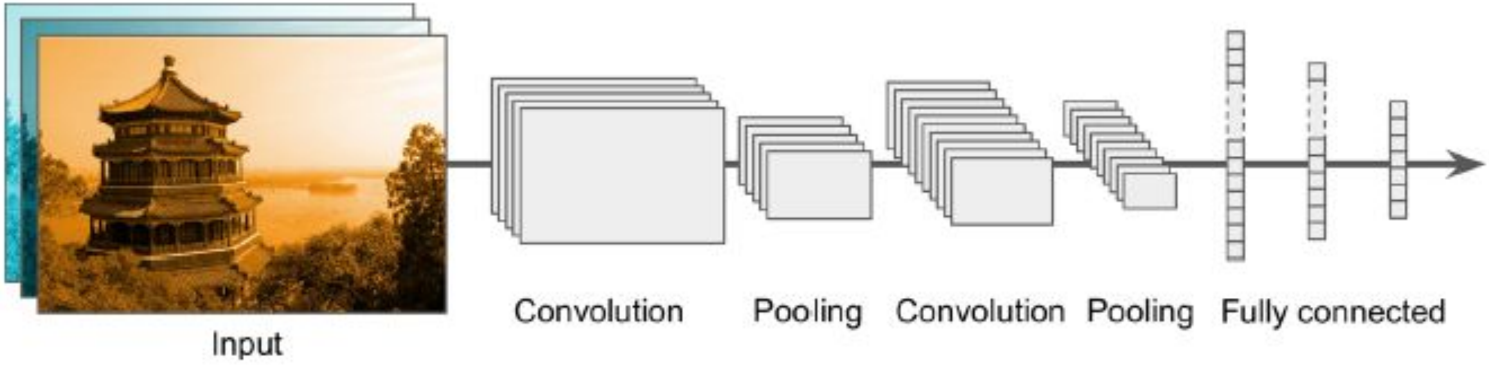
Convolutional Neural Networks (CNN) Architectures Cont.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Lesson #13

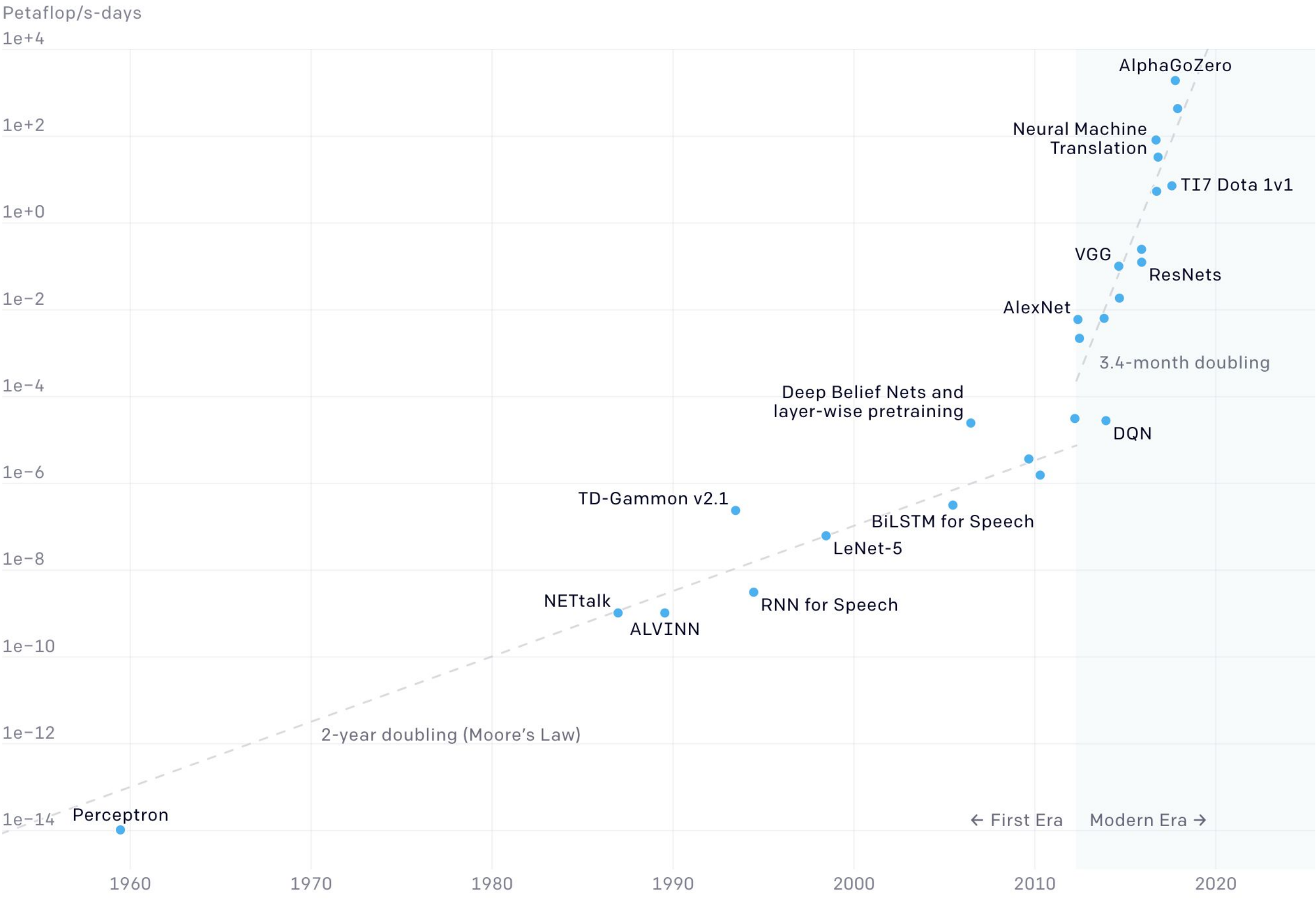
AlexNet (help), VGG, Resnet
Transfer learning & Fine-tuning
Working HDFS and Large Dataset

