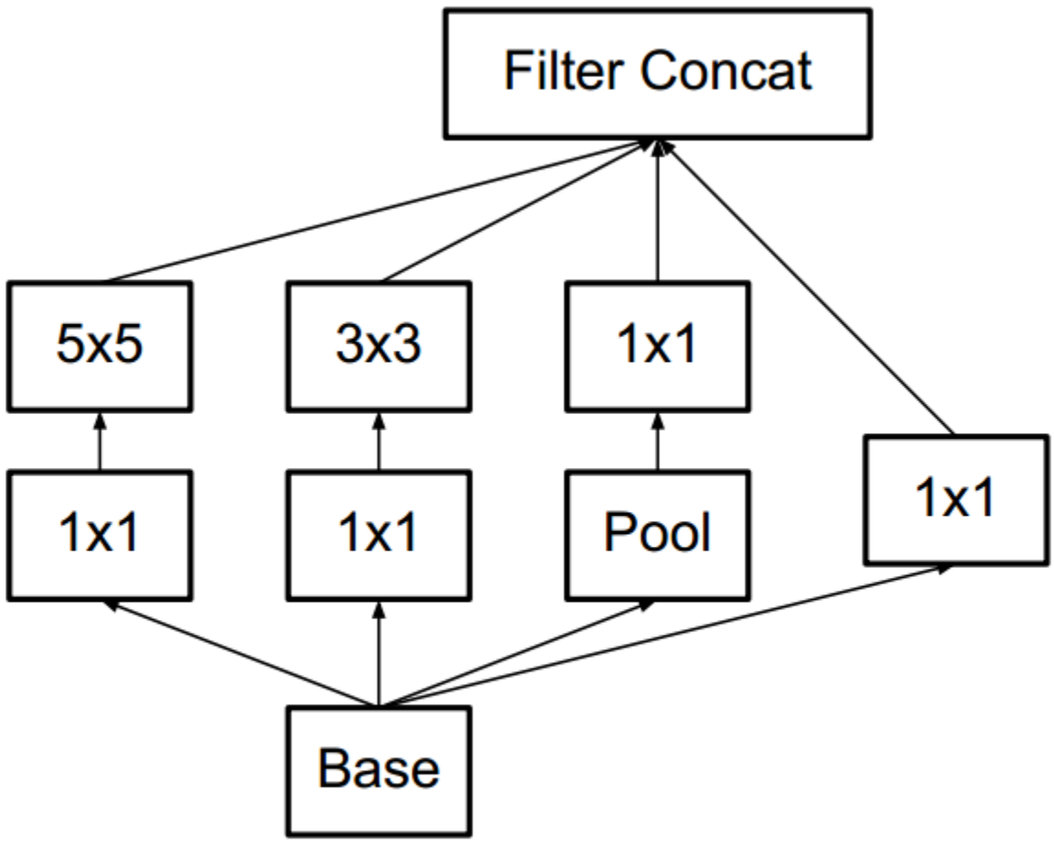


Figure 4. Original Inception module as described in [20].

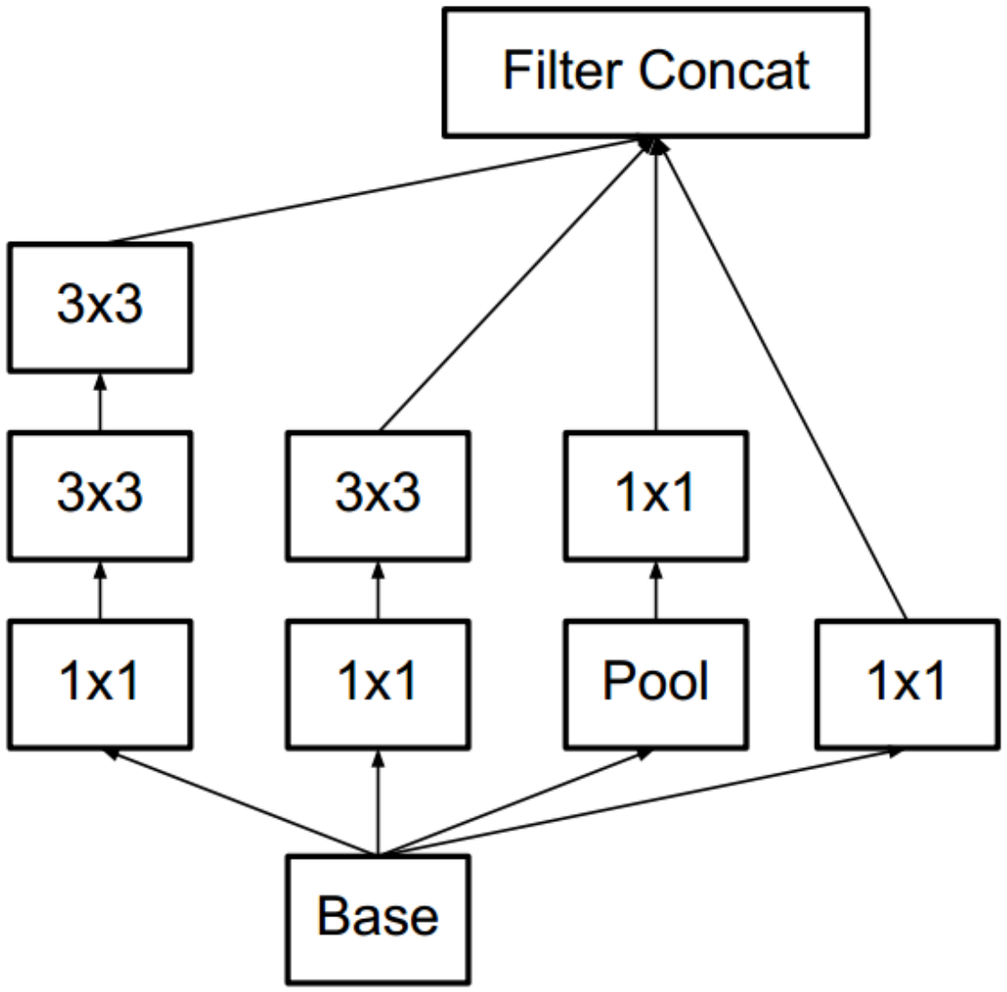


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2.

Spatial Factorization into Asymmetric Convolutions

We are wondering if the convolution kernel can be made smaller, such as 2x2, but there is an asymmetric method that can be better than this method. That is to use nx1 size convolution. For example, using the [3x1 + 1x3] convolution layer. In this case, a single 3x3 convolution has the same receptive field. Refer to Figure 3. This asymmetric method can save [((3x3)-(3 + 3)) / (3x3) = 33%] computing resources, and replacing two 2x2 only saves [11%] Computing resources.

In theory, we can have a deeper discussion and use the convolution of [1xn + nx1] instead of the convolutional layer of nxn. Refer to Figure 6. But this situation is not very good in the previous layer, but it can perform better on a medium-sized feature map [mxm, m is between 12 and 20]. In this case, use [1x7 + 7x1] convolutional layer can get a very good result.

Utility of Auxiliary Classifiers

Inception-v1 introduced some auxiliary classifiers (referring to some branches of the previous layer adding the softmax layer to calculate the loss back propagation) to improve the aggregation problem in deep networks. The original motive is to pass the gradient back to the previous convolutional layer , so that they can effectively and improve the aggregation of features and avoid the problem of vanishing gradients.

Traditionally, pooling layers are used in convolutional networks to reduce the size of feature maps . In order to avoid bottlenecks in the expression of spatial information, the number of convolution kernels in the network can be expanded before using max pooling or average pooling.

For example, for a dxd network layer with K feature maps, to generate a network layer with 2K [d / 2 xd / 2] feature maps, we can use 2K convolution kernels with a step size of 1. Convolution and then add a pooling layer to get it, then this operation requires [2d 2 K 2]. But using pooling instead of convolution, the approximate operation is [2 * (d / 2) 2 xK 2], which reduces the operation by four times. However, this will cause a description bottleneck, because the feature map is reduced to [(d / 2) 2 xK], which will definitely cause the loss of spatial information on the network. Refer to Figure 9. However, we have adopted a different method to avoid this bottleneck, refer to Figure 10. That is, two parallel channels are used , one is a pooling layer (max or average), the step size is 2, and the other is a convolution layer , and then it is concatenated during output.

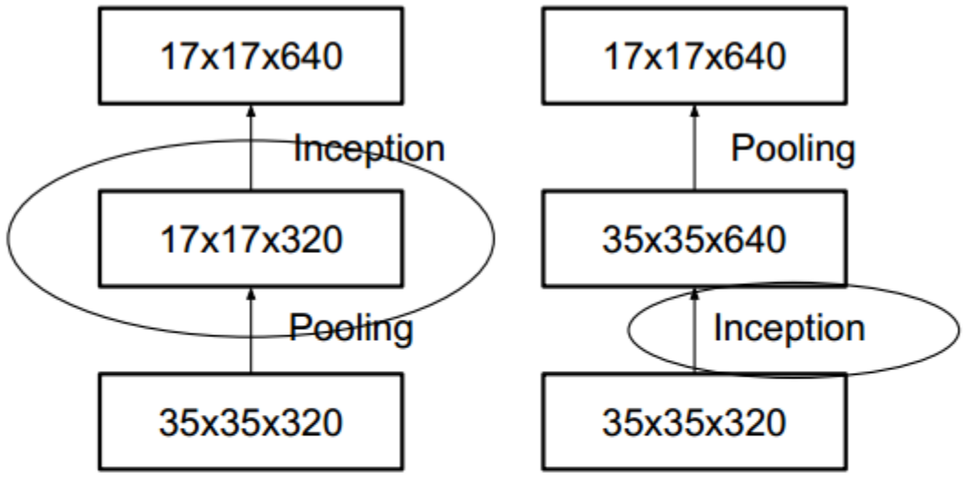


Figure 9. Two alternative ways of reducing the grid size. The solution on the left violates the principle 1 of not introducing an representational bottleneck from Section 2. The version on the right is 3 times more expensive computationally.

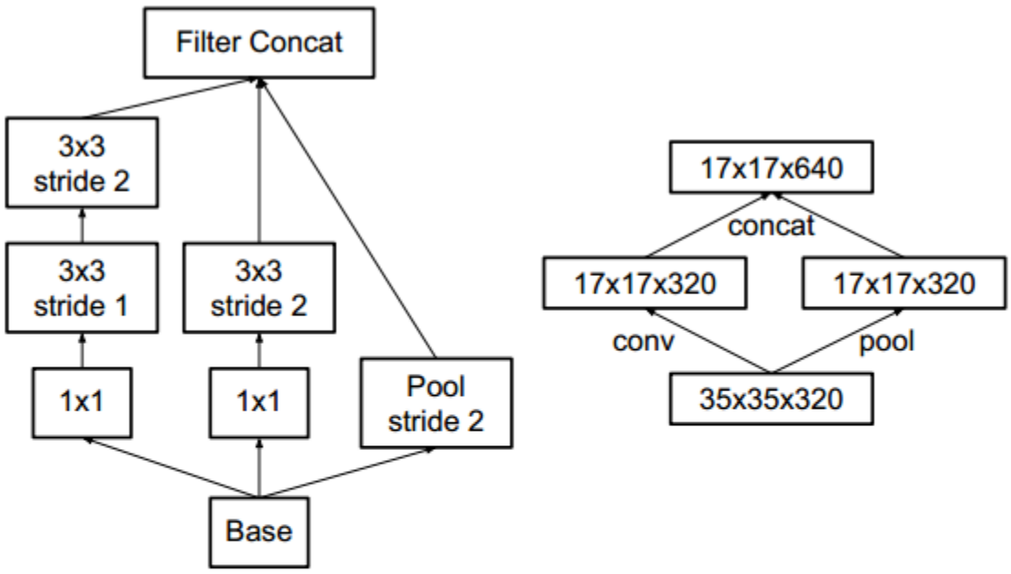


Figure 10. Inception module that reduces the grid-size while expands the filter banks. It is both cheap and avoids the representational bottleneck as is suggested by principle 1. The diagram on the right represents the same solution but from the perspective of grid sizes rather than the operations.

