



Project-Traffic Sign Classifier

Intro to Traffic Sign Classifier

- Next, you're going to build a deep neural network to classify traffic signs.
- Hands-on practical experience with deep neural network
- Your network should be able to take an image and decide if it's a stop sign, a yield sign, a different type of sign or maybe no sign at all.



yield sign– This sign indicates that you need to slow down, but you may not have to come to a full stop

<https://www.youtube.com/watch?v=IX1lym7Kbhl>

LeNet-5 (1998)

LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel greyscale input images. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources.

CNN Important papers:

1998: [LeNet](#)

2012: [AlexNet](#)

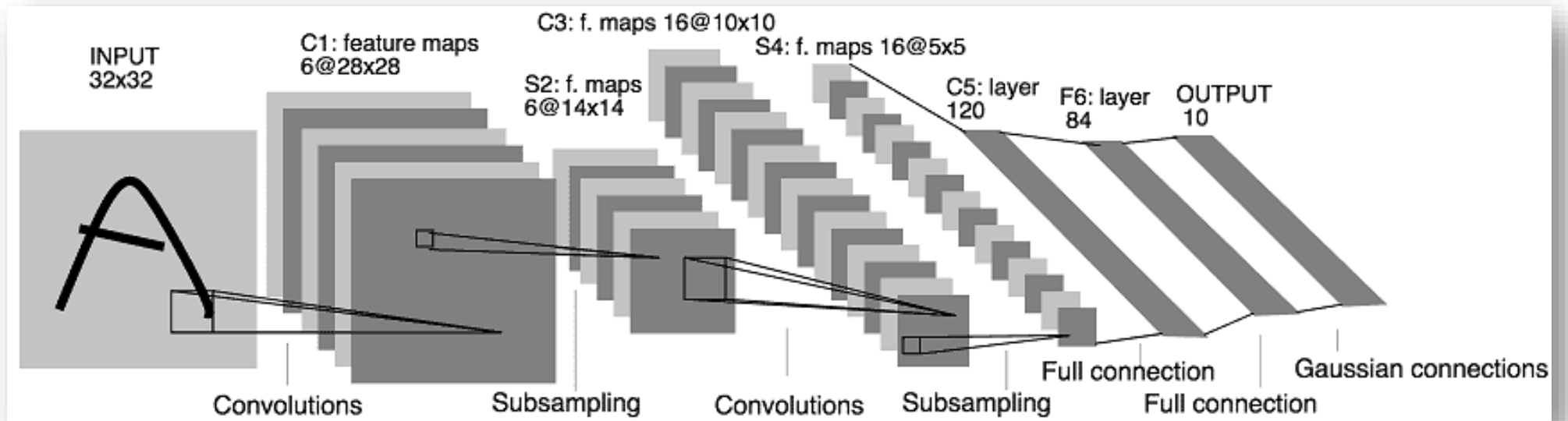
2013: [ZFNet](#)

2014: [VGG](#), [GoogLeNet \(Inception v1\)](#)

2015: [ResNet](#), [Inception v2](#), [Inception v3](#)

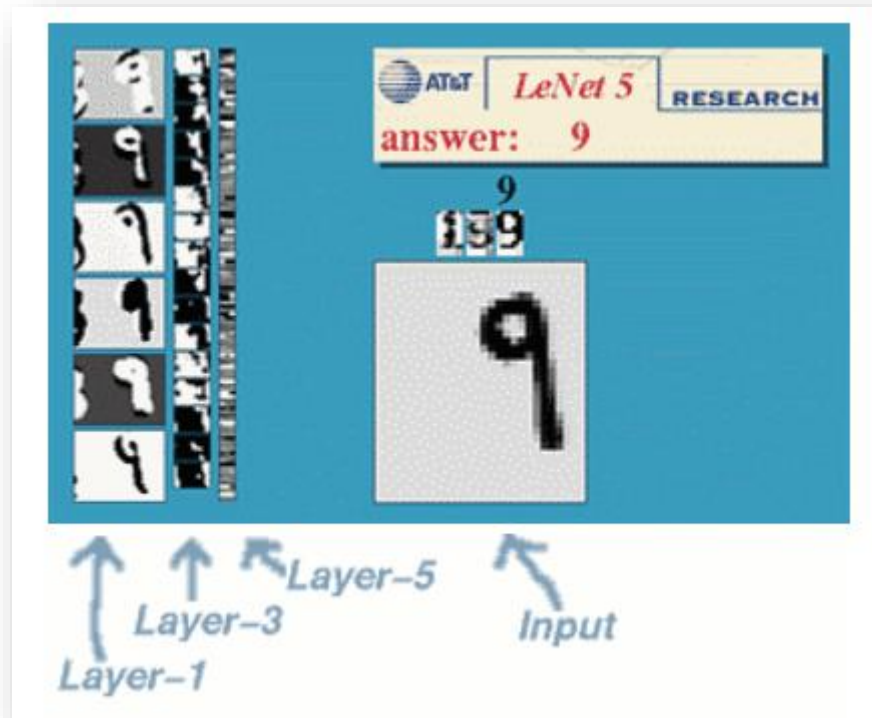
2016: [Inception v4](#), [SqueezeNet](#), [DarkNet](#)

2017: [MobileNet](#)



LeNet-5 ,convolutional neural networks

Convolutional Neural Networks are a special kind of multi-layer neural networks. Like almost every other neural networks they are trained with a version of the back-propagation algorithm.



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

[LeCun et al., 1998]

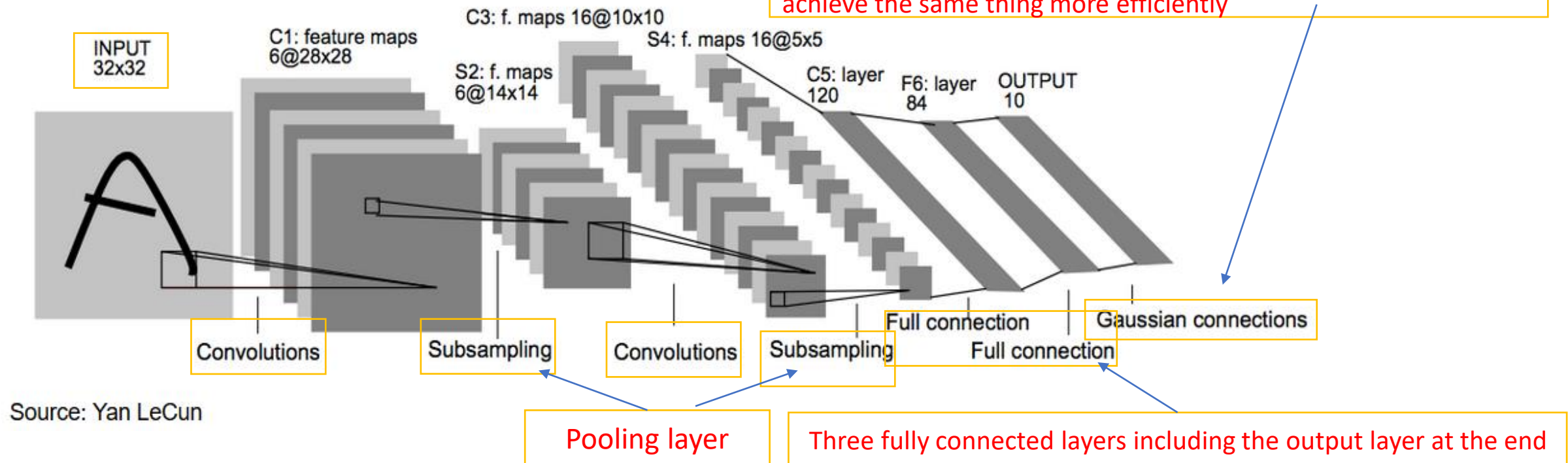
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.

Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998.

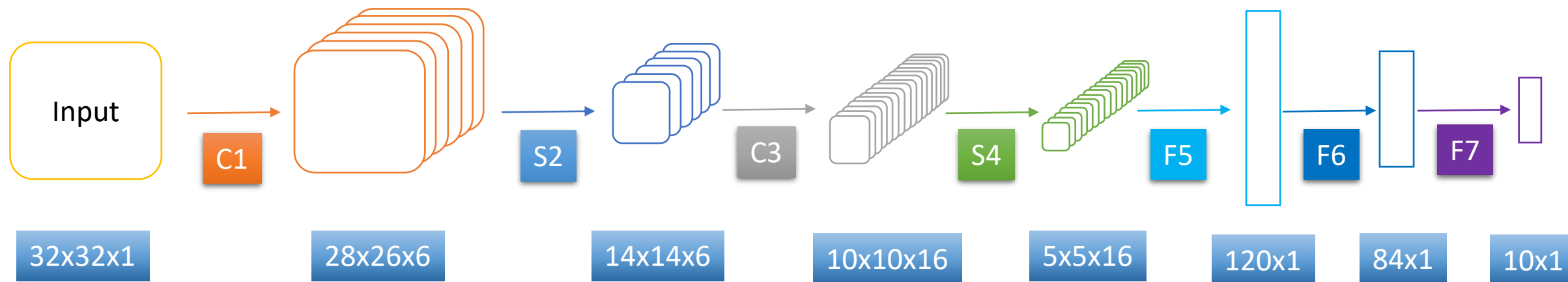
LeNet Architecture

Before we adapt LeNet for the Traffic Sign Classifier project, let's walk through the lab solution and review what happens on each line.