Typical CNN Architecture



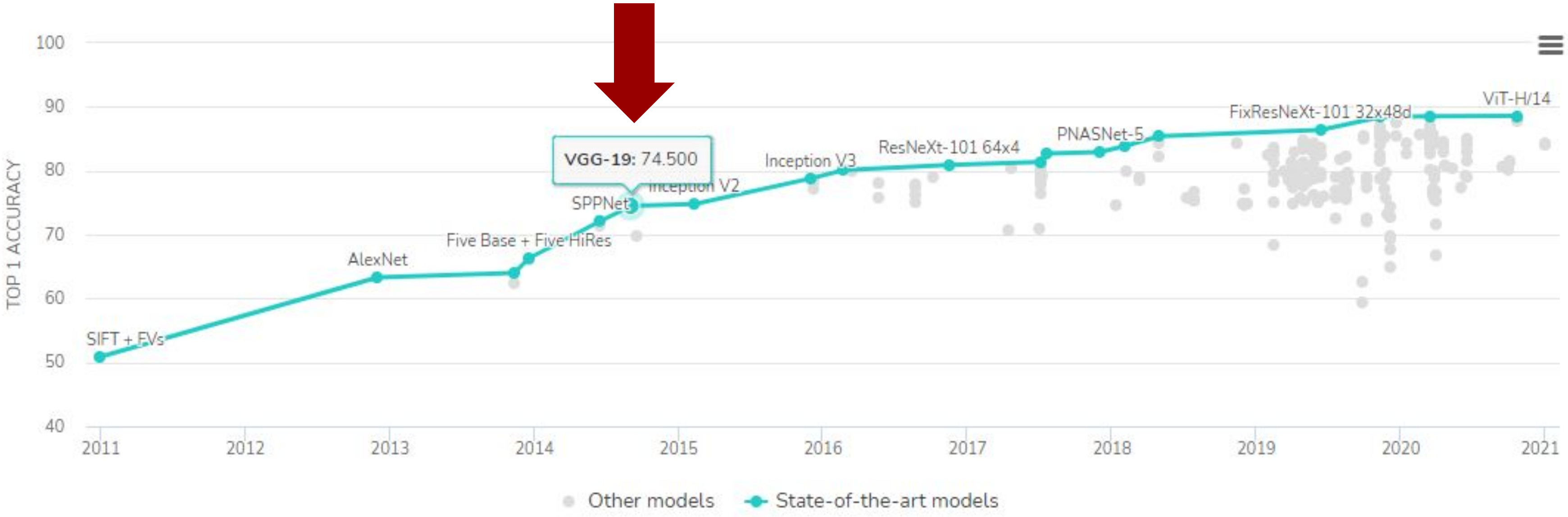
INPUT => [[CONV => RELU]*N => POOL?]*M => [FC => RELU]*K => FC

Two Distinct Eras of Compute Usage in Training AI Systems



http://bit.do/twoeras

Image Classification on ImageNet



https://paperswithcode.com/sota/image-classification-on-imagenet

Published as a conference paper at ICLR 2015

Very Deep Convolutional Networks for Large-Scale Image Recognition

Karen Simonyan* & **Andrew Zisserman**+
Visual Geometry Group, Department of Engineering Science, University of Oxford
{karen,az}@robots.ox.ac.uk

Abstract

In this work we investigate the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. Our main contribution is a thorough evaluation of networks of increasing depth using an architecture with very small (3×3) convolution filters, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16–19 weight layers. These findings were the basis of our ImageNet Challenge 2014 submission, where our team secured the first and the second places in the localisation and classification tracks respectively. We also show that our representations generalise well to other datasets, where they achieve state-of-the-art results. We have made our two best-performing ConvNet models publicly available to facilitate further research on the use of deep visual representations in computer vision.

