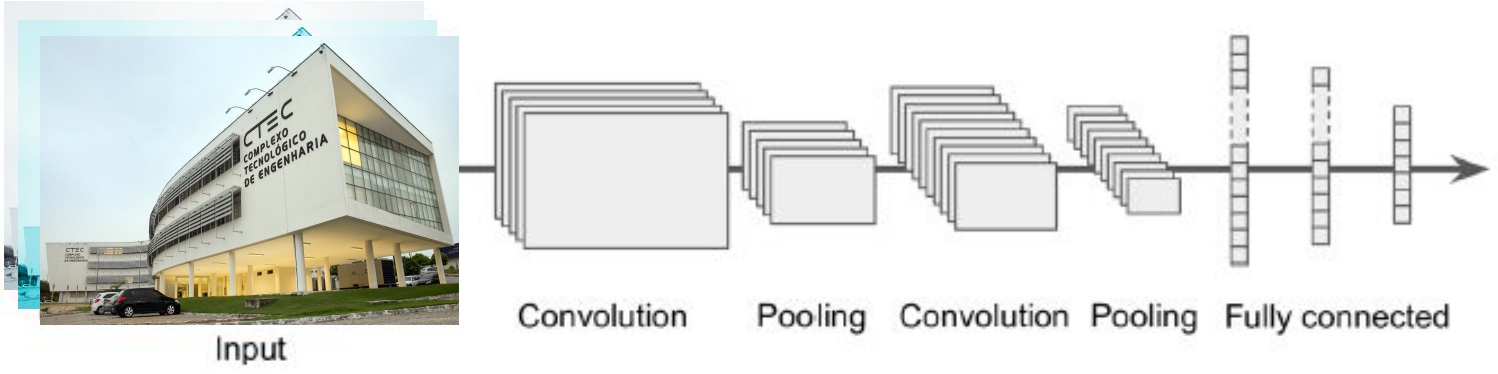
Convolutional Neural Networks (CNN)

Architectures I

CNN Architectures

Typical CNN Architecture

[Layer Patterns]




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

https://cs231n.github.io/convolutional-networks

```
INPUT => CONV => RELU => FC
```

Shallow CNN

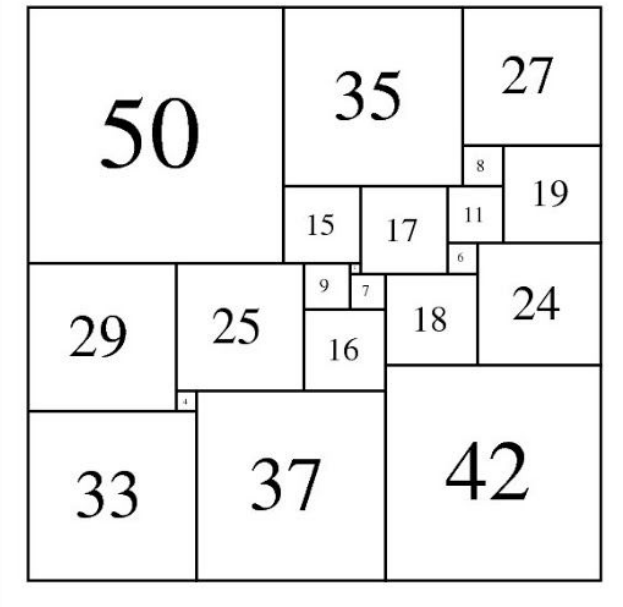
```
INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU] * 3 => POOL =>
     [FC => RELU => DO] * 2 => SOFTMAX
```

AlexNet-like CNN architecture which has multiple CONV => RELU => POOL layer sets, followed by FC layers

```
INPUT => [CONV => RELU] * 2 => POOL => [CONV => RELU] * 2 => POOL =>
     [CONV => RELU] * 3 => POOL => [CONV => RELU] * 3 => POOL =>
     [FC => RELU => DO] * 2 => SOFTMAX
```

For deeper network architectures, such as VGG, we'll stack two (or more) layers before every POOL layer

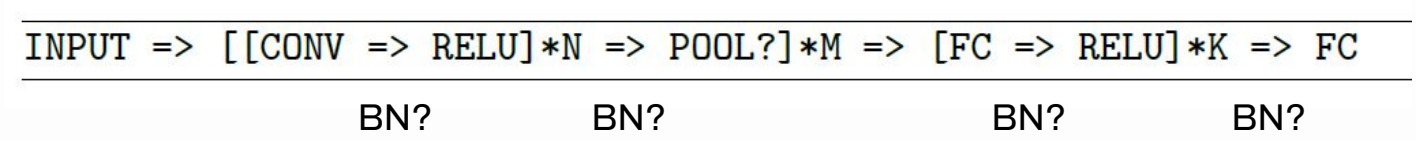
Best Practices



- To start, the images presented to the input layer should be **square**. Using square inputs allows us to take **advantage of linear algebra optimization libraries**.
- Common input layer sizes include 32 x 32, 64 x 64, 96 x 96, 224 x 224, 227 x 227 and 229 x 229 (leaving out the number of channels for notational convenience).

Best Practices

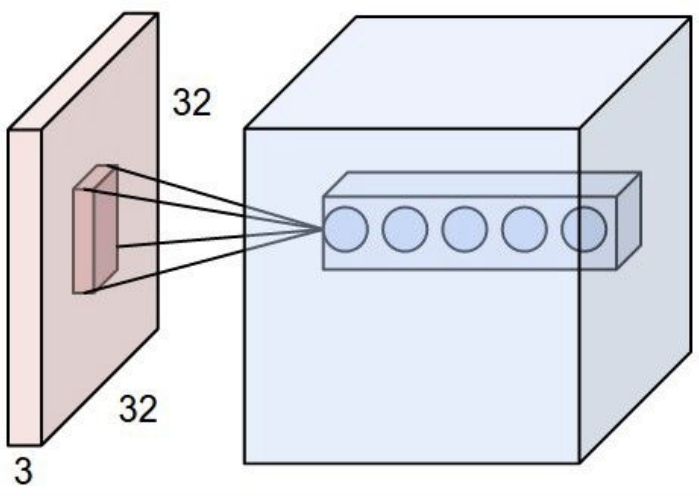
Where do the Batch Normalization layers go?



- Batch normalization has been shown to be extremely effective at **reducing the number of epochs** it takes to train a neural network.
- Batch normalization also has the added benefit of helping **"stabilize" training**, allowing for a larger variety of learning rates and regularization strengths.
- Using batch normalization doesn't alleviate the need to tune these parameters of course, but it will make your life easier by making **learning rate and regularization less volatile** and more **straightforward to tune**.
- The **biggest drawback** of batch normalization is that it can actually slow down the wall time it takes to train your network (even though you'll need fewer epochs to obtain reasonable accuracy) by **2-3x due to the computation** of per-batch statistics and normalization.

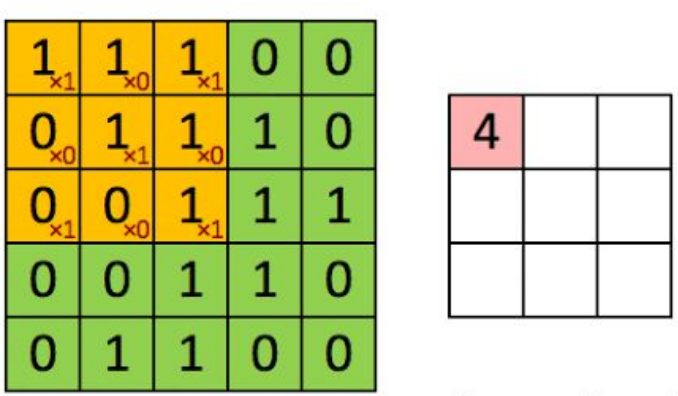
Adrian Rosebrock & Jason Brownlee. Deep Learning for Computer Vision.

Best Practices



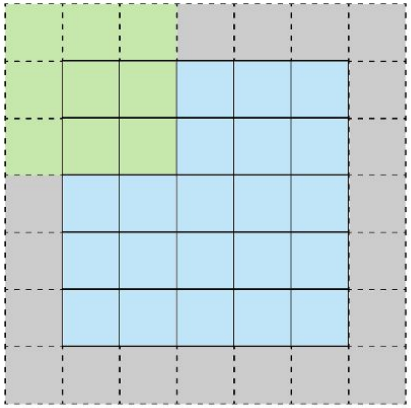
- In general, your CONV layers should use smaller **filter sizes such as 3x3 and 5x5**.
- Tiny **1x1 filters** are used to learn local features, but only in your more **advanced network architectures.**
- Larger filter sizes such as **7x7 and 11x11** may be used as the first CONV layer in the network (> 200x200 pixels). However, after this initial CONV layer the filter size should drop dramatically, otherwise you will reduce the spatial dimensions of your volume too quickly.

Best Practices

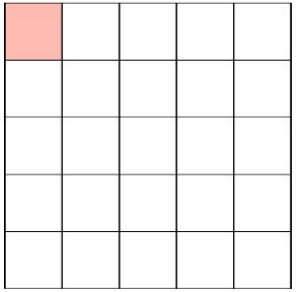


- Use a stride of **S = 1 for CONV layers**, at least for **smaller** spatial **input** volumes
- Networks that accept **larger input volumes** use a stride **S >= 2** in the first CONV layer to help reduce spatial dimensions.
- Using a stride of **S = 1 enables our CONV layers to learn filters while the POOL layer is responsible for downsampling**. However, keep in mind that not all network architectures follow this pattern – some architectures skip max pooling altogether and rely on the CONV stride to reduce volume size.