LeNet Lab Solution



Source Code: <https://github.com/udacity/CarND-LeNet-Lab/blob/master/LeNet-Lab-Solution.ipynb>



Layer	# filters / neurons	Filter size	Stride	Size of feature map	Activation function
Input	-	-	-	$32 \times 32 \times 1$	
Conv 1	6	5×5	1	$28 \times 28 \times 6$	tanh
Avg. pooling 1		2×2	2	$14 \times 14 \times 6$	
Conv 2	16	5×5	1	$10 \times 10 \times 16$	tanh
Avg. pooling 2		2×2	2	$5 \times 5 \times 16$	
Conv 3	120	5×5	1	120	tanh
Fully Connected 1	-	-	-	84	tanh
Fully Connected 2	-	-	-	10	Softmax

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

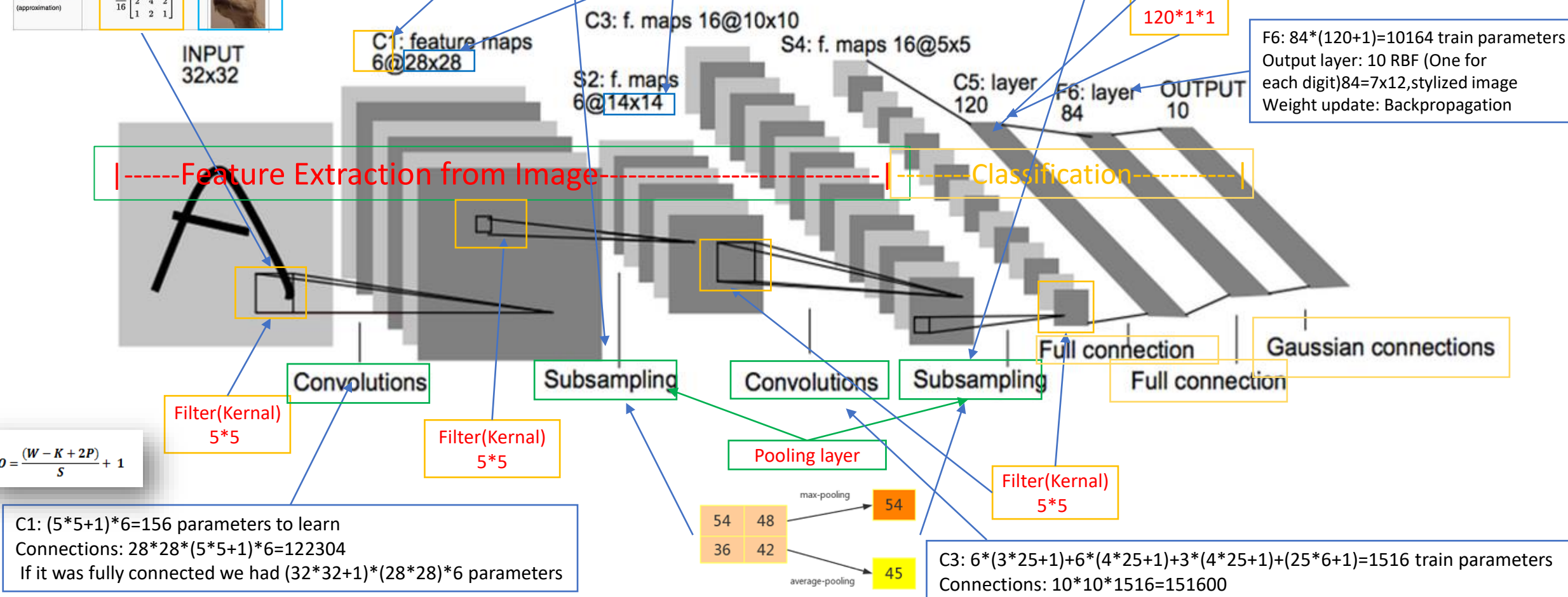
S2: Subsampling layer with 6 feature maps of size 14x14 2x2 non overlapping receptive fields in C1 Layer.
 $S2: (1+1)*6=12$ trainable parameters
 \Rightarrow trainable parameters = (coefficient + bias) * filter
 Connections: $14*14*(2*2+1)*6=5880$

http://www.cs.cmu.edu/~aarti/Class/10701_Spring14/slides/DeepLearning.pdf
<https://engmrk.com/lenet-5-a-classic-cnn-architecture/>

S4: Subsampling layer with 16 feature maps of size 5x5, Each unit in S4 is connected to the corresponding 2x2 receptive field at C3
 $S4: 16*2=32$ train parameters
 Connections: $5*5*(2*2+1)*16=2000$

F5: $120*(16*25+1)=48120$ train parameters
 Connections: Fully connected

F6: $84*(120+1)=10164$ train parameters
 Output layer: 10 RBF (One for each digit) $84=7*12$, stylized image
 Weight update: Backpropagation



C3 layer

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE 1

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED
BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

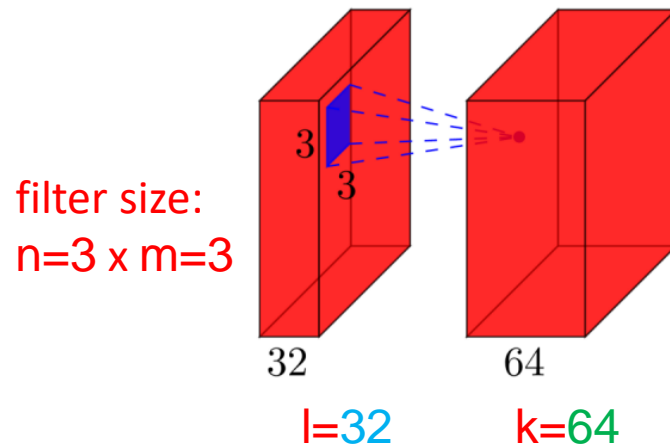
Snapshot of TABLE 1 from [LeCun et al., 1998]

How to calculate the number of parameters for convolutional neural network?

- **Pooling layers:** The pooling layers e.g. do the following: "replace a 2x2 neighborhood by its maximum value". So there is **no parameter you could learn** in a pooling layer.
- **Fully-connected layers:** In a fully-connected layer, **all input units have a separate weight to each output unit**. For n inputs and m outputs, the number of weights is $n*m$. Additionally, you have a bias for each output node, so you are at $(n+1)*m$ parameters.
- **Output layer:** The output layer is a normal fully-connected layer, so $(n+1)*m$ parameters, where n is the number of inputs and m is the number of outputs.

Input layer: All the input layer does is read the input image, so there are no parameters you could learn here.

Convolutional layers: Consider a convolutional layer which takes l feature maps at the **input**, and has k feature maps as **output**. The filter size is $n \times m$. For example, this will look like this:



Here, the input has $l=32$ feature maps as input, $k=64$ feature maps as output, and the filter size is $n=3 \times m=3$.

It is important to understand, that we don't simply have a 3x3 filter, but actually a 3x3x32 filter, as our input has 32 dimensions. And we learn 64 different 3x3x32 filters. Thus, the total number of weights is $n*m*k*l$. Then, there is also a bias term for each feature map, so we have a total number of parameters of $(n*m*l+1)*k$.