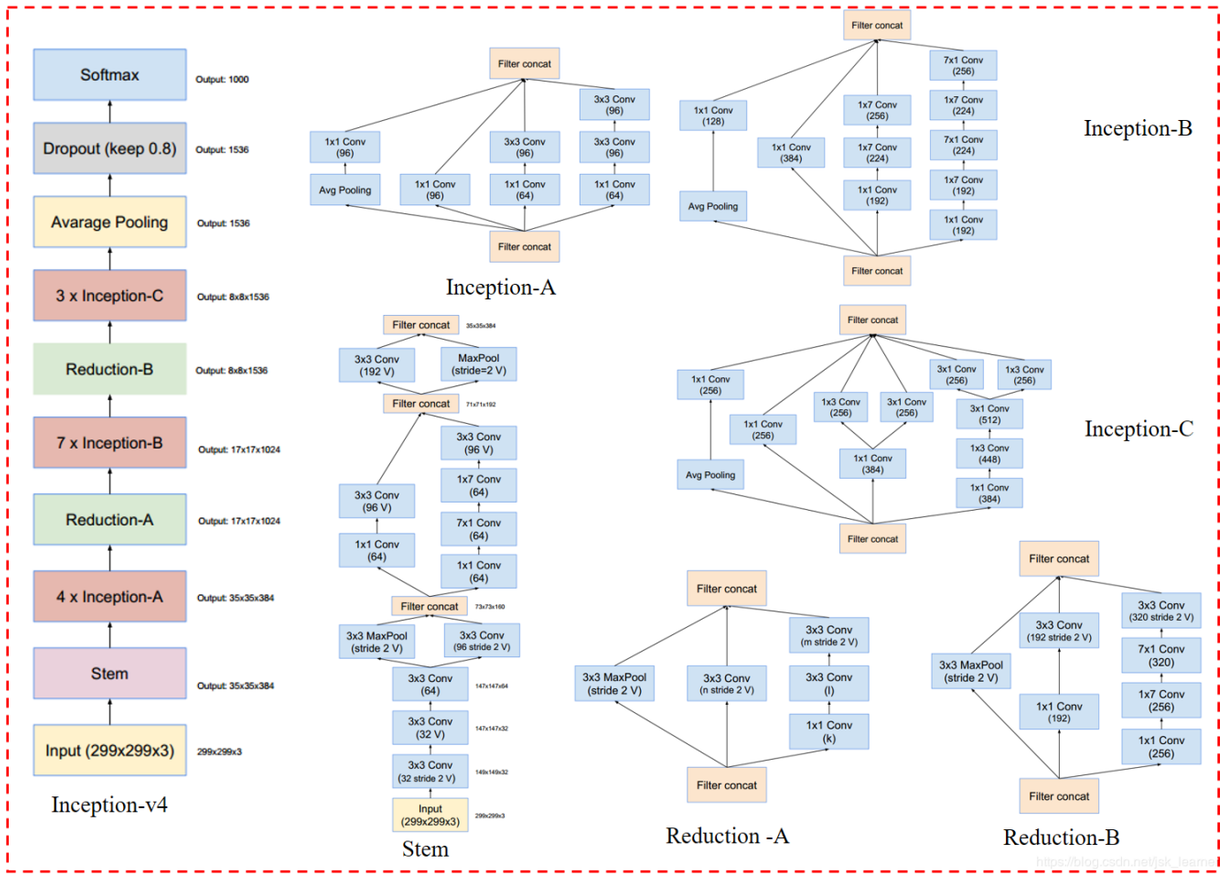
Inception-v4-2016

After ResNet appeared, ResNet residual structure was added.

It is based on Inception-v3 and added the skip connection structure in ResNet. Finally, under the structure of 3 residual and 1 inception-v4 , it reached the top-5 error 3.08% in CLS (ImageNet calssification) .

1-Introduction Residual conn works well when training very deep networks. Because the Inception network architecture can be very deep, it is reasonable to use residual conn instead of concat.

Compared with v3, Inception-v4 has more unified simplified structure and more inception modules.



The big picture of Inception-v4:

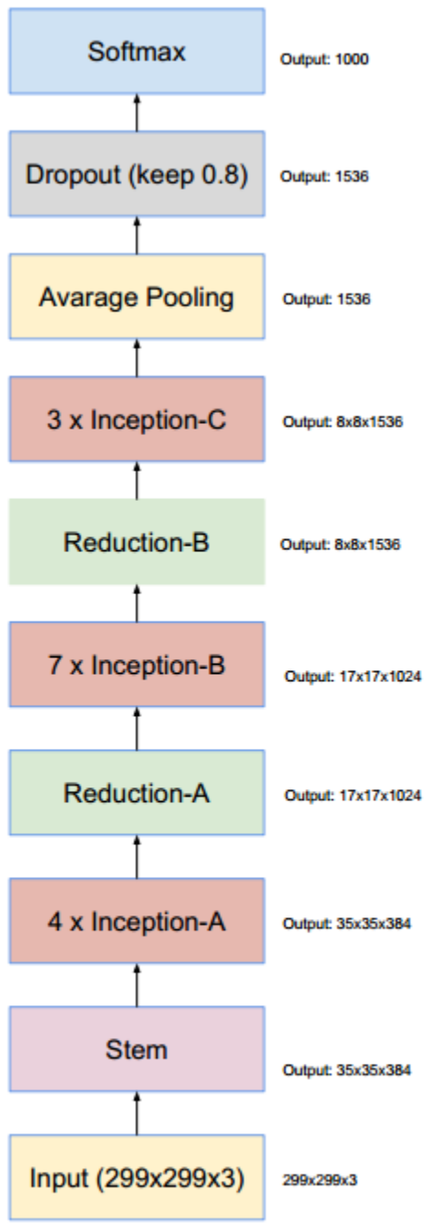


Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

Fig9 is an overall picture, and Fig3,4,5,6,7,8 are all local structures. For the specific structure of each module, see the end of the article.

Residual Inception Blocks

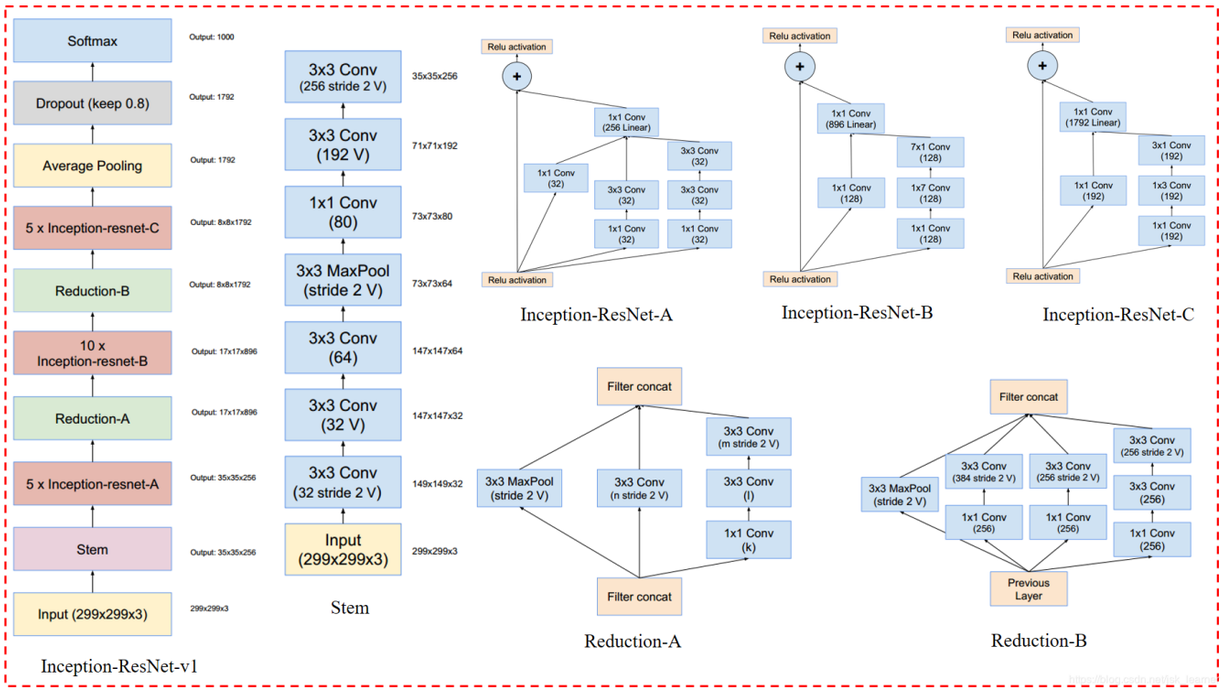
For the residual version in the Inception network, we use an Inception module that consumes less than the original Inception. The convolution kernel (followed by 1x1) of each Inception module is used to modify the dimension, which can compensate the reduction of the Inception dimension to some extent.

One is named **Inception-ResNet-v1**, which is consistent with the calculation cost of Inception-v3. One is named **Inception-ResNet-v2**, which is consistent with the calculation cost of Inception-v4.

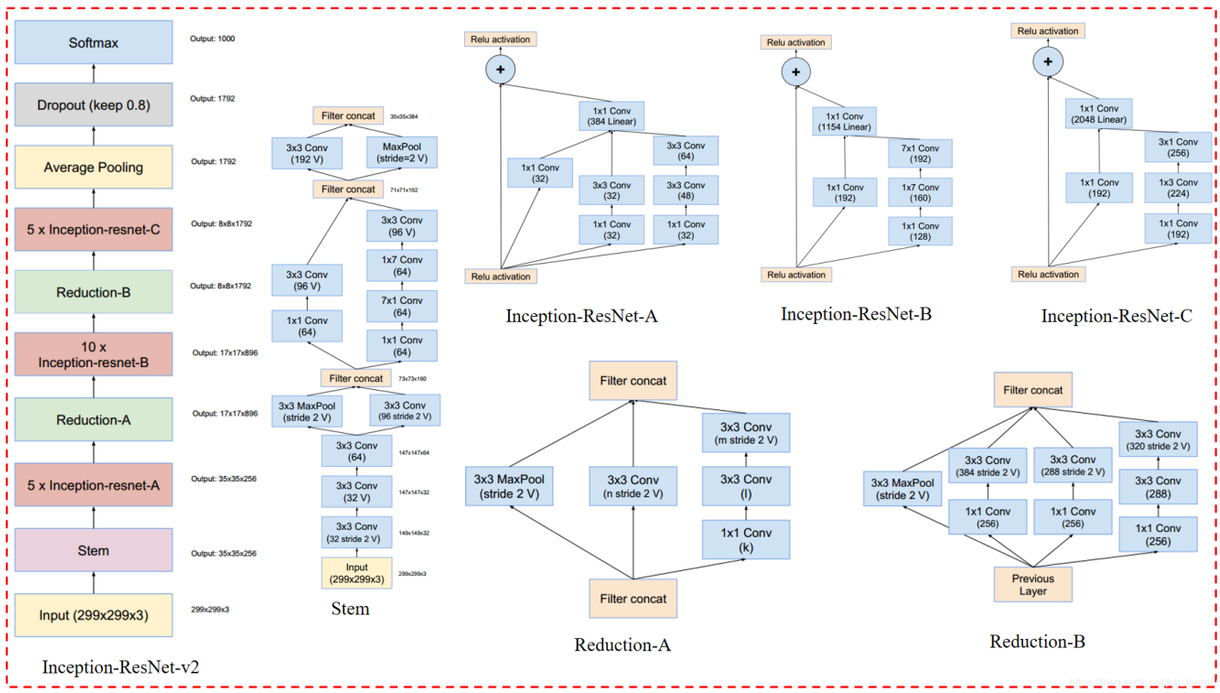
Figure 15 shows the structure of both. However, Inception-v4 is actually slower in practice, probably because it has more layers.

Another small technique is that we use the BN layer in the header of the traditional layer in the Inception-ResNet module, but not in the header of the summations. ** There is reason to believe that the BN layer is effective. But in order to add more Inception modules, we made a compromise between the two.

Inception-ResNet-v1



Inception-ResNet-v2



Scaling of the Residuals

This paper finds that when the number of convolution kernels exceeds 1,000 , the residual variants will start to show instability , and the network will die in the early stages of training, which means that the last layer before the average pooling layer is in the Very few iterations start with just a zero value . This situation

cannot be prevented by reducing the learning rate or by adding a BN layer . Hekaiming's ResNet article also mentions this phenomenon.

This article finds that scale can stabilize the training process before adding the residual module to the activation layer . This article sets the scale coefficient between 0.1 and 0.3.

In order to prevent the occurrence of unstable training of deep residual networks, He suggested in the article that it is divided into two stages of training. The first stage is called warm-up (preheating) , that is, training the model with a very low learning first. In the second stage, a higher learning rate is used. And this article finds that if the convolution sum is very high, even a learning rate of 0.00001 cannot solve this training instability problem, and the high learning rate will also destroy the effect. But this article considers scale residuals to be more reliable than warm-up.

Even if scal is not strictly necessary, it has no effect on the final accuracy, but it can stabilize the training process.

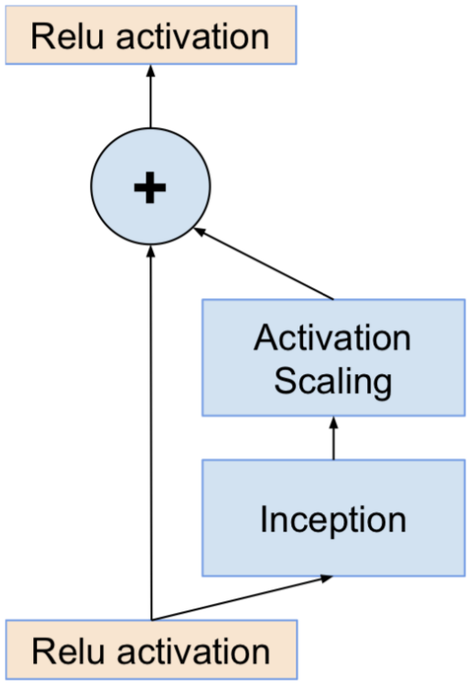


Figure 20. The general schema for scaling combined Inception-resnet moduels. We expect that the same idea is useful in the general resnet case, where instead of the Inception block an arbitrary subnetwork is used. The scaling block just scales the last linear activations by a suitable constant, typically around 0.1.