Best Practices



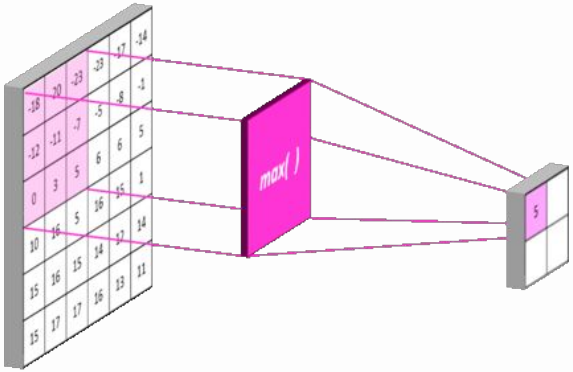
Stride 1 with Padding



Feature Map

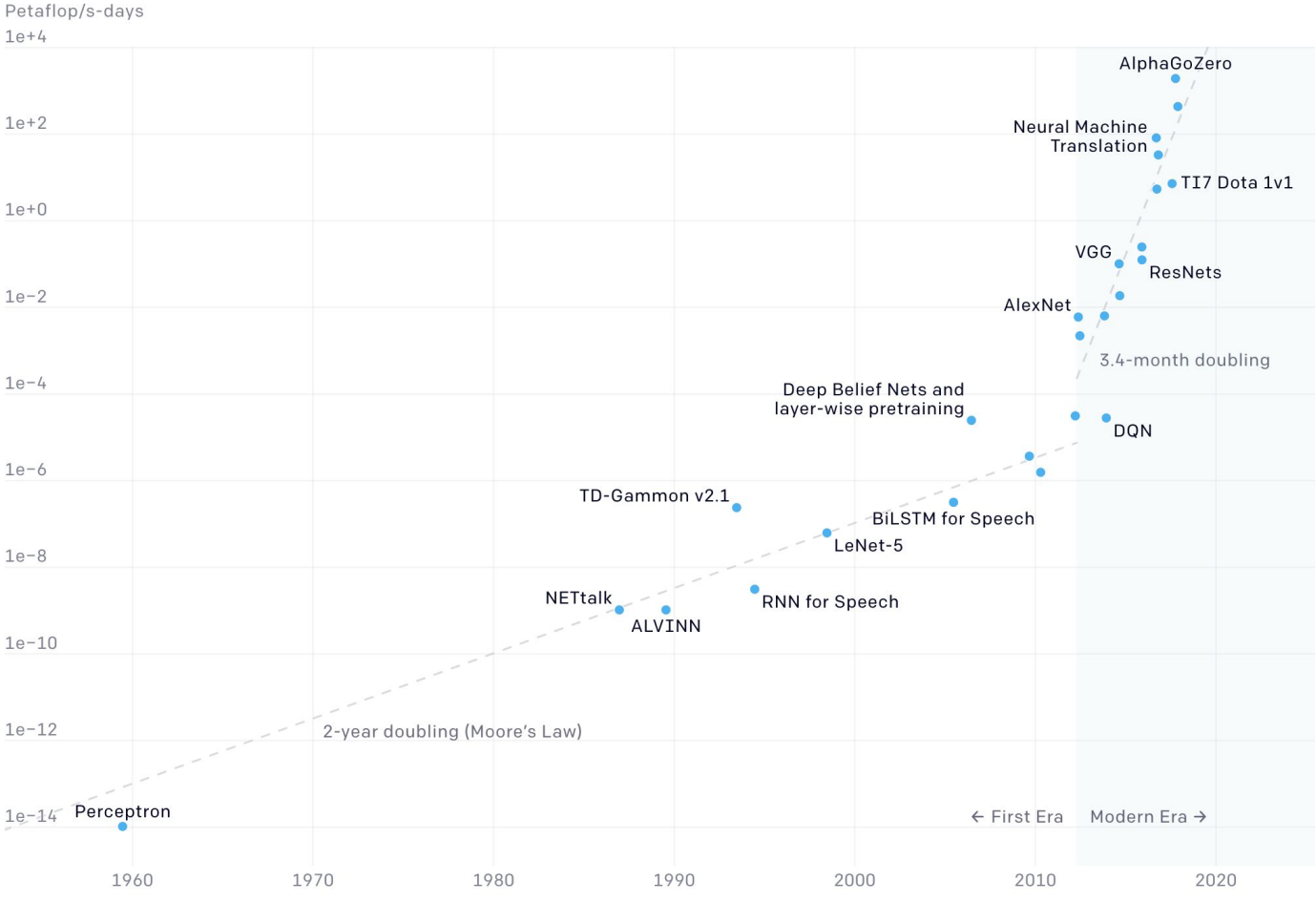
- Apply **zero-padding (same)** to my CONV layers to ensure the output dimension size matches the input dimension size – the only exception to this rule is if I want to purposely reduce spatial dimensions via convolution.
- Applying zero-padding when stacking multiple CONV layers on top of each other has also demonstrated to increase classification accuracy in practice.

Best Practices



- It is recommended to **use POOL layers** (rather than CONV layers) **to reduce the spatial dimensions** of your input, at least until you become more experienced constructing your own CNN architectures. Once you reach that point, you should start experimenting with using CONV layers to reduce spatial input size and try removing max pooling layers from your architecture.
- Most commonly, you'll see **max pooling** applied over a **2x2 receptive field size** and a stride of **S = 2**.
- You might also see a **3x3 receptive field** early in the network architecture to help reduce image size.

Two Distinct Eras of Compute Usage in Training AI Systems



http://bit.do/twoeras

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Abstract—

Multilayer Neural Networks trained with the backpropagation algorithm constitute the best example of a successful Gradient-Based Learning technique. Given an appropriate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten characters, with minimal preprocessing. This paper reviews various methods applied to handwritten character recognition and compares them on a standard handwritten digit recognition task. Convolutional Neural Networks, that are specifically designed to deal with the variability of 2D shapes, are shown to outperform all other techniques.

Real-life document recognition systems are composed of multiple modules including field extraction, segmentation, recognition, and language modeling. A new learning paradigm, called Graph Transformer Networks (GTN), allows such multi-module systems to be trained globally using Gradient-Based methods so as to minimize an overall performance measure.

Two systems for on-line handwriting recognition are described. Experiments demonstrate the advantage of global training, and the flexibility of Graph Transformer Networks.

A Graph Transformer Network for reading bank check is also described. It uses Convolutional Neural Network character recognizers combined with global training techniques to provides record accuracy on business and personal checks. It is deployed commercially and reads several million checks per day.

Keywords— **Neural Networks, OCR, Document Recognition, Machine Learning, Gradient-Based Learning, Convolutional Neural Networks, Graph Transformer Networks, Finite State Transducers.**

I. Introduction

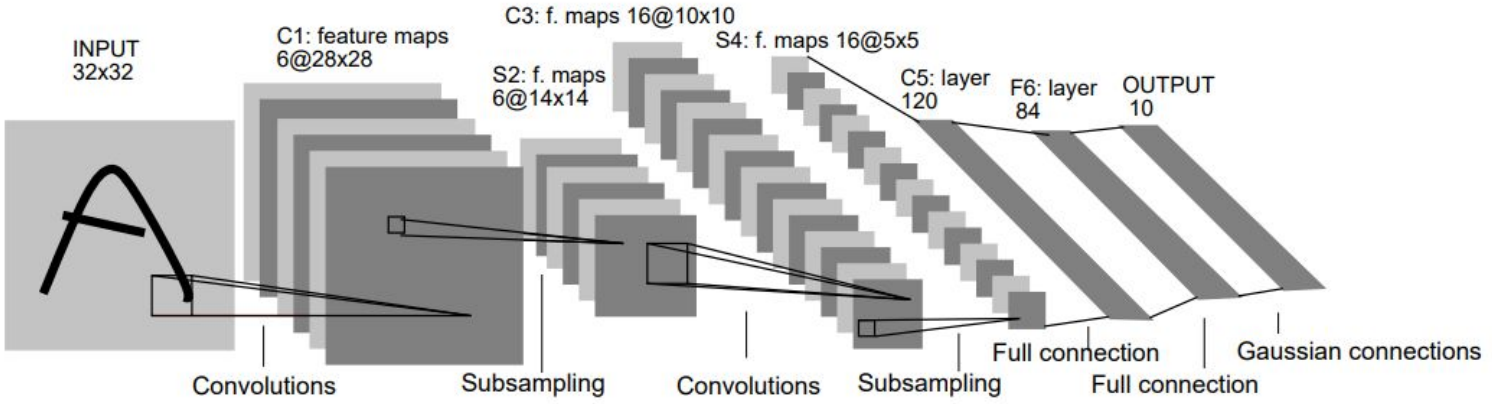
Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the availability of learning techniques has been a crucial factor in the recent success of pattern recognition applications such as continuous speech recognition and handwriting recognition.

The main message of this paper is that better pattern recognition systems can be built by relying more on automatic learning, and less on hand-designed heuristics. This is made possible by recent progress in machine learning and computer technology. Using character recognition as a case study, we show that hand-crafted feature extraction can be advantageously replaced by carefully designed learning machines that operate directly on pixel images. Using document understanding as a case study, we show that the traditional way of building recognition systems by manually integrating individually designed modules can be replaced by a unified and well-principled design paradigm, called *Graph Transformer Networks*, that allows training all the modules to optimize a global performance criterion.