$$(3*3*32+1)*64=18,496$$

```

net1 = NeuralNet(
    layers=[('input', layers.InputLayer),
            ('conv2d1', layers.Conv2DLayer),
            ('maxpool1', layers.MaxPool2DLayer),
            ('conv2d2', layers.Conv2DLayer),
            ('maxpool2', layers.MaxPool2DLayer),
            ('dropout1', layers.DropoutLayer),
            ('dense', layers.DenseLayer),
            ('dropout2', layers.DropoutLayer),
            ('output', layers.DenseLayer),],

    # input layer
    input_shape=(None, 1, 28, 28),

    # layer conv2d1
    conv2d1_num_filters=32,
    conv2d1_filter_size=(5, 5),
    conv2d1_nonlinearity=lasagne.nonlinearities.rectify,
    conv2d1_W=lasagne.init.GlorotUniform(),

    # layer maxpool1
    maxpool1_pool_size=(2, 2),

    # layer conv2d2
    conv2d2_num_filters=32,
    conv2d2_filter_size=(3, 3),
    conv2d2_nonlinearity=lasagne.nonlinearities.rectify,

    # layer maxpool2
    maxpool2_pool_size=(2, 2),

    # dropout1
    dropout1_p=0.5,

    # dense
    dense_num_units=256,
    dense_nonlinearity=lasagne.nonlinearities.rectify,

    # dropout2
    dropout2_p=0.5,

    # output
    output_nonlinearity=lasagne.nonlinearities.softmax,
    output_num_units=10,

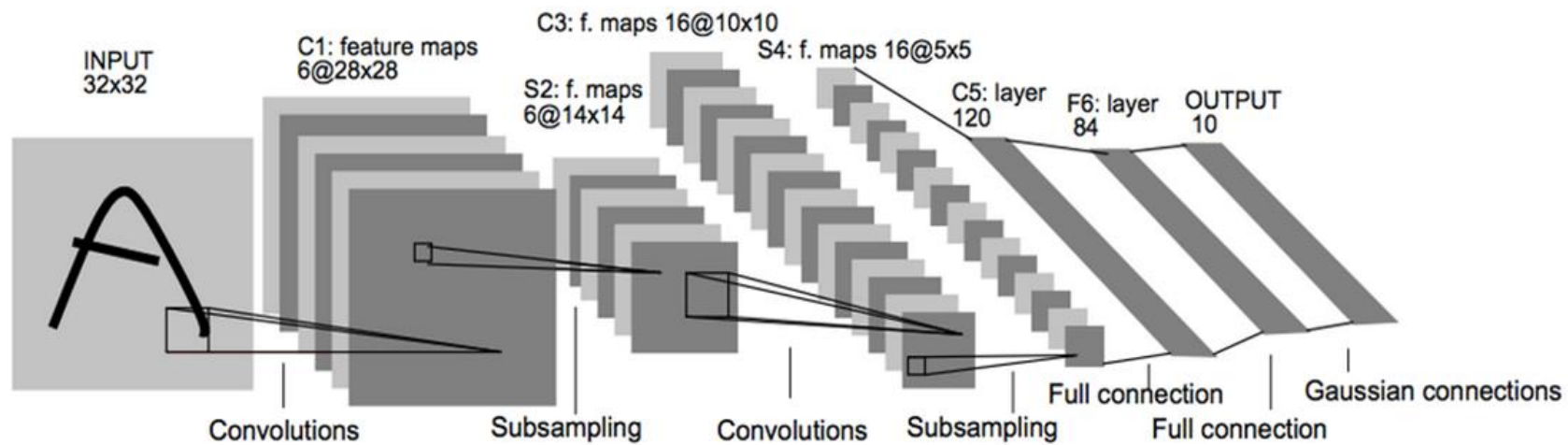
    # optimization method params
    update=nesterov_momentum,
    update_learning_rate=0.01,
    update_momentum=0.9,max_epochs=10,verbose=1,)

# Train the network
nn = net1.fit(X_train, y_train)

```

#	name	size	parameters
0	input	1x28x28	0
1	conv2d1	$(28 - (5 - 1)) = 24 \rightarrow 32 \times 24 \times 24$	$(5 * 5 * 1 + 1) * 32 = 832$
2	maxpool1	32x12x12	0
3	conv2d2	$(12 - (3 - 1)) = 10 \rightarrow 32 \times 10 \times 10$	$(3 * 3 * 32 + 1) * 32 = 9'248$
4	maxpool2	32x5x5	0
5	dense	256	$(32 * 5 * 5 + 1) * 256 = 205'056$
6	output	10	$(256 + 1) * 10 = 2'570$

So in your network, you have a total of $832 + 9'248 + 205'056 + 2'570 = 217'706$ learnable parameters, which is exactly what Lasagne reports.



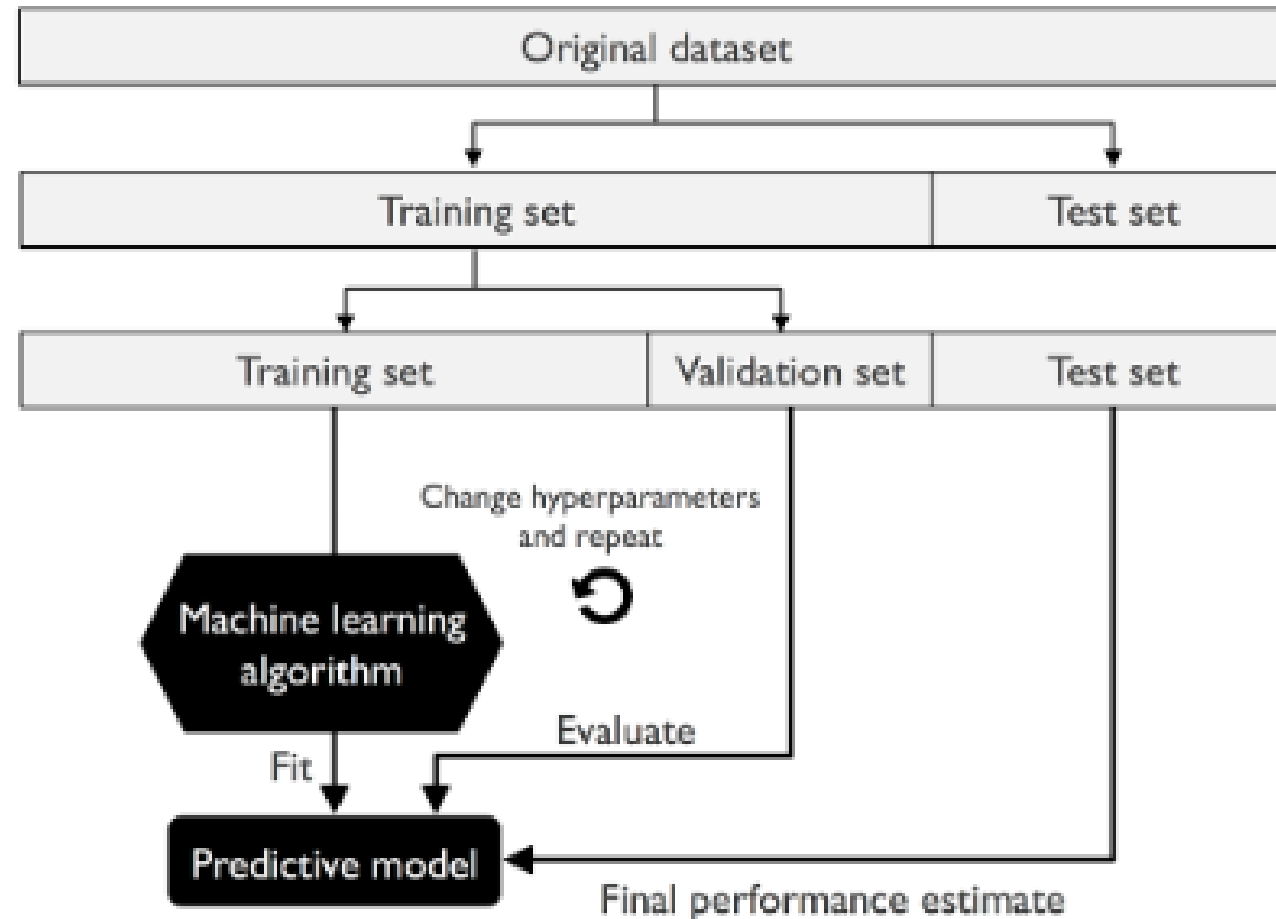
```
model = models.Sequential()
# Conv 32x32x1 => 28x28x6.
model.add(layers.Conv2D(filters = 6, kernel_size = (5, 5), strides=(1, 1), padding='valid',
activation='relu', data_format = 'channels_last', input_shape = (32, 32, 1)))
# Maxpool 28x28x6 => 14x14x6
model.add(layers.MaxPooling2D((2, 2)))
# Conv 14x14x6 => 10x10x16
model.add(layers.Conv2D(16, (5, 5), activation='relu'))
# Maxpool 10x10x16 => 5x5x16
model.add(layers.MaxPooling2D((2, 2)))
# Flatten 5x5x16 => 400
model.add(layers.Flatten())
# Fully connected 400 => 120
model.add(layers.Dense(120, activation='relu'))
# Fully connected 120 => 84
model.add(layers.Dense(84, activation='relu'))
# Dropout
model.add(layers.Dropout(0.2))
# Fully connected, output layer 84 => 43
model.add(layers.Dense(43, activation='softmax'))
```

Model: "sequential"