Conclusion

Inception-ResNet-v1 : a network architecture combining inception module and resnet module with similar calculation cost to Inception-v3;

Inception-ResNet-v2 : A more expensive but better performing network architecture.

Inception-v4 : A pure inception module, without residual connections, but with performance similar to Inception-ResNet-v2.

A big picture of the various module structures of Inception-v4 / Inception-ResNet-v1 / v2:

- Fig3-Stem: (Inception-v4 & Inception-ResNet-v2)

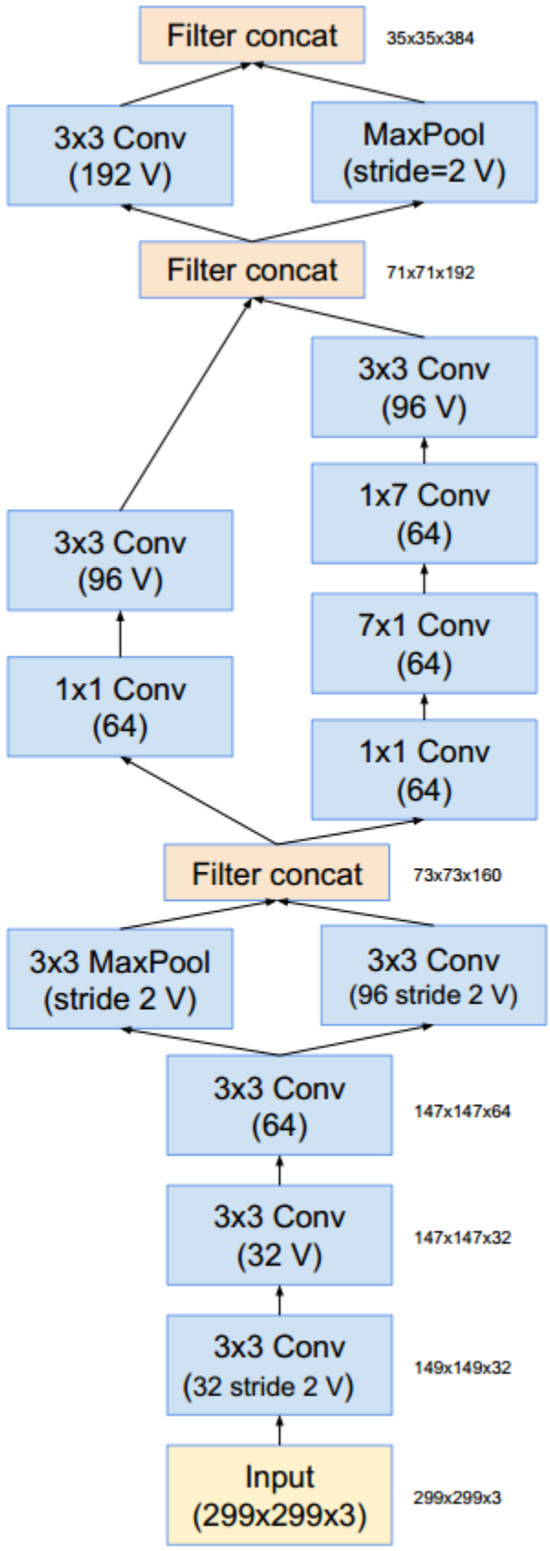


Figure 3. The schema for stem of the pure Inception-v4 and Inception-ResNet-v2 networks. This is the input part of those networks. Cf. Figures 9 and 15

- Fig4-Inception-A: (Inception-v4)

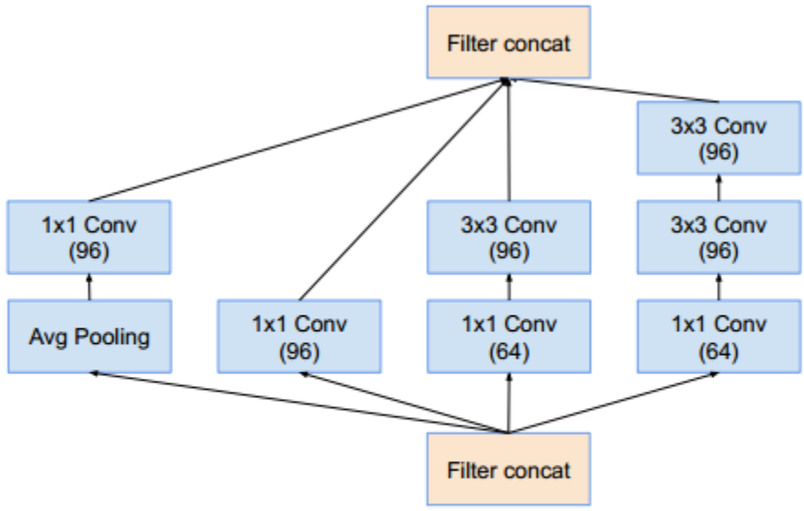


Figure 4. The schema for 35 × 35 grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

- Fig5-Inception-B: (Inception-v4)

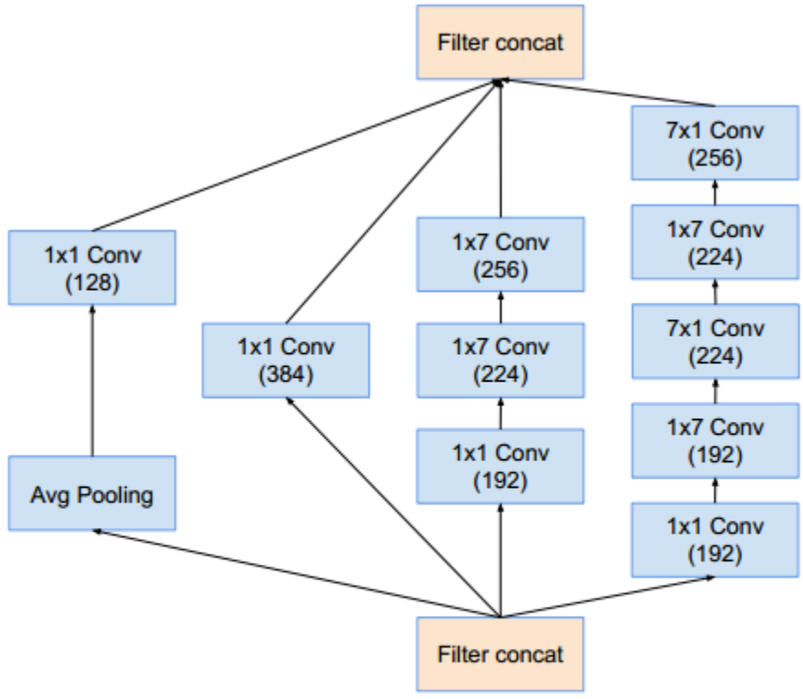


Figure 5. The schema for 17 × 17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

- Fig6-Inception-C: (Inception-v4)

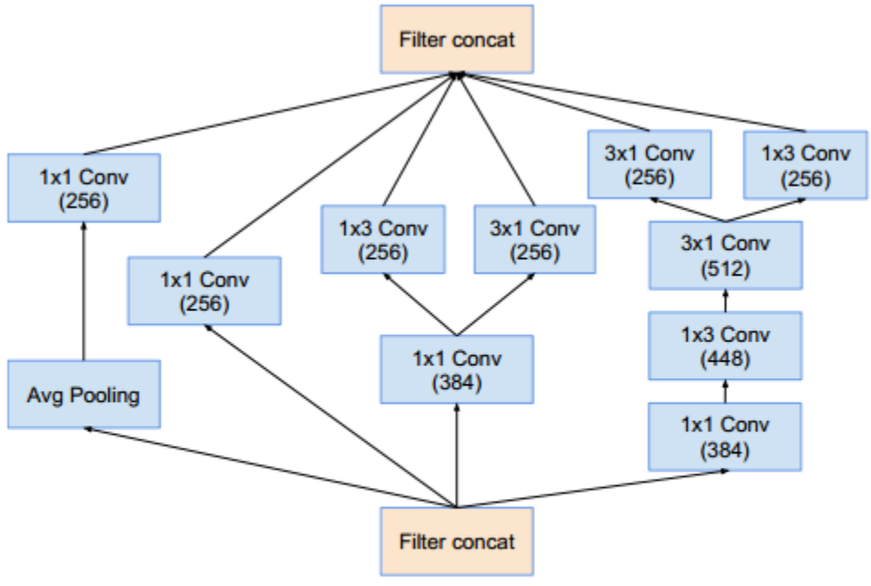


Figure 6. The schema for 8 × 8 grid modules of the pure Inception-v4 network. This is the Inception-C block of Figure 9.

- Fig7-Reduction-A: (Inception-v4 & Inception-ResNet-v1 & Inception-ResNet-v2)

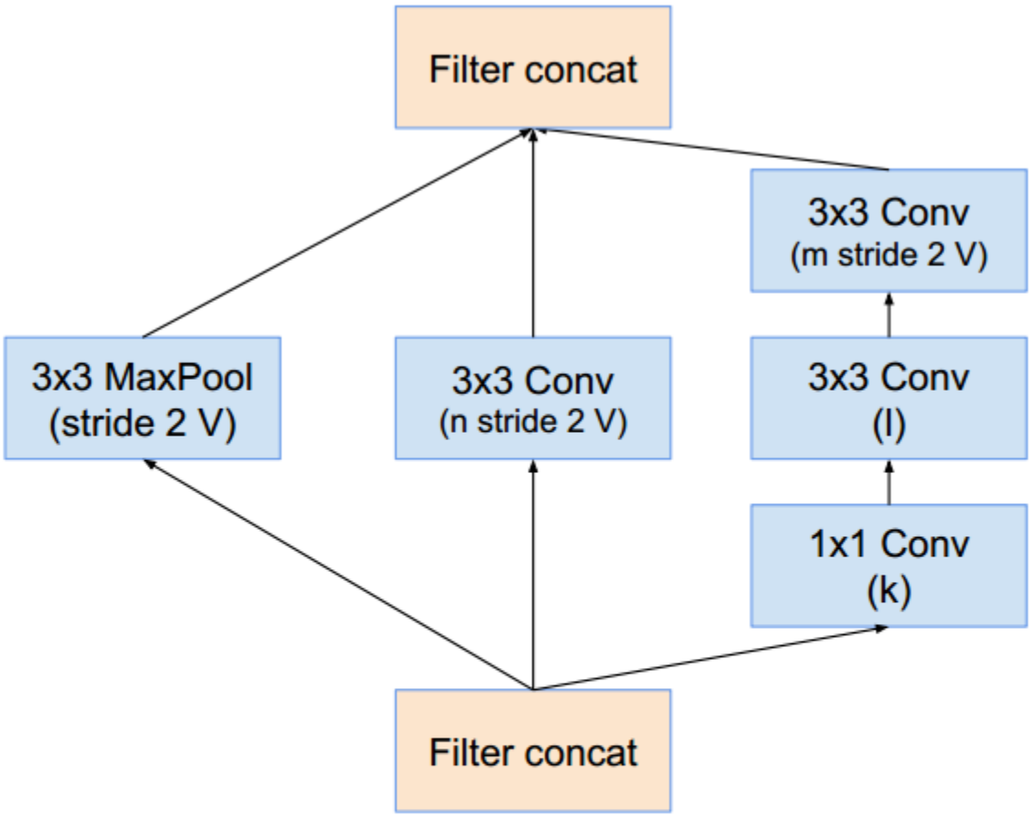


Figure 7. The schema for 35×35 to 17×17 reduction module. Different variants of this blocks (with various number of filters) are used in Figure 9, and 15 in each of the new Inception(-v4, -ResNet-v1, -ResNet-v2) variants presented in this paper. The k, l, m, n numbers represent filter bank sizes which can be looked up in Table 1.

- Fig8-Reduction-B: (Inception-v4)

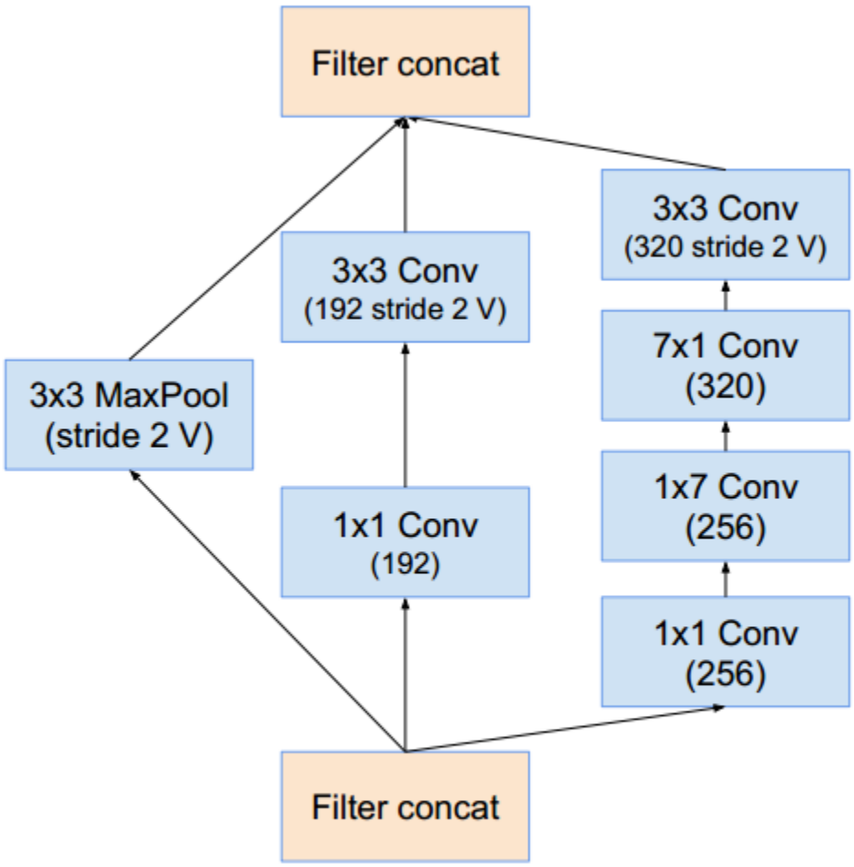


Figure 8. The schema for 17×17 to 8×8 grid-reduction module. This is the reduction module used by the pure Inception-v4 network in Figure 9.