Since the early days of pattern recognition it has been known that the variability and richness of natural data, be it speech, glyphs, or other types of patterns, make it almost impossible to build an accurate recognition system

http://yann.lecun.com/exdb/lenet/

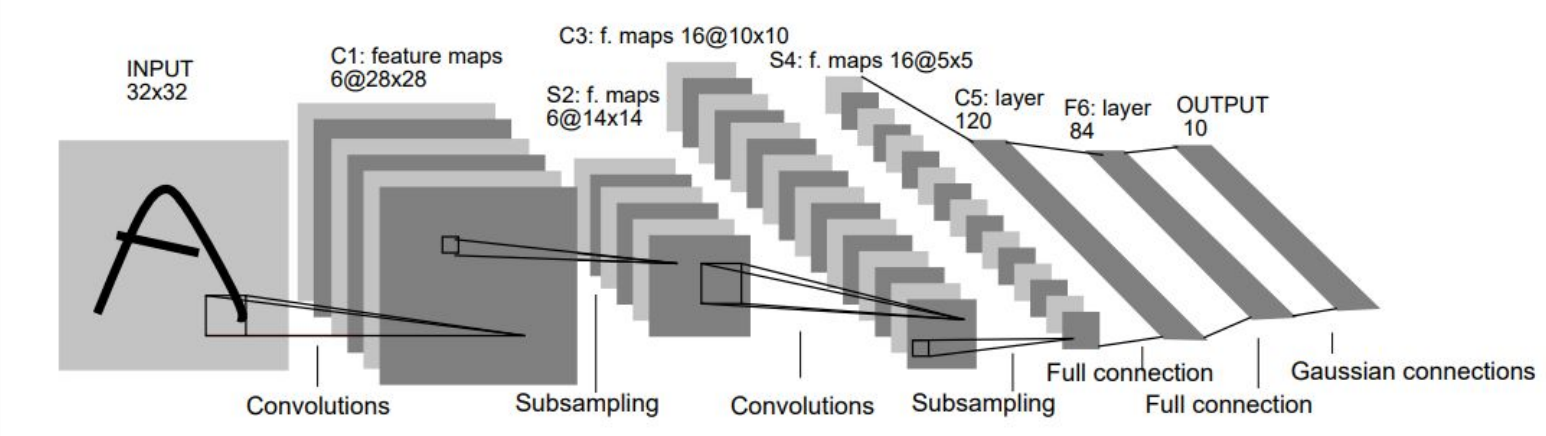
LeNet-5




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

- Fixed-sized input images.
- Group convolutional and pooling layers into blocks.
- Repetition of convolutional-pooling blocks in the architecture.
- Increase in the number of filters with the depth of the network.
- Distinct feature extraction and classifier parts of the architecture.