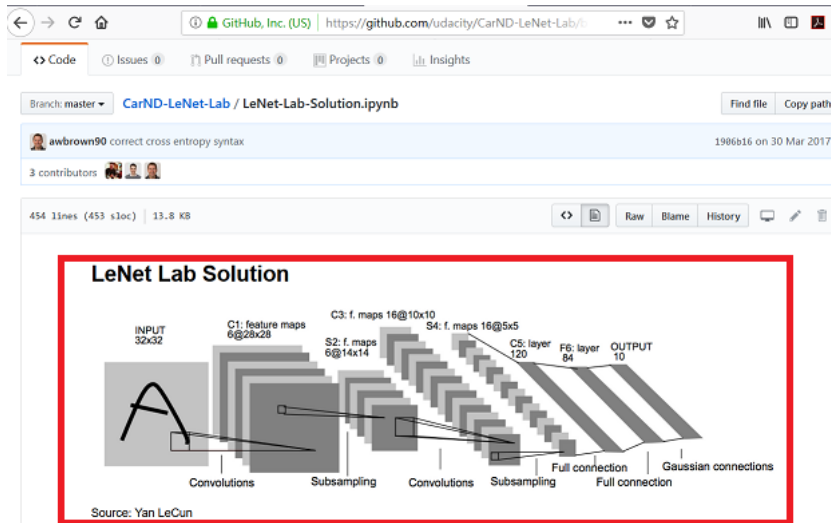
Layer (type)	Output Shape	Param #	
conv2d (Conv2D)	(None, 28, 28, 6)	156	$(5*5*1+1)*6$
max_pooling2d (MaxPooling2D)	(None, 14, 14, 6)	0	
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2416	$(5*5*6+1)*16$
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 16)	0	
flatten (Flatten)	(None, 400)	0	
dense (Dense)	(None, 120)	48120	$(400+1)*120$
dense_1 (Dense)	(None, 84)	10164	$(120+1)*84$
dropout (Dropout)	(None, 84)	0	
dense_2 (Dense)	(None, 43)	3655	$(84+1)*43$

Total params: 64,511
Trainable params: 64,511
Non-trainable params: 0

How to use training, validation, and test datasets



LeNet Data



LeNet Architecture from Yan LeCun 1998 Paper

Load Data

Load the MNIST data, which comes pre-loaded with TensorFlow.

You do not need to modify this section.

load the MNIST data set which comes pre-installed with TensorFlow

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", reshape=False)
X_train, y_train = mnist.train.images, mnist.train.labels
X_validation, y_validation = mnist.validation.images, mnist.validation.labels
X_test, y_test = mnist.test.images, mnist.test.labels
```

Then we store the training validation and test sets

```
assert(len(X_train) == len(y_train))
assert(len(X_validation) == len(y_validation))
assert(len(X_test) == len(y_test))
```

Verify the number of images in each set matches the number of labels in the same set.

```
print()
print("Image Shape: {}".format(X_train[0].shape))
print()
print("Training Set: {} samples".format(len(X_train)))
print("Validation Set: {} samples".format(len(X_validation)))
print("Test Set: {} samples".format(len(X_test)))
```

Then, we print out the shape of one image so that we know what the dimensions of the data are.



Image Shape: (28, 28, 1)
Training Set: 55000 samples
Validation Set: 5000 samples
Test Set: 10000 samples

```
import numpy as np

# Pad images with 0s
X_train      = np.pad(X_train, ((0,0),(2,2),(2,2),(0,0)), 'constant')
X_validation  = np.pad(X_validation, ((0,0),(2,2),(2,2),(0,0)), 'constant')
X_test       = np.pad(X_test, ((0,0),(2,2),(2,2),(0,0)), 'constant')

print("Updated Image Shape: {}".format(X_train[0].shape))
```

The MNIST data that TensorFlow pre-loads comes as 28x28x1 images.
However, the LeNet architecture only accepts 32x32xC images, where C is the number of color channels.
In order to reformat the MNIST data into a shape that LeNet will accept, we pad the data with two rows of zeros on the top and bottom, and two columns of zeros on the left and right (28+2+2 = 32).
You do not need to modify this section.

Updated Image Shape: (32, 32, 1)

```
import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

index = random.randint(0, len(X_train))
image = X_train[index].squeeze()

plt.figure(figsize=(1,1))
plt.imshow(image, cmap="gray")
print(y_train[index])
```

Visualize Data

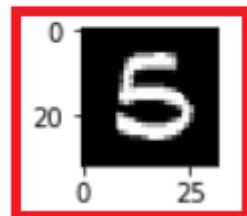
View a sample from the dataset.
You do not need to modify this section.

```
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train)
```

Shuffle the training data.
You do not need to modify this section.

5



We preprocess the data which in this case just amounts to shuffling the training set. It's important to shuffle the training data otherwise the ordering of the data might have a huge effect on how well the network trends.

LeNet Implementation

We start actually building our deep neural network.
First, we load TensorFlow.

```
import tensorflow as tf  
  
EPOCHS = 10  
BATCH_SIZE = 128
```

Setup TensorFlow

The EPOCH and BATCH_SIZE values affect the training speed and model accuracy. You do not need to modify this section.

We will use this **EPOCHS** variable, to tell TensorFlow how many times to run our training data through the network. In general, the more EPOCHS, the better our model will train, but also the longer training will take.

We will also use the **BATCH_SIZE** variable, to tell TensorFlow how many training images to run through the network at a time. The larger the batch size, the faster our model will train, but our processor may have a memory limit on how large a batch it can run.

SOLUTION: Implement LeNet-5

Implement the [LeNet-5](#) neural network architecture. This is the only cell you need to edit.

Input

The LeNet architecture accepts a 32x32xC image as input, where C is the number of color channels. Since MNIST images are grayscale, C is 1 in this case.

Architecture

Layer 1: Convolutional. The output shape should be 28x28x6.

Activation. Your choice of activation function.

Pooling. The output shape should be 14x14x6.

Layer 2: Convolutional. The output shape should be 10x10x16.

Activation. Your choice of activation function.

Pooling. The output shape should be 5x5x16.

Flatten. Flatten the output shape of the final pooling layer such that it's 1D instead of 3D. The easiest way to do is by using `tf.contrib.layers.flatten`, which is already imported for you.

Layer 3: Fully Connected. This should have 120 outputs.

Activation. Your choice of activation function.

Layer 4: Fully Connected. This should have 84 outputs.

Activation. Your choice of activation function.

Layer 5: Fully Connected (Logits). This should have 10 outputs.

Output

Return the result of the 2nd fully connected layer.

Now We come to LeNet, which is the core of the lab.

First we set more hyper parameters. In this case both hyper parameters relate to how we initialize our weights. You can experiment with these values and see if you can do better than what we have here.

```
from tensorflow.contrib.layers import flatten

def LeNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

    # SOLUTION: Activation.
    conv1 = tf.nn.relu(conv1)

    # SOLUTION: Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev = sigma))
    conv2_b = tf.Variable(tf.zeros(16))
    conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

    # SOLUTION: Activation.
    conv2 = tf.nn.relu(conv2)

    # SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
    fc0 = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(120))
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # SOLUTION: Activation.
    fc1 = tf.nn.relu(fc1)

    # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))
    fc2_b = tf.Variable(tf.zeros(84))
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # SOLUTION: Activation.
    fc2 = tf.nn.relu(fc2)

    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 10.
    fc3_W = tf.Variable(tf.truncated_normal(shape=(84, 10), mean = mu, stddev = sigma))
    fc3_b = tf.Variable(tf.zeros(10))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

Initialize the weight & bias

We activate the output of the convolutional layer, in this case with a **ReLU** activation function.

We pool the output, using the 2*2 kernel with a 2*2 stride, which gives us a pooling output of 14*14*6.

We flatten this output into a vector. The length of the vector is 5*5*16, which equals 400.

We apply a **ReLU** activation to the output of this fully connected layer.

We repeat that pattern again this time with a layer width of 84.

$32-5+1=28$

The first convolutional layer. This layer has a 5*5 filter with an input depth of 1, and an output depth of 6.

Use the conv2D function to convolve the filter over the images, and we add the bias at the end.

The network then runs through another set of convolutional activation and pooling layers, giving an output of 5*5*16.

We pass this vector into a fully connected layer, with a width of 120.

Finally, we attach a fully connected output layer, with a width, equal to the number of classes in our label set. In this case, we have 10 classes, one for each digit, so the width of the output layer is 10. These outputs are also known as our logits, which is what we return from the LeNet function.

LeNet Training Pipeline

THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 of a larger set available from NIST. The digits have been size normalized and centered in a fixed size image.

It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spe preprocessing and formatting.

Four files are available on this site:

```
train-images-idx3-ubyte.gz  training set images (9912472 bytes)
train-labels-idx1-ubyte.gz  training set labels (2948 bytes)
test-images-idx3-ubyte.gz    test set images (1648877 bytes)
test-labels-idx1-ubyte.gz    test set labels (1432 bytes)
```

Please note that once you have downloaded the files, you must use the following command to extract them:

X is a placeholder that will store our input batches. We initialize the batch size to **None**, which allow the placeholder to later accept a batch of any size, and we set the image dimensions to 32*32*1. y stores our labels. In this case, our label come through with sparse variables, which just means that they're integers. They aren't one-hot encoded yet.

Here, we set up TensorFlow variables.

```
x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, 10)
```

Features and Labels

Train LeNet to classify **MNIST** data. x is a placeholder for a batch of input images. y is a placeholder for a batch of output labels. You do not need to modify this section.