1 Introduction

Convolutional networks (ConvNets) have recently enjoyed a great success in large-scale image and video recognition (Krizhevsky et al., 2012; Zeiler & Fergus, 2013; Sermanet et al., 2014; Simonyan & Zisserman, 2014) which has become possible due to the large public image repositories, such as ImageNet (Deng et al., 2009), and high-performance computing systems, such as GPUs or large-scale distributed clusters (Dean et al., 2012). In particular, an important role in the advance of deep visual recognition architectures has been played by the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) (Russakovsky et al., 2014), which has served as a testbed for a few generations of large-scale image classification systems, from high-dimensional shallow feature encodings (Perronnin et al., 2010) (the winner of ILSVRC-2011) to deep ConvNets (Krizhevsky et al., 2012) (the winner of ILSVRC-2012).

With ConvNets becoming more of a commodity in the computer vision field, a number of attempts have been made to improve the original architecture of Krizhevsky et al. (2012) in a bid to achieve better accuracy. For instance, the best-performing submissions to the ILSVRC-2013 (Zeiler & Fergus, 2013; Sermanet et al., 2014) utilised smaller receptive window size and smaller stride of the first convolutional layer. Another line of improvements dealt with training and testing the networks densely over the whole image and over multiple scales (Sermanet et al., 2014; Howard, 2014). In this paper, we address another important aspect of ConvNet architecture design – its depth. To this end, we fix other parameters of the architecture, and steadily increase the depth of the network by adding more convolutional layers, which is feasible due to the use of very small (3×3) convolution filters in all layers.

As a result, we come up with significantly more accurate ConvNet architectures, which not only achieve the state-of-the-art accuracy on ILSVRC classification and localisation tasks, but are also applicable to other image recognition datasets, where they achieve excellent performance even when used as a part of a relatively simple pipelines (e.g. deep features classified by a linear SVM without fine-tuning). We have released our two best-performing models[1] to facilitate further research.

VGG

#16 #19

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					

VGG #16 or
VGG #19

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

- Filter size is constant
- Number of filters increase along the architecture

```
# import the necessary packages
from tensorflow.keras.applications import VGG16

model = VGG16(weights="imagenet")
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

Transfer Learning

Large Dataset → Trained DL Model

IMAGENET

Small Dataset → Transfer Learning

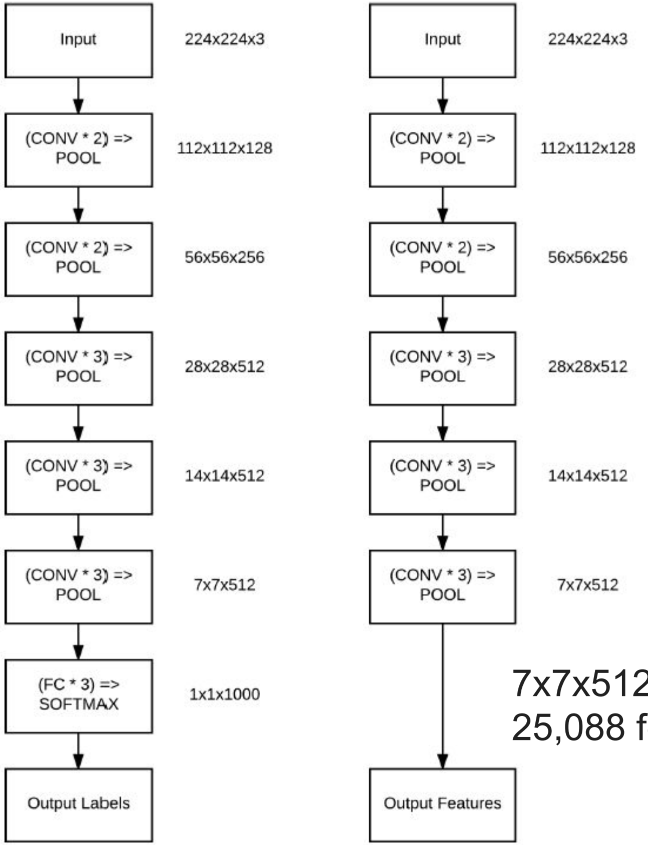
Trained DL Model → Transfer Learning → New DL Model

In general, there are **two types of transfer learning** when applied to deep learning for **computer vision**:

1. Treating networks as arbitrary feature extractors.
2. Removing the fully-connected layers of an existing network, placing new FC layer set on top of the CNN, and fine-tuning these weights (and optionally previous layers) to recognize object classes.

Transfer Learning: Extracting features with a pre-trained CNN

VGG 16

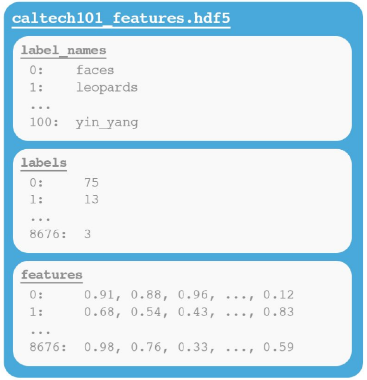


7x7x512
25,088 features

