In [1]:

```
# Load necessary packages
import numpy as np
import pandas as pd
import matplotlib as mpl
import os, glob

# Visualization modules
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

# Data-processing modules
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array

# Modeling and evaluation modules
from keras.models import model_from_yaml
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten
from sklearn import metrics

import warnings
warnings.filterwarnings("ignore")

# Set seed for reproducibility
SEED = 42
```

```
Using TensorFlow backend.
```

Part I. Data Processing

The train & test data is pretty clean in terms of image data, but we will need to do a bit of prep work to use in our model.

a) Use the "ImageDataGenerator()" class from keras.processing.image to build out an instance called "train_datagen" with the following parameters:

- rescale = 1./255
- shear_range = 0.2
- zoom_range = 0.2
- horizontal_flip = True

In [2]:

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)
```

b) Then build your training set by using the method ".flow_from_directory()"

- path (where training data is stored)

- target_size = (64, 64)
- batch_size = 32
- class_mode = categorical

In [3]:

```
train_generator = train_datagen.flow_from_directory(
    directory="dataset_train/",
    target_size=(64, 64),
    batch_size=32,
    class_mode="categorical",
    seed=SEED)
```

```
Found 88 images belonging to 4 classes.
```

c) Take a look at your training set:

- What is the image shape of each training observation?
- How many total classes do we need to predict on?

In [4]:

```
# Iterate over each train image in each category to count
train_files = glob.glob("dataset_train/*")
img_count = {}
for i in train_files:
    img_by_cat = len(os.listdir(i+'/'))
    img_count[i] = img_by_cat

img_count
```

Out[4]:

```
{'dataset_train/category 2': 22,
 'dataset_train/category 4': 22,
 'dataset_train/category 3': 22,
 'dataset_train/category 1': 22}
```

There are 4 classes in total that we need to predict on (categories 1 to 4). For each category, we have 22 images for training.

In [5]:

```
for i in train_files:
    sample_images = list(os.listdir(i+'/'))

sample_images
```

Out[5]:

```
['1031.png',
 '1025.png',
 '1024.png',
 '1030.png',
 '1032.png',
 '1033.png',
 '1023.png',
 '1020.png',
 '1021.png',
 '1052.png',
 '1051.png',
 '1050.png',
 '1040.png',
 '1041.png',
 '1043.png',
 '1042.png',
 '1010.png',
 '1011.png',
 '1013.png',
 '1012.png',
 '1000.png',
 '1014.png']
```

In [6]:

```
img = load_img('dataset_train/category 1/1000.png')
x = img_to_array(img)  # Convert to a numpy array
x.shape # This is the shape of a training image
```

Out[6]:

```
(800, 800, 3)
```

In [7]:

```
# Get shape after data processing
train_generator.target_size
```

Out[7]:

```
(64, 64)
```

Part II. Build Initial Classifier

Create an instance of Sequential called "classifier"

Add a Conv2D layer with the following parameters:

- filters = 32
- kernel_size = (3,3)

- kernel_size = (3,3)
- input_shape = image shape found in part 1
- activation = relu

Add a MaxPooling2D layer where pool_size = (2,2)

Add another Conv2D layer:

- filters = 64
- kernel_size = (3,3)
- activation = relu

Add a MaxPooling2D layer where pool_size = (2,2)

Add a Flatten layer

Add a Dense layer

- units = 128
- activation = relu

Add a final Dense layer (this will output our probabilities):

- units = # of classes
- activation = softmax

Compile with the following:

- optimize = adam
- loss = categorical cross entropy
- metric = accuracy

In [8]:

```
classifier = Sequential()

classifier.add(Conv2D(filters=32, kernel_size=(3, 3), input_shape=(64, 64, 3), activ
classifier.add(MaxPooling2D(pool_size=(2, 2)))

classifier.add(Conv2D(filters=64, kernel_size=(3, 3), activation="relu"))
classifier.add(MaxPooling2D(pool_size=(2, 2)))
classifier.add(Flatten())  # this converts our feature maps to 1D feature vectors

classifier.add(Dense(units=128, activation="relu"))
classifier.add(Dense(units=4, activation="softmax"))

classifier.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accu

classifier.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 32)        896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 29, 29, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 12544)             0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1605760
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 516
=================================================================
Total params: 1,625,668
Trainable params: 1,625,668
Non-trainable params: 0
_________________________________________________________________
```

Part III. Model Runs

This will be run various times with different numbers of steps_per_epoch and epochs.

a) Use .fit_generator() with the training set. For the first run, use the following parameters:

- steps_per_epoch = 10
- epochs = 10

In [9]:

```
classifier.fit_generator(train_generator, steps_per_epoch=10, epochs=10)
```

```
Epoch 1/10
10/10 [==============================] - 2s 219ms/step - loss: 1.0824
- accuracy: 0.6115
Epoch 2/10
10/10 [==============================] - 2s 200ms/step - loss: 0.2689
- accuracy: 0.9236
Epoch 3/10
10/10 [==============================] - 2s 205ms/step - loss: 0.1048
- accuracy: 0.9662
Epoch 4/10
10/10 [==============================] - 2s 218ms/step - loss: 0.0708
- accuracy: 0.9831
Epoch 5/10
10/10 [==============================] - 2s 204ms/step - loss: 0.0445
- accuracy: 0.9831
Epoch 6/10
10/10 [==============================] - 2s 204ms/step - loss: 0.0165
- accuracy: 1.0000
Epoch 7/10
10/10 [==============================] - 2s 217ms/step - loss: 0.0157
- accuracy: 0.9932
Epoch 8/10
10/10 [==============================] - 2s 215ms/step - loss: 0.0090
- accuracy: 1.0000
Epoch 9/10
10/10 [==============================] - 2s 208ms/step - loss: 0.0132
- accuracy: 0.9932
Epoch 10/10
10/10 [==============================] - 2s 207ms/step - loss: 0.0030
- accuracy: 1.0000
```

Out[9]:

```
<keras.callbacks.callbacks.History at 0x14185a710>
```

b) Write out each model & model_weights to a file.

In [10]:

```
# Write model and model weights to disk
model_yaml = classifier.to_yaml()
with open("model_1.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)

classifier.save_weights("model_1.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

c) Predict using the model built in step 2.