We use the **tf.one_hot** function to one-hot encode the labels.

The learning rate tells TensorFlow how quickly to update the network's weights.

We pass the input data to the LeNet function to calculate our logits.

```
rate = 0.001
logits = LeNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

Training Pipeline

Create a training pipeline that uses the model to classify MNIST data. You do not need to modify this section.

The **tf.reduce_mean** function averages the cross entropy from all of the training images. AdamOptimizer uses the Adam algorithm to minimize the loss function similarly to what stochastic gradient descent does. The Adam algorithm is a little more sophisticated than stochastic gradient descent, so it's a good default choice for an optimizer. This is where we use the learning rate hyperparameter that we set earlier. Finally, we run the minimize function on the optimizer which uses backpropagation to update the network and minimize our training loss.

We used the **tf.nn.softmax_cross_entropy_with_logits** function to compare those logits to the ground truth labels and calculate the cross entropy. Cross entropy is just a measure of how different the logits are from the ground truth training labels.

LeNet Evaluation Pipeline

The training pipeline we just set up is what trains the model, but the evaluation pipeline we create here will evaluate how good the model is.

In this code cell, we set up another pipeline, this time for evaluating the model.

```
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()
```

These two lines are the entire evaluation pipeline.

```
def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

Takes a dataset as input

Model Evaluation
Evaluate how well the loss and accuracy of the model for a given dataset. You do not need to modify this section.

To measure whether a given prediction is correct by comparing the logit prediction to the one-hot encoded ground truth label.

To calculate the model's overall accuracy by averaging the individual prediction accuracies.

The evaluate function averages the accuracy of each batch to calculate the total accuracy of the model.

Then batches the dataset and runs it through the evaluation pipeline.

LeNet Training the Model

First, we create the TensorFlow session and initialize the variables.

At the beginning of each epoch, we shuffle our training data to ensure that our training isn't biased by the order of the images.

Then, we break our training data into batches and train the model on each batch

We evaluate the model on our validation data.

We train over whatever number of epochs has been set in the **EPOCHS** hyperparameter.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})

        validation_accuracy = evaluate(X_validation, y_validation)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

    saver.save(sess, './lenet')
    print("Model saved")
```

Once we have completely trained the model, we save it.

Train the Model

Run the training data through the training pipeline to train the model. Before each epoch, shuffle the training set.

After each epoch, measure the loss and accuracy of the validation set. Save the model after training. You do not need to modify this section.

lenet.data-00000-of-00001
lenet.index
lenet.meta

Evaluate the Model ¶

Once you are completely satisfied with your model, evaluate the performance of the model on the test set. Be sure to only do this once!

```
with tf.Session() as sess:  
    saver.restore(sess, tf.train.latest_checkpoint('.'))  
  
    test_accuracy = evaluate(X_test, y_test)  
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

← You do not need to modify this section

If you were to measure the performance of your trained model on the test set, then improve your model, and then measure the performance of your model on the test set again, that would invalidate your test results. You wouldn't get a true measure of how well your model would perform against real data.

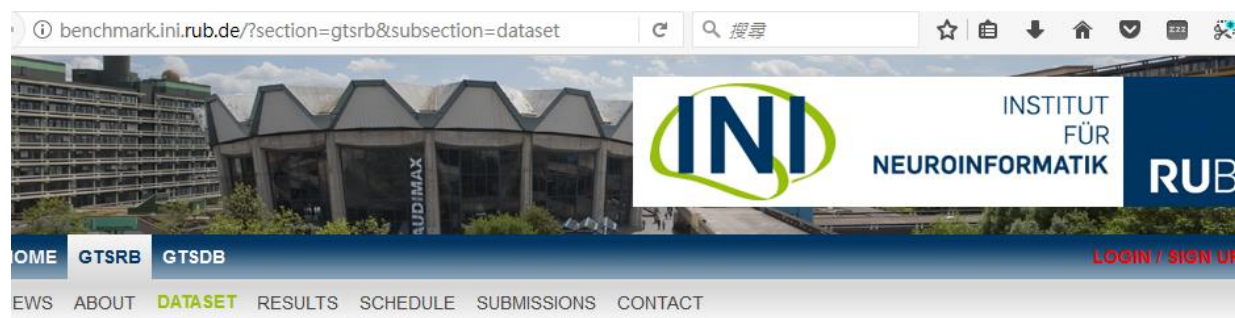
LeNet for Traffic Signs

- First, reset the kernel and clear the output to ensure you have a fresh start.
- Next, clear the cell that loads the MNIST data and replace it with code to load the traffic sign data.
- You should also delete the code that pads the images, since the traffic sign images are already 32*32 pixels.
- The traffic sign data does not come with a validation set. You can use the `train_test_split()` function in the sklearn library though to slice off a validation set from the training set.
- The traffic sign images are in color not grayscale like the MNIST images, so the input depth should be 3 to match 3RGB color channels.
- The traffic sign classifier has 43 classes where is MNIST only had 10, so you'll have to change that.

Traffic Sign Classifier

Traffic Sign Classifier Project

In this project, you will use what you've learned about deep neural networks and convolutional neural networks to classify traffic signs. Specifically, you'll train a model to classify traffic signs from the [German Traffic Sign Dataset](#).



Dataset

- [Overview](#)
- [Structure](#)
- [Image format](#)
- [Annotation format](#)
- [Result format](#)
- [Pre-calculated features](#)
- [Code snippets](#)
- [Citation](#)
- [Result analysis application](#)
- [Downloads](#)



Set Up Your Environment

TensorFlow

If you have access to a GPU, you should follow the TensorFlow instructions for [installing TensorFlow with GPU support](#).

Once you've installed all of the necessary dependencies, you can install the tensorflow-gpu package:

```
pip install tensorflow-gpu
```

Traffic Sign Classifier



Start the Project

1. [Download the dataset](#). This is a pickled dataset in which we've already resized the images to 32x32.
2. Clone the project and start the notebook.

```
git clone https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project  
cd CarND-Traffic-Sign-Classifier-Project
```

3. Launch the Jupyter notebook: `jupyter notebook Traffic_Sign_Classifier.ipynb`

```
conda install scikit-learn
```

Demo

Migrate your TensorFlow 1 code to TensorFlow 2

```
import tensorflow as tf
```



```
import tensorflow.compat.v1 as tf  
tf.disable_v2_behavior()
```

```
from tensorflow.contrib.layers import flatten  
conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')  
fc0 = flatten(conv2)
```

```
fc0 = tf.compat.v1.layers.flatten(conv2)
```

```
x = tf.placeholder(tf.float32, (None, 32, 32, 1))  
y = tf.placeholder(tf.int32, (None))
```



```
x = tf.compat.v1.placeholder(tf.float32, (None, 32, 32, 1))  
y = tf.compat.v1.placeholder(tf.int32, (None))
```

<https://www.tensorflow.org/guide/migrate>

Traffic-Sign-Classifier-Project Demo

Step 0: Load The Data Download the dataset & unzip



C:\ncku\2020\traffic-signs-data

名稱

test.p
train.p
valid.p

```
import pickle
```

```
training_file = "traffic-signs-data/train.p"  
validation_file = "traffic-signs-data/valid.p"  
testing_file = "traffic-signs-data/test.p"
```

```
with open(training_file, mode='rb') as f:  
    train = pickle.load(f)  
with open(validation_file, mode='rb') as f:  
    valid = pickle.load(f)  
with open(testing_file, mode='rb') as f:  
    test = pickle.load(f)
```

```
x_train, y_train = train['features'],  
train['labels']  
x_valid, y_valid = valid['features'],  
valid['labels']  
x_test, y_test = test['features'], test['labels']
```

```
print("x_train shape:", x_train.shape)  
print("y_train shape:", y_train.shape)  
print("x_valid shape:", x_valid.shape)  
print("y_valid shape:", y_valid.shape)  
print("X_test shape:", x_test.shape)  
print("y_test shape:", y_test.shape)
```

• Step 1: Dataset Summary & Exploration

Provide a Basic Summary of the Data Set Using Python, Numpy and/or Pandas

pip install matplotlib
pip install opencv-python

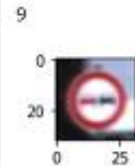
```
import numpy as np
n_train = len(x_train)#Number of training examples
n_test = len(x_test)#number of testing examples
image_shape = x_train[0].shape
n_classes = len(np.unique(y_train))#find the unique_classes of traffic light
print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

Number of training examples = 34799
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43

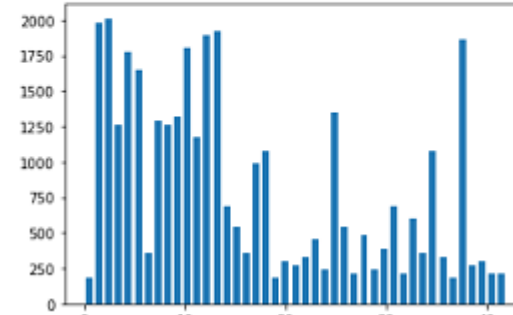
```
### Data exploration visualization goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import random
import numpy as np
# Visualizations will be shown in the notebook.
%matplotlib inline

index = random.randint(0, len(x_train))
image = x_train[index].squeeze()

plt.figure(figsize=(1,1))
plt.imshow(image)
print(y_train[index])
```



```
# plot calculate histogram of Label frequency
hist, bins = np.histogram(y_train, bins=n_classes)
width = 0.7 * (bins[1] - bins[0])
center = (bins[:-1] + bins[1:]) / 2
plt.bar(center, hist, align='center', width=width)
plt.show()
```