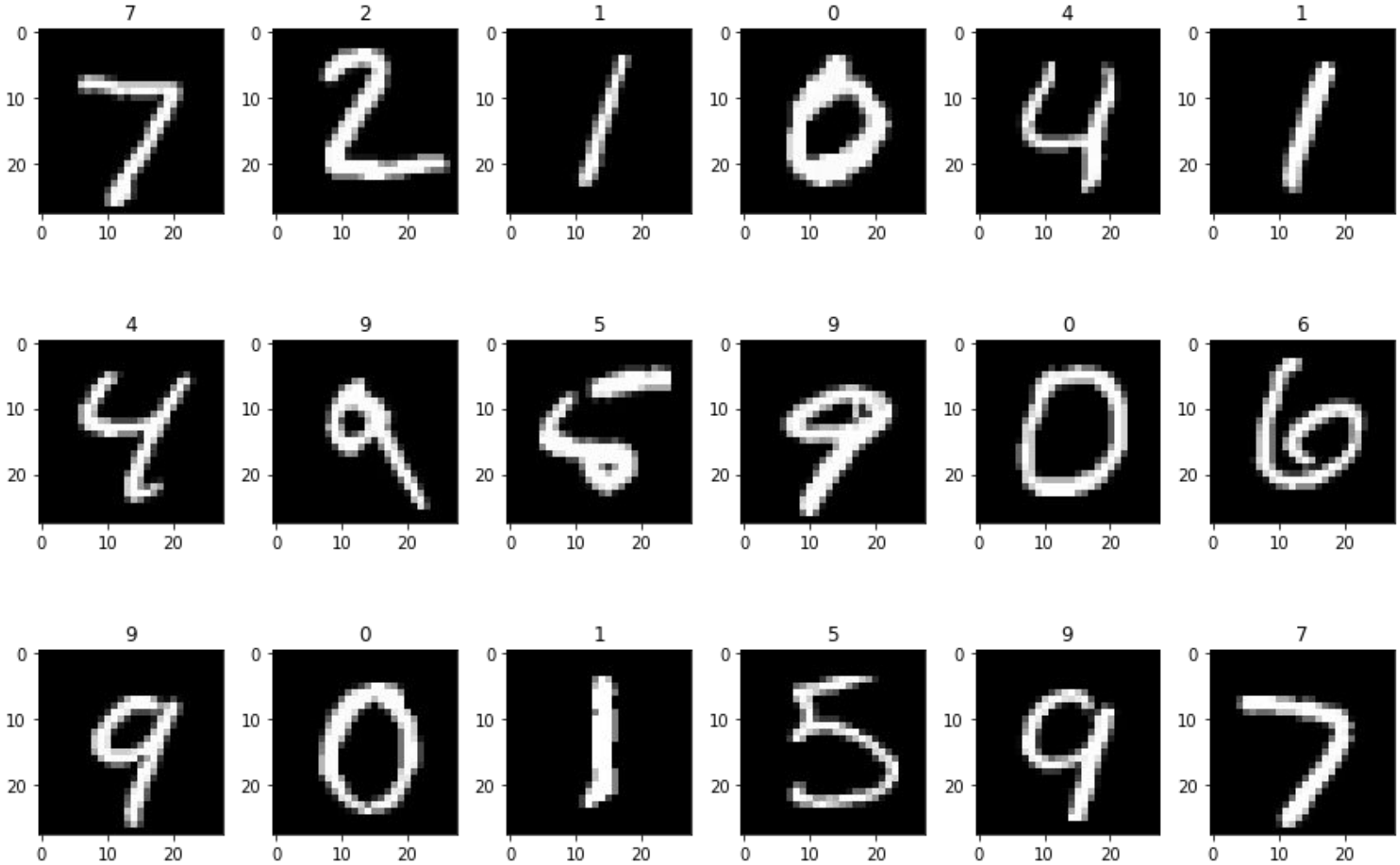
LeNet-5

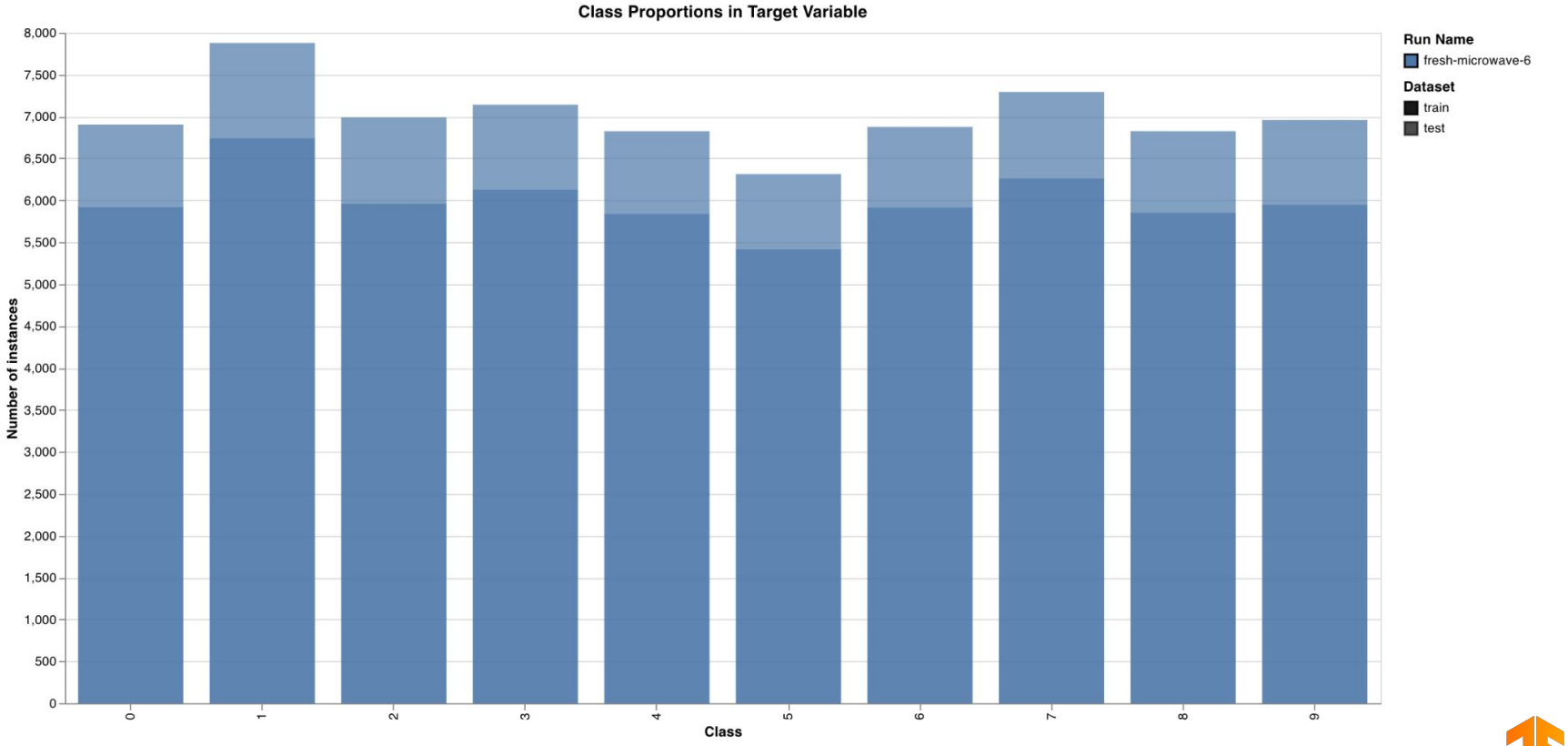


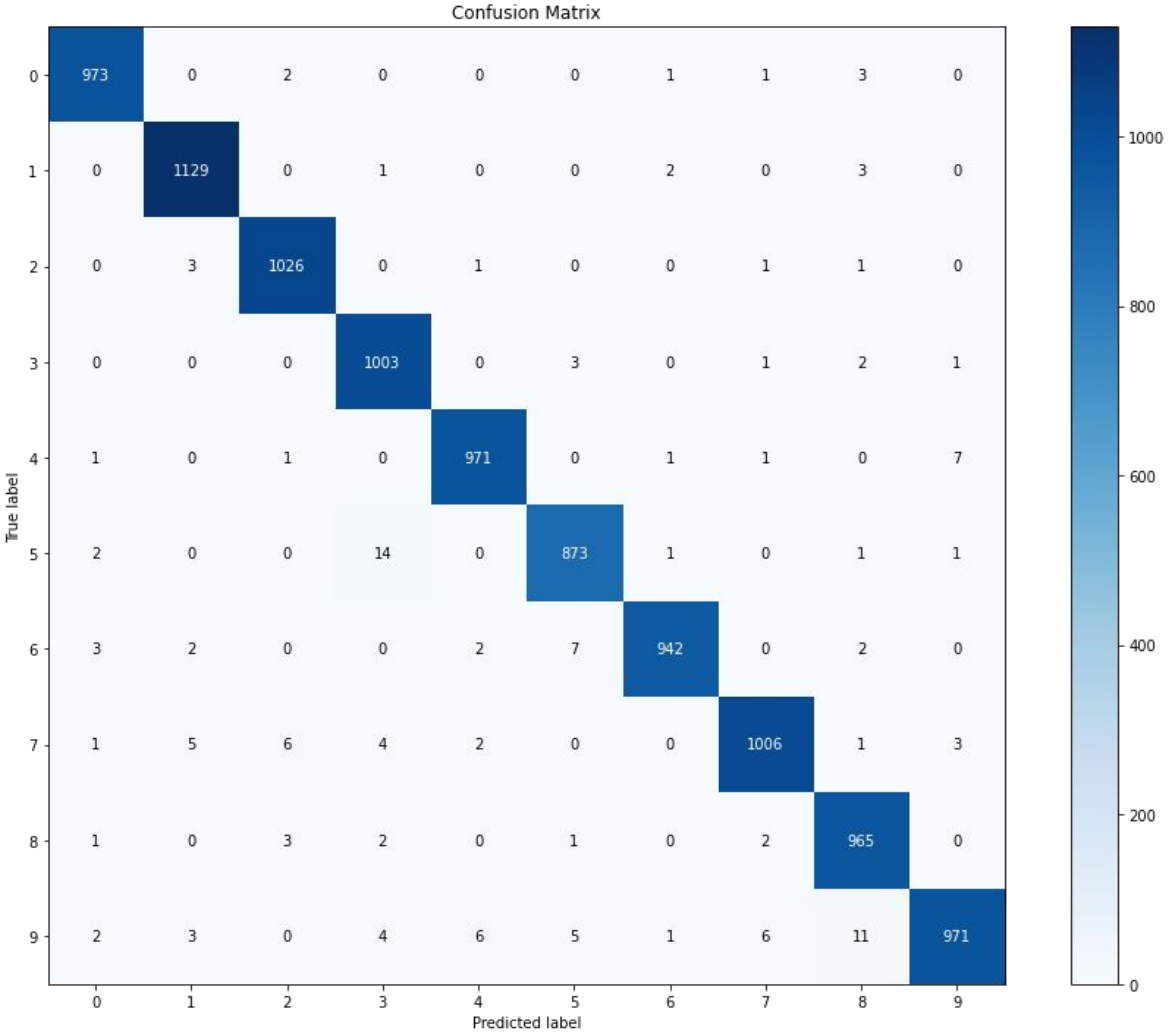

```
lenet5 = Sequential()
lenet5.add(Conv2D(6, (5,5), strides=1,  activation='tanh',input_shape=(28,28,1),padding='same')) #C1
lenet5.add(AveragePooling2D()) #S2
lenet5.add(Conv2D(16, (5,5), strides=1, activation='tanh', padding='valid')) #C3
lenet5.add(AveragePooling2D()) #S4
lenet5.add(Flatten()) #Flatten
lenet5.add(Dense(120, activation='tanh')) #C5
lenet5.add(Dense(84, activation='tanh')) #F6
lenet5.add(Dense(10, activation='softmax')) #Output layer
```

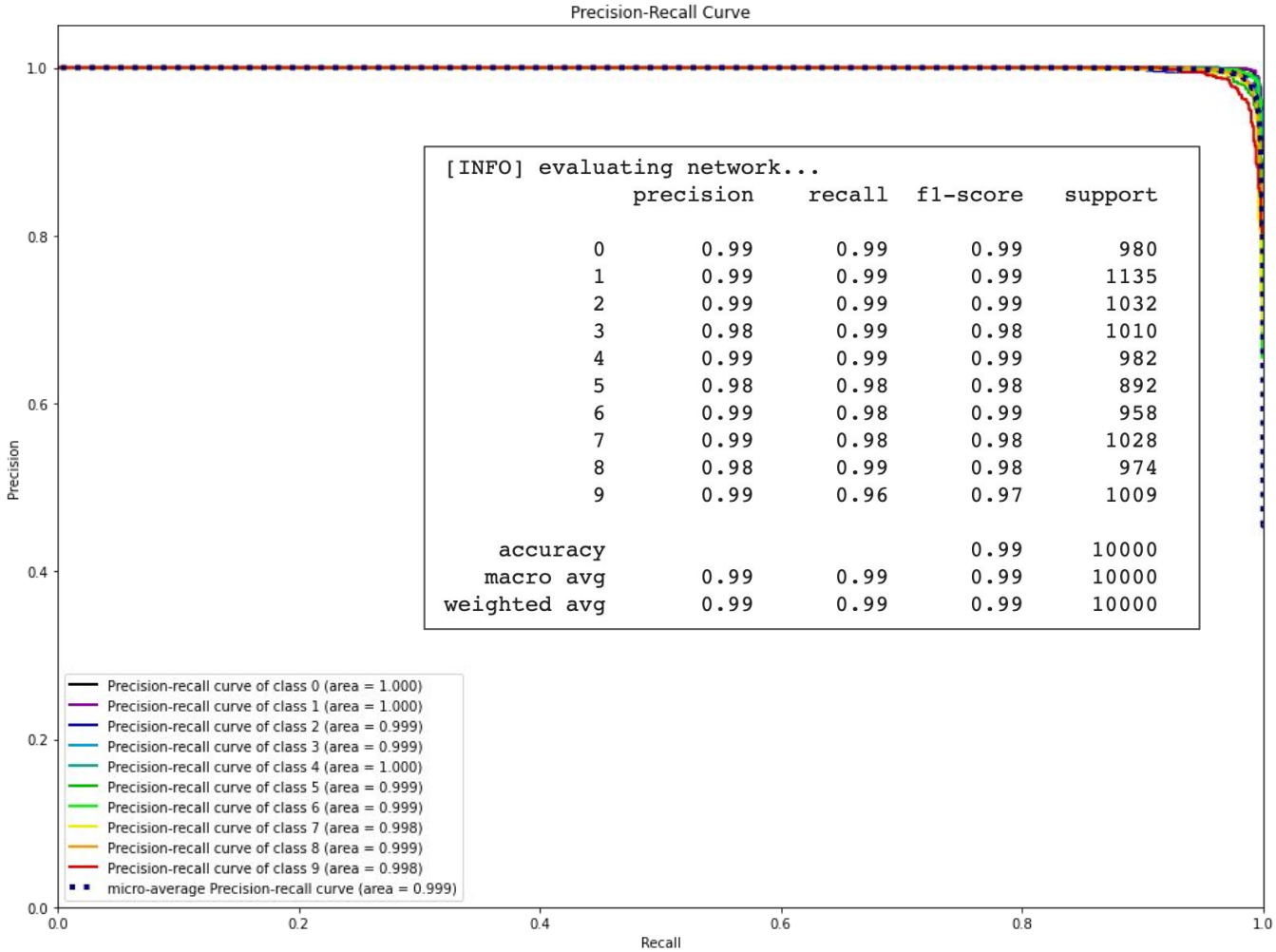
LeNet-5


```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 6)         156
_________________________________________________________________
average_pooling2d (AveragePo (None, 14, 14, 6)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 16)        2416
_________________________________________________________________
average_pooling2d_1 (Average (None, 5, 5, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 120)               48120
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```


7
2
1
0
4
1
4
9
5
9
0
6
9
0
1
5
9
7


Class Proportions in Target Variable
Number of instances
8,000
7,500
7,000
6,500
6,000
5,500
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
1
2
3
4
5
6
7
8
9
Class
Run Name
fresh-microwave-6
Dataset
train
test


Confusion Matrix
True label
Predicted label
0 973 0 2 0 0 0 1 1 3 0
1 0 1129 0 1 0 0 2 0 3 0
2 0 3 1026 0 1 0 0 1 1 0
3 0 0 0 1003 0 3 0 1 2 1
4 1 0 1 0 971 0 1 1 0 7
5 2 0 0 14 0 873 1 0 1 1
6 3 2 0 0 2 7 942 0 2 0
7 1 5 6 4 2 0 0 1006 1 3
8 1 0 3 2 0 1 0 2 965 0
9 2 3 0 4 6 5 1 6 11 971
0 1 2 3 4 5 6 7 8 9
1000
800
600
400
200
0

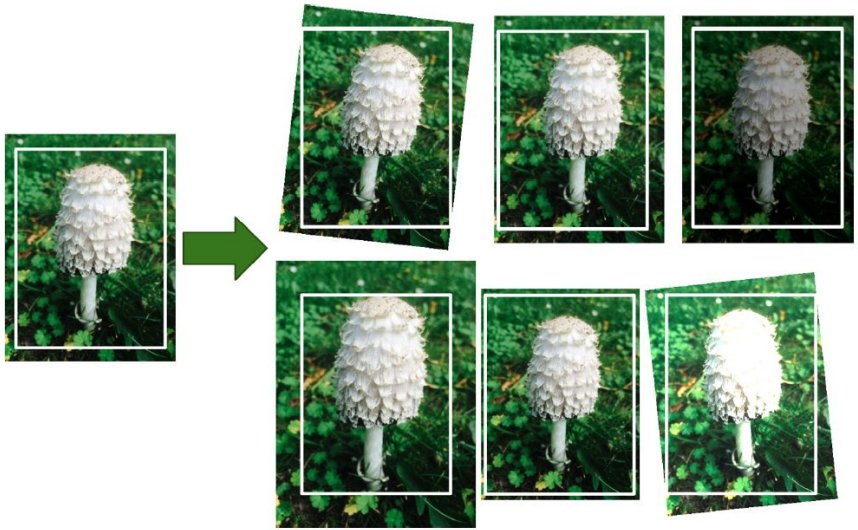


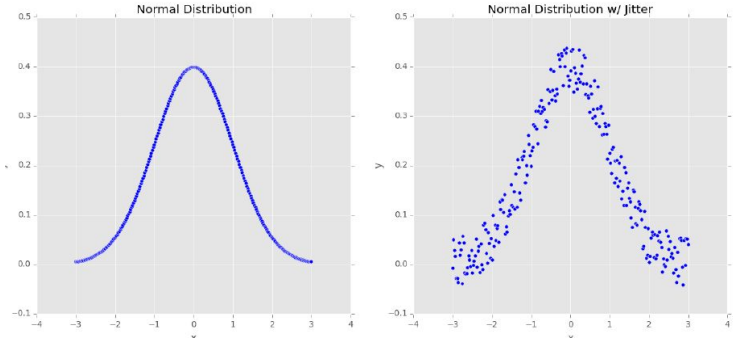
[INFO] evaluating network...

	precision	recall	f1-score	support
0	0.99	0.99	0.99	980
1	0.99	0.99	0.99	1135
2	0.99	0.99	0.99	1032
3	0.98	0.99	0.98	1010
4	0.99	0.99	0.99	982
5	0.98	0.98	0.98	892
6	0.99	0.98	0.99	958
7	0.99	0.98	0.98	1028
8	0.98	0.99	0.98	974
9	0.99	0.96	0.97	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

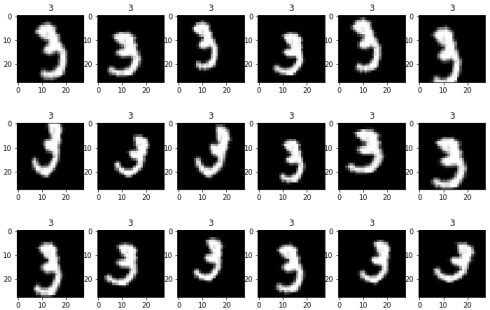
Data Augmentation

- When applying data augmentation is to increase the generalizability of the model
- In most cases, you'll see an increase in testing accuracy, perhaps at the expense at a slight dip in training accuracy





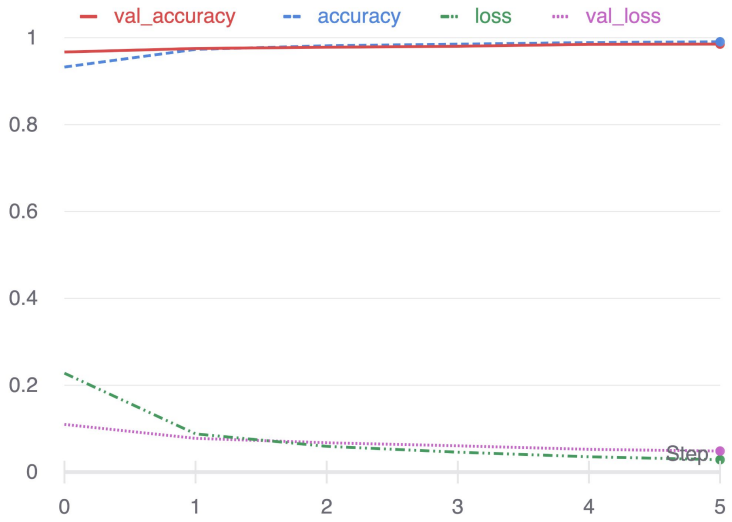
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# construct the image generator for data augmentation then
# initialize the total number of images generated thus far
aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
                         height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
                         horizontal_flip=False, fill_mode="nearest")
total = 0
image = train_x[10:11,:,:,:]

# construct the actual Python generator
print("[INFO] generating images...")
imageGen = aug.flow(image, batch_size=1)

# loop over examples from our image data augmentation generator
for img in imageGen:

  show_image(img, train_y[10], total)

  # increment our counter
  total += 1

  # if we have reached 10 examples, break from the loop
  if total == 18:
    break
```