```python
# import the necessary packages
from tensorflow.keras.applications import VGG16

model = VGG16(weights="imagenet", include_top=False)
features = model.predict(batchImages, batch_size=bs)
```

Shallow ML Classifier
(Logistic Regression, RF, Xgboost, etc)

Image	Features 25088 columns	Class
#01		Cat
#02		Cat
#N		Dog



Caltech 101
8677 instances

Animals: Cat, Dog & Panda
3000 instances

Flowers 17
1360 instances

```python
# INPUTS
# path to input dataset
dataset = "animals"

# path to output HDF5 file
output  = "animals/hdf5/features.hdf5"

# size of feature extraction buffer
buffer_size = 1000

# store the batch size in a convenience variable
bs = 32

# feature extraction
feature_extraction(dataset,output,buffer_size,bs)

# train and evaluate
train_and_evaluate(output)
```

```
Database keys ['features', 'label_names', 'labels']
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 0.1}
[INFO] evaluating...
              precision    recall  f1-score   support

        cats       0.96      0.99      0.97       253
        dogs       0.98      0.96      0.97       263
       panda       1.00      0.99      0.99       234

    accuracy                           0.98       750
   macro avg       0.98      0.98      0.98       750
weighted avg       0.98      0.98      0.98       750
```

	precision	recall	f1-score	support
Faces	0.98	0.99	0.99	119
Faces_easy	0.99	0.99	0.99	109
Leopards	0.98	1.00	0.99	55
Motorbikes	1.00	1.00	1.00	195
accordion	1.00	1.00	1.00	12