- Fig10-Inception-ResNet-A: (Inception-ResNet-v1)

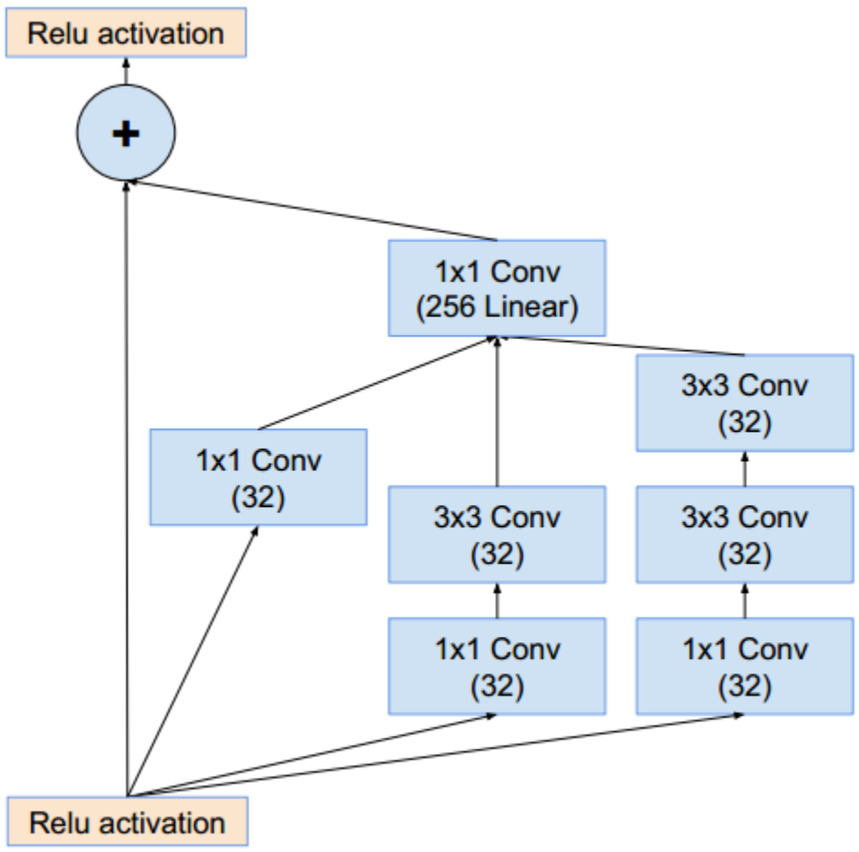


Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

- Fig11-Inception-ResNet-B: (Inception-ResNet-v1)

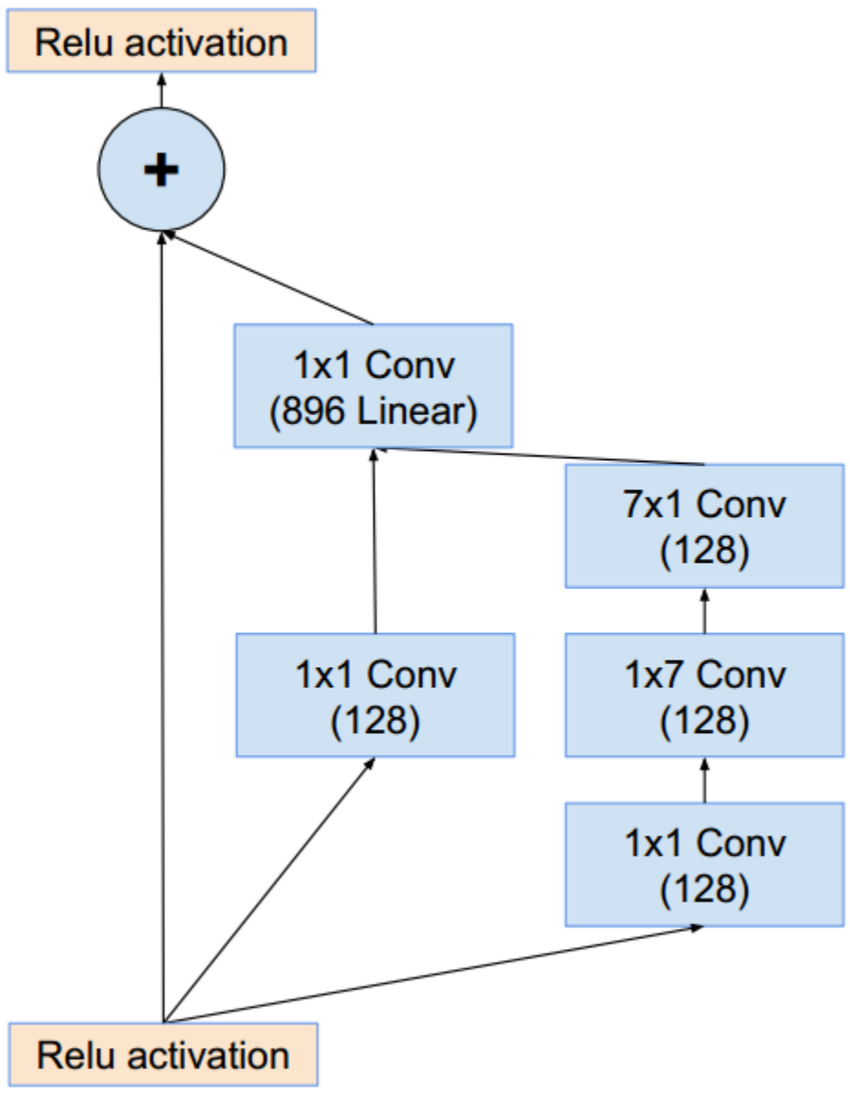


Figure 11. The schema for 17 × 17 grid (Inception-ResNet-B) module of Inception-ResNet-v1 network.

- Fig12-Reduction-B: (Inception-ResNet-v1)

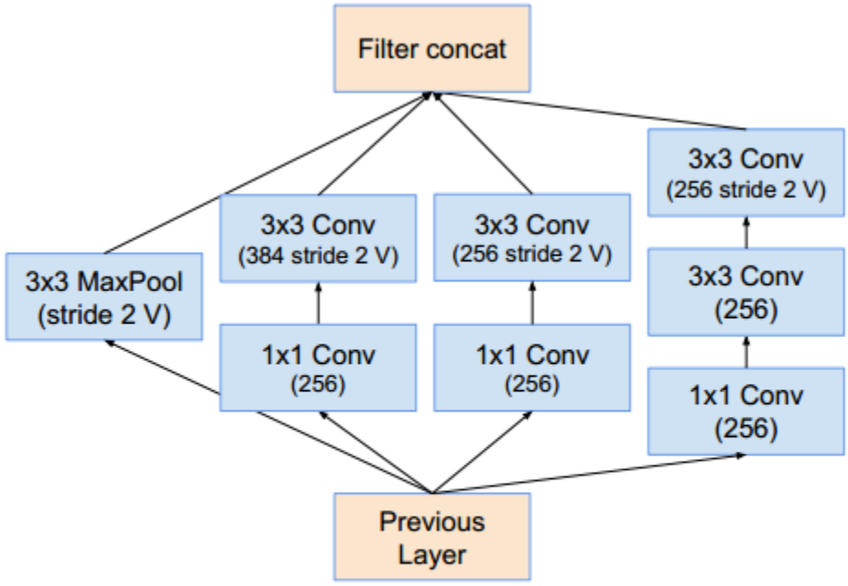


Figure 12. "Reduction-B" 17×17 to 8×8 grid-reduction module. This module used by the smaller Inception-ResNet-v1 network in Figure 15.

- Fig13-Inception-ResNet-C: (Inception-ResNet-v1)

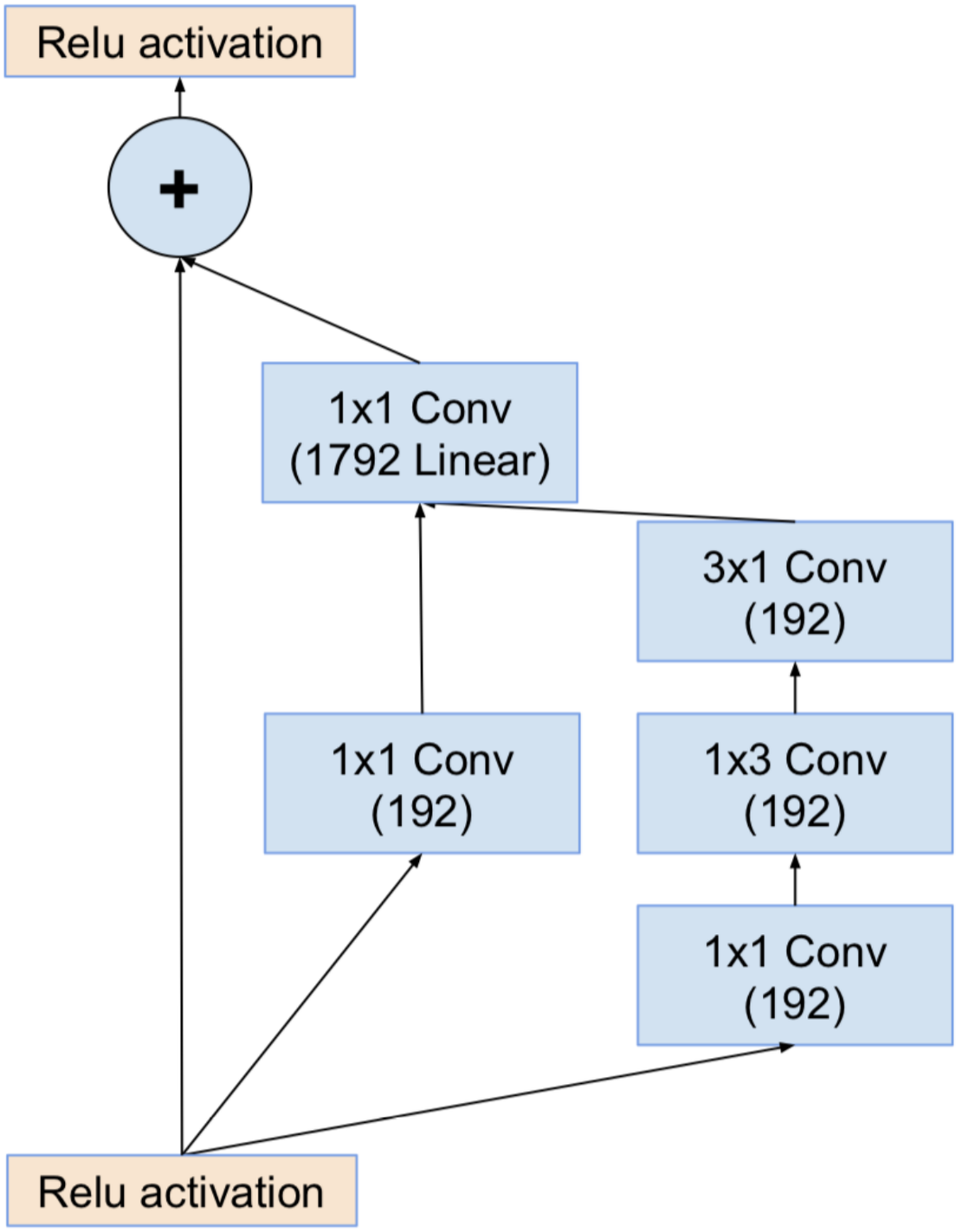
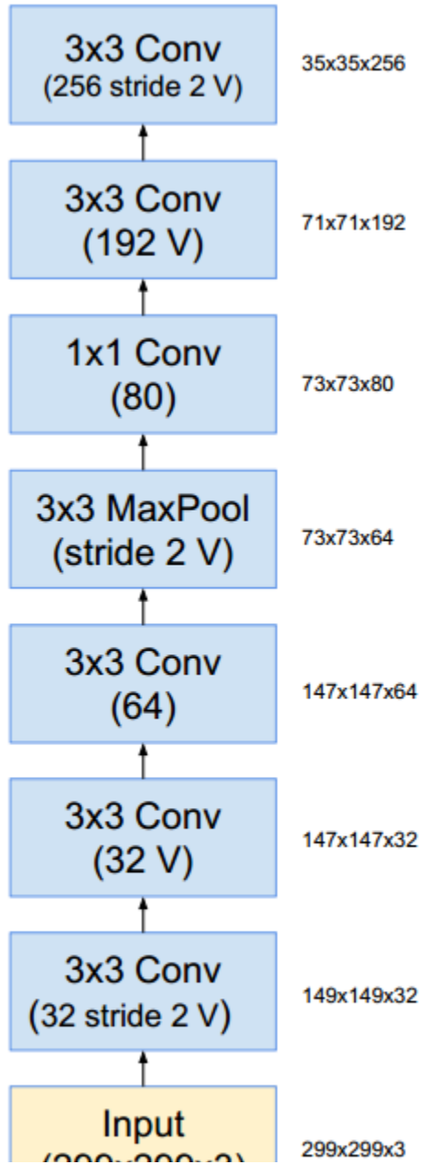


Figure 13. The schema for 8×8 grid (Inception-ResNet-C) module of Inception-ResNet-v1 network.