In [11]:

```
# Load model from disk
yaml_file = open('model_1.yaml', 'r')
loaded_model_yaml = yaml_file.read()
yaml_file.close()
model = model_from_yaml(loaded_model_yaml)

# Load weights into new model
model.load_weights("model_1.h5")
print("Loaded model from disk")

# Test data path
img_dir = "dataset_test/" # Enter Directory of all images

# Iterate over each test image
# Make a prediction and add to results
data_path = os.path.join(img_dir, '*g')
files = glob.glob(data_path)
data = []
results = []
for f1 in files:
    img = load_img(f1, target_size = (64, 64))
    img = img_to_array(img)
    img = np.expand_dims(img, axis = 0)
    data.append(img)
    result = model.predict(img)
    r = np.argmax(result, axis=1)
    results.append(r)

results
```

```
Loaded model from disk
```

Out[11]:

```
[array([3]),
 array([0]),
 array([0]),
 array([0]),
 array([1]),
 array([2]),
 array([1]),
 array([1])]
```

d) Determine accuracy. Note: To determine accuracy, you will need to manually check the labels given to each class in the training data. This will require you to go and look in the training data, and then determine how a category was coded in keras.

In [12]:

```
# Check category labels
train_generator.class_indices
```

Out[12]:

```
{'category 1': 0, 'category 2': 1, 'category 3': 2, 'category 4': 3}
```

After you do this, you will need to compare predicted values to the actual values for the test set (there are only

8 test observations).

In [13]:

```
# Understand the order of file reading of the test images
for f1 in files:
    print(f1)
```

```
dataset_test/C033.png
dataset_test/1022.png
dataset_test/4011.png
dataset_test/1053.png
dataset_test/6051.png
dataset_test/4053.png
dataset_test/C014.png
dataset_test/6023.png
```

In [14]:

```
# Manually label the test data
test_labels = [3,0,2,0,1,2,3,1]
```

In [15]:

```
# Determine accuracy
acc = metrics.accuracy_score(test_labels, results)
acc
```

Out[15]:

```
0.75
```

e) Run this process for the following combinations:

- (steps_per_epoch: 10, epochs: 10) <- the one we just did
- (steps_per_epoch: 10, epochs: 20)
- (steps_per_epoch: 10, epochs: 30)
- (steps_per_epoch: 30, epochs: 10)
- (steps_per_epoch: 30, epochs: 20)
- (steps_per_epoch: 30, epochs: 30)
- (steps_per_epoch: 50, epochs: 10)
- (steps_per_epoch: 50, epochs: 20)
- (steps_per_epoch: 50, epochs: 30)
- (steps_per_epoch: 50, epochs: 100) (Please note: This one will take some time so you should consider running and saving the model outputs so you don't have to keep an eye on your code)

In [16]:

```
# Write a function that allows us to train the same model on differnet steps per epo
def fit_gen(model, s, e):
    model.fit_generator(train_generator, steps_per_epoch=s, epochs=e)
    return model
```

In [18]:

```
mod2 = fit_gen(classifier, 10, 20)
```

```
Epoch 1/20
10/10 [==============================] - 2s 217ms/step - loss: 5.0645e
-04 - accuracy: 1.0000
Epoch 2/20
10/10 [==============================] - 2s 237ms/step - loss: 4.2707e
-04 - accuracy: 1.0000
Epoch 3/20
10/10 [==============================] - 2s 219ms/step - loss: 2.8854e
-04 - accuracy: 1.0000
Epoch 4/20
10/10 [==============================] - 2s 216ms/step - loss: 2.1609e
-04 - accuracy: 1.0000
Epoch 5/20
10/10 [==============================] - 2s 214ms/step - loss: 3.4656e
-04 - accuracy: 1.0000
Epoch 6/20
10/10 [==============================] - 2s 216ms/step - loss: 1.9113e
-04 - accuracy: 1.0000
Epoch 7/20
10/10 [==============================] - 2s 229ms/step - loss: 7.5189e
-04 - accuracy: 1.0000
Epoch 8/20
10/10 [==============================] - 2s 232ms/step - loss: 4.3758e
-04 - accuracy: 1.0000
Epoch 9/20
10/10 [==============================] - 2s 224ms/step - loss: 1.5342e
-04 - accuracy: 1.0000
Epoch 10/20
10/10 [==============================] - 2s 226ms/step - loss: 1.3853e
-04 - accuracy: 1.0000
Epoch 11/20
10/10 [==============================] - 2s 215ms/step - loss: 1.7344e
-04 - accuracy: 1.0000
Epoch 12/20
10/10 [==============================] - 2s 225ms/step - loss: 1.3575e
-04 - accuracy: 1.0000
Epoch 13/20
10/10 [==============================] - 2s 203ms/step - loss: 2.1481e
-04 - accuracy: 1.0000
Epoch 14/20
10/10 [==============================] - 2s 206ms/step - loss: 1.8518e
-04 - accuracy: 1.0000
Epoch 15/20
10/10 [==============================] - 2s 207ms/step - loss: 1.5538e
-04 - accuracy: 1.0000
Epoch 16/20
10/10 [==============================] - 2s 202ms/step - loss: 1.1764e
-04 - accuracy: 1.0000
Epoch 17/20
10/10 [==============================] - 2s 220ms/step - loss: 1.7216e
-04 - accuracy: 1.0000
Epoch 18/20
10/10 [==============================] - 2s 212ms/step - loss: 8.4724e
-05 - accuracy: 1.0000
Epoch 19/20
10/10 [==============================] - 2s 218ms/step - loss: 8.1489e
-05 - accuracy: 1.0000
```

```
Epoch 20/20
10/10 [==============================] - 2s 223ms/step - loss: 8.6301e
-05 - accuracy: 1.0000
```