```
# show image of 49 random data points
fig, axes = plt.subplots(7,7, figsize=(15, 12))
fig.subplots_adjust(hspace=.2, wspace=.001)
axes = axes.ravel()
for i in range(49):
    index = random.randint(0, len(x_train))
    image = x_train[index]
    axes[i].axis('off')
    axes[i].imshow(image)
    axes[i].set_title(y_train[index])
```



signnames.csv - Excel	
ClassId	SignName
0	Speed limit (20km/h)
1	Speed limit (30km/h)
2	Speed limit (50km/h)
3	Speed limit (60km/h)
4	Speed limit (70km/h)
5	Speed limit (80km/h)
6	End of speed limit (80km/h)
7	Speed limit (100km/h)
8	Speed limit (120km/h)
9	No passing

```
def getSignNames():
    return pd.read_csv('./signnames.csv').values
```


- Step 2: Design and Test a Model Architecture
 - Pre-process the Data Set (normalization, grayscale, etc.)

```
# 1. rgb to grayscale
# 2. apply histogram equalization
# 3. normalize the training and testing data to [-1 1]
import cv2
def grayAndEqualizeHist(img):
    """
    :param img: input RGB image
    :return: histogram equalized grayscale image
    """
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    equ = cv2.equalizeHist(gray)
    #equ = equ - 128. / 128.
    return equ

x_train = np.array([grayAndEqualizeHist(img) for img in x_train])
x_test = np.array([grayAndEqualizeHist(img) for img in x_test])

print('preprocessed the data')
```

Model Architecture

```
from tensorflow.contrib.layers import flatten

def LeNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

    # SOLUTION: Activation.
    conv1 = tf.nn.relu(conv1)

    # SOLUTION: Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev = sigma))
    conv2_b = tf.Variable(tf.zeros(16))
    conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

    # SOLUTION: Activation.
    conv2 = tf.nn.relu(conv2)

    # SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
    fc0 = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(120))
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # SOLUTION: Activation.
    fc1 = tf.nn.relu(fc1)

    # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))
    fc2_b = tf.Variable(tf.zeros(84))
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # SOLUTION: Activation.
    fc2 = tf.nn.relu(fc2)

    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 10.
    fc3_W = tf.Variable(tf.truncated_normal(shape=(84, 10), mean = mu, stddev = sigma))
    fc3_b = tf.Variable(tf.zeros(10))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

LeNet Training Pipeline & Evaluation Pipeline

```
from sklearn.utils import shuffle
```

```
rate = 0.001
EPOCHS = 30
BATCH_SIZE = 128
#x = tf.placeholder(tf.float32, (None, 32, 32, 1))
#y = tf.placeholder(tf.int32, (None))
x = tf.compat.v1.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.compat.v1.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)
#keep_prob = tf.placeholder(tf.float32)
keep_prob = tf.compat.v1.placeholder(tf.float32)

#logits = ConvNet(x)
logits = LeNet(x)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits = logits, labels=one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()
```

```
def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y, keep_prob: 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

LeNet Training the Model

```
# Training and evaluation
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(x_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        x_train, y_train = shuffle(x_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = x_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep_prob: 0.5})

        validation_accuracy = evaluate(x_validation, y_validation)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

    saver.save(sess, './lenet')
    print("Model saved")
```

Training...

EPOCH 1 ...

Validation Accuracy = 0.812

EPOCH 2 ...

Validation Accuracy = 0.875

`print(tf.trainable_variables())`

Evaluate the Model

```
# Test with testing data
with tf.Session() as sess:
    #saver.restore(sess, tf.train.latest_checkpoint('.'))
    sess.run(tf.global_variables_initializer())
    saver1 = tf.train.import_meta_graph('./lenet.meta')
    saver1.restore(sess, "./lenet")

    test_accuracy = evaluate(x_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

INFO:tensorflow:Restoring parameters from ./lenet
Test Accuracy = 0.923

Step 3: Test a Model on New Images

```
n = 6
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.import_meta_graph('./lenet.meta')
    saver.restore(sess, "./lenet")
    top_k = sess.run(tf.nn.top_k(tf.nn.softmax(logits), k=5), feed_dict={x: x_test_new, keep_prob: 1.0})

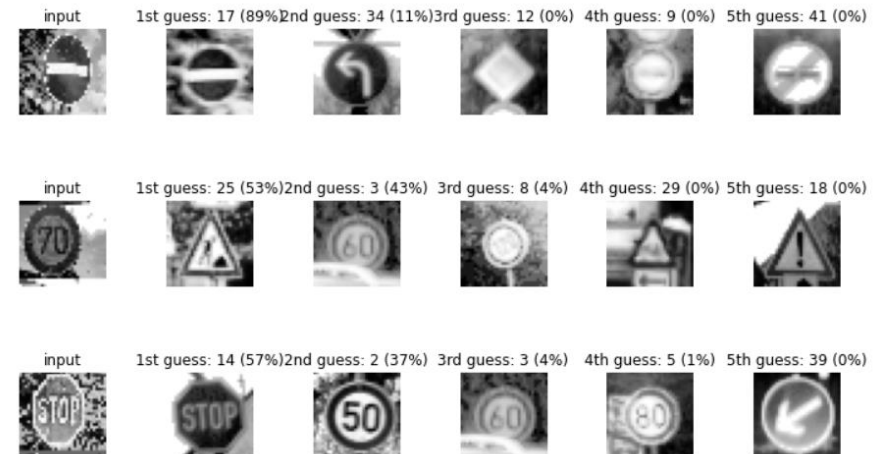
    fig, axs = plt.subplots(len(x_test_new), n, figsize=(12, 12))
    fig.subplots_adjust(hspace = .5, wspace=.7)
    axs = axs.ravel()

    for i, image in enumerate(x_test_new):

        axs[n*i].axis('off')
        axs[n*i].imshow(image.squeeze(), cmap='gray')
        axs[n*i].set_title('input')

        top1 = top_k[1][i][0]
        idx1 = np.argmax(y_test == top1)[0]
        axs[n*i+1].axis('off')
        axs[n*i+1].imshow(x_test[idx1].squeeze(), cmap='gray')
        axs[n*i+1].set_title('1st guess: {} ({:.0f}%)' .format(top1, 100*top_k[0][i][0]))

        top2 = top_k[1][i][1]
```



Traffic Sign Recognition with TensorFlow 2.x

It is a big change from TensorFlow 1.0 to 2.0 with a tighter Keras integration, where the focus is more on higher level APIs

- **Data overview**(pip install pandas)
- **Model construction**
- **Model training and evaluation**

Kaggle- GTSRB - German Traffic Sign Recognition Benchmark

kaggle

Dataset

GTSRB - German Traffic Sign Recognition Benchmark

Multi-class, single-image classification challenge

Mykola • updated a year ago (Version 1)

Data Tasks Kernels (36) Discussion (2) Activity Metadata

Download (300 MB) New Notebook

URL: <https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign>

MYKOLA • UPDATED 3 YEARS AGO

804

New Notebook

Download (642 MB)

⋮

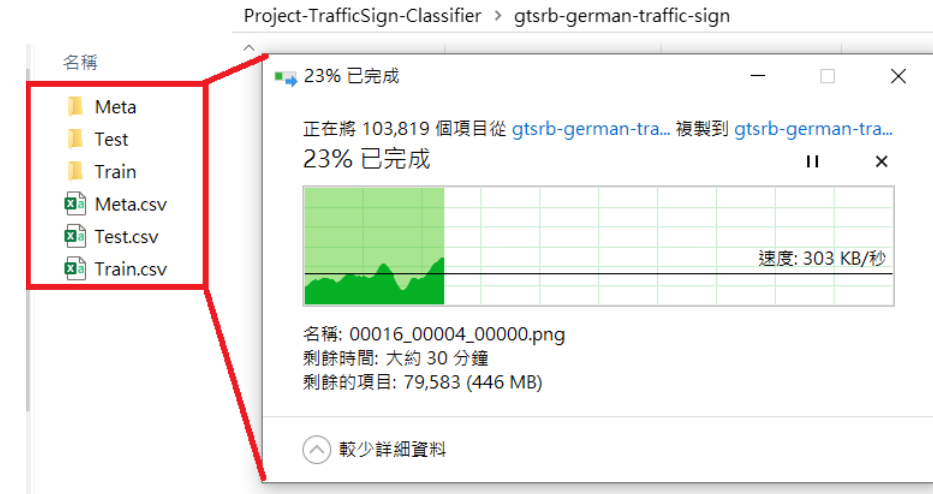
GTSRB - German Traffic Sign Recognition Benchmark

Multi-class, single-image classification challenge



Data overview

- The images are of different sizes ranging from 20x20 to 70x70, and all have 3 channels: RGB
- The data package includes folders of Train, Test and a test.csv. There are a meta.csv and a Meta folder to show the standard image for each traffic sign. There is also a signname.csv for mapping a label to its description. Train folder contains 43 sub-folders whose names are the labels of the images in them. For example, all the images in folder 0 has a class label of 0 and so on...
- So the first thing I have to do is to resize all the images to 32x32x3 and read them into a numpy array as training features. At the same time, I created another numpy array with labels of each image, which is from the folder name where the image loaded from.
- In the GTSRB dataset, there 51,839 German traffic signs in 43 classes.