- Fig14-Stem: (Inception-ResNet-v1)



Code implementation

From Scratch

In [2]:
```
!pip install tflearn
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) https://us-pyth
on.pkg.dev/colab-wheels/public/simple/ (https://us-python.pkg.dev/colab-wheels/public/s
imple/)
Collecting tflearn
  Downloading tflearn-0.5.0.tar.gz (107 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 107.3/107.3 kB 6.7 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from t
flearn) (1.22.4)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from tfl
earn) (1.16.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from
tflearn) (8.4.0)
Building wheels for collected packages: tflearn
  Building wheel for tflearn (setup.py) ... done
  Created wheel for tflearn: filename=tflearn-0.5.0-py3-none-any.whl size=127283 sha256
=f0daf99db3e5e956ce8947fe092b62231f5895e4149dfeaf4134e3d979b1729b
  Stored in directory: /root/.cache/pip/wheels/55/fb/7b/e06204a0ceefa45443930b9a250cb5e
be31def0e4e8245a465
Successfully built tflearn
Installing collected packages: tflearn
Successfully installed tflearn-0.5.0
```

In [1]:
```
from tensorflow import keras
from keras.models import Model
from keras.layers import Input, Conv2D, MaxPooling2D, concatenate, Flatten, Dense, Avera
from keras.optimizers import Adam
```

In [4]:
```
# Get Data
import tflearn.datasets.oxflower17 as oxflower17
from keras.utils import to_categorical

x, y = oxflower17.load_data()

x_train = x.astype('float32') / 255.0
y_train = to_categorical(y, num_classes=17)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/compa
t/v2_compat.py:107: disable_resource_variables (from tensorflow.python.ops.variable_sco
pe) is deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term

Downloading Oxford 17 category Flower Dataset, Please wait...

100.0% 60276736 / 60270631

Succesfully downloaded 17flowers.tgz 60270631 bytes.
File Extracted
Starting to parse images...
Parsing Done!
```

In [5]:
```
print(x_train.shape)
print(y_train.shape)
```

```
(1360, 224, 224, 3)
(1360, 17)
```

In [3]:
```
# Inception block
def inception_block(x, filters):
    tower_1 = Conv2D(filters[0], (1, 1), padding='same', activation='relu')(x)
    tower_1 = Conv2D(filters[1], (3, 3), padding='same', activation='relu')(tower_1)

    tower_2 = Conv2D(filters[2], (1, 1), padding='same', activation='relu')(x)
    tower_2 = Conv2D(filters[3], (5, 5), padding='same', activation='relu')(tower_2)

    tower_3 = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(x)
    tower_3 = Conv2D(filters[4], (1, 1), padding='same', activation='relu')(tower_3)

    output = concatenate([tower_1, tower_2, tower_3], axis=3)
    return output

# Build the Inception model
def inception(input_shape, num_classes):
    inputs = Input(shape=input_shape)

    x = Conv2D(64, (3, 3), padding='same', activation='relu')(inputs)
    x = MaxPooling2D((2, 2))(x)

    x = inception_block(x, filters=[64, 96, 128, 16, 32])
    x = inception_block(x, filters=[128, 128, 192, 32, 96])
    x = MaxPooling2D((2, 2))(x)

    x = inception_block(x, filters=[192, 96, 208, 16, 48])
    x = inception_block(x, filters=[160, 112, 224, 24, 64])
    x = inception_block(x, filters=[128, 128, 256, 24, 64])
    x = inception_block(x, filters=[112, 144, 288, 32, 64])
    x = MaxPooling2D((2, 2))(x)

    x = inception_block(x, filters=[256, 160, 320, 32, 128])
    x = inception_block(x, filters=[256, 160, 320, 32, 128])
    x = inception_block(x, filters=[384, 192, 384, 48, 128])

    x = AveragePooling2D((4, 4))(x)
    x = Flatten()(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=inputs, outputs=outputs)
    return model
```

In [6]:
```
# Create the Inception model
model = inception(input_shape=(224, 224, 3), num_classes=17)

# Compile the model
model.compile(optimizer=Adam(lr=0.001), loss='categorical_crossentropy', metrics=['accur

# Print a summary of the model
model.summary()

# Train
model.fit(x_train, y_train, batch_size=64, epochs=5, verbose=1,validation_split=0.2, shu
```

```
Model: "model"
__________________________________________________________________________________
_______________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================
==============
 input_1 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 conv2d (Conv2D)                (None, 224, 224, 64  1792        ['input_1[0][0]']
                                )

 max_pooling2d (MaxPooling2D)   (None, 112, 112, 64  0           ['conv2d[0][0]']
                                )

 conv2d_1 (Conv2D)              (None, 112, 112, 64  4160        ['max_pooling2d[0]
[0]']
                                )
```

Pretrained

In [7]:
```
# download the data from g drive

import gdown
url = "https://drive.google.com/file/d/12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP/view?usp=sharin
file_id = url.split("/")[-2]
print(file_id)
prefix = 'https://drive.google.com/uc?/export=download&id='
gdown.download(prefix+file_id, "catdog.zip")
```

```
12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP

Downloading...
From: https://drive.google.com/uc?/export=download&id=12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP
(https://drive.google.com/uc?/export=download&id=12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP)
To: /content/catdog.zip
100%|██████████| 9.09M/9.09M [00:01<00:00, 7.59MB/s]
```

Out[7]:
```
'catdog.zip'
```

In [8]:
```
!unzip catdog.zip
```

```
Archive:  catdog.zip
   creating: train/
   creating: train/Cat/
  inflating: train/Cat/0.jpg
  inflating: train/Cat/1.jpg
  inflating: train/Cat/2.jpg
  inflating: train/Cat/cat.2405.jpg
  inflating: train/Cat/cat.2406.jpg
  inflating: train/Cat/cat.2436.jpg
  inflating: train/Cat/cat.2437.jpg
  inflating: train/Cat/cat.2438.jpg
  inflating: train/Cat/cat.2439.jpg
  inflating: train/Cat/cat.2440.jpg
  inflating: train/Cat/cat.2441.jpg
  inflating: train/Cat/cat.2442.jpg
  inflating: train/Cat/cat.2443.jpg
  inflating: train/Cat/cat.2444.jpg
  inflating: train/Cat/cat.2445.jpg
  inflating: train/Cat/cat.2446.jpg
```

In [9]:

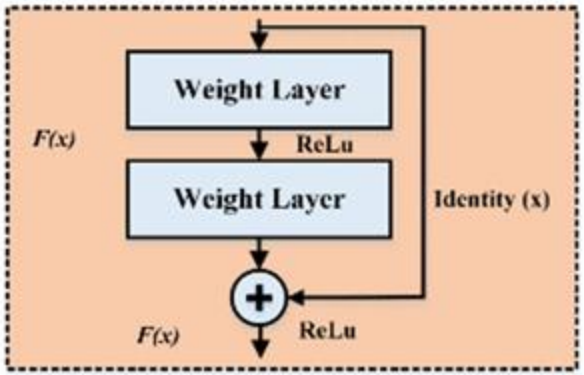
```
from tensorflow import keras
from tensorflow.keras.applications.inception_v3 import InceptionV3, preprocess_input
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten

# Set the path to your training and validation data
train_data_dir = '/content/train'
validation_data_dir = '/content/validation'

# Set the number of training and validation samples
num_train_samples = 2000
num_validation_samples = 800

# Set the number of epochs and batch size
epochs = 5
batch_size = 16

# Load the InceptionV3 model without the top layer
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3

# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Create a new model
model = Sequential()

# Add the base model as a layer
model.add(base_model)

# Add custom layers on top of the base model
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Preprocess the training and validation data
train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
validation_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='binary')

validation_generator = validation_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='binary')

# Train the model
model.fit(
    train_generator,
    steps_per_epoch=num_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=num_validation_samples // batch_size)

# Save the trained model
model.save('dog_cat_classifier.h5')
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/keras/layers/normalizat
ion/batch_normalization.py:581: _colocate_with (from tensorflow.python.framework.ops) i
s deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

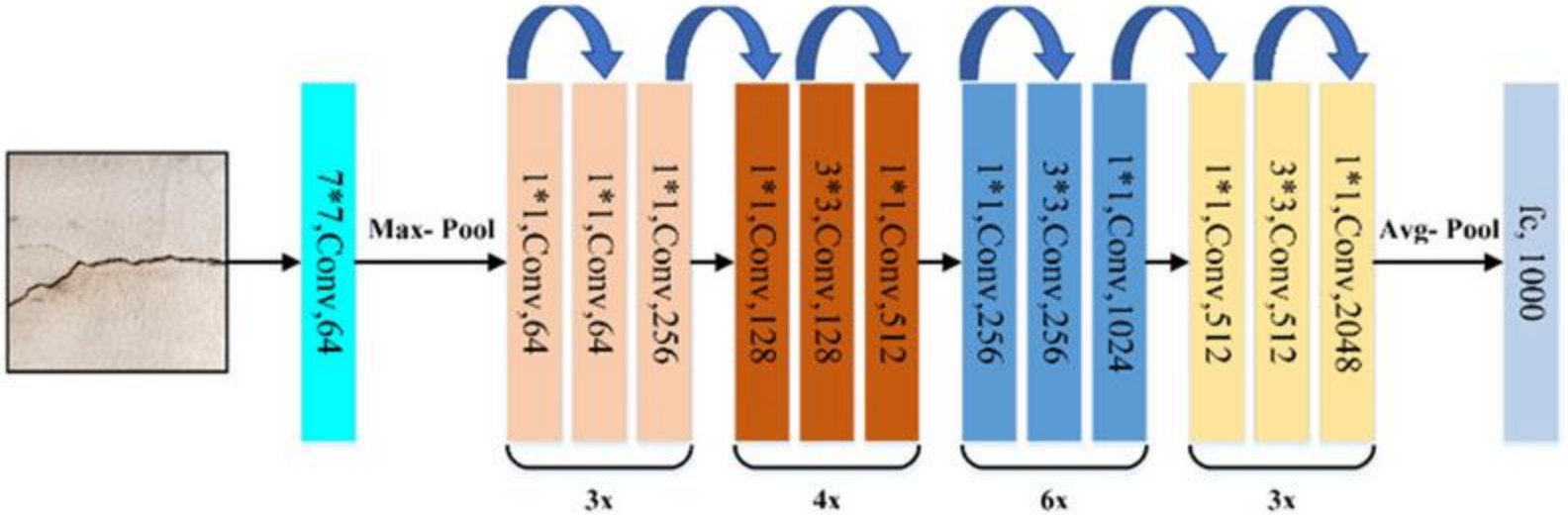
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5 (https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5)
87910968/87910968 [==============================] - 3s 0us/step
Found 337 images belonging to 2 classes.
Found 59 images belonging to 2 classes.
Epoch 1/5
125/125 [==============================] - 20s 135ms/step - batch: 62.0000 - size: 15.2800 - loss: 2.2705 - acc: 0.8613 - val_loss: 0.4283 - val_acc: 0.9500
Epoch 2/5
125/125 [==============================] - 15s 118ms/step - batch: 62.0000 - size: 15.2800 - loss: 1.0321 - acc: 0.8927 - val_loss: 0.3787 - val_acc: 0.9662
Epoch 3/5
125/125 [==============================] - 14s 109ms/step - batch: 62.0000 - size: 15.4000 - loss: 0.1464 - acc: 0.9605 - val_loss: 0.1556 - val_acc: 0.9676
Epoch 4/5
125/125 [==============================] - 14s 116ms/step - batch: 62.0000 - size: 15.2800 - loss: 0.3796 - acc: 0.9314 - val_loss: 0.2708 - val_acc: 0.9676
Epoch 5/5
125/125 [==============================] - 14s 109ms/step - batch: 62.0000 - size: 15.4000 - loss: 0.2161 - acc: 0.9626 - val_loss: 0.2489 - val_acc: 0.9662
```