What change in training step when using data augmentation?

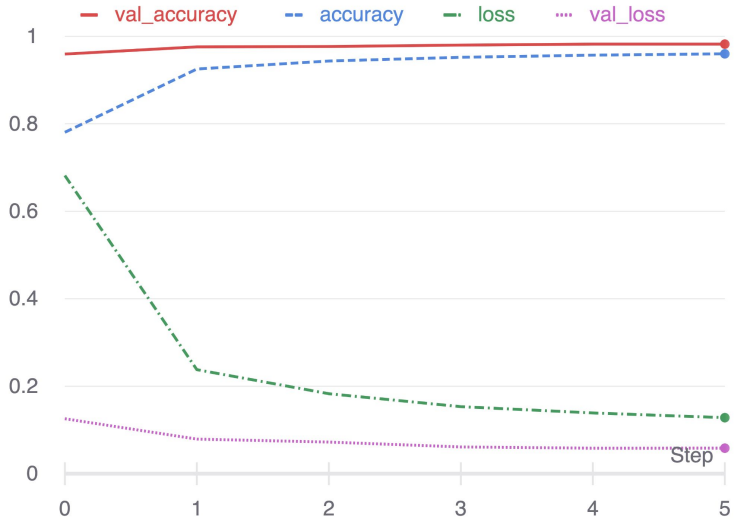
```
# construct the image generator for data augmentation then
# initialize the total number of images generated thus far
aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
                         height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
                         horizontal_flip=False, fill_mode="nearest")

print("[INFO] training network...")
history = lenet5.fit(aug.flow(train_x, train_y, batch_size=32),
                     validation_data=(test_x, test_y),
                     epochs=5, verbose=1,
                     callbacks=[MyCustomCallback()])
```

Without Data Augmentation




```
[INFO] evaluating network...
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.99      0.99      0.99      1032
           3       0.98      0.99      0.98      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.98      0.98       892
           6       0.99      0.98      0.99       958
           7       0.99      0.98      0.98      1028
           8       0.98      0.99      0.98       974
           9       0.99      0.96      0.97      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

Data Augmentation




```
[INFO] evaluating network...
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       0.99      1.00      0.99      1135
           2       0.98      0.98      0.98      1032
           3       0.97      0.99      0.98      1010
           4       0.97      0.99      0.98       982
           5       0.97      0.99      0.98       892
           6       0.99      0.98      0.99       958
           7       0.98      0.98      0.98      1028
           8       0.98      0.96      0.97       974
           9       0.99      0.95      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

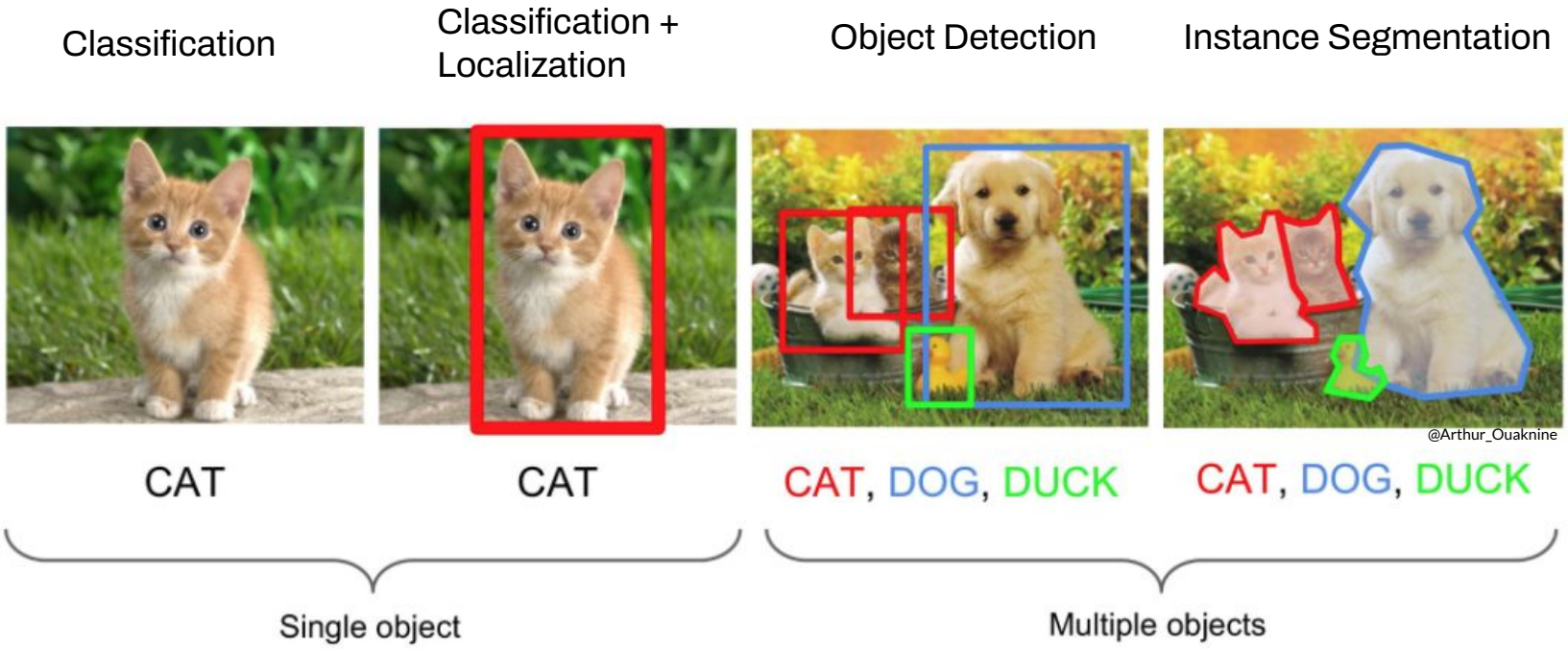
14,197,122 images, 21841 synsets indexed, 1M images with bounding box annotations