In [19]:

```
mod3 = fit_gen(classifier, 10, 30)
```

```
Epoch 1/30
10/10 [==============================] - 2s 210ms/step - loss: 7.3900e
-05 - accuracy: 1.0000
Epoch 2/30
10/10 [==============================] - 2s 202ms/step - loss: 5.7942e
-05 - accuracy: 1.0000
Epoch 3/30
10/10 [==============================] - 2s 227ms/step - loss: 9.6341e
-05 - accuracy: 1.0000
Epoch 4/30
10/10 [==============================] - 2s 215ms/step - loss: 9.2406e
-05 - accuracy: 1.0000
Epoch 5/30
10/10 [==============================] - 2s 220ms/step - loss: 1.2448e
-04 - accuracy: 1.0000
Epoch 6/30
10/10 [==============================] - 2s 221ms/step - loss: 1.4940e
-04 - accuracy: 1.0000
Epoch 7/30
10/10 [==============================] - 2s 229ms/step - loss: 2.0914e
-04 - accuracy: 1.0000
Epoch 8/30
10/10 [==============================] - 3s 253ms/step - loss: 6.2939e
-05 - accuracy: 1.0000
Epoch 9/30
10/10 [==============================] - 2s 235ms/step - loss: 8.1611e
-05 - accuracy: 1.0000
Epoch 10/30
10/10 [==============================] - 3s 250ms/step - loss: 1.1389e
-04 - accuracy: 1.0000
Epoch 11/30
10/10 [==============================] - 3s 268ms/step - loss: 9.8356e
-05 - accuracy: 1.0000
Epoch 12/30
10/10 [==============================] - 2s 214ms/step - loss: 4.4802e
-05 - accuracy: 1.0000
Epoch 13/30
10/10 [==============================] - 2s 202ms/step - loss: 5.7308e
-05 - accuracy: 1.0000
Epoch 14/30
10/10 [==============================] - 2s 214ms/step - loss: 3.9151e
-05 - accuracy: 1.0000
Epoch 15/30
10/10 [==============================] - 2s 248ms/step - loss: 4.5719e
-05 - accuracy: 1.0000
Epoch 16/30
10/10 [==============================] - 2s 247ms/step - loss: 1.2158e
-04 - accuracy: 1.0000
Epoch 17/30
10/10 [==============================] - 2s 249ms/step - loss: 4.2672e
-05 - accuracy: 1.0000
Epoch 18/30
10/10 [==============================] - 2s 222ms/step - loss: 6.5367e
-05 - accuracy: 1.0000
Epoch 19/30
10/10 [==============================] - 2s 226ms/step - loss: 3.7332e
-05 - accuracy: 1.0000
```

```
Epoch 20/30
10/10 [==============================] - 2s 210ms/step - loss: 5.1064e
-05 - accuracy: 1.0000
Epoch 21/30
10/10 [==============================] - 2s 198ms/step - loss: 7.5069e
-05 - accuracy: 1.0000
Epoch 22/30
10/10 [==============================] - 2s 209ms/step - loss: 2.9340e
-05 - accuracy: 1.0000
Epoch 23/30
10/10 [==============================] - 2s 198ms/step - loss: 2.5805e
-05 - accuracy: 1.0000
Epoch 24/30
10/10 [==============================] - 2s 210ms/step - loss: 7.3019e
-05 - accuracy: 1.0000
Epoch 25/30
10/10 [==============================] - 2s 224ms/step - loss: 5.8883e
-05 - accuracy: 1.0000
Epoch 26/30
10/10 [==============================] - 2s 231ms/step - loss: 3.5969e
-05 - accuracy: 1.0000
Epoch 27/30
10/10 [==============================] - 2s 229ms/step - loss: 5.0213e
-05 - accuracy: 1.0000
Epoch 28/30
10/10 [==============================] - 2s 213ms/step - loss: 5.3858e
-05 - accuracy: 1.0000
Epoch 29/30
10/10 [==============================] - 2s 233ms/step - loss: 2.9400e
-05 - accuracy: 1.0000
Epoch 30/30
10/10 [==============================] - 2s 241ms/step - loss: 2.0359e
-05 - accuracy: 1.0000
```

In [20]:

```
mod4 = fit_gen(classifier, 30, 10)
```

```
Epoch 1/10
30/30 [==============================] - 7s 228ms/step - loss: 3.5582e
-05 - accuracy: 1.0000
Epoch 2/10
30/30 [==============================] - 7s 230ms/step - loss: 2.0062e
-05 - accuracy: 1.0000
Epoch 3/10
30/30 [==============================] - 7s 223ms/step - loss: 3.1618e
-05 - accuracy: 1.0000
Epoch 4/10
30/30 [==============================] - 7s 230ms/step - loss: 3.0374e
-05 - accuracy: 1.0000
Epoch 5/10
30/30 [==============================] - 7s 225ms/step - loss: 2.2273e
-05 - accuracy: 1.0000
Epoch 6/10
30/30 [==============================] - 7s 217ms/step - loss: 3.5320e
-05 - accuracy: 1.0000
Epoch 7/10
30/30 [==============================] - 7s 219ms/step - loss: 1.6409e
-05 - accuracy: 1.0000
Epoch 8/10
30/30 [==============================] - 7s 219ms/step - loss: 2.0606e
-05 - accuracy: 1.0000
Epoch 9/10
30/30 [==============================] - 7s 220ms/step - loss: 2.3477e
-05 - accuracy: 1.0000
Epoch 10/10
30/30 [==============================] - 6s 214ms/step - loss: 1.6656e
-05 - accuracy: 1.0000
```