airplanes	1.00	1.00	1.00	214
⋮	⋮		⋮	
watch	1.00	0.98	0.99	63
water_lilly	1.00	0.33	0.50	12
wheelchair	1.00	1.00	1.00	14
wild_cat	0.90	0.90	0.90	10
windsor_chair	1.00	1.00	1.00	15
wrench	1.00	0.89	0.94	9
yin_yang	0.88	0.88	0.88	16
accuracy			0.95	2170
macro avg	0.94	0.93	0.93	2170
weighted avg	0.96	0.95	0.95	2170

Caltech 101

	precision	recall	f1-score	support
bluebell	1.00	1.00	1.00	15
buttercup	0.86	0.90	0.88	21
coltsfoot	0.94	0.81	0.87	21
cowslip	0.70	0.82	0.76	17
crocus	0.90	0.90	0.90	20
daffodil	0.86	0.86	0.86	22
daisy	1.00	0.94	0.97	16
dandelion	1.00	0.95	0.98	22
fritillary	1.00	0.95	0.97	19
iris	1.00	0.94	0.97	18
lilyvalley	0.75	1.00	0.86	18
pansy	0.90	1.00	0.95	18
snowdrop	0.73	0.65	0.69	17
sunflower	1.00	1.00	1.00	25
tigerlily	0.95	1.00	0.98	21
tulip	0.74	0.67	0.70	21
windflower	0.96	0.90	0.93	29
accuracy			0.90	340
macro avg	0.90	0.90	0.90	340
weighted avg	0.90	0.90	0.90	340

Flowers 17

Large Dataset → Trained DL Model

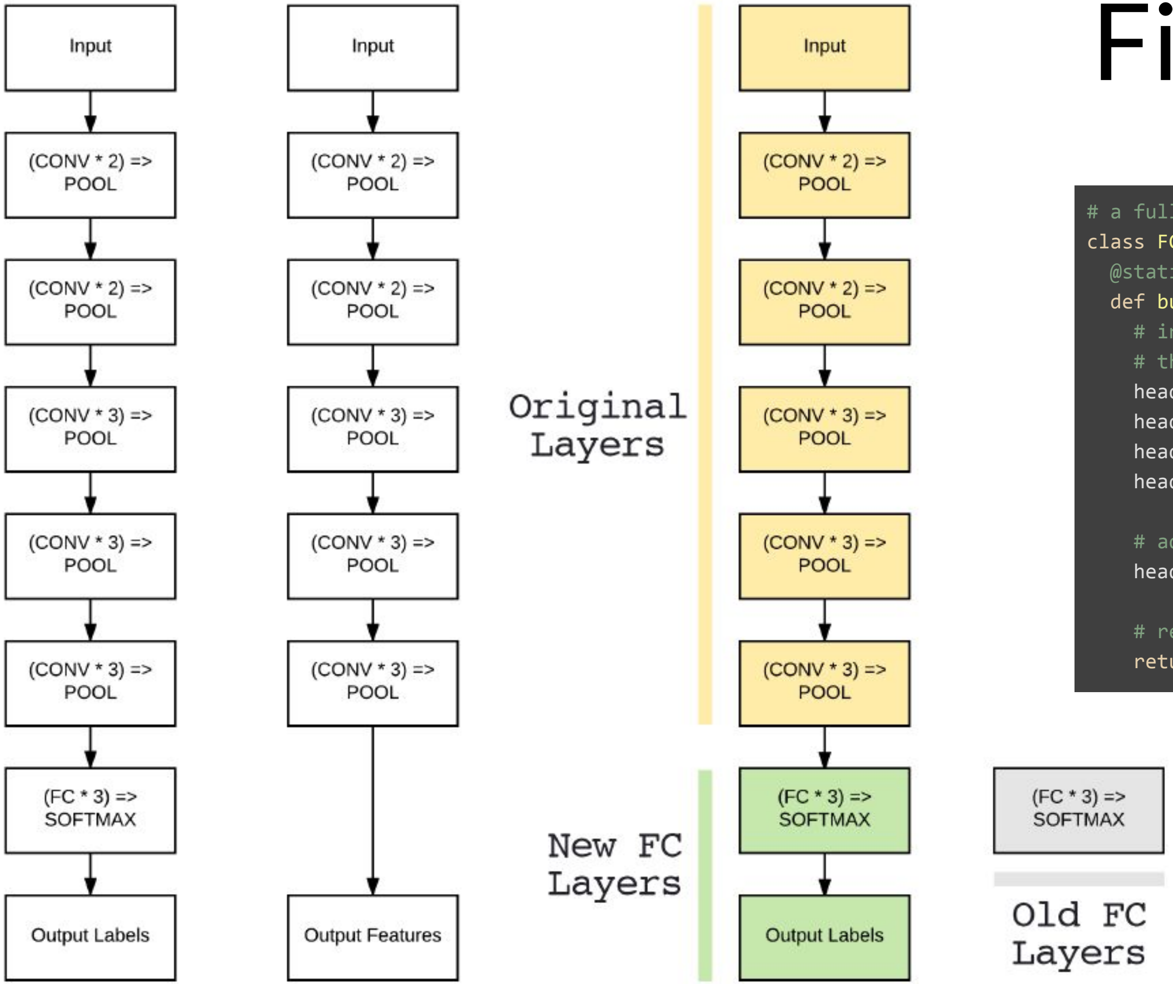
IMAGENET

Small Dataset → Transfer Learning → New DL Model

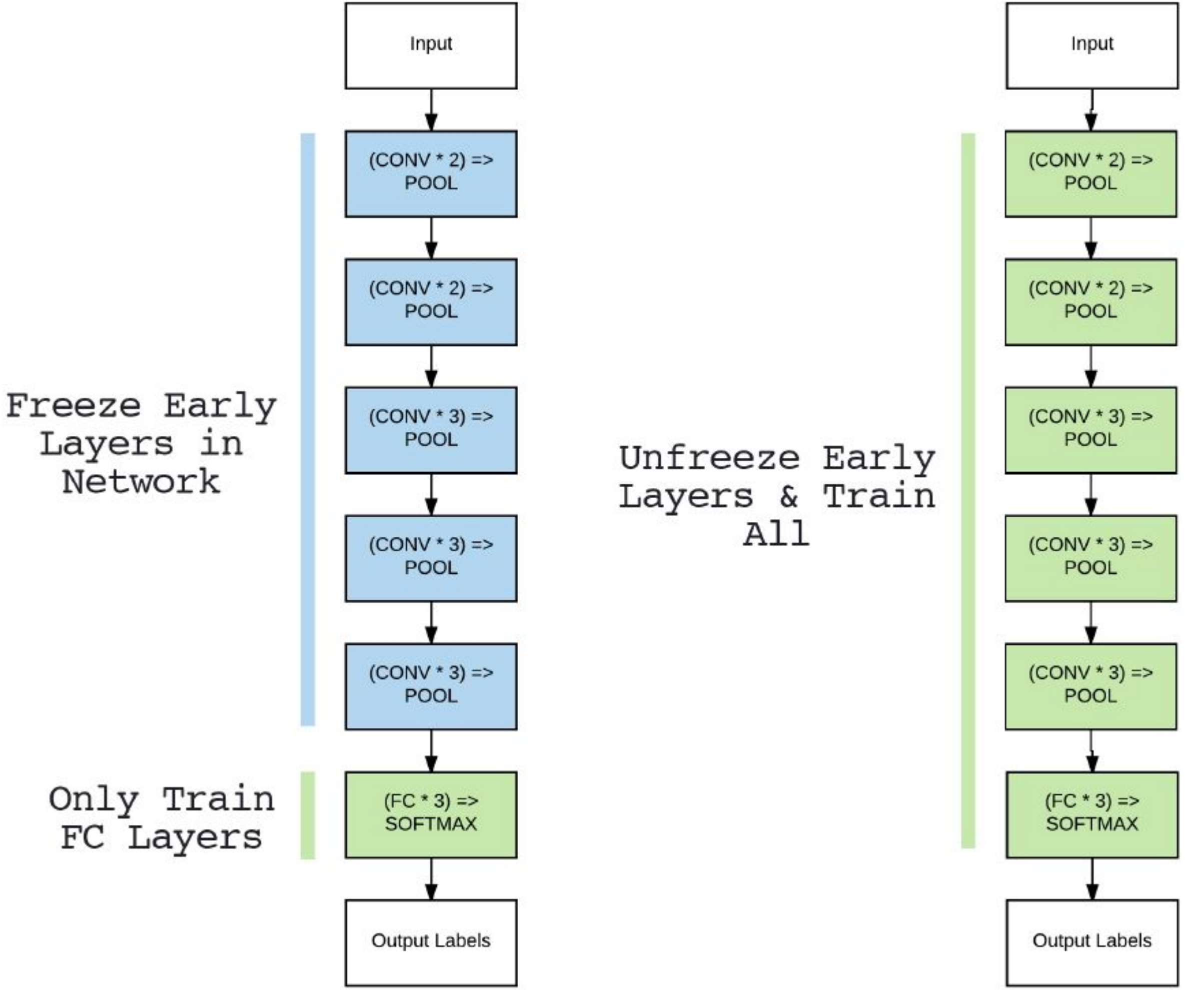
In general, there are **two types of transfer learning** when applied to deep learning for **computer vision**:

1. ~~Treating networks as arbitrary feature extractors.~~
2. Removing the fully-connected layers of an existing network, placing new FC layer set on top of the CNN, and **fine-tuning** these weights (and optionally previous layers) to recognize object classes.

Fine-Tuning