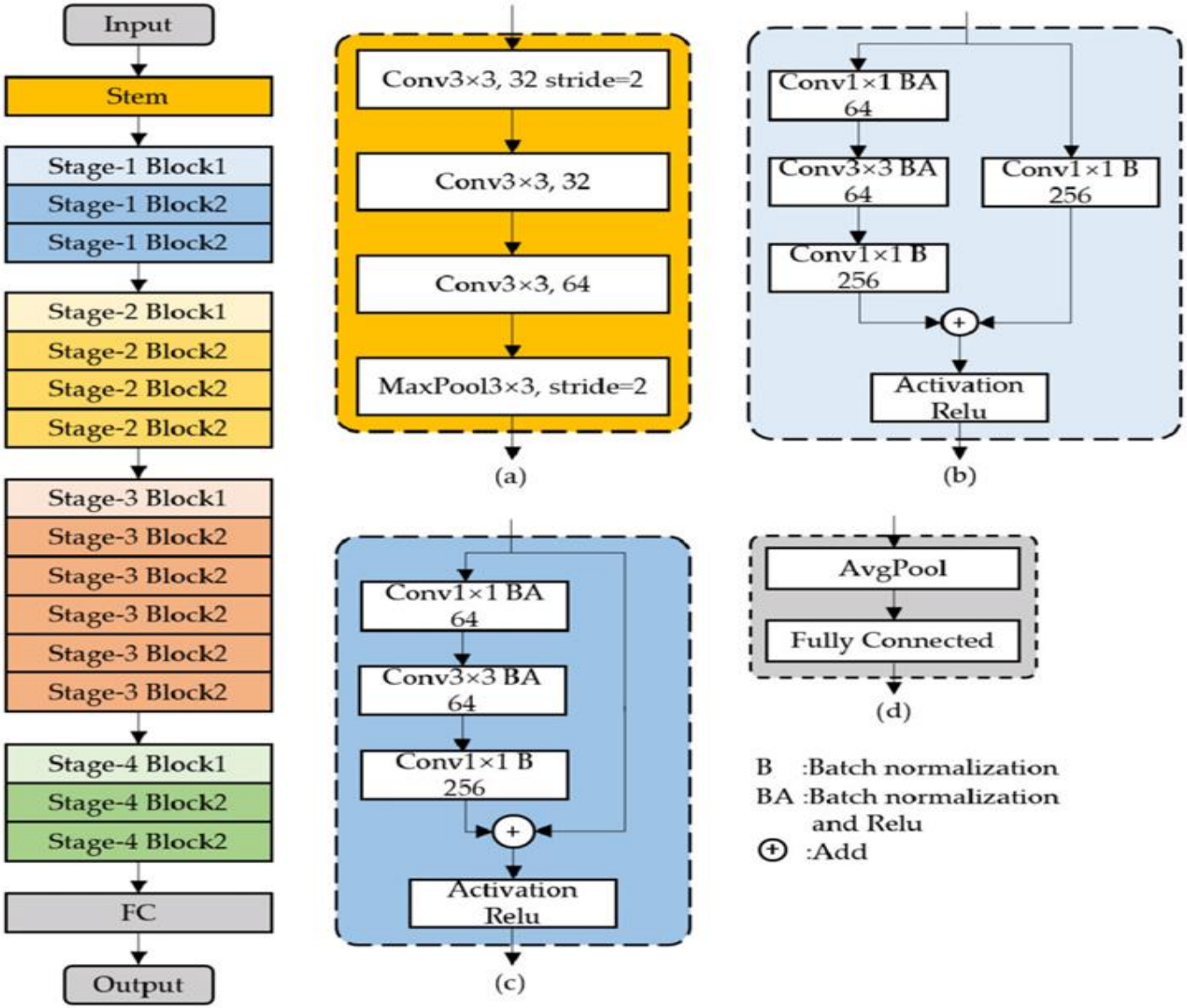
In [ ]:

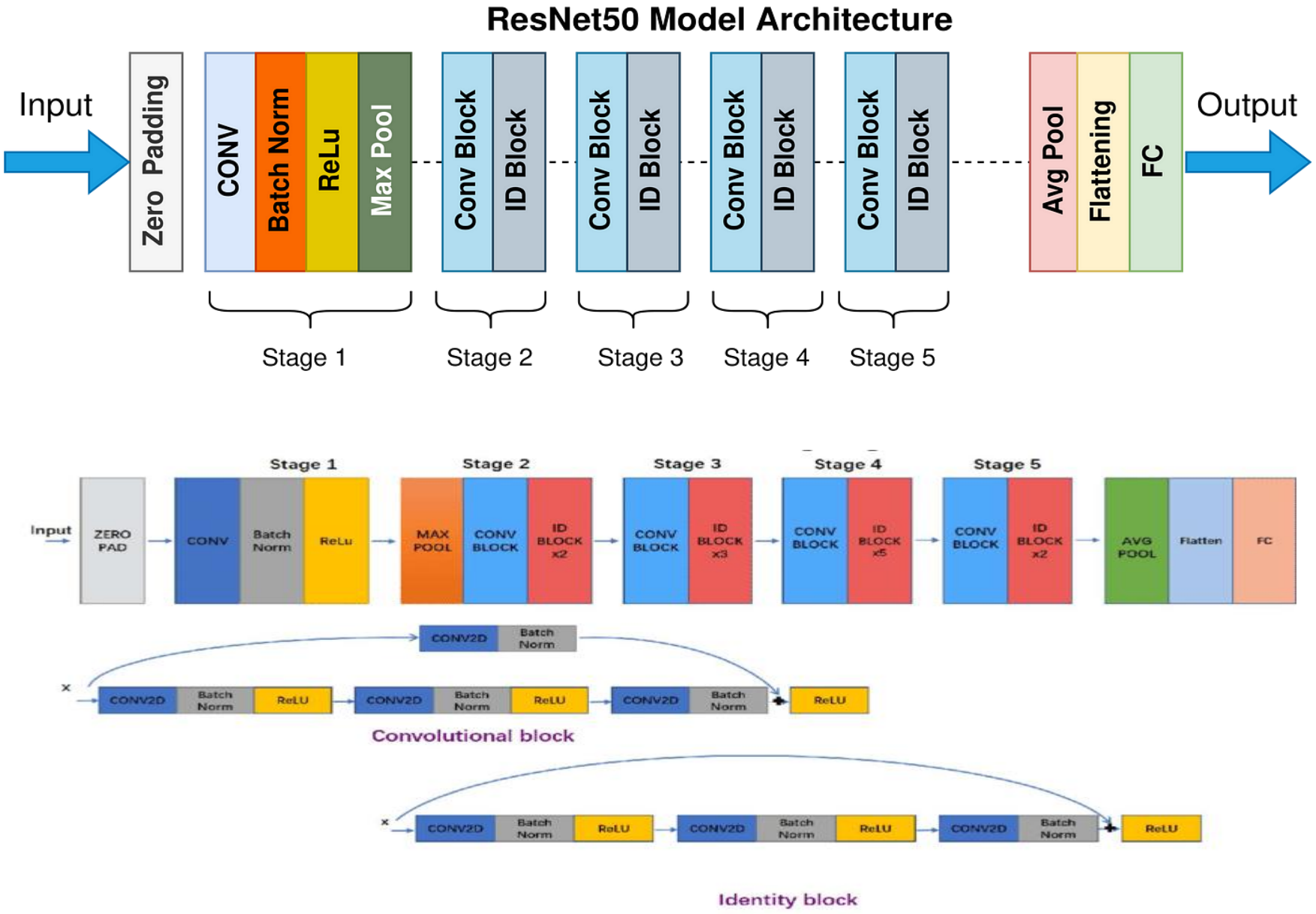
RSNeT

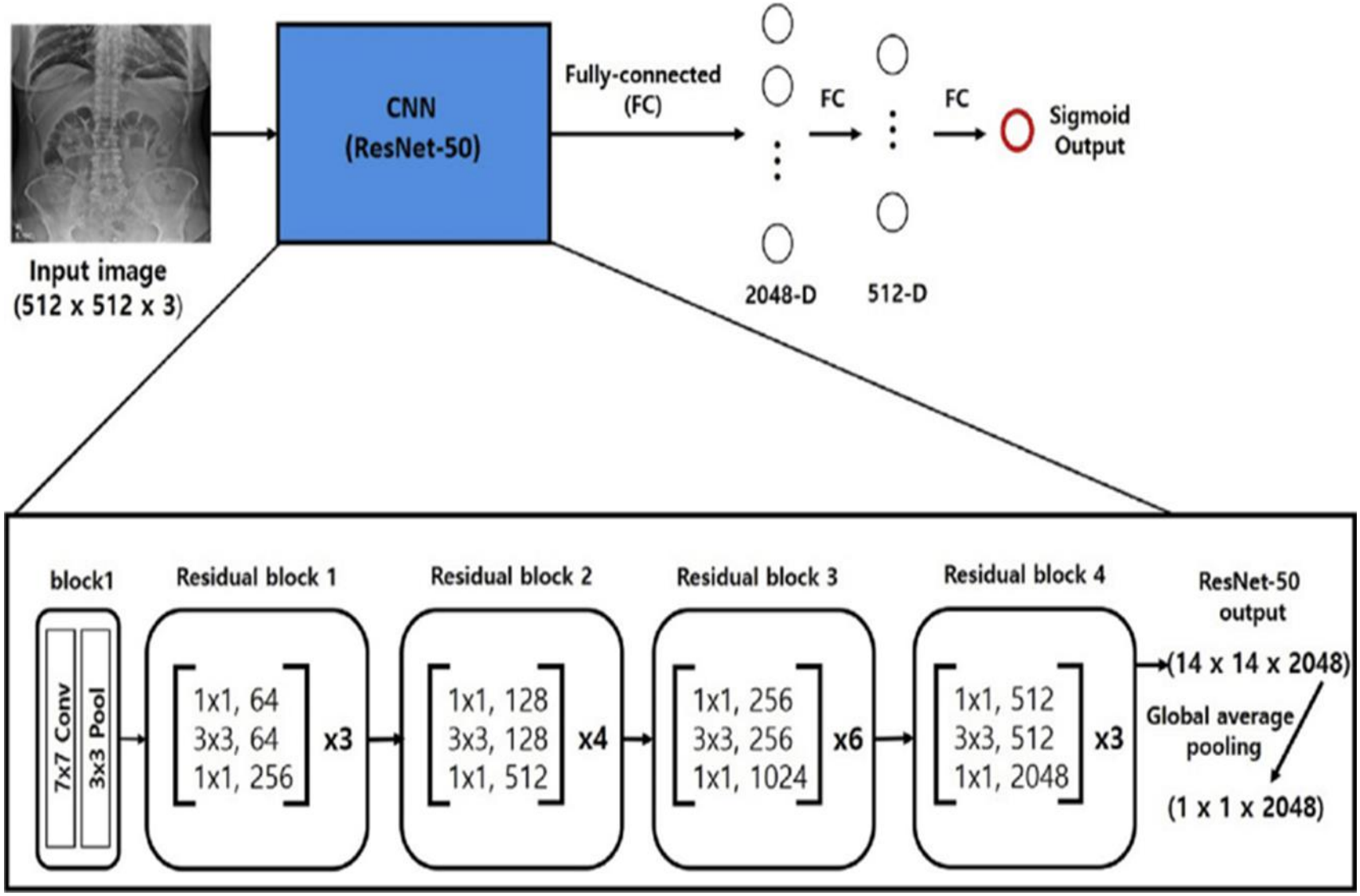


Residual Learning Block




Input
Stem
Stage-1 Block1
Stage-1 Block2
Stage-1 Block2
Stage-2 Block1
Stage-2 Block2
Stage-2 Block2
Stage-2 Block2
Stage-3 Block1
Stage-3 Block2
Stage-3 Block2
Stage-3 Block2
Stage-3 Block2
Stage-3 Block2
Stage-4 Block1
Stage-4 Block2
Stage-4 Block2
FC
Output
Conv3×3, 32 stride=2
Conv3×3, 32
Conv3×3, 64
MaxPool3×3, stride=2
(a)
Conv1×1 BA 64
Conv3×3 BA 64
Conv1×1 B 256
Conv1×1 B 256
Activation Relu
(b)
Conv1×1 BA 64
Conv3×3 BA 64
Conv1×1 B 256
Activation Relu
(c)
AvgPool
Fully Connected
(d)
B :Batch normalization
BA :Batch normalization and Relu
:Add


ResNet50 Model Architecture
Input
Zero Padding
CONV
Batch Norm
ReLu
Max Pool
Conv Block
ID Block
Conv Block
ID Block
Conv Block
ID Block
Conv Block
ID Block
Avg Pool
Flattening
FC
Output
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Input
ZERO PAD
CONV
Batch Norm
ReLu
MAX POOL
CONV BLOCK
ID BLOCK x2
CONV BLOCK
ID BLOCK x3
CONV BLOCK
ID BLOCK x5
CONV BLOCK
ID BLOCK x2
AVG POOL
Flatten
FC
x
CONV2D
Batch Norm
ReLU
Convolutional block
Identity block


Input image
(512 x 512 x 3)
CNN
(ResNet-50)
Fully-connected
(FC)
2048-D
FC
512-D
FC
Sigmoid
Output
block1
7x7 Conv
3x3 Pool
Residual block 1
1x1, 64
3x3, 64
1x1, 256
x3
Residual block 2
1x1, 128
3x3, 128
1x1, 512
x4
Residual block 3
1x1, 256
3x3, 256
1x1, 1024
x6
Residual block 4
1x1, 512
3x3, 512
1x1, 2048
x3
ResNet-50
output
(14 x 14 x 2048)
Global average
pooling
(1 x 1 x 2048)

Resnet

Introduction

ResNet is a network structure proposed by the He Kaiming, Sun Jian and others of Microsoft Research Asia in 2015, and won the first place in the ILSVRC-2015 classification task. At the same time, it won the first place in ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation tasks. It was a sensation at the time.