From the training set, randomly spitted 20% as validation set for use during the process of model training. The model accuracy of training and validation will give us information about underfitting or overfitting.

Converted images to grayscale and normalized each pixels. Normalization makes model to converge more quickly

```
# shuffle training data and split them into training and validation
indices = np.random.permutation(trainx.shape[0])
# 20% to val
split_idx = int(trainx.shape[0]*0.8)
train_idx, val_idx = indices[:split_idx], indices[split_idx:]
X_train, X_validation = trainx[train_idx:], trainx[val_idx:]
y_train, y_validation = trainy[train_idx], trainy[val_idx]
```

```
# get overall stat of the whole dataset
n_train = X_train.shape[0]
n_validation = X_validation.shape[0]
n_test = X_test.shape[0]
image_shape = X_train[0].shape
n_classes = len(np.unique(y_train))
print("There are {} training examples ".format(n_train))
print("There are {} validation examples".format(n_validation))
print("There are {} testing examples".format(n_test))
print("Image data shape is {}".format(image_shape))
print("There are {} classes".format(n_classes))
```

```
There are 31367 training examples
There are 7842 validation examples
There are 12630 testing examples
Image data shape is (32, 32, 3)
There are 43 classes
```

```
# convert the images to grayscale
X_train_gry = np.sum(X_train/3, axis=3, keepdims=True)
X_validation_gry = np.sum(X_validation/3, axis=3, keepdims=True)
X_test_gry = np.sum(X_test/3, axis=3, keepdims=True)

# Normalize data
X_train_normalized_gry = (X_train_gry-128)/128
X_validation_normalized_gry = (X_validation_gry-128)/128
X_test_normalized_gry = (X_test_gry-128)/128
```

Model construction

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

model = models.Sequential()
# Conv 32x32x1 => 28x28x6.
model.add(layers.Conv2D(filters = 6, kernel_size = (5, 5), strides=(1, 1), padding='valid',
                        activation='relu', data_format = 'channels_last', input_shape = (32, 32, 1)))
# Maxpool 28x28x6 => 14x14x6
model.add(layers.MaxPooling2D((2, 2)))
# Conv 14x14x6 => 10x10x16
model.add(layers.Conv2D(16, (5, 5), activation='relu'))
# Maxpool 10x10x16 => 5x5x16
model.add(layers.MaxPooling2D((2, 2)))
# Flatten 5x5x16 => 400
model.add(layers.Flatten())
# Fully connected 400 => 120
model.add(layers.Dense(120, activation='relu'))
# Fully connected 120 => 84
model.add(layers.Dense(84, activation='relu'))
# Dropout
model.add(layers.Dropout(0.2))
# Fully connected, output layer 84 => 43
model.add(layers.Dense(43, activation='softmax'))
```

Model training and evaluation

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 6)	156
max_pooling2d (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2416
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 120)	48120
dense_1 (Dense)	(None, 84)	10164
dropout (Dropout)	(None, 84)	0
dense_2 (Dense)	(None, 43)	3655
Total params: 64,511		
Trainable params: 64,511		
Non-trainable params: 0		

```
# specify optimizer, loss function and metric
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# training batch_size=128, epochs=20
conv = model.fit(X_train, y_train, batch_size=128, epochs=20,
                 validation_data=(X_validation, y_validation))
```

Train on 31367 samples, validate on 7842 samples

Epoch 1/20

31367/31367 [=====] - 10s 309us/sample - loss: 2.1147 - accuracy: 0.4291 - val_loss: 0.8104 - val_accuracy: 0.7733

Epoch 2/20

31367/31367 [=====] - 9s 295us/sample - loss: 0.6644 - accuracy: 0.8058 - val_loss: 0.3500 - val_accuracy: 0.9142

Epoch 3/20

31367/31367 [=====] - 9s 288us/sample - loss: 0.3574 - accuracy: 0.8998 - val_loss: 0.2320 - val_accuracy: 0.9458

Epoch 4/20

31367/31367 [=====] - 9s 286us/sample - loss: 0.2348 - accuracy: 0.9340 - val_loss: 0.1523 - val_accuracy: 0.9672

Epoch 5/20

31367/31367 [=====] - 9s 296us/sample - loss: 0.1724 - accuracy: 0.9531 - val_loss: 0.1229 - val_accuracy: 0.9697

Epoch 6/20

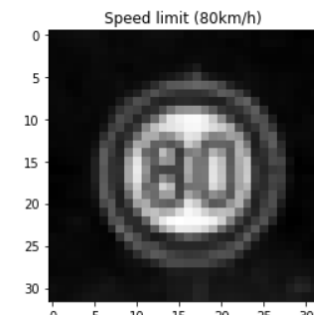
31367/31367 [=====] - 9s 290us/sample - loss: 0.1422 - accuracy: 0.9601 - val_loss: 0.1241 - val_accuracy: 0.9660

```
model.evaluate(x=X_test, y=y_test)
```

12630/12630 [=====] - 4s 348us/sample - loss: 0.6001 - accuracy: 0.9218