```python
# a fully connect network
class FCHeadNet:
  @staticmethod
  def build(baseModel, classes, D):
    # initialize the head model that will be placed on top of
    # the base, then add a FC layer
    headModel = baseModel.output
    headModel = Flatten(name="flatten")(headModel)
    headModel = Dense(D, activation="relu")(headModel)
    headModel = Dropout(0.5)(headModel)

    # add a softmax layer
    headModel = Dense(classes,activation="softmax")(headModel)

    # return the model
    return headModel
```



RMSprop is frequently used in situations where we need to quickly obtain reasonable performance (first stage - left image).

SGD using a very small learning rate (second stage - right image)

Tip!

```
# loop over all layers in the base model and freeze them so they
# will *not* be updated during the training process
for layer in baseModel.layers:
  layer.trainable = False
```

Let's do a fine tuning using VGG 16 over Flowers 17

Flowers 17
1360 instances

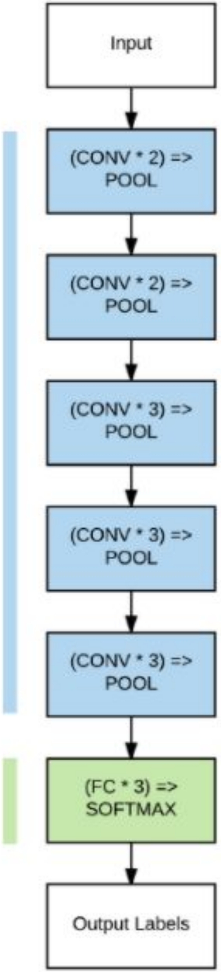
Previous result using feature extraction

	precision	recall	f1-score	support
bluebell	1.00	1.00	1.00	15
buttercup	0.86	0.90	0.88	21
coltsfoot	0.94	0.81	0.87	21
cowslip	0.70	0.82	0.76	17
crocus	0.90	0.90	0.90	20
daffodil	0.86	0.86	0.86	22
daisy	1.00	0.94	0.97	16
dandelion	1.00	0.95	0.98	22
fritillary	1.00	0.95	0.97	19
iris	1.00	0.94	0.97	18
lilyvalley	0.75	1.00	0.86	18
pansy	0.90	1.00	0.95	18
snowdrop	0.73	0.65	0.69	17
sunflower	1.00	1.00	1.00	25
tigerlily	0.95	1.00	0.98	21
tulip	0.74	0.67	0.70	21
windflower	0.96	0.90	0.93	29
accuracy			0.90	340
macro avg	0.90	0.90	0.90	340
weighted avg	0.90	0.90	0.90	340

Stage #01

Freeze Early Layers in Network

Only Train FC Layers



Epochs: 25, RMSProp (0.001), FC (256)

	precision	recall	f1-score	support
bluebell	0.93	0.64	0.76	22
buttercup	0.94	0.74	0.83	23
coltsfoot	0.79	0.95	0.86	20
cowslip	0.55	0.75	0.63	16
crocus	0.93	0.88	0.90	16
daffodil	0.89	0.63	0.74	27
daisy	1.00	0.93	0.97	15
dandelion	1.00	0.70	0.82	20
fritillary	0.89	0.85	0.87	20
iris	0.75	1.00	0.86	18
lilyvalley	0.67	0.86	0.75	21
pansy	0.96	0.96	0.96	23
snowdrop	0.64	0.89	0.74	18
sunflower	1.00	0.96	0.98	23
tigerlily	0.95	0.95	0.95	19
tulip	0.73	0.80	0.76	20
windflower	0.94	0.89	0.92	19
accuracy			0.84	340
macro avg	0.86	0.84	0.84	340
weighted avg	0.86	0.84	0.84	340

Stage #02

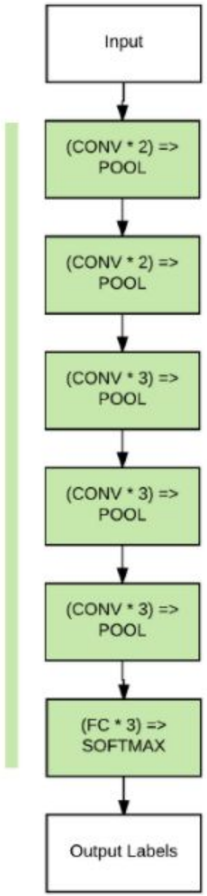
```python
# import the necessary packages
from tensorflow.keras.applications import VGG16