In [21]:

```
mod5 = fit_gen(classifier, 30, 20)
```

```
Epoch 1/20
30/30 [==============================] - 7s 221ms/step - loss: 1.4814e
-05 - accuracy: 1.0000
Epoch 2/20
30/30 [==============================] - 7s 221ms/step - loss: 1.7685e
-05 - accuracy: 1.0000
Epoch 3/20
30/30 [==============================] - 7s 225ms/step - loss: 1.6668e
-05 - accuracy: 1.0000
Epoch 4/20
30/30 [==============================] - 7s 223ms/step - loss: 2.4128e
-05 - accuracy: 1.0000
Epoch 5/20
30/30 [==============================] - 6s 213ms/step - loss: 1.2991e
-05 - accuracy: 1.0000
Epoch 6/20
30/30 [==============================] - 7s 220ms/step - loss: 2.4361e
-05 - accuracy: 1.0000
Epoch 7/20
30/30 [==============================] - 7s 218ms/step - loss: 1.5190e
-05 - accuracy: 1.0000
Epoch 8/20
30/30 [==============================] - 7s 221ms/step - loss: 1.1301e
-05 - accuracy: 1.0000
Epoch 9/20
30/30 [==============================] - 7s 217ms/step - loss: 8.9801e
-06 - accuracy: 1.0000
Epoch 10/20
30/30 [==============================] - 7s 219ms/step - loss: 9.2447e
-06 - accuracy: 1.0000
Epoch 11/20
30/30 [==============================] - 7s 224ms/step - loss: 7.2097e
-06 - accuracy: 1.0000
Epoch 12/20
30/30 [==============================] - 7s 217ms/step - loss: 2.6104e
-05 - accuracy: 1.0000
Epoch 13/20
30/30 [==============================] - 6s 214ms/step - loss: 1.6678e
-05 - accuracy: 1.0000
Epoch 14/20
30/30 [==============================] - 7s 219ms/step - loss: 9.4415e
-06 - accuracy: 1.0000
Epoch 15/20
30/30 [==============================] - 7s 227ms/step - loss: 1.3661e
-05 - accuracy: 1.0000
Epoch 16/20
30/30 [==============================] - 7s 218ms/step - loss: 6.7877e
-06 - accuracy: 1.0000
Epoch 17/20
30/30 [==============================] - 7s 233ms/step - loss: 6.3362e
-06 - accuracy: 1.0000
Epoch 18/20
30/30 [==============================] - 7s 232ms/step - loss: 1.0614e
-05 - accuracy: 1.0000
Epoch 19/20
30/30 [==============================] - 7s 222ms/step - loss: 6.1763e
-06 - accuracy: 1.0000
```

```
Epoch 20/20
30/30 [==============================] - 7s 218ms/step - loss: 4.9600e
-06 - accuracy: 1.0000
```

In [22]:

```
mod6 = fit_gen(classifier, 30, 30)
```

```
Epoch 1/30
30/30 [==============================] - 7s 226ms/step - loss: 7.3888e
-06 - accuracy: 1.0000
Epoch 2/30
30/30 [==============================] - 7s 239ms/step - loss: 6.5597e
-06 - accuracy: 1.0000
Epoch 3/30
30/30 [==============================] - 7s 234ms/step - loss: 1.0710e
-05 - accuracy: 1.0000
Epoch 4/30
30/30 [==============================] - 7s 218ms/step - loss: 3.8526e
-06 - accuracy: 1.0000
Epoch 5/30
30/30 [==============================] - 7s 232ms/step - loss: 5.7177e
-06 - accuracy: 1.0000
Epoch 6/30
30/30 [==============================] - 7s 227ms/step - loss: 3.0311e
-06 - accuracy: 1.0000
Epoch 7/30
30/30 [==============================] - 6s 210ms/step - loss: 3.3013e
-06 - accuracy: 1.0000
Epoch 8/30
30/30 [==============================] - 7s 226ms/step - loss: 2.6586e
-06 - accuracy: 1.0000
Epoch 9/30
30/30 [==============================] - 7s 226ms/step - loss: 3.7232e
-06 - accuracy: 1.0000
Epoch 10/30
30/30 [==============================] - 7s 234ms/step - loss: 4.2270e
-06 - accuracy: 1.0000
Epoch 11/30
30/30 [==============================] - 7s 225ms/step - loss: 5.3649e
-06 - accuracy: 1.0000
Epoch 12/30
30/30 [==============================] - 7s 228ms/step - loss: 4.4423e
-06 - accuracy: 1.0000
Epoch 13/30
30/30 [==============================] - 7s 219ms/step - loss: 7.1878e
-06 - accuracy: 1.0000
Epoch 14/30
30/30 [==============================] - 7s 221ms/step - loss: 3.7346e
-06 - accuracy: 1.0000
Epoch 15/30
30/30 [==============================] - 6s 217ms/step - loss: 3.7167e
-06 - accuracy: 1.0000
Epoch 16/30
30/30 [==============================] - 6s 217ms/step - loss: 3.4968e
-06 - accuracy: 1.0000
Epoch 17/30
30/30 [==============================] - 7s 225ms/step - loss: 3.9759e
-06 - accuracy: 1.0000
Epoch 18/30
30/30 [==============================] - 6s 215ms/step - loss: 2.3889e
-06 - accuracy: 1.0000
Epoch 19/30
30/30 [==============================] - 6s 216ms/step - loss: 3.1286e
-06 - accuracy: 1.0000
```

```
Epoch 20/30
30/30 [==============================] - 7s 232ms/step - loss: 2.0494e
-06 - accuracy: 1.0000
Epoch 21/30
30/30 [==============================] - 7s 229ms/step - loss: 2.6431e
-06 - accuracy: 1.0000
Epoch 22/30
30/30 [==============================] - 7s 230ms/step - loss: 2.9569e
-06 - accuracy: 1.0000
Epoch 23/30
30/30 [==============================] - 7s 226ms/step - loss: 3.2812e
-06 - accuracy: 1.0000
Epoch 24/30
30/30 [==============================] - 7s 219ms/step - loss: 2.0847e
-06 - accuracy: 1.0000
Epoch 25/30
30/30 [==============================] - 6s 207ms/step - loss: 1.4299e
-06 - accuracy: 1.0000
Epoch 26/30
30/30 [==============================] - 6s 211ms/step - loss: 3.6356e
-06 - accuracy: 1.0000
Epoch 27/30
30/30 [==============================] - 6s 215ms/step - loss: 3.3085e
-06 - accuracy: 1.0000
Epoch 28/30
30/30 [==============================] - 6s 212ms/step - loss: 2.4021e
-06 - accuracy: 1.0000
Epoch 29/30
30/30 [==============================] - 7s 220ms/step - loss: 2.8901e
-06 - accuracy: 1.0000
Epoch 30/30
30/30 [==============================] - 7s 224ms/step - loss: 1.5800e
-06 - accuracy: 1.0000
```