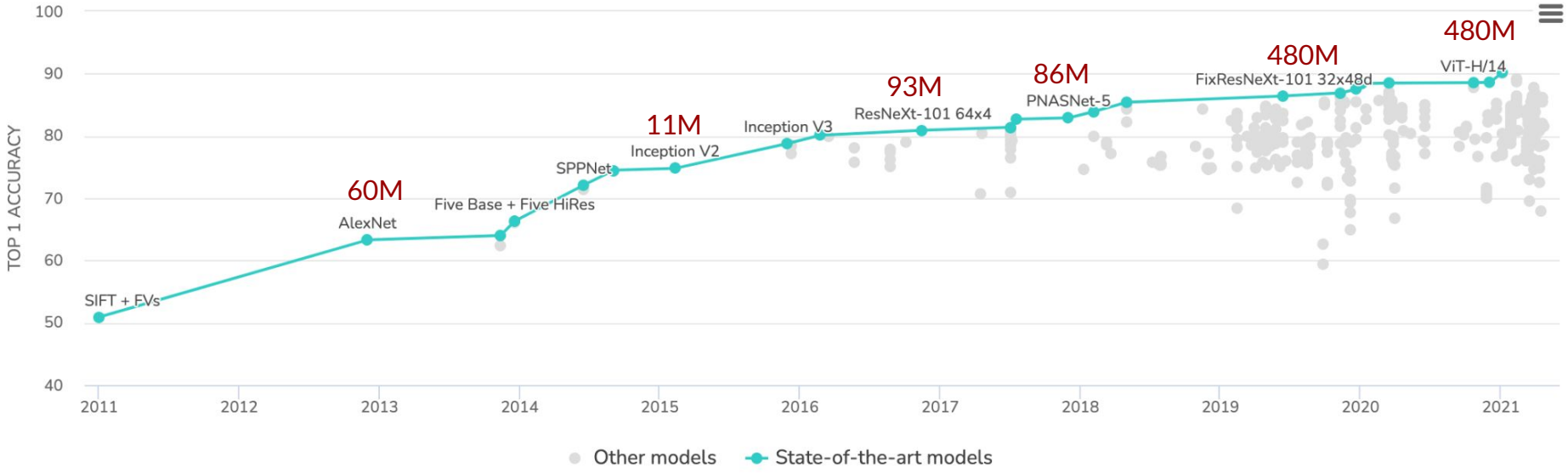
http://www.image-net.org

1,000 image categories

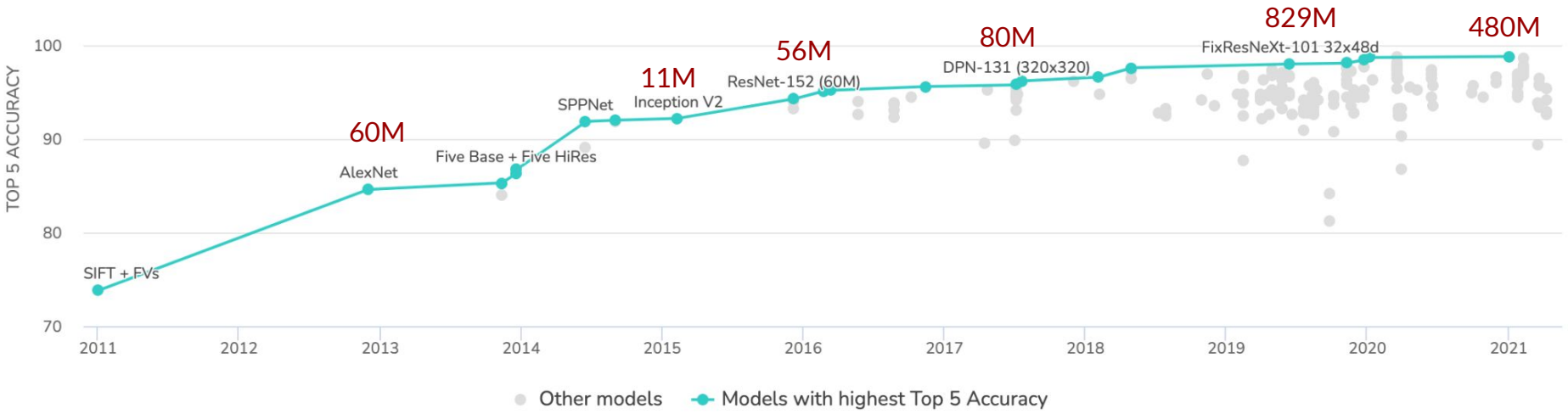
Models are trained on ≈ 1.2M training images
50k images for validations (50 images per synset)
100k images for testing (100 images per synset)
rank-1 and rank-5 accuracies

The general tasks of ILSVRC for most years





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



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

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

1 Introduction

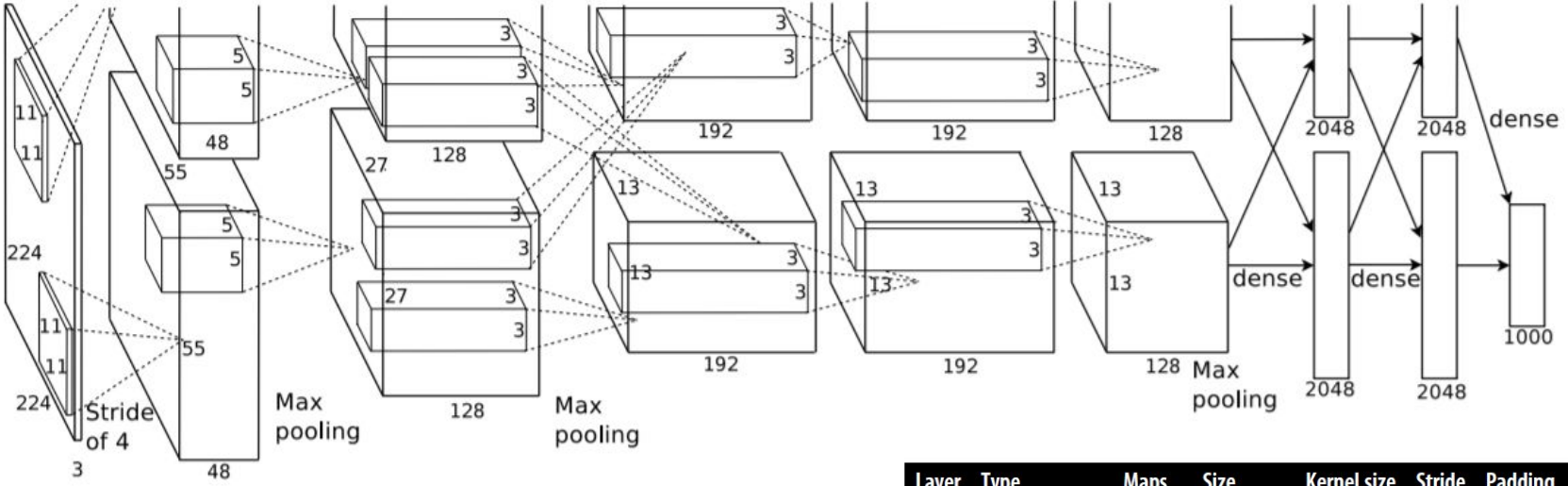
Current approaches to object recognition make essential use of machine learning methods. To improve their performance, we can collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting. Until recently, datasets of labeled images were relatively small — on the order of tens of thousands of images (e.g., NORB [16], Caltech-101/256 [8, 9], and CIFAR-10/100 [12]). Simple recognition tasks can be solved quite well with datasets of this size, especially if they are augmented with label-preserving transformations. For example, the current-best error rate on the MNIST digit-recognition task (<0.3%) approaches human performance [4]. But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets. And indeed, the shortcomings of small image datasets have been widely recognized (e.g., Pinto et al. [21]), but it has only recently become possible to collect labeled datasets with millions of images. The new larger datasets include LabelMe [23], which consists of hundreds of thousands of fully-segmented images, and ImageNet [6], which consists of over 15 million labeled high-resolution images in over 22,000 categories.

To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don't have. Convolutional neural networks (CNNs) constitute one such class of models [16, 11, 13, 18, 15, 22, 26]. Their capacity can be con-

The ILSVRC was a competition designed to spur innovation in the field of computer vision. Before the development of AlexNet, the task was thought very difficult and far beyond the capability of modern computer vision methods.

Innovations

1. Develop deep and effective end-to-end models
2. Use of ReLU
3. Average pooling was replaced with a max-pooling (non-overlapping)
4. The newly proposed dropout method was used to address overfitting between the fully connected layers of the classifier
5. Data augmentation
6. Split the model into two pipelines to train on multiple GPUs.



Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	`valid`	–
C7	Convolution	256	13×13	3×3	1	`same`	ReLU
C6	Convolution	384	13×13	3×3	1	`same`	ReLU
C5	Convolution	384	13×13	3×3	1	`same`	ReLU
S4	Max pooling	256	13×13	3×3	2	`valid`	–
C3	Convolution	256	27×27	5×5	1	`same`	ReLU
S2	Max pooling	96	27×27	3×3	2	`valid`	–
C1	Convolution	96	55×55	11×11	4	`valid`	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

```
INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU] * 3 => POOL =>
     [FC => RELU => DO] * 2 => SOFTMAX
```

```
# create a model
model = Sequential()

# Block #1: first CONV => RELU => POOL layer set
model.add(Conv2D(96, (11, 11), strides=(4, 4),
                 input_shape=(227,227,3), padding="valid",
                 kernel_regularizer=l2(0.0002),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(Dropout(0.25))

# Block #2: second CONV => RELU => POOL layer set
model.add(Conv2D(256, (5, 5), padding="same",
                 kernel_regularizer=l2(0.0002),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(Dropout(0.25))
```

AlexNet
Blocks #01 and #02

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$227 \times 227 \times 3$	
CONV	$55 \times 55 \times 96$	$11 \times 11/4 \times 4, K = 96$
ACT	$55 \times 55 \times 96$	
BN	$55 \times 55 \times 96$	
POOL	$27 \times 27 \times 96$	$3 \times 3/2 \times 2$
DROPOUT	$27 \times 27 \times 96$	
CONV	$27 \times 27 \times 256$	$5 \times 5, K = 256$
ACT	$27 \times 27 \times 256$	
BN	$27 \times 27 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$13 \times 13 \times 256$	

```
# Block #3: CONV => RELU => CONV => RELU => CONV => RELU
model.add(Conv2D(384, (3, 3), padding="same",
                 kernel_regularizer=l2(0.0002),activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(384, (3, 3), padding="same",
                 kernel_regularizer=l2(0.002),activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(256, (3, 3), padding="same",
                 kernel_regularizer=l2(0.002),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(Dropout(0.25))
```