ResNet, also known as residual neural network, refers to the idea of adding residual learning to the traditional convolutional neural network, which solves the problem of gradient dispersion and accuracy degradation (training set) in deep networks, so that the network can get more and more The deeper, both the accuracy and the speed are controlled.

> Deep Residual Learning for Image Recognition Original link : ResNet Paper (https://arxiv.org/pdf/1512.03385.pdf)

The problem caused by increasing depth

- The first problem brought by increasing depth is the problem of gradient explosion / dissipation . This is because as the number of layers increases, the gradient of backpropagation in the network will become unstable with continuous multiplication, and become particularly large or special. small. Among them , the problem of gradient dissipation often occurs .
- In order to overcome gradient dissipation, many solutions have been devised, such as using BatchNorm, replacing the activation function with ReLu, using Xaiver initialization, etc. It can be said that gradient dissipation has been well solved
- Another problem of increasing depth is the problem of network degradation, that is, as the depth increases, the performance of the network will become worse and worse, which is directly reflected in the decrease in accuracy on the training set. The residual network article solves this problem. And after this problem is solved, the depth of the network has increased by several orders of magnitude.

Degradation of deep network

> With network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a favored deep model leads to higher training error.

Degradation problem

"with the network depth increasing, accuracy gets saturated"

Not caused by overfitting:

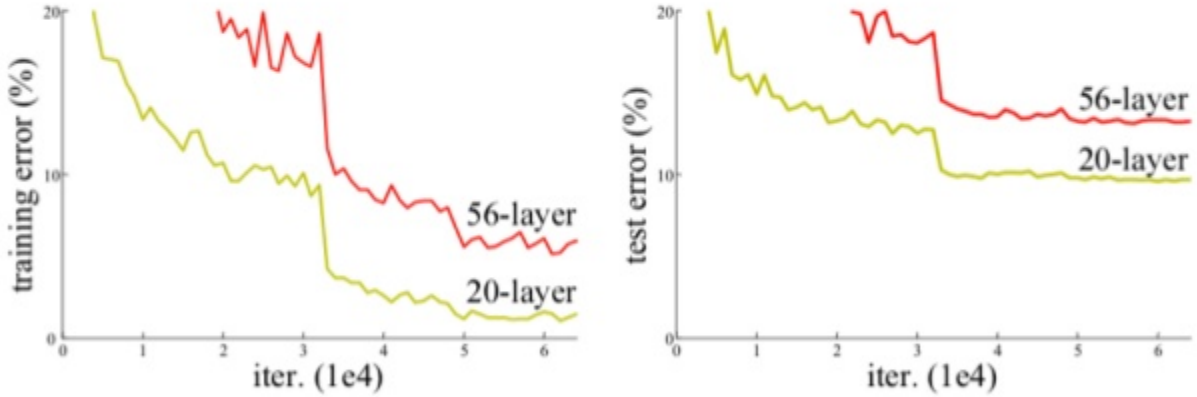


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error.

- The above figure is the error rate of the training set classified by the network on the CIFAR10-data set with the increase of the network depth . It can be seen that if we directly stack the convolutional layers, as the number of layers increases, the error rate increases significantly. The trend is that the deepest 56-layer network has the worst accuracy . We verified it on the VGG network. For the CIFAR-10 dataset, it took 5 minutes on the 18-layer VGG network to get the full network training. The 80% accuracy rate was achieved, and the 34-layer VGG model took 8 minutes to get the 72% accuracy rate. The problem of network degradation does exist.
- The decrease in the training set error rate indicates that the problem of degradation is not caused by overfitting. The specific reason is that it is left for further study. The author's other paper "Identity Mappings in Deep Residual Networks" proved the occurrence of degradation. It is because the optimization performance is not good, which indicates that the deeper the network, the more difficult the reverse gradient is to conduct.

Deep Residual Networks

From 10 to 100 layers

We can imagine that *when we simply stack the network directly to a particularly long length, the internal characteristics of the network have reached the best situation in one of the layers. At this time, the remaining layers should not make any changes to the characteristics and learn automatically. The form of identity mapping*. That is to say, for a particularly deep deep network, the solution space of the shallow form of the network should be a subset of the solution space of the deep network, in other words, a network deeper than the shallow network will not have at least Worse effect, but this is not true because of network degradation.

Then, we settle for the second best. In the case of network degradation, if we do not add depth, we can improve the accuracy. Can we at least make the deep network achieve the same performance as the shallow network, that is, let the layers behind the deep network achieve at least The role of identity mapping . Based on this idea, the author proposes a residual module to help the network achieve identity mapping.

To understand ResNet, we must first understand what kind of problems will occur when the network becomes deeper.

The first problem brought by increasing the network depth is the disappearance and explosion of the gradient.

This problem was successfully solved after Szegedy proposed the **BN (Batch Normalization)** structure. The BN layer can normalize the output of each layer. The size can still be kept stable after the reverse layer transfer, and it will not be too small or too large.

Is it easy to converge after adding BN and then increasing the depth?

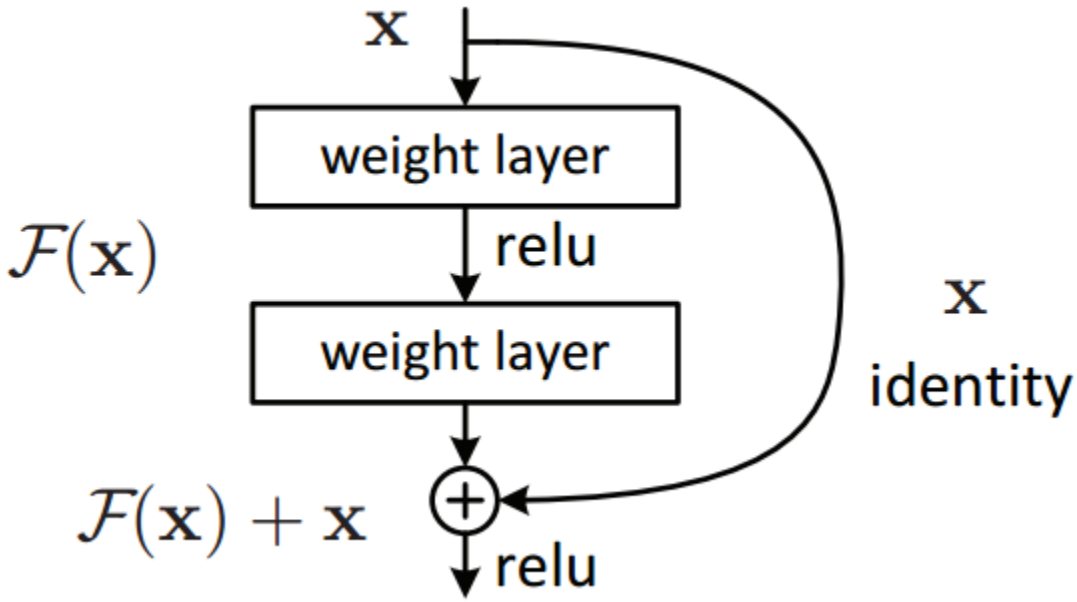
The answer is still **negative**. The author mentioned the second problem-**the degradation problem**: when the level reaches a certain level, the accuracy will saturate and then decline rapidly. This decline is not caused by the disappearance of the gradient. It is not caused by overfit, but because the network is so complicated that it is difficult to achieve the ideal error rate by unconstrained stocking training alone.

The degradation problem is not a problem of the network structure itself, but is caused by the current insufficient training methods. The currently widely used training methods, whether it is SGD, AdaGrad, or RMSProp, cannot reach the theoretically optimal convergence result after the network depth becomes larger.

We can also prove that as long as there is an ideal training method, deeper networks will definitely perform better than shallow networks.

The proof process is also very simple: Suppose that several layers are added behind a network A to form a new network B. If the added level is just an identity mapping of the output of A, that is, the output of A is after the level of B becomes the output of B, there is no change, so the error rates of network A and network B are equal, which proves that the deepened network will not be worse than the network before deepening.

He Kaiming proposed a residual structure to implement the above identity mapping (Below Figure): In addition to the normal convolution layer output, the entire module has a branch directly connecting the input to the output. The output and the output of the convolution do The final output is obtained by arithmetic addition. The formula is H (x) = F (x) + x, x is the input, F (x) is the output of the convolution branch, and H (x) is the output of the entire structure. It can be shown that if all parameters in the F (x) branch are 0, H (x) is an identity mapping. The residual structure artificially creates an identity map, which can make the entire structure converge in the direction of the identity map, ensuring that the final error rate will not become worse because the depth becomes larger. If a network can achieve the desired result by simply setting the parameter values by hand, then this structure can easily converge to the result through training. This is a rule that is unsuccessful when designing complex networks. Recall that in order to restore the original distribution after BN processing, the formula y = rx + delta is used. When r is manually set to standard deviation and delta is the mean, y is the distribution before BN processing. This is the use of this Rules.



What does residual learning mean?

The idea of residual learning is the above picture, which can be understood as a block, defined as follows:

$$y = F(x, \{W_i\}) + x$$

The residual learning block contains two branches or two mappings:

1. Identity mapping refers to the curved curve on the right side of the figure above. As its name implies, identity mapping refers to its own mapping, which is x itself;
2. *F*(x) Residual mapping refers to another branch, that is, part. This part is called residual mapping (y -x) .

What role does the residual module play in back propagation?

- The residual module will significantly reduce the parameter value in the module, so that the parameters in the network have a more sensitive response ability to the loss of reverse conduction, although the fundamental It does not solve the problem that the loss of backhaul is too small, but it reduces the parameters. Relatively speaking, it increases the effect of backhaul loss and also generates a certain regularization effect.
- Secondly, because there are branches of the identity mapping in the forward process, the gradient conduction in the back-propagation process also has more simple paths , and the gradient can be transmitted to the previous module after only one relu.
- The so-called backpropagation is that the network outputs a value, and then compares it with the real value to an error loss. At the same time, the loss is changed to change the parameter. The returned loss depends on the original loss and gradient. Since the purpose is to change the parameter, The problem is that if the intensity of changing the parameter is too small, the value of the parameter can be reduced, so that the loss of the intensity of changing the parameter is relatively greater.
- Therefore, the most important role of the residual module is to change the way of forward and backward information transmission, thereby greatly promoting the optimization of the network.
- Using the four criteria proposed by Inceptionv3, we will use them again to improve the residual module. Using criterion 3, the dimensionality reduction before spatial aggregation will not cause information loss, so the same method is also used here, adding 1 * 1 convolution The kernel is used to increase the non-linearity and reduce the depth of the output to reduce the computational cost. You get the form of a residual module that becomes a bottleneck. The figure above shows the basic form on the left and the bottleneck form on the right.

- To sum up, the shortcut module will help the features in the network perform identity mapping in the forward process, and help conduct gradients in the reverse process, so that deeper models can be successfully trained.

Why can the residual learning solve the problem of "the accuracy of the network deepening declines"?

For a neural network model, if the model is optimal, then training can easily optimize the residual mapping to 0, and only identity mapping is left at this time. No matter how you increase the depth, the network will always be in an optimal state in theory. Because it is equivalent to all the subsequent added networks to carry information transmission along the identity mapping (self), it can be understood that the number of layers behind the optimal network is discarded (without the ability to extract features), and it does not actually play a role. . In this way, the performance of the network will not decrease with increasing depth.

The author used two types of data, **ImageNet** and **CIFAR**, to prove the effectiveness of ResNet:

The first is ImageNet. The authors compared the training effect of ResNet structure and traditional structure with the same number of layers. The left side of Figure is a VGG-19 network with a traditional structure (each followed by BN), the middle is a 34-layer network with a traditional structure (each followed by BN), and the right side is 34 layers ResNet (the solid line indicates a direct connection, and the dashed line indicates a dimensional change using 1x1 convolution to match the number of features of the input and output). Figure 3 shows the results after training these types of networks.