In [23]:

```
mod7 = fit_gen(classifier, 50, 10)
```

```
Epoch 1/10
50/50 [==============================] - 12s 231ms/step - loss: 2.0730
e-06 - accuracy: 1.0000
Epoch 2/10
50/50 [==============================] - 11s 221ms/step - loss: 1.9959
e-06 - accuracy: 1.0000
Epoch 3/10
50/50 [==============================] - 11s 220ms/step - loss: 1.7614
e-06 - accuracy: 1.0000
Epoch 4/10
50/50 [==============================] - 11s 220ms/step - loss: 1.8586
e-06 - accuracy: 1.0000
Epoch 5/10
50/50 [==============================] - 11s 221ms/step - loss: 1.9674
e-06 - accuracy: 1.0000
Epoch 6/10
50/50 [==============================] - 11s 226ms/step - loss: 1.1028
e-06 - accuracy: 1.0000
Epoch 7/10
50/50 [==============================] - 11s 228ms/step - loss: 1.2493
e-06 - accuracy: 1.0000
Epoch 8/10
50/50 [==============================] - 11s 216ms/step - loss: 1.1934
e-06 - accuracy: 1.0000
Epoch 9/10
50/50 [==============================] - 11s 225ms/step - loss: 4.7192
e-06 - accuracy: 1.0000
Epoch 10/10
50/50 [==============================] - 11s 224ms/step - loss: 2.1672
e-06 - accuracy: 1.0000
```

In [24]:

```
mod8 = fit_gen(classifier, 50, 20)
```

```
Epoch 1/20
50/50 [==============================] - 12s 232ms/step - loss: 1.4267
e-06 - accuracy: 1.0000
Epoch 2/20
50/50 [==============================] - 11s 229ms/step - loss: 1.8330
e-06 - accuracy: 1.0000
Epoch 3/20
50/50 [==============================] - 11s 219ms/step - loss: 1.3528
e-06 - accuracy: 1.0000
Epoch 4/20
50/50 [==============================] - 10s 208ms/step - loss: 1.1391
e-06 - accuracy: 1.0000
Epoch 5/20
50/50 [==============================] - 10s 207ms/step - loss: 1.0450
e-06 - accuracy: 1.0000
Epoch 6/20
50/50 [==============================] - 10s 205ms/step - loss: 8.3772
e-07 - accuracy: 1.0000
Epoch 7/20
50/50 [==============================] - 10s 207ms/step - loss: 1.1138
e-06 - accuracy: 1.0000
Epoch 8/20
50/50 [==============================] - 10s 206ms/step - loss: 7.2383
e-07 - accuracy: 1.0000
Epoch 9/20
50/50 [==============================] - 10s 204ms/step - loss: 6.3499
e-07 - accuracy: 1.0000
Epoch 10/20
50/50 [==============================] - 10s 205ms/step - loss: 8.5991
e-07 - accuracy: 1.0000
Epoch 11/20
50/50 [==============================] - 10s 207ms/step - loss: 8.8973
e-07 - accuracy: 1.0000
Epoch 12/20
50/50 [==============================] - 10s 204ms/step - loss: 7.6997
e-07 - accuracy: 1.0000
Epoch 13/20
50/50 [==============================] - 10s 206ms/step - loss: 7.3311
e-07 - accuracy: 1.0000
Epoch 14/20
50/50 [==============================] - 10s 207ms/step - loss: 6.1388
e-07 - accuracy: 1.0000
Epoch 15/20
50/50 [==============================] - 10s 205ms/step - loss: 8.1938
e-07 - accuracy: 1.0000
Epoch 16/20
50/50 [==============================] - 10s 205ms/step - loss: 7.0267
e-07 - accuracy: 1.0000
Epoch 17/20
50/50 [==============================] - 10s 205ms/step - loss: 6.2061
e-07 - accuracy: 1.0000
Epoch 18/20
50/50 [==============================] - 10s 205ms/step - loss: 5.0481
e-07 - accuracy: 1.0000s - loss: 5
Epoch 19/20
50/50 [==============================] - 10s 206ms/step - loss: 7.0008
e-07 - accuracy: 1.0000
```

```
Epoch 20/20
50/50 [==============================] - 10s 206ms/step - loss: 6.5961
e-07 - accuracy: 1.0000
```