AlexNet Block #03

CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 256$	$3 \times 3, K = 256$
ACT	$13 \times 13 \times 256$	
BN	$13 \times 13 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$6 \times 6 \times 256$	

```
# Block #4: first set of FC => RELU layers
model.add(Flatten())
model.add(Dense(4096,kernel_regularizer=l2(0.0002),activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

# Block #5: second set of FC => RELU layers
model.add(Dense(4096, kernel_regularizer=l2(0.0002),activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

# softmax classifier
model.add(Dense(1000, kernel_regularizer=l2(0.0002)))
model.add(Activation("softmax"))
```

AlexNet Blocks #04, #05

FC	4096
ACT	4096
BN	4096
DROPOUT	4096
FC	4096
ACT	4096
BN	4096
DROPOUT	4096
FC	1000
SOFTMAX	1000

AlexNet

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 55, 55, 96)	34944
batch_normalization (BatchNo	(None, 55, 55, 96)	384
max_pooling2d (MaxPooling2D)	(None, 27, 27, 96)	0
dropout (Dropout)	(None, 27, 27, 96)	0
conv2d_1 (Conv2D)	(None, 27, 27, 256)	614656
batch_normalization_1 (Batch	(None, 27, 27, 256)	1024
max_pooling2d_1 (MaxPooling2	(None, 13, 13, 256)	0
dropout_1 (Dropout)	(None, 13, 13, 256)	0
conv2d_2 (Conv2D)	(None, 13, 13, 384)	885120
batch_normalization_2 (Batch	(None, 13, 13, 384)	1536
conv2d_3 (Conv2D)	(None, 13, 13, 384)	1327488
batch_normalization_3 (Batch	(None, 13, 13, 384)	1536
conv2d_4 (Conv2D)	(None, 13, 13, 256)	884992
batch_normalization_4 (Batch	(None, 13, 13, 256)	1024
max_pooling2d_2 (MaxPooling2	(None, 6, 6, 256)	0
dropout_2 (Dropout)	(None, 6, 6, 256)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 4096)	37752832
batch_normalization_5 (Batch	(None, 4096)	16384
dropout_3 (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16781312
batch_normalization_6 (Batch	(None, 4096)	16384
dropout_4 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 1000)	4097000
activation (Activation)	(None, 1000)	0

Total params: 62,416,616
Trainable params: 62,397,480
Non-trainable params: 19,136

https://www.kaggle.com/c/dogs-vs-cats/overview

Dogs vs. Cats

Create an algorithm to distinguish dogs from cats

Kaggle · 213 teams · 7 years ago

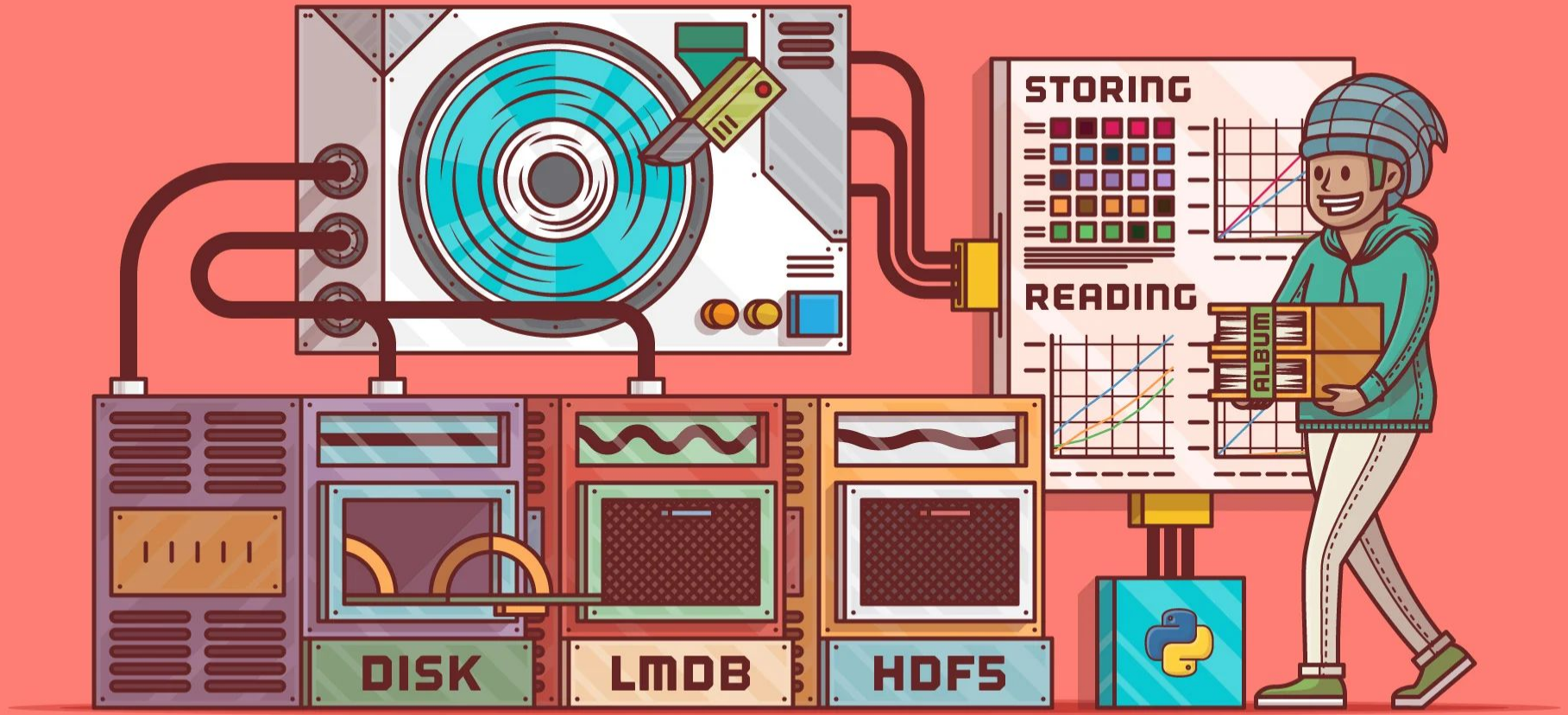
Overview | Data | Notebooks | Discussion | Leaderboard | Rules | Team

Overview

The training archive contains 25,000 images (different sizes) of dogs and cats (.zip 543MB)

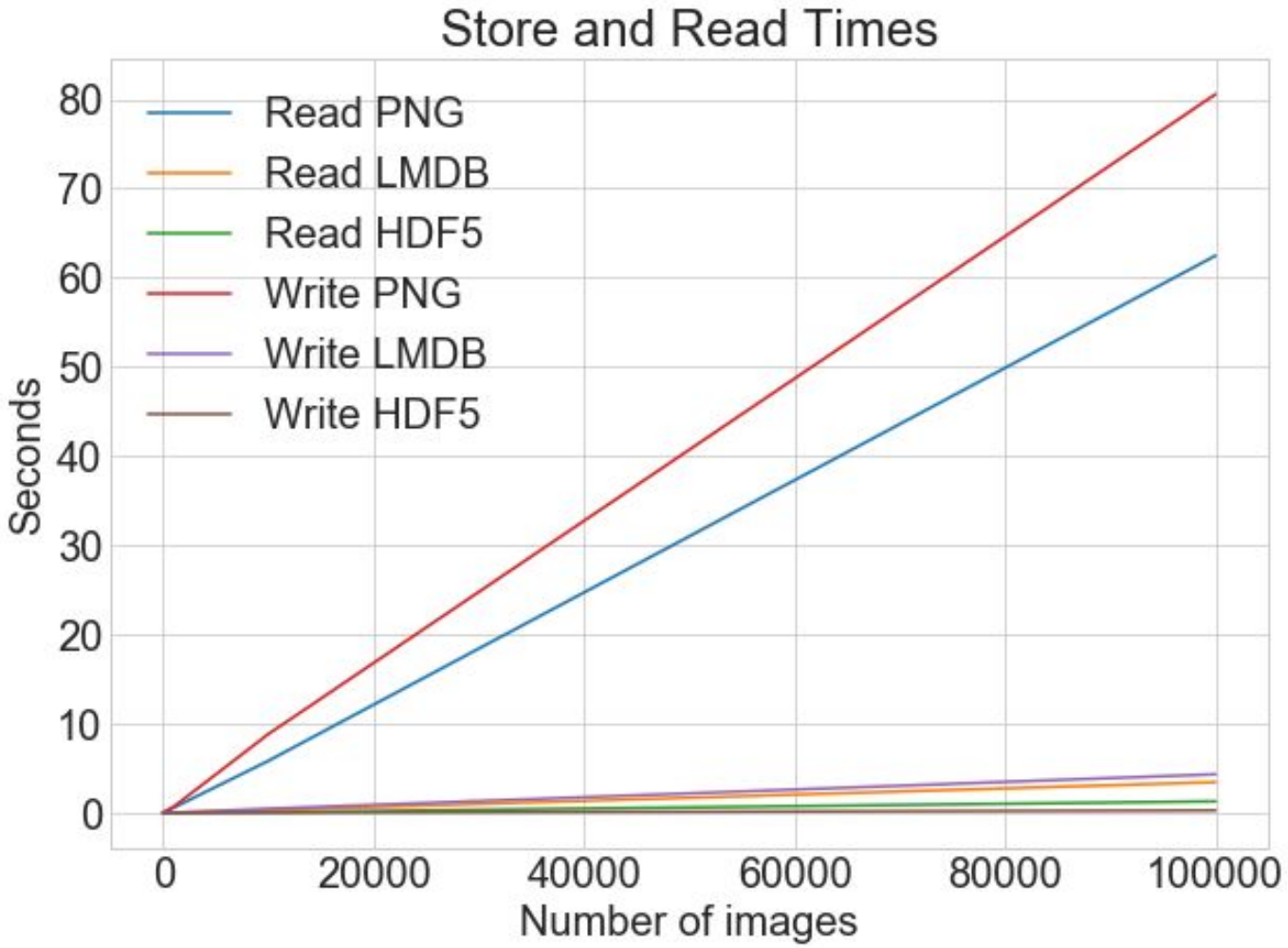
- Description
- Prizes
- Evaluation
- Winners

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.