```
index = np.random.randint(0, n_test)
im = X_test[index]
fig, ax = plt.subplots()
ax.set_title(sign.loc[sign['ClassId'] == np.argmax(model.predict(np.array([im])), 'SignName'].values[0])
ax.imshow(im.squeeze(), cmap = 'gray')
```

<matplotlib.image.AxesImage at 0x1d63befa708>



Traffic Sign Recognition with TensorFlow 2.x (advanced)

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
data_aug = ImageDataGenerator(
    featurewise_center=False,
    featurewise_std_normalization=False,
    rotation_range=10,
    zoom_range=0.2,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.11,
    horizontal_flip=False,
    vertical_flip=False)
```

```
# Define a Callback class that stops training once accuracy reaches 98.0%
class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy')>0.97):
            print("\nReached 97.0% accuracy so cancelling training!")
            self.model.stop_training = True
```

```
callbacks = myCallback()
# specify optimizer, loss function and metric
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# training batch_size=128, epochs=30
# conv = model.fit(data_aug.flow(X_train, y_train, batch_size=128), epochs=30,
#                   validation_data=(X_validation, y_validation))
conv = model.fit(data_aug.flow(X_train, y_train, batch_size=128), epochs=200,
                 validation_data=(X_validation, y_validation), verbose = 2,
                 callbacks=[callbacks])
```

```
print(conv.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

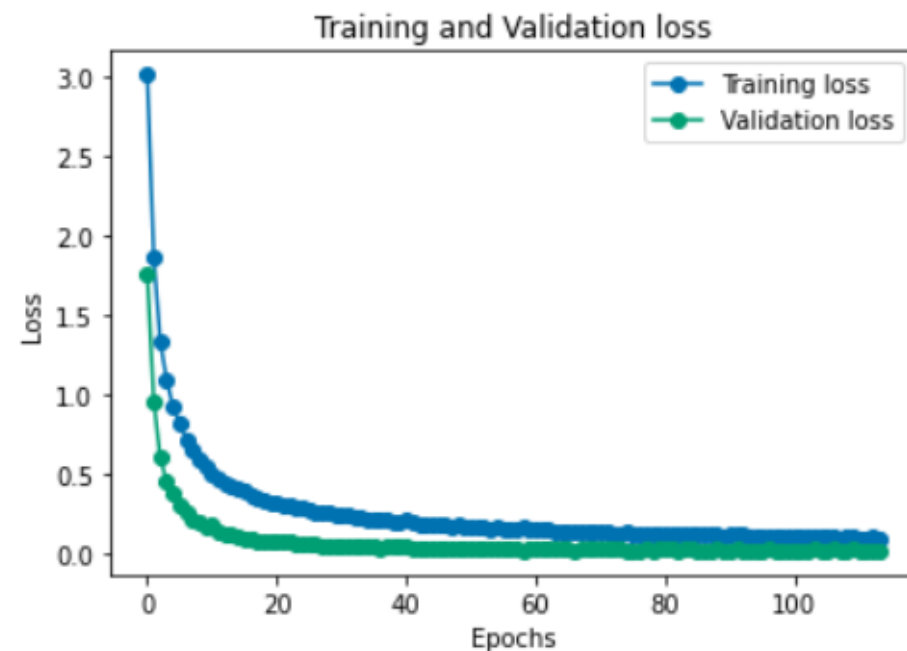
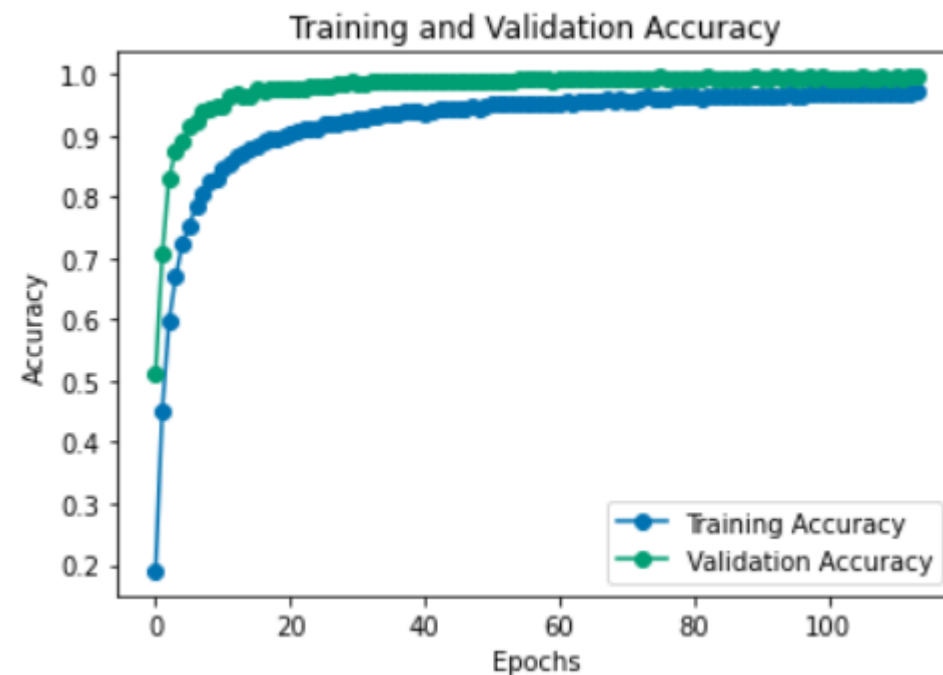
```
# summarize history for accuracy
plt.plot(conv.history['accuracy'], '-o')
plt.plot(conv.history['val_accuracy'], '-o')
plt.title('Training and Validation Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(['Training Accuracy', 'Validation Accuracy'], loc='lower right')
plt.show()

# summarize history for loss
plt.plot(conv.history['loss'], '-o')
plt.plot(conv.history['val_loss'], '-o')
plt.title('Training and Validation loss')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['Training loss', 'Validation loss'], loc='upper right')
plt.show()
```

```
model.evaluate(x=X_test, y=y_test)
```

```
12630/12630 [=====] - 2s 138us/sample - loss: 0.2630 - accuracy: 0.9486
```

```
[0.2630225523550105, 0.9486144]
```

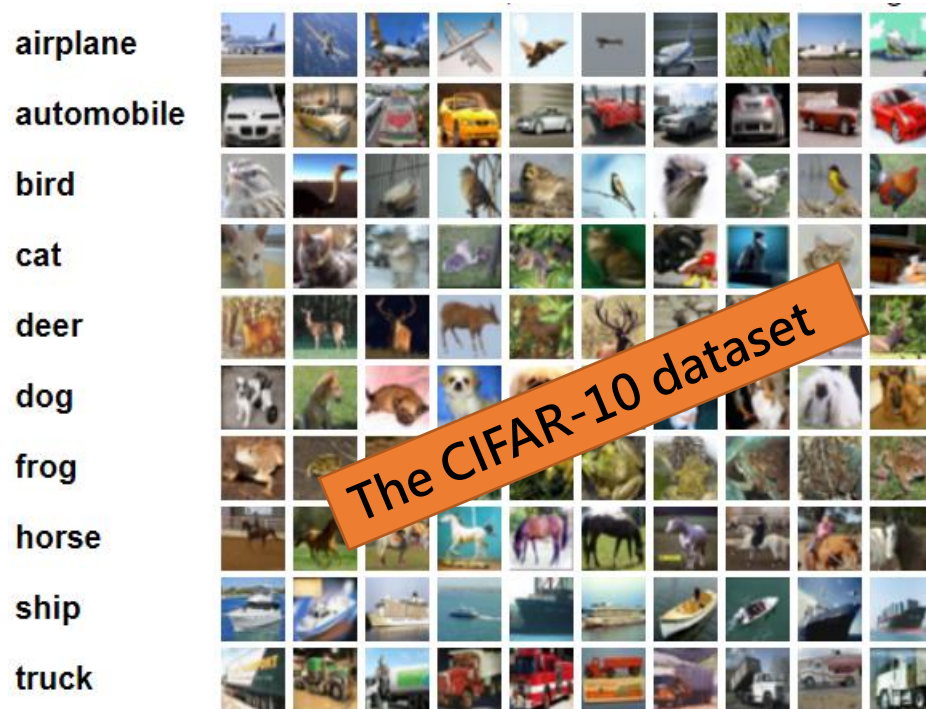


CIFAR-100 classification with Keras

CIFAR-100 classification

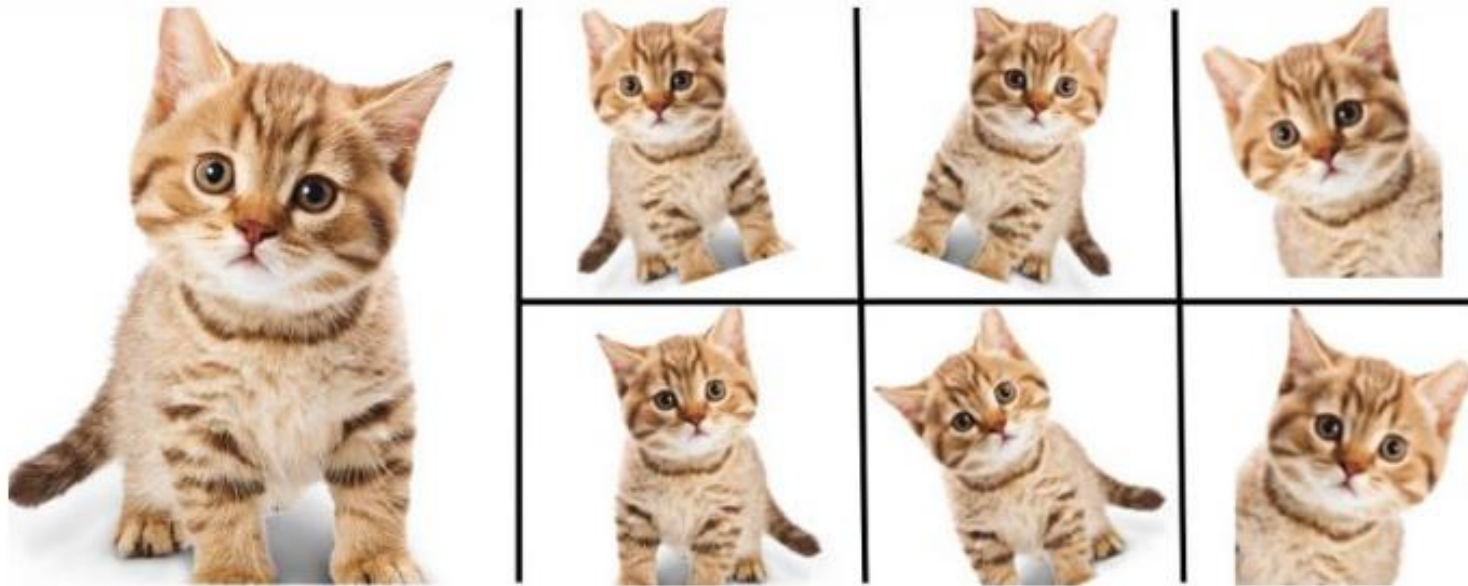
CIFAR dataset-Krizhevsky & Hinton (2009)

Reference : <https://www.cs.toronto.edu/~kriz/cifar.html>



CIFAR-100 classification
with Keras
(Tensorflow 2.X)

Data Augmentation-Enlarge your Dataset



Enlarge your Dataset

Complements

- ***Keras Tutorial:*** [The Ultimate Beginner's Guide to Deep Learning in Python](#)
- ***Keras Tutorial -*** [Traffic Sign Recognition](#)
- [Feeding your own data set into the CNN model in Keras](#)