The data on the left shows that the 34-layer network (red line) with the traditional structure has a higher error rate than the VGG-19 (blue-green line). Because the BN structure is added to each layer Therefore, the high error is not caused by the gradient disappearing after the level is increased, but by the degradation problem; the ResNet structure on the right side of Figure 3 shows that the 34-layer network (red line) has a higher error rate than the 18-layer network (blue-green line). Low, this is because the ResNet structure has overcome the degradation problem. In addition, the final error rate of the ResNet 18-layer network on the right is similar to the error rate of the traditional 18-layer network on the left. This is because the 18-layer network is simpler and can converge to a more ideal result even without the ResNet structure.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

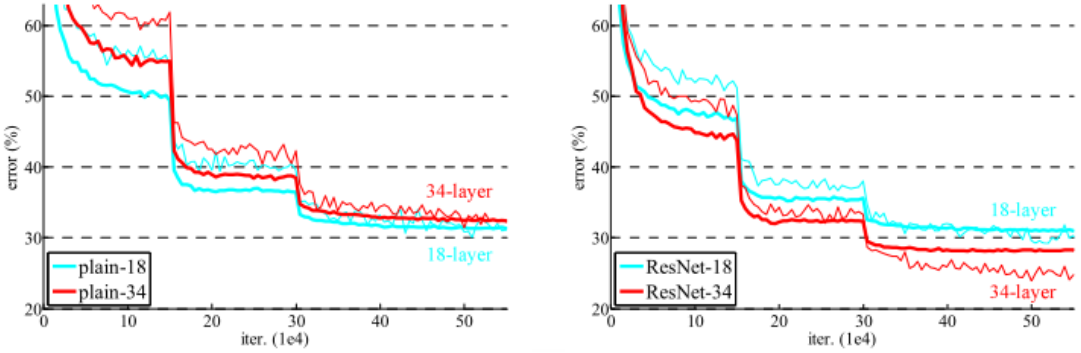
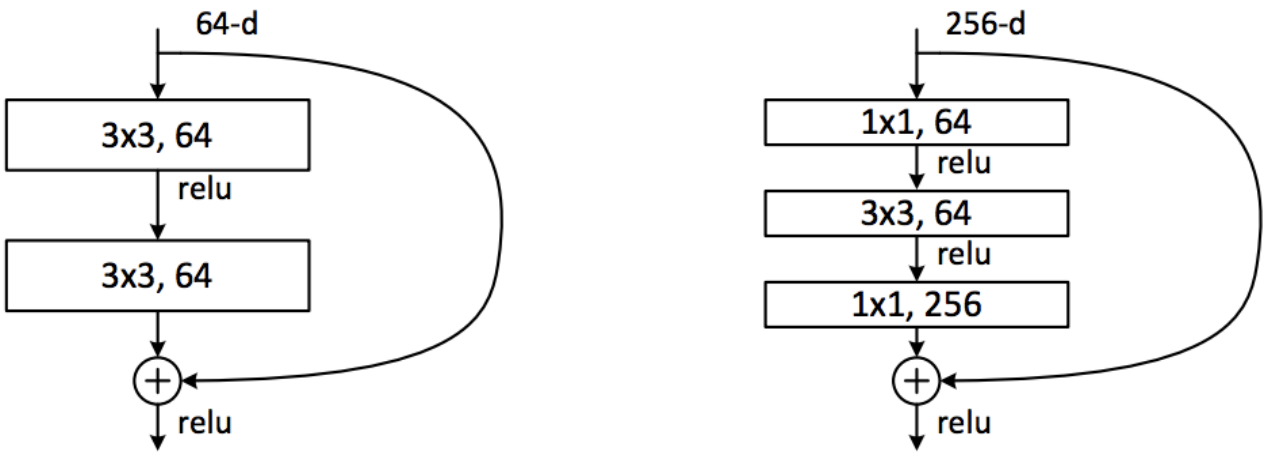


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

The ResNet structure like the left side of Fig. 4 is only used for shallow ResNet networks. If there are many network layers, the dimensions near the output end of the network will be very large. Still using the structure on the left side of Fig. 4 will cause a huge amount of calculation. For deeper networks, we all use the bottleneck structure on the right side of Figure 4, first using a 1x1 convolution for dimensionality reduction, then 3x3 convolution, and finally using 1x1 dimensionality to restore the original dimension.

In practice, considering the cost of the calculation, the residual block is calculated and optimized, that is, the two 3x3 convolution layers are replaced with 1x1 + 3x3 + 1x1 , as shown below. The middle 3x3 convolutional layer in the new structure first reduces the calculation under one dimensionality-reduced 1x1 convolutional layer , and then restores it under another 1x1 convolutional layer , both maintaining accuracy and reducing the amount of calculation .



This is equivalent to reducing the amount of parameters for the same number of layers , so it can be extended to deeper models. So the author proposed ResNet with 50, 101 , and 152 layers , and not only did not have degradation problems, the error rate was greatly reduced, and the computational complexity was also kept at a very low level .

At this time, the error rate of ResNet has already dropped other networks a few streets, but it does not seem to be satisfied. Therefore, a more abnormal 1202 layer network has been built. For such a deep network, optimization is still not difficult, but it appears The problem of overfitting is quite normal. The author also said that the 1202 layer model will be further improved in the future.

Diffrent Variants : -

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$

Code Implementation

From Scratch

In [1]:
```
!pip install tflearn
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) https://us-python.pkg.dev/colab-wheels/public/simple/ (https://us-python.pkg.dev/colab-wheels/public/simple/)
Collecting tflearn
  Downloading tflearn-0.5.0.tar.gz (107 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 107.3/107.3 kB 9.7 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from tflearn) (1.22.4)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from tflearn) (1.16.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from tflearn) (8.4.0)
Building wheels for collected packages: tflearn
  Building wheel for tflearn (setup.py) ... done
  Created wheel for tflearn: filename=tflearn-0.5.0-py3-none-any.whl size=127283 sha256=8a88a556a32a46e730ec7bc096cd085cfa75e78702e726ebedc481ef0341cda6
  Stored in directory: /root/.cache/pip/wheels/55/fb/7b/e06204a0ceefa45443930b9a250cb5ebe31def0e4e8245a465
Successfully built tflearn
Installing collected packages: tflearn
Successfully installed tflearn-0.5.0
```

In [2]:
```
from tensorflow import keras
from keras.models import Model
from keras.layers import Conv2D, BatchNormalization, Activation, Add, AveragePooling2D,
from keras.optimizers import Adam
```

In [3]:
```
# Get Data
import tflearn.datasets.oxflower17 as oxflower17
from keras.utils import to_categorical

x, y = oxflower17.load_data()

x_train = x.astype('float32') / 255.0
y_train = to_categorical(y, num_classes=17)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/compa
t/v2_compat.py:107: disable_resource_variables (from tensorflow.python.ops.variable_sco
pe) is deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term

Downloading Oxford 17 category Flower Dataset, Please wait...

100.0% 60276736 / 60270631

Succesfully downloaded 17flowers.tgz 60270631 bytes.
File Extracted
Starting to parse images...
Parsing Done!
```

In [4]:
```
print(x_train.shape)
print(y_train.shape)
```

```
(1360, 224, 224, 3)
(1360, 17)
```

In [5]:

```
# Residual block
def residual_block(x, filters, downsample=False):
    strides = (2, 2) if downsample else (1, 1)

    # First convolutional layer of the block
    y = Conv2D(filters, kernel_size=(3, 3), strides=strides, padding='same')(x)
    y = BatchNormalization()(y)
    y = Activation('relu')(y)

    # Second convolutional layer of the block
    y = Conv2D(filters, kernel_size=(3, 3), strides=(1, 1), padding='same')(y)
    y = BatchNormalization()(y)

    # Skip connection if downsample or number of filters change
    if downsample:
        x = Conv2D(filters, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

    # Add skip connection
    y = Add()([x, y])
    y = Activation('relu')(y)
    return y

# Build the ResNet model
def resnet(input_shape, num_classes):
    inputs = keras.Input(shape=input_shape)

    # Initial convolutional layer
    x = Conv2D(16, kernel_size=(3, 3), strides=(1, 1), padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # Residual blocks
    x = residual_block(x, filters=16)
    x = residual_block(x, filters=16)
    x = residual_block(x, filters=32, downsample=True)
    x = residual_block(x, filters=32)
    x = residual_block(x, filters=64, downsample=True)
    x = residual_block(x, filters=64)

    # Average pooling and output layer
    x = AveragePooling2D(pool_size=(8, 8))(x)
    x = Flatten()(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=inputs, outputs=outputs)
    return model
```

In [6]:
```
# Create the ResNet model
model = resnet(input_shape=(224, 224, 3), num_classes=17)

# Compile the model
model.compile(optimizer=Adam(lr=0.001), loss='categorical_crossentropy', metrics=['accur

# Train
model.fit(x_train, y_train, batch_size=64, epochs=5, verbose=1,validation_split=0.2, shu
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/keras/layers/normalizat
ion/batch_normalization.py:581: _colocate_with (from tensorflow.python.framework.ops) i
s deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
/usr/local/lib/python3.10/dist-packages/keras/optimizers/legacy/adam.py:117: UserWarnin
g: The `lr` argument is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)

Train on 1088 samples, validate on 272 samples
Epoch 1/5
1088/1088 [==============================] - ETA: 0s - loss: 2.8100 - acc: 0.1967

/usr/local/lib/python3.10/dist-packages/keras/engine/training_v1.py:2335: UserWarning:
`Model.state_updates` will be removed in a future version. This property should not be
used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates

1088/1088 [==============================] - 24s 22ms/sample - loss: 2.8100 - acc: 0.19
67 - val_loss: 2.8402 - val_acc: 0.0699
Epoch 2/5
1088/1088 [==============================] - 9s 9ms/sample - loss: 1.6519 - acc: 0.4550
- val_loss: 2.8888 - val_acc: 0.0735
Epoch 3/5
1088/1088 [==============================] - 9s 9ms/sample - loss: 1.2091 - acc: 0.5928
- val_loss: 2.9678 - val_acc: 0.0735
Epoch 4/5
1088/1088 [==============================] - 10s 9ms/sample - loss: 0.9970 - acc: 0.676
5 - val_loss: 3.1419 - val_acc: 0.0735
Epoch 5/5
1088/1088 [==============================] - 10s 9ms/sample - loss: 0.7557 - acc: 0.744
5 - val_loss: 3.3201 - val_acc: 0.0404
```

Out[6]:
```
<keras.callbacks.History at 0x7f2260d58df0>
```

Pretrained

In [7]:
```
# download the data from g drive

import gdown
url = "https://drive.google.com/file/d/12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP/view?usp=sharin
file_id = url.split("/")[-2]
print(file_id)
prefix = 'https://drive.google.com/uc?/export=download&id='
gdown.download(prefix+file_id, "catdog.zip")
```

```
12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP

Downloading...
From: https://drive.google.com/uc?/export=download&id=12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP
(https://drive.google.com/uc?/export=download&id=12jiQxJzYSYl3wnC8x5wHAhRzzJmmsCXP)
To: /content/catdog.zip
100%|██████████| 9.09M/9.09M [00:00<00:00, 113MB/s]
```

Out[7]:
```
'catdog.zip'
```

In [8]:
```
!unzip catdog.zip
```

```
Archive:  catdog.zip
   creating: train/
   creating: train/Cat/
  inflating: train/Cat/0.jpg
  inflating: train/Cat/1.jpg
  inflating: train/Cat/2.jpg
  inflating: train/Cat/cat.2405.jpg
  inflating: train/Cat/cat.2406.jpg
  inflating: train/Cat/cat.2436.jpg
  inflating: train/Cat/cat.2437.jpg
  inflating: train/Cat/cat.2438.jpg
  inflating: train/Cat/cat.2439.jpg
  inflating: train/Cat/cat.2440.jpg
  inflating: train/Cat/cat.2441.jpg
  inflating: train/Cat/cat.2442.jpg
  inflating: train/Cat/cat.2443.jpg
  inflating: train/Cat/cat.2444.jpg
  inflating: train/Cat/cat.2445.jpg
  inflating: train/Cat/cat.2446.jpg
```

In [9]:

```
from tensorflow import keras

from keras.datasets import cifar10
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten

# Set the path to your training and validation data
train_data_dir = '/content/train'
validation_data_dir = '/content/validation'

# Set the number of training and validation samples
num_train_samples = 2000
num_validation_samples = 800

# Set the number of epochs and batch size
epochs = 5
batch_size = 16

# Load the VGG16 model without the top layer
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Create a new model
model = Sequential()

# Add the base model as a layer
model.add(base_model)

# Add custom layers on top of the base model
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Preprocess the training and validation data
train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
validation_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='binary')

validation_generator = validation_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='binary')

# Train the model
```

```
model.fit(
    train_generator,
    steps_per_epoch=num_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=num_validation_samples // batch_size)

# Save the trained model
model.save('dog_cat_classifier.h5')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5 (https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5)
94765736/94765736 [==============================] - 6s 0us/step
Found 337 images belonging to 2 classes.
Found 59 images belonging to 2 classes.
Epoch 1/5
125/125 [==============================] - 19s 134ms/step - batch: 62.0000 - size: 15.2800 - loss: 1.5985 - acc: 0.9497 - val_loss: 2.8061 - val_acc: 0.9311
Epoch 2/5
125/125 [==============================] - 15s 118ms/step - batch: 62.0000 - size: 15.4000 - loss: 0.1633 - acc: 0.9906 - val_loss: 2.4300 - val_acc: 0.9486
Epoch 3/5
125/125 [==============================] - 16s 129ms/step - batch: 62.0000 - size: 15.2800 - loss: 0.0889 - acc: 0.9916 - val_loss: 2.3081 - val_acc: 0.9649
Epoch 4/5
125/125 [==============================] - 15s 119ms/step - batch: 62.0000 - size: 15.2800 - loss: 0.0966 - acc: 0.9911 - val_loss: 1.9620 - val_acc: 0.9176
Epoch 5/5
125/125 [==============================] - 15s 118ms/step - batch: 62.0000 - size: 15.2800 - loss: 0.0648 - acc: 0.9953 - val_loss: 2.2139 - val_acc: 0.9676
```