# whether or not to include top of CNN
include_top = 0

# load the VGG16 network
print("[INFO] loading network...")
model = VGG16(weights="imagenet", include_top= include_top > 0)
print("[INFO] showing layers...")

# loop over the layers in the network and display them to the
# console
for (i, layer) in enumerate(model.layers):
  print("[INFO] {}\t{}".format(i, layer.__class__.__name__))
```

VGG-16 without FC

```
[INFO] loading network...
[INFO] showing layers...
[INFO] 0        InputLayer
[INFO] 1        Conv2D
[INFO] 2        Conv2D
[INFO] 3        MaxPooling2D
[INFO] 4        Conv2D
[INFO] 5        Conv2D
[INFO] 6        MaxPooling2D
[INFO] 7        Conv2D
[INFO] 8        Conv2D
[INFO] 9        Conv2D
[INFO] 10       MaxPooling2D
[INFO] 11       Conv2D
[INFO] 12       Conv2D
[INFO] 13       Conv2D
[INFO] 14       MaxPooling2D
[INFO] 15       Conv2D
[INFO] 16       Conv2D
[INFO] 17       Conv2D
[INFO] 18       MaxPooling2D
```

Stage #02

Unfreeze Early Layers & Train All



VGG-16 without FC

```
[INFO] loading network...
[INFO] showing layers...
[INFO] 0         InputLayer
[INFO] 1         Conv2D
[INFO] 2         Conv2D
[INFO] 3         MaxPooling2D
[INFO] 4         Conv2D
[INFO] 5         Conv2D
[INFO] 6         MaxPooling2D
[INFO] 7         Conv2D
[INFO] 8         Conv2D
[INFO] 9         Conv2D
[INFO] 10        MaxPooling2D
[INFO] 11        Conv2D
[INFO] 12        Conv2D
[INFO] 13        Conv2D
[INFO] 14        MaxPooling2D
[INFO] 15        Conv2D
[INFO] 16        Conv2D
[INFO] 17        Conv2D
[INFO] 18        MaxPooling2D
```