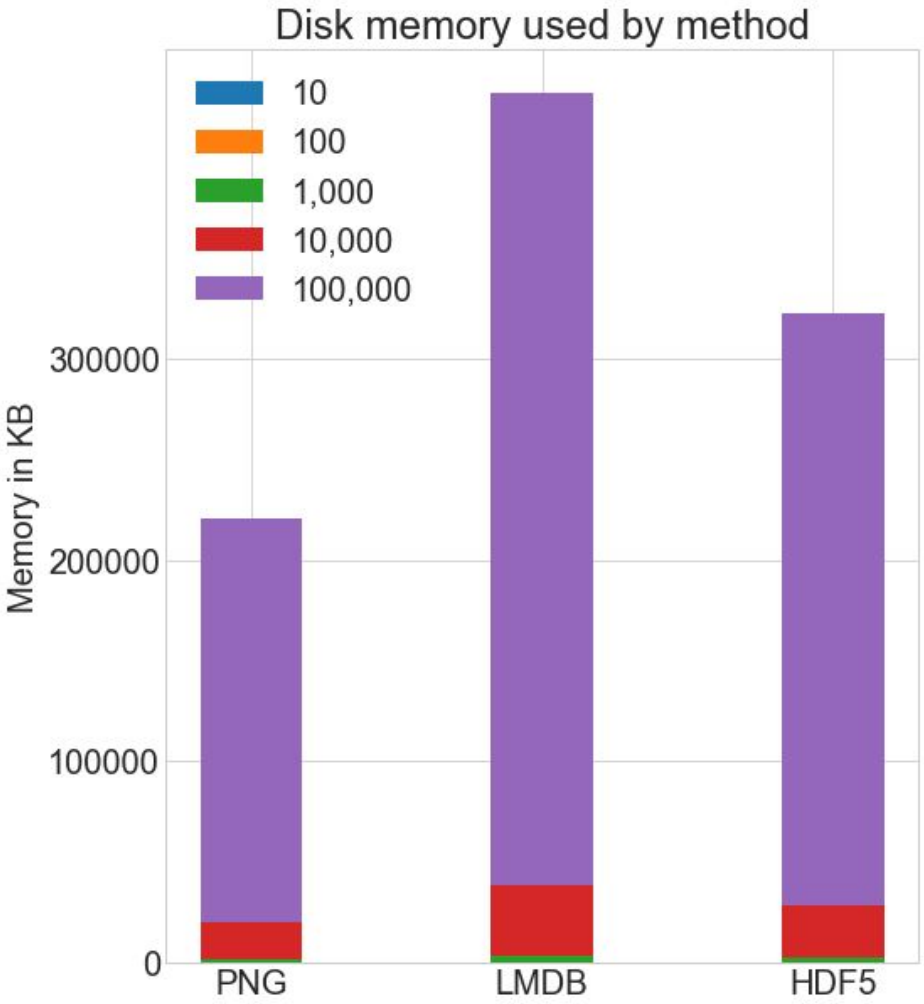


Real Python

https://realpython.com/storing-images-in-python



https://realpython.com/storing-images-in-python



Both **HDF5** and **LMDB** take up more disk space than if you store using normal **.png** images. It's important to note that both LMDB and HDF5 disk usage and **performance depend highly on various factors**, including operating system and, more critically, the size of the data you store.

- Parallel access
- Documentation
- Integration for other libraries
- Support to slicing
- Data compression

https://realpython.com/storing-images-in-python

dog.9909.jpg
cat.9653.jpg
dog.2499.jpg
dog.6350.jpg
cat.997.jpg
dog.3829.jpg

HDF5DatasetWriter (gzip)

train.hdf5
val.hdf5
test.hdf5

dogs_cats.hdf5

label_names
0: cat
1: dog
...
24999: dog

labels
0: 0
1: 1
...
24999: 1

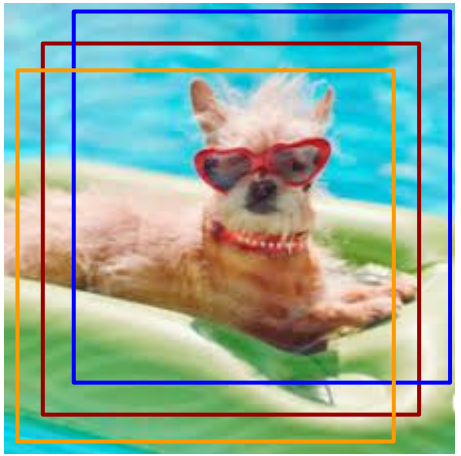
features
0: 0.91, 0.73, 0.99, ..., 0.10
1: 0.12, 0.22, 0.98, ..., 0.20
...
24999: 0.13, 0.34, 0.97, ..., 0.23

During the transformation of dataset to hdf5 format

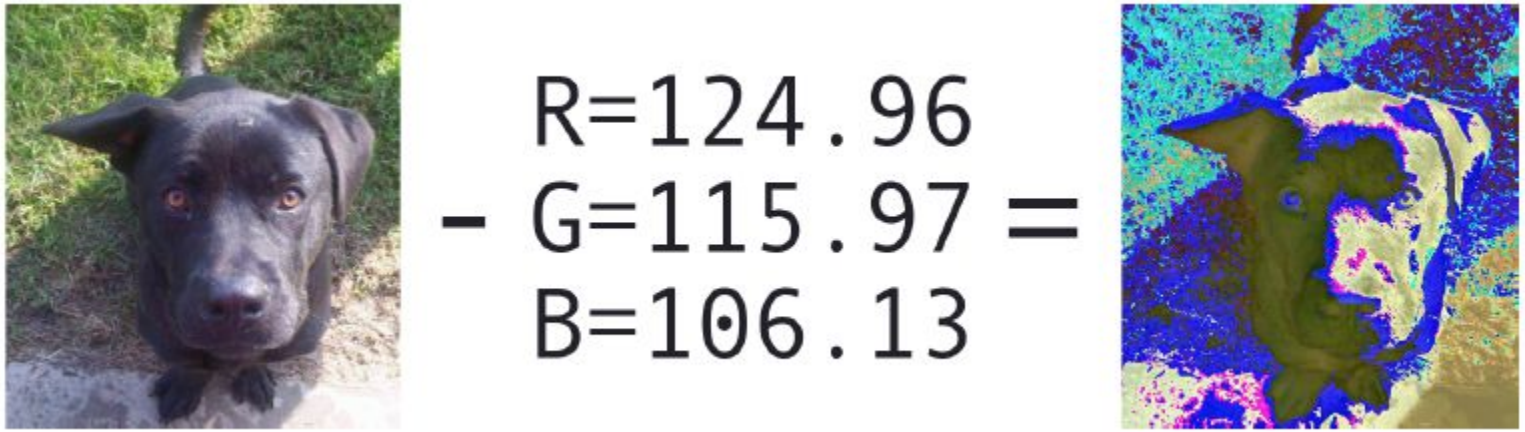
```
AspectAwarePreprocessor(256, 256) >> hdf5
```

Pre-processing before the training step

```
PatchPreprocessor(227, 227)
```



```
MeanPreprocessor(means["R"], means["G"], means["B"])
```



A mean subtraction pre-processor designed to subtract the mean Red, Green, and Blue pixel intensities across a dataset from an input image (which is a form of **data normalization)** <designed to increase classification accuracy>.

Before we can implement the AlexNet architecture and train it on the Kaggle Dogs vs. Cats dataset, we first need to define a class responsible for yielding **batches of images and labels** from our **HDF5** dataset.

Part #01

```
class HDF5DatasetGenerator:
 def __init__(self, dbPath, batchSize, preprocessors=None, aug=None,  binarize=True,classes=2):
   # store the batch size, preprocessors, and data augmentor,
   # whether or not the labels should be binarized, along with
   # the total number of classes
   self.batchSize = batchSize
   self.preprocessors = preprocessors
   self.aug = aug
   self.binarize = binarize
   self.classes = classes

   # open the HDF5 database for reading and determine the total
   # number of entries in the database
   self.db = h5py.File(dbPath, "r")
   self.numImages = self.db["labels"].shape[0]
```

Part #02

```
def generator(self, passes=np.inf):
   # initialize the epoch count
   epochs = 0

   # keep looping infinitely -- the model will stop once we have
   # reach the desired number of epochs
   while epochs < passes:
     # loop over the HDF5 dataset
     for i in np.arange(0, self.numImages, self.batchSize):
       # extract the images and labels from the HDF dataset
       images = self.db["images"][i: i + self.batchSize]
       labels = self.db["labels"][i: i + self.batchSize]
```

```
       # check to see if the labels should be binarized
       if self.binarize:
         labels = to_categorical(labels,self.classes)
```

Part #03

```
# check to see if our preprocessors are not None
if self.preprocessors is not None:
  # initialize the list of processed images
  procImages = []

  # loop over the images
  for image in images:
    # loop over the preprocessors and apply each
    # to the image
    for p in self.preprocessors:
      image = p.preprocess(image)

    # update the list of processed images
    procImages.append(image)

  # update the images array to be the processed
  # images
  images = np.array(procImages)
```

Part #04

```
      # if the data augmentation exists, apply it
      if self.aug is not None:
        (images, labels) = next(self.aug.flow(images,labels,batch_size=self.batchSize))
```

Finally, we can yield a 2-tuple of the batch of images and labels to the calling Keras generator:

```
     # yield a tuple of images and labels
      yield (images, labels)

    # increment the total number of epochs
    epochs += 1

def close(self):
  # close the database
  self.db.close()
```

epochs:045-val_acc:0.921.hdf5

https://abre.ai/alexnet-dogs-cats