LeNet Lab

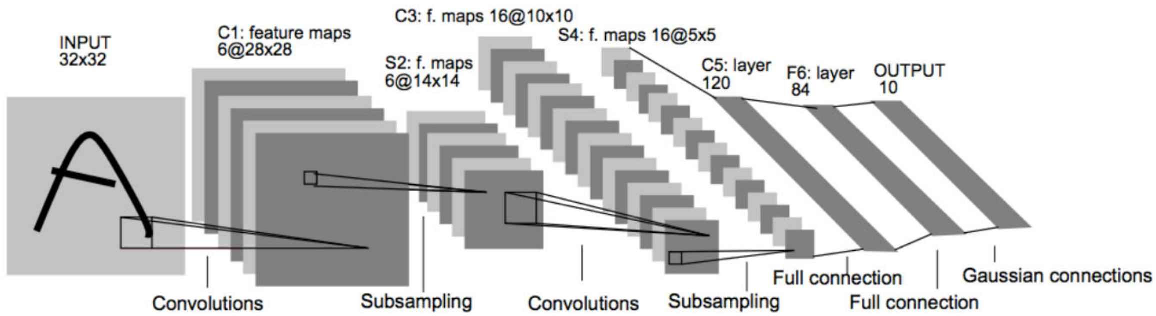


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet. Source: Yann Lecun.

Load Data

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

You do not need to modify this section.

In [12]:

```
import numpy as np
from sklearn.model_selection import train_test_split

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
from tensorflow import keras
from keras import backend as k



(x_train, y_train) , (x_test, y_test) = keras.datasets.mnist.load_data()

# Splitting the validation set out of training set
x_train , x_validation, y_train, y_validation = train_test_split(x_train, y_train, test_size = 0.08333)

# normalization our training, validation, test image set.
x_train = x_train/255.0
x_validation = x_validation/255.0
x_test = x_test/255.0


# Applying Channels in the image
img_rows, img_cols=28, 28

if k.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    x_validation = x_validation.reshape(x_validation.shape[0], 1, img_rows, img_cols)
```

```
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    x_validation = x_validation.reshape(x_validation.shape[0], img_rows, img_cols, 1)

#Verifying the number of images in each set matches the number of labels in same set
assert(len(x_train) == len(y_train))
assert(len(x_validation) == len(y_validation))
assert(len(x_test) == len(y_test))



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)))
```

```
Image Shape: (28, 28, 1)

Training Set:   55000 samples
Validation Set: 5000 samples
Test Set:       10000 samples
```

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.

In [2]:

```
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))
```

```
Updated Image Shape: (32, 32, 1)
```

Visualize Data

View a sample from the dataset.

You do not need to modify this section.

In [3]:

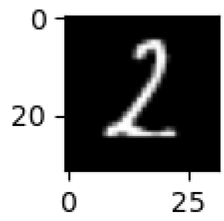
```
import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

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])
```

2



Preprocess Data

Shuffle the training data.

You do not need to modify this section

In [4]:
```
from sklearn.utils import shuffle

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

Setup TensorFlow

The EPOCH and BATCH_SIZE values affect the training speed and model accuracy.

You do not need to modify this section

In [5]:
```
EPOCHS = 10
BATCH_SIZE = 128
```

TODO: 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.

In [6]:

```
def LeNet(x):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each
    w = {'wc1': tf.Variable(tf.truncated_normal(shape=[5,5,1,6], mean = mu, stddev = sigma)),
      'wc2': tf.Variable(tf.truncated_normal(shape=[5,5,6,16], mean = mu, stddev = sigma)),
      'wd1': tf.Variable(tf.truncated_normal(shape=[5*5*16,120], mean = mu, stddev = sigma)),
      'wd2': tf.Variable(tf.truncated_normal(shape=[120,84], mean = mu, stddev = sigma)),
      'out': tf.Variable(tf.truncated_normal(shape=[84,10], mean = mu, stddev = sigma)) }

    b ={'bc1': tf.Variable(tf.zeros([6])),
     'bc2': tf.Variable(tf.zeros([16])),
     'bd1': tf.Variable(tf.zeros([120])),
     'bd2': tf.Variable(tf.zeros([84])),
     'out': tf.Variable(tf.zeros([10]))
     }

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1 = tf.nn.conv2d(x, w['wc1'], strides = [1,1,1,1], padding = 'VALID')
    conv1 = tf.nn.bias_add(conv1,b['bc1'])
    # Activation.
    conv1 = tf.nn.relu(conv1)
    # Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize = [1,2,2,1], strides = [1,2,2,1], padding = 'SAME')


    # Layer 2: Convolutional. Output = 10x10x16.
    conv2 = tf.nn.conv2d(conv1, w['wc2'], strides = [1,1,1,1], padding = 'VALID')
```

```python
    conv2 = tf.nn.bias_add(conv2,b['bc2'])
    # Activation.
    conv2 = tf.nn.relu(conv2)
    # Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize = [1,2,2,1], strides = [1,2,2,1], padding = 'SAME')


    # Flatten. Input = 5x5x16. Output = 400.
    conv2_flat = tf.reshape(conv2, [-1, w['wd1'].get_shape().as_list()[0]])


    # Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1 = tf.add(tf.matmul(conv2_flat,w['wd1']), b['bd1'])
    # Activation.
    fc1 = tf.nn.relu(fc1)


    # Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2 = tf.add(tf.matmul(fc1,w['wd2']), b['bd2'])
    # Activation.
    fc2 = tf.nn.relu(fc2)


    # Layer 5: Fully Connected. Input = 84. Output = 10.
    logits = tf.add(tf.matmul(fc2,w['out']), b['out'])

    return logits
```

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.

In [7]:

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

Training Pipeline

Create a training pipeline that uses the model to classify MNIST data.

You do not need to modify this section.

In [8]:

```python
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)
```

```
WARNING:tensorflow:From C:\Users\akislaya\Anaconda3\envs\IntroToTensorFlow\lib\site-packages\tensorflo
w\python\util\dispatch.py:1176: softmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is de
```

```
precated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.
```

Model Evaluation

Evaluate how well the loss and accuracy of the model for a given dataset.

You do not need to modify this section.

In [9]:

```
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})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

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.

In [10]:

```
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")
```

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.963

EPOCH 2 ...
Validation Accuracy = 0.975

EPOCH 3 ...
Validation Accuracy = 0.981

EPOCH 4 ...
Validation Accuracy = 0.975

EPOCH 5 ...
Validation Accuracy = 0.985

EPOCH 6 ...
Validation Accuracy = 0.986

EPOCH 7 ...
Validation Accuracy = 0.984

EPOCH 8 ...
Validation Accuracy = 0.989

EPOCH 9 ...
Validation Accuracy = 0.985

EPOCH 10 ...
Validation Accuracy = 0.990

Model saved
```

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!

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.

You do not need to modify this section.

In [11]:

```
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))
```

```
INFO:tensorflow:Restoring parameters from .\lenet
Test Accuracy = 0.990
```