In [25]:

```
mod9 = fit_gen(classifier, 50, 30)
```

```
Epoch 1/30
50/50 [==============================] - 10s 206ms/step - loss: 4.7505
e-07 - accuracy: 1.0000
Epoch 2/30
50/50 [==============================] - 10s 205ms/step - loss: 7.1972
e-07 - accuracy: 1.0000
Epoch 3/30
50/50 [==============================] - 10s 204ms/step - loss: 5.7931
e-07 - accuracy: 1.0000
Epoch 4/30
50/50 [==============================] - 10s 206ms/step - loss: 4.6246
e-07 - accuracy: 1.0000
Epoch 5/30
50/50 [==============================] - 10s 204ms/step - loss: 6.3627
e-07 - accuracy: 1.0000
Epoch 6/30
50/50 [==============================] - 10s 206ms/step - loss: 6.2630
e-07 - accuracy: 1.0000
Epoch 7/30
50/50 [==============================] - 10s 205ms/step - loss: 3.5899
e-07 - accuracy: 1.0000
Epoch 8/30
50/50 [==============================] - 10s 205ms/step - loss: 6.0903
e-07 - accuracy: 1.0000
Epoch 9/30
50/50 [==============================] - 10s 206ms/step - loss: 3.6952
e-07 - accuracy: 1.0000
Epoch 10/30
50/50 [==============================] - 10s 206ms/step - loss: 3.0183
e-07 - accuracy: 1.0000
Epoch 11/30
50/50 [==============================] - 10s 204ms/step - loss: 5.3983
e-07 - accuracy: 1.0000
Epoch 12/30
50/50 [==============================] - 10s 205ms/step - loss: 7.0563
e-07 - accuracy: 1.0000
Epoch 13/30
50/50 [==============================] - 10s 208ms/step - loss: 4.6708
e-07 - accuracy: 1.0000
Epoch 14/30
50/50 [==============================] - 10s 210ms/step - loss: 4.4983
e-07 - accuracy: 1.0000
Epoch 15/30
50/50 [==============================] - 11s 210ms/step - loss: 5.9046
e-07 - accuracy: 1.0000
Epoch 16/30
50/50 [==============================] - 10s 209ms/step - loss: 3.3683
e-07 - accuracy: 1.0000
Epoch 17/30
50/50 [==============================] - 11s 211ms/step - loss: 4.0399
e-07 - accuracy: 1.0000
Epoch 18/30
50/50 [==============================] - 11s 210ms/step - loss: 3.2145
e-07 - accuracy: 1.0000
Epoch 19/30
50/50 [==============================] - 11s 212ms/step - loss: 2.7294
e-07 - accuracy: 1.0000
```

```
Epoch 20/30
50/50 [==============================] - 11s 215ms/step - loss: 3.0314
e-07 - accuracy: 1.0000
Epoch 21/30
50/50 [==============================] - 10s 210ms/step - loss: 2.0765
e-07 - accuracy: 1.0000
Epoch 22/30
50/50 [==============================] - 11s 211ms/step - loss: 4.1007
e-07 - accuracy: 1.0000
Epoch 23/30
50/50 [==============================] - 11s 213ms/step - loss: 2.5035
e-07 - accuracy: 1.0000
Epoch 24/30
50/50 [==============================] - 10s 210ms/step - loss: 2.9815
e-07 - accuracy: 1.0000
Epoch 25/30
50/50 [==============================] - 11s 211ms/step - loss: 3.1048
e-07 - accuracy: 1.0000
Epoch 26/30
50/50 [==============================] - 11s 212ms/step - loss: 2.9718
e-07 - accuracy: 1.0000
Epoch 27/30
50/50 [==============================] - 10s 209ms/step - loss: 2.2961
e-07 - accuracy: 1.0000
Epoch 28/30
50/50 [==============================] - 11s 212ms/step - loss: 2.8186
e-07 - accuracy: 1.0000
Epoch 29/30
50/50 [==============================] - 11s 212ms/step - loss: 2.0660
e-07 - accuracy: 1.0000
Epoch 30/30
50/50 [==============================] - 11s 211ms/step - loss: 2.3164
e-07 - accuracy: 1.0000
```

In [26]:

```
mod10 = fit_gen(classifier, 50, 100)
```

```
Epoch 1/100
50/50 [==============================] - 11s 210ms/step - loss: 2.0080
e-07 - accuracy: 1.0000
Epoch 2/100
50/50 [==============================] - 11s 212ms/step - loss: 2.6425
e-07 - accuracy: 1.0000
Epoch 3/100
50/50 [==============================] - 10s 208ms/step - loss: 1.9127
e-07 - accuracy: 1.0000
Epoch 4/100
50/50 [==============================] - 11s 211ms/step - loss: 3.6142
e-07 - accuracy: 1.0000
Epoch 5/100
50/50 [==============================] - 11s 212ms/step - loss: 1.8565
e-07 - accuracy: 1.0000
Epoch 6/100
50/50 [==============================] - 10s 210ms/step - loss: 2.6103
e-07 - accuracy: 1.0000
Epoch 7/100
50/50 [==============================] - 11s 213ms/step - loss: 1.6177
```

In [27]:

```
# Write the models and model weights to disk

model2_yaml = mod2.to_yaml()
with open('model_2.yaml', 'w') as yaml_file:
    yaml_file.write(model2_yaml)
mod2.save_weights('model_2.h5')
print('Saved model2 to disk')

model3_yaml = mod3.to_yaml()
with open('model_3.yaml', 'w') as yaml_file:
    yaml_file.write(model3_yaml)
mod3.save_weights('model_3.h5')
print('Saved model3 to disk')

model4_yaml = mod4.to_yaml()
with open('model_4.yaml', 'w') as yaml_file:
    yaml_file.write(model4_yaml)
mod4.save_weights('model_4.h5')
print('Saved model4 to disk')

model5_yaml = mod5.to_yaml()
with open('model_5.yaml', 'w') as yaml_file:
    yaml_file.write(model5_yaml)
mod5.save_weights('model_5.h5')
print('Saved model5 to disk')

model6_yaml = mod6.to_yaml()
with open('model_6.yaml', 'w') as yaml_file:
    yaml_file.write(model6_yaml)
mod6.save_weights('model_6.h5')
print('Saved model6 to disk')

model7_yaml = mod7.to_yaml()
with open('model_7.yaml', 'w') as yaml_file:
    yaml_file.write(model7_yaml)
mod7.save_weights('model_7.h5')
print('Saved model7 to disk')

model8_yaml = mod8.to_yaml()
with open('model_8.yaml', 'w') as yaml_file:
    yaml_file.write(model8_yaml)
mod8.save_weights('model_8.h5')
print('Saved model8 to disk')

model9_yaml = mod9.to_yaml()
with open('model_9.yaml', 'w') as yaml_file:
    yaml_file.write(model2_yaml)
mod9.save_weights('model_9.h5')
print('Saved model9 to disk')

model10_yaml = mod10.to_yaml()
with open('model_10.yaml', 'w') as yaml_file:
    yaml_file.write(model10_yaml)
mod10.save_weights('model_10.h5')
print('Saved model10 to disk')
```

```
Saved model2 to disk
Saved model3 to disk
Saved model4 to disk
```

```
Saved model14 to disk
Saved model15 to disk
Saved model16 to disk
Saved model17 to disk
Saved model18 to disk
Saved model19 to disk
Saved model110 to disk
```