```python
# now that the head FC layers have been trained/initialized, lets
# unfreeze the final set of CONV layers and make them trainable
for layer in baseModel.layers[15:]:
  layer.trainable = True
```

Feature extraction

	precision	recall	f1-score	support
bluebell	1.00	1.00	1.00	15
buttercup	0.86	0.90	0.88	21
coltsfoot	0.94	0.81	0.87	21
cowslip	0.70	0.82	0.76	17
crocus	0.90	0.90	0.90	20
daffodil	0.86	0.86	0.86	22
daisy	1.00	0.94	0.97	16
dandelion	1.00	0.95	0.98	22
fritillary	1.00	0.95	0.97	19
iris	1.00	0.94	0.97	18
lilyvalley	0.75	1.00	0.86	18
pansy	0.90	1.00	0.95	18
snowdrop	0.73	0.65	0.69	17
sunflower	1.00	1.00	1.00	25
tigerlily	0.95	1.00	0.98	21
tulip	0.74	0.67	0.70	21
windflower	0.96	0.90	0.93	29
accuracy			0.90	340
macro avg	0.90	0.90	0.90	340
weighted avg	0.90	0.90	0.90	340

Stage #02 fine tuning
SGD (0.001), epochs = 100

	precision	recall	f1-score	support
bluebell	0.91	0.91	0.91	22
buttercup	0.96	0.96	0.96	23
coltsfoot	0.83	0.95	0.88	20
cowslip	0.79	0.94	0.86	16
crocus	0.88	0.94	0.91	16
daffodil	0.96	0.85	0.90	27
daisy	1.00	0.93	0.97	15
dandelion	1.00	0.80	0.89	20
fritillary	0.95	0.95	0.95	20
iris	1.00	1.00	1.00	18
lilyvalley	1.00	0.86	0.92	21
pansy	1.00	0.96	0.98	23
snowdrop	0.75	0.83	0.79	18
sunflower	1.00	0.96	0.98	23
tigerlily	1.00	1.00	1.00	19
tulip	0.83	1.00	0.91	20
windflower	0.95	0.95	0.95	19
accuracy			0.93	340
macro avg	0.93	0.93	0.93	340
weighted avg	0.93	0.93	0.93	340

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [41] but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions[1], where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

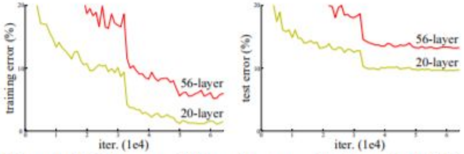


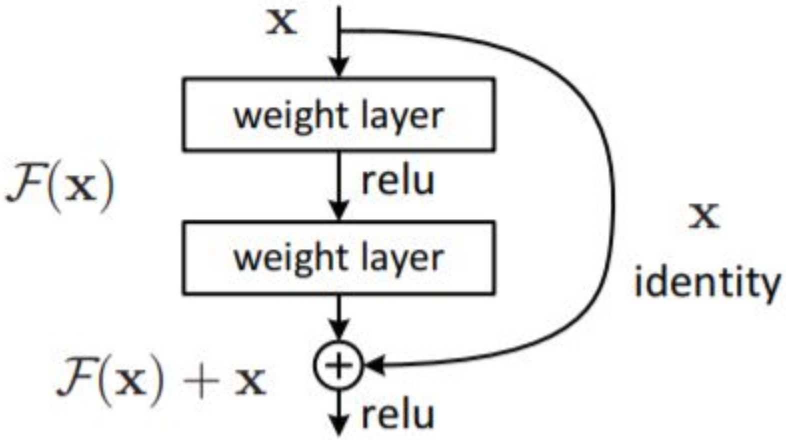
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

greatly benefited from very deep models.

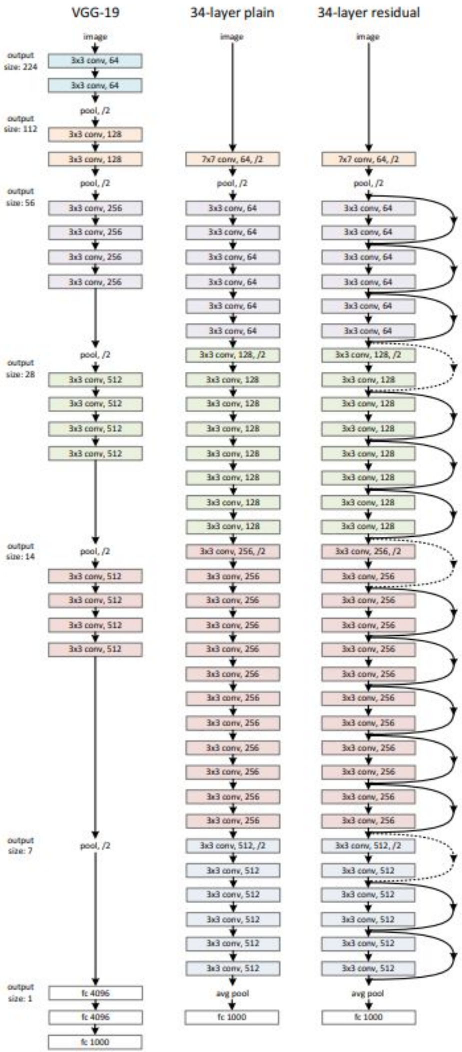
Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [1, 9], which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization [23, 9, 37, 13] and intermediate normalization layers [16], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with backpropagation [22].

When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network

ResNet 152

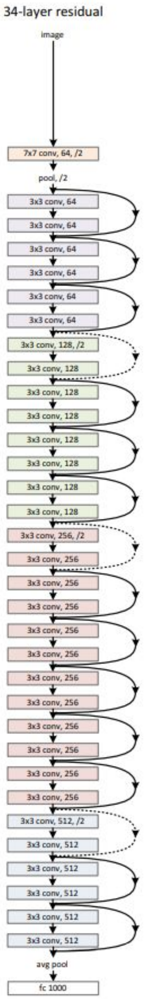


Residual learning block



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

```python
from tensorflow.keras.applications import ResNet50

Model = ResNet50(weights="imagenet")
```



Playground Prediction Competition

Dogs vs. Cats

Create an algorithm to distinguish dogs from cats

Kaggle · 213 teams · 7 years ago

Overview Data Notebooks Discussion Leaderboard Rules Team

Overview

- Description
- Prizes
- Evaluation
- Winners

In this competition, you'll write an algorithm to classify whether images contain either a dog or a cat. This is easy for humans, dogs, and cats. Your computer will find it a bit more difficult.

HDF5
AlexNet
ResNet50
Transfer Learning

http://bit.do/alexnet_epoch45