In [28]:

```
def pred_result(model):
    results = []
    for f1 in files:
        img = load_img(f1, target_size = (64, 64))
        img = img_to_array(img)
        img = np.expand_dims(img, axis = 0)
        data.append(img)
        result = model.predict(img)
        r = np.argmax(result, axis=1)
        results.append(r)
    return results
```

In [30]:

```
r2 = pred_result(mod2)
r3 = pred_result(mod3)
r4 = pred_result(mod4)
r5 = pred_result(mod5)
r6 = pred_result(mod6)
r7 = pred_result(mod7)
r8 = pred_result(mod8)
r9 = pred_result(mod9)
r10 = pred_result(mod10)
```

In [31]:

```
# Determine accuracy
acc2 = metrics.accuracy_score(test_labels, r2)
acc3 = metrics.accuracy_score(test_labels, r3)
acc4 = metrics.accuracy_score(test_labels, r4)
acc5 = metrics.accuracy_score(test_labels, r5)
acc6 = metrics.accuracy_score(test_labels, r6)
acc7 = metrics.accuracy_score(test_labels, r7)
acc8 = metrics.accuracy_score(test_labels, r8)
acc9 = metrics.accuracy_score(test_labels, r9)
acc10 = metrics.accuracy_score(test_labels, r10)
```

f) Create a final dataframe that combines the accuracy across each combination.

In [32]:

```
compare = pd.DataFrame({
    'Steps per Epoch': [10,10,10,30,30,30,50,50,50,50],
    'Epochs': [10,20,30,10,20,30,10,20,30,100],
    'Accuracy': [acc, acc2, acc3, acc4, acc5, acc6, acc7, acc8, acc9, acc10]})

compare
```

Out[32]:

	Steps per Epoch	Epochs	Accuracy
0	10	10	0.75
1	10	20	0.75
2	10	30	0.75
3	30	10	0.75
4	30	20	0.75
5	30	30	0.75
6	50	10	0.75
7	50	20	0.75
8	50	30	0.75
9	50	100	0.75

Part IV. Conceptual Questions

4. Discuss the effect of the following on accuracy and loss (train & test):
    - Increasing the steps_per_epoch
    - Increasing the number of epochs

Steps per epoch is the number of batch iterations before a training epoch is considered finished. The model's weights get one update through gradient descent after each iteration. As our output suggests, increasing the steps_per_epoch would result in more computation due to the weight-update and thus increase the training time for each epoch. The more iterations, the more weight update and the closer the loss function is to being minimized. Therefore, it would increase the accuracy on the training data. However, given the bias-variance tradeoff, decrease in bias comes at the expense of increasing variance, and this could potentially lead to overfitting on the training data.

An epoch is a measure of the number of times all of the training vectors are used once to update the weights. In neural network, an epoch is completed when an entire dataset is passed both forward and backward through the network once. A visualization is shown below that shows the relationship between the train and generalization error and the number of epochs. As the number of epochs increases, the accuracy of the training set improves and the error continues to decrease. However, for the validation set, the error reaches the lowest at a certain epoch, after which the error increases again. This is because the model is starting to overfit the training data and is not generalizable to unseen data. Therefore, in practice we can employ early stopping and stop training when the validation error reaches the minimum to prevent the model from overfitting.

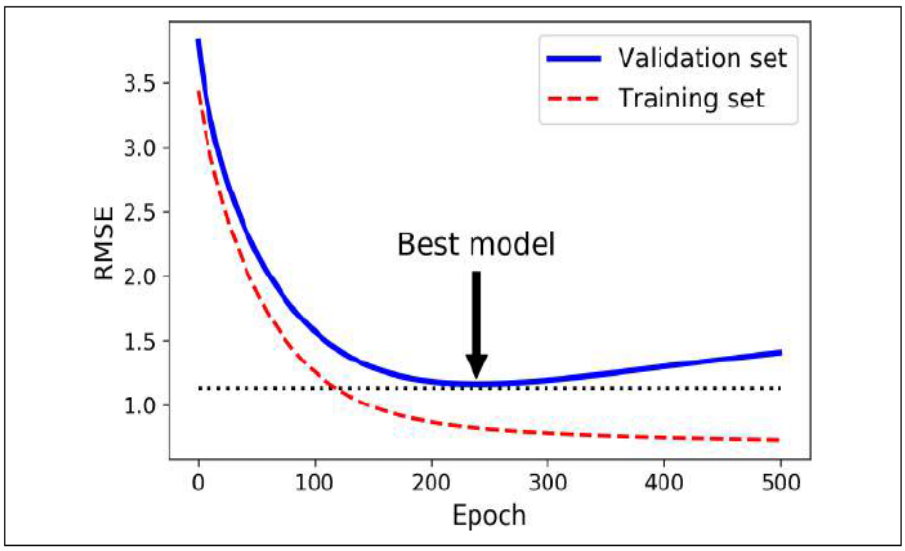


Figure 4-20. Early stopping regularization

5. Name two uses of zero padding in CNN.

Padding works by extending the area of which a convolutional neural network processes an image. In order for a later to have the same dimension as the previous layer, it's common to add zeros around the image, which is referred to as zero padding. There are a couple of uses of zero padding:

- Padding enables us to control the spatial size of the output volumes. The spatial size of the output volume is given by (W−F+2P)/S + 1, where W represents the input volume size (W), F the receptive field size of the Conv Layer neurons, S the stride with which they are applied, and P the amount of zero padding used on the border. Let's plug in some numbers to better illustrate the idea. Say we have W = 5, F = 3, S = 1, without zero padding, the output volume would be (5-3+0)/1 + 1 = 3, which is smaller than the input volume (i.e. we have a reduction in volume size). However, with zero padding, we can control the spatial size of the output volume and prevent the reduction in volume from happening too quickly. This also is the key to allowing us to build deeper networks.
- Padding allows for more accurate analysis of the images by retaining information at the borders. In CNN, the kernel is the neural networks filter which moves across the image, scanning each pixel and converting the data into a smaller, or sometimes larger, format. In order to assist the kernel with processing the image, padding is added to the frame of the image to allow for more space for the kernel to cover the image. As shown in the picture below, without padding, the pixels in the corner of the input would get ignored, while the pixels in the center could repeated contribute to the output. When several no-padding layers get stacked together, as the size the the volumes reduces by a small amount after each convolution layer, the boarder pixels would be "washed away" too quickly. By implementing zero padding, the pixels at the border wouldn't be lost and the model can achieve better performance.

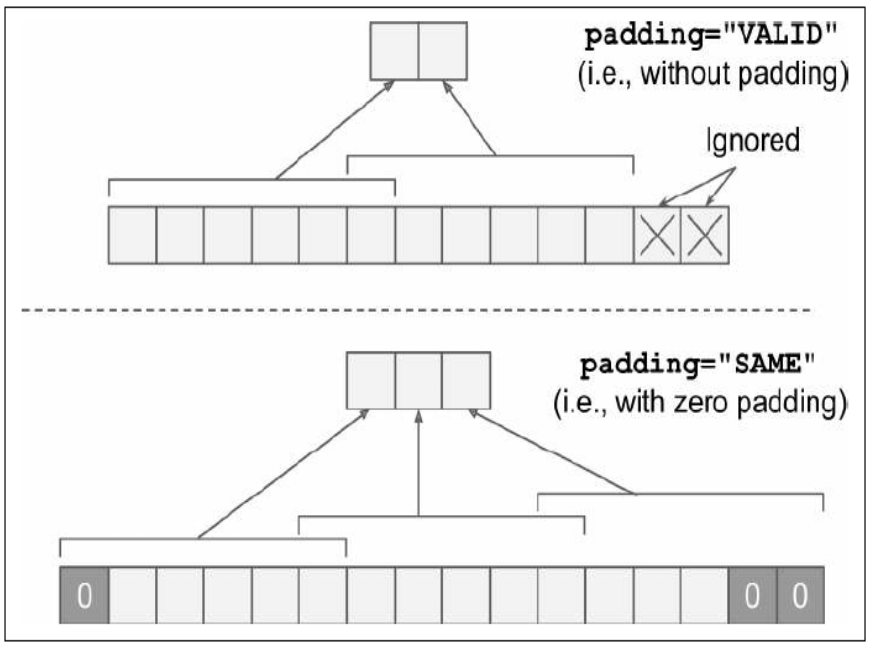
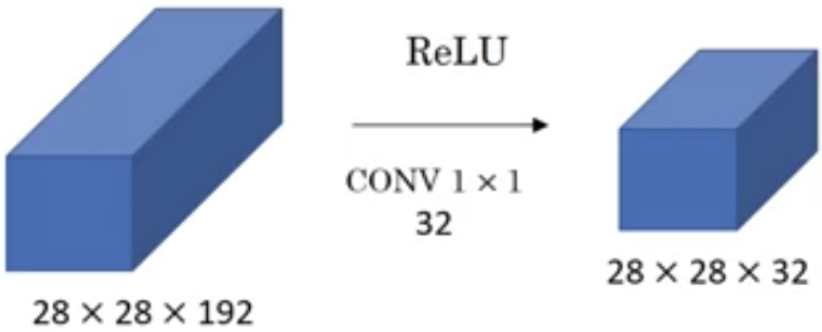


Figure 14-7. Padding options—input width: 13, filter width: 6, stride: 5

6. What is the use of a 1 x 1 kernel in CNN?

A convolutional layer with a 1×1 filter can be used at any point in a convolutional neural network to control the number of feature maps. While a simple 1×1 filter provides a way to usefully summarize the input feature maps, the use of multiple 1×1 filters allows the tuning of the number of summaries of the input feature maps, effectively allowing the depth of the feature maps to be increased or decreased as needed.

The most common application of a 1×1 kernel in CNN is dimension reduction. Using the graphical illustration below, we start with a input volume of 2828192. We can use the pooling layer to shrink the height and width, but to shrink the number of channels to 32, we have to use 32 1×1 filters (each filter has volume of 11192). 1×1 filters act like coordinate-dependent transformation in the filter space, which is stricly linear, then it is usually succeeded by a non-linear activation layer like ReLu. By shrinking on the number of channels, we can improve save on computation and increase modeling efficiency.



7. What are the advantages of a CNN over a fully connected DNN for this image classification problem?

There are several advantages of a CNN over a fully connected DNN for image classification problem:

- Increased efficiency due to reduction in the number of parameters: One main feature of CNN is weight sharing. In CNN, all neurons in a feature map share the same parameters, drastically reducing the number of parameters needed in the model. Say for example that we have one-layer CNN with 10 filters of size

5x5. We can simply calculate parameters of such a CNN, which is the sum of 5510 weights and 10 biases, i.e 5510 + 10 = 260 parameters. Now let's take a simple one layered NN with 250 neurons, here the number of weight parameters depending on the size of images is ‘250 x K’ where size of the image is P X M and K = (P x M). Additionally, you need ‘M’ biases. Adding these up can lead to a very large number of paramaters. Therefore CNN is more efficient in terms of memory and complexity. On a large image dataset where billions of neurons are needed, a CNN would be less complex and memory-efficient compared to a fully connected DNN.

- Agility in pattern recognition due to CNN being location invariant: once the CNN has leraned to recognize a pattern in one location, it can recognize it in any other location, which is what we call the feature extraction property of CNN. In contrast, when a DNN learned to recognize a pattern in one location, it can only recognize it in that particular location.

In [ ]: