COMP 576 Final Report

Classification of Dogs and Cats

Wuhao Wang (ww43)

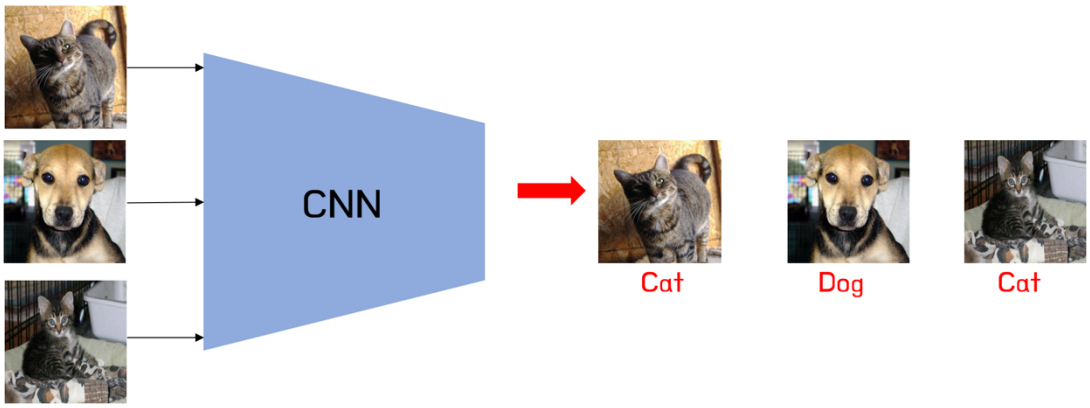
Github link: https://github.com/wuhao-git/COMP-576-Final-Project

December 12, 2022

1 Introduction

1.1 Background Introduction

Who doesn't like animals, right? Personally, if I could put them in all of my projects, I'd be living the life. In this report, I'm going to perform how to take a dataset full of lovely animal images, do some black magic, and end up with a model that can classify even your personal pet pictures. This model can then be used in your personal applications so that you can show your friends and family the cool stuff you've been learning.



1.2 Dataset

Dataset is released in Kaggle. Original dataset has 12500 images of dogs and 12500 images of cats, in 25000 images in total. That's a huge amount to train the model. But in our case, we just only use 1000 images for training, 500 images for validation, and 1000 images for test.

Actually, 1000 images are not enough datasets for training. But mentioned earlier, we already learn about how to size-up the dataset with transformation. Yes, it is data augmentation. There are several techniques to transform the image. In this case, we will use following transformations:

Random Crop: from original image, we just choose random size of bounding box and crop it.

Random Rotation: We can rotate the original image with random angle.

flip_left_right: We can imagine the transformation with mirrors that flips left to right.

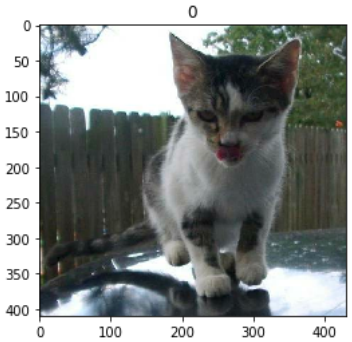
Of course, model input must be the same size. So after data augmentation, we need to resize our transformed image to fixed size. In this case, we choose 150x150 for the input image.

At first, we implement image load method.

```
def load(f, label):
    # load the file into tensor
    image = tf.io.read_file(f)
    # Decode it to JPEG format
    image = tf.image.decode_jpeg(image)
    # Convert it to tf.float32
    image = tf.cast(image, tf.float32)

    return image, label
```

So let's test it for checking functionality.

```
image, label = load('./dataset/my_cat_dog/train/cat/cat.11.jpg', 0)

fig, ax = plt.subplots()
ax.imshow(image /255.)
ax.set_title(label)
plt.show()
```



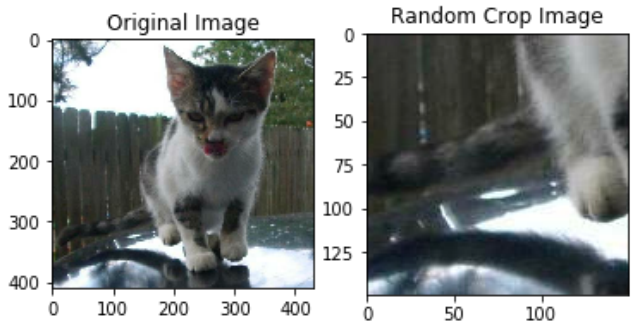
And it is required to fix the input format. For this purpose, we need to implement resize function. Tensorflow has image class(tf.image) to handle the image processing in advance. we can use it. Note that size argument must have an order of [height, width].

```
def resize(input_image, size):
    return tf.image.resize(input_image, size)
```

So this is a sample image of Cat, which has label of '0'(Cats). Then we will implement random crop function. Actually, Tensorflow already contains random_crop API for convenience.

```
def random_crop(input_image):
    return tf.image.random_crop(input_image, size=[150, 150, 3])
```

```
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image / 255.)
ax[0].set_title("Original Image")

ax[1].imshow(random_crop(image) / 255.)
ax[1].set_title("Random Crop Image")
plt.show()
```

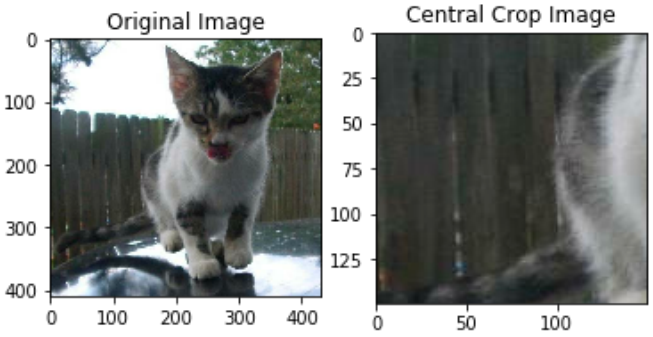


Because validation and test processs don't affect the model training, it just measure the accuracy. So we don't need to data augmentation process in validation and test data. And sometimes random crop may crop the useless section of image that cannot classify correctly. In that case, central_crop function is required, not random crop. So We implement it using tensorflow.

From documentation, it needs to define central_fraction as an argument. It means that this API crops from the center point based on the fraction. Our purpose is to made an input data with 150x150x3. But the size of each image may be different. So we need to resize it in advance.

```
def central_crop(input_image):
    image = resize(input_image, [176, 176])
    return tf.image.central_crop(image, central_fraction=0.84)
```

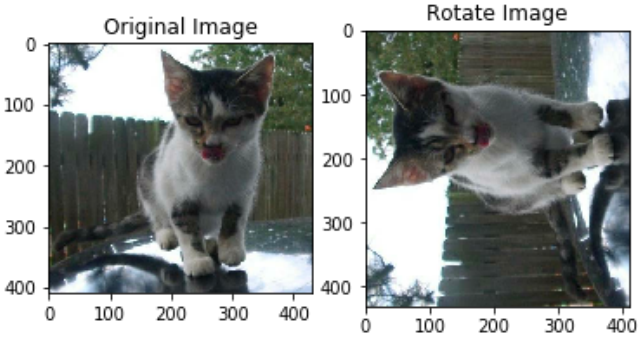
```
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image / 255.)
ax[0].set_title("Original Image")

ax[1].imshow(random_crop(image) / 255.)
ax[1].set_title("Central Crop Image")
plt.show()
```



Then we can implement random_rotation API.

```
def random_rotation(input_image):
    angles = np.random.randint(0, 3, 1)
    return tf.image.rot90(input_image, k=angles[0])
```

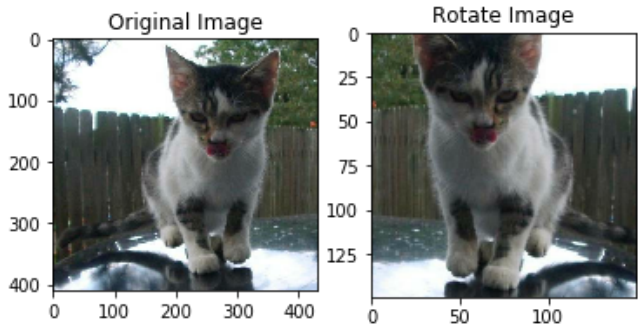
```
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image / 255.)
ax[0].set_title("Original Image")

ax[1].imshow(random_rotation(image) / 255.)
ax[1].set_title("Rotate Image")
plt.show()
```



Once we define several helper functions for data augmentation, we can merge it in one API, called random_jitter. And we can add another image transformation function, like random_flip_left_right

```
def random_jitter(input_image):
    # Resize it to 176 x 176 x 3
    image = resize(input_image, [176, 176])
    # Randomly Crop to 150 x 150 x 3
    image = random_crop(image)
    # Randomly rotation
    image = random_rotation(image)
    # Randomly mirroring
    image = tf.image.random_flip_left_right(image)
    return image
```

```
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image / 255.)
ax[0].set_title("Original Image")

ax[1].imshow(random_jitter(image) / 255.)
ax[1].set_title("Rotate Image")
plt.show()
```



One more API we need to implement is normalize. Normalization is one of method for rescaling. There are several techniques for normalization. But in this API, our normalize function will be

convert the value range from [0, 255] to [0, 2]

move the value range from [0, 2] to [-1, 1]

After that, whole value in image will be in range of [-1, 1]

```
# normalizing the images to [-1, 1]
def normalize(input_image):
    mid = (tf.reduce_max(input_image) + tf.reduce_min(input_image)) / 2
    input_image = input_image / mid - 1
    return input_image
```

And it will be helpful to make train data and validation data in single API.

```
def load_image_train(image_file, label):
    image, label = load(image_file, label)
    image = random_jitter(image)
    image = normalize(image)
    return image, label

def load_image_val(image_file, label):
    image, label = load(image_file, label)
    image = central_crop(image)
    image = normalize(image)
    return image, label
```

1.21 Data Pipeline

Usually dataset pipeline is built for training and test dataset. Actually, it is very efficiency for memory usage, because its type is python generator.

Our data is already separated in each speicies through folder.

- dataset
  - train
    - cat
    - dog
  - test
    - cat
    - dog
  - val
    - cat
    - dog

So we need to extract folder name as an label and add it into the data pipeline. So we are doing as follows:

- Build temp_ds from cat images (usually have *.jpg)
- Add label (0) in train_ds
- Build temp_ds from dog images (usually have *.jpg)
- Add label (1) in temp_ds
- Merge two datasets into one

```
temp_ds = tf.data.Dataset.list_files(os.path.join('./dataset/my_cat_dog', 'train', 'cat', '*.jpg'))
temp_ds = temp_ds.map(lambda x: (x, 0))

temp2_ds = tf.data.Dataset.list_files(os.path.join('./dataset/my_cat_dog', 'train', 'dog', '*.jpg'))
temp2_ds = temp2_ds.map(lambda x: (x, 1))

train_ds = temp_ds.concatenate(temp2_ds)
```

Then, we can make it suffle or split it with batch size and so on.

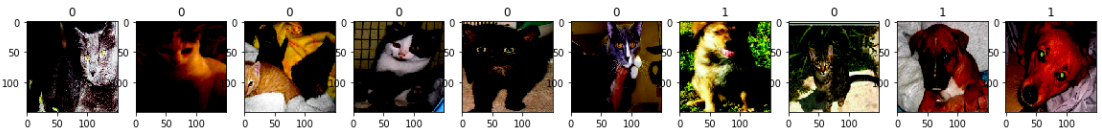
```
buffer_size = tf.data.experimental.cardinality(train_ds).numpy()
train_ds = train_ds.shuffle(buffer_size)\
                   .map(load_image_train, num_parallel_calls=16)\
                   .batch(20)\
                   .repeat()
```

Same in Validation and Test set, we will make each dataset pipeline through same process. But mentioned before, we don't need to apply data augmentation for these dataset.

```
temp_ds = tf.data.Dataset.list_files(os.path.join('./dataset/my_cat_dog', 'val', 'cat', '*.jpg'))
temp_ds = temp_ds.map(lambda x: (x, 0))

temp2_ds = tf.data.Dataset.list_files(os.path.join('./dataset/my_cat_dog', 'val', 'dog', '*.jpg'))
temp2_ds = temp2_ds.map(lambda x: (x, 1))

val_ds = temp_ds.concatenate(temp2_ds)

val_ds = val_ds.map(load_image_val, num_parallel_calls=16)\
               .batch(20)\
               .repeat()
```

```
temp_ds = tf.data.Dataset.list_files(os.path.join('./dataset/my_cat_dog', 'test', 'cat', '*.jpg'))
temp_ds = temp_ds.map(lambda x: (x, 0))

temp2_ds = tf.data.Dataset.list_files(os.path.join('./dataset/my_cat_dog', 'test', 'dog', '*.jpg'))
temp2_ds = temp2_ds.map(lambda x: (x, 1))

test_ds = temp_ds.concatenate(temp2_ds)

test_ds = test_ds.map(load_image_val, num_parallel_calls=16)\
                 .shuffle(buffer_size)\
                 .batch(20)\
                 .repeat()
```

We built the datapipe line for the training, validation and test. Now, let's chcek whether it is correct or not.



1.3 Models

Here we will build the CNN classifier. Unlike general Convolution Layer, we will define custom Convolution Layer class with Batch normalization.

When we use Batch normalization, we need to define whether it is used in training mode or not. Because Batch normalization is one of approaches to help training easily, but in test/validation mode, weight may not be updated. At that case, training argument must be False.

```
class Conv(tf.keras.Model):
    def __init__(self, filters, kernel_size):
        super(Conv, self).__init__()

        self.conv = tf.keras.layers.Conv2D(filters=filters, kernel_size=kernel_size)
        self.bn = tf.keras.layers.BatchNormalization()
        self.relu = tf.keras.layers.ReLU()
        self.pool = tf.keras.layers.MaxPool2D(pool_size=(2, 2))

    def call(self, inputs, training=True):
        x = self.conv(inputs)
        x = self.bn(x, training=training)
        x = self.relu(x)
        x = self.pool(x)
        return x
```

Using this class, we implement CNN model with Sequential API. And output node will be 2 since our classifier can classify two labels: cat and dog (or 0 and 1).

```
model = tf.keras.Sequential(name='Cat_Dog_CNN')

model.add(Conv(filters=32, kernel_size=(3, 3)))
model.add(Conv(filters=64, kernel_size=(3, 3)))
model.add(Conv(filters=128, kernel_size=(3, 3)))
model.add(Conv(filters=128, kernel_size=(3, 3)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
model.add(tf.keras.layers.Dense(units=2, activation=tf.keras.activations.softmax))
```

```
model(images[:1])
```

```
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[0.48711136, 0.5128886 ]], dtype=float32)>
```

1.3.1 Model Checkpoint

We used ModelCheckPoint for saving weight of model. Through this, we can save the weight that trained model can perform best accuracy. Or we can load the best model to enhance the performance.

```
checkpoint_path = "./train/cat_dog_cnn/cp-{epoch:04d}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_best_only=True,
                                                 save_weights_only=True,
                                                 verbose=1)
```

1.3.2 Compile Model

To use the model for training, it is required to define the optimizer and loss function.

Adaptive Momentum estimation (Adam for short) is widely used optimizer to find the optimal solution for minimum loss. There needs to be defined learning_rate for step_size. In this case, we use 1e-4 (or 0.00004) as a learning_rate.

And There are many loss function for classification. Maybe someone confuses about what kind of loss function that can we choose from SparseCategoricalCrossentropy or Cross Entropy Loss. Cross Entropy Loss is used for classification when there are two or more label classes. But mentioned in documentation, if the label is provided as an integer (not float or whatever), we need to use SparseCategoricalCrossentropy.

The last argument we need to define is metrics. Metrics is the numerical result to check the performance. Our case is classification, and the performance of classification can be measured from comparing predicted label and ground truth label. So the metrics we need to define is accuracy.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
model.summary()
```

```
Model: "Cat_Dog_CNN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv (Conv)                  multiple                  1024
_________________________________________________________________
conv_1 (Conv)                multiple                  18752
_________________________________________________________________
conv_2 (Conv)                multiple                  74368
_________________________________________________________________
conv_3 (Conv)                multiple                  148096
_________________________________________________________________
flatten (Flatten)            multiple                  0
_________________________________________________________________
dense (Dense)                multiple                  3211776
_________________________________________________________________
dense_1 (Dense)              multiple                  1026
=================================================================
Total params: 3,455,042
Trainable params: 3,454,338
Non-trainable params: 704
_________________________________________________________________
```

Finally, we can extract the summary of our CNN model that have almost 3.5 million parameters.

1.3.3 Train the model

We can train our model with .fit() method. And also we need to use checkpoint callback that we defined earlier.

```
train_len = len(glob(os.path.join('./dataset/my_cat_dog', 'train', 'cat', '*.jpg'))) * 2
val_len = len(glob(os.path.join('./dataset/my_cat_dog', 'val', 'cat', '*.jpg'))) * 2
test_len = len(glob(os.path.join('./dataset/my_cat_dog', 'test', 'cat', '*.jpg'))) * 2
```

```
model.fit(train_ds, steps_per_epoch=train_len / 20,
          validation_data=val_ds,
          validation_steps=val_len/20,
          epochs=30,
          callbacks=[cp_callback]
          )
```

```
Epoch 1/30
 97/100 [============================>.] - ETA: 0s - loss: 0.8350 - accuracy: 0.5840
Epoch 00001: val_loss improved from inf to 0.79115, saving model to ./train/cat_dog_cnn/cp-0001.ckpt
100/100 [==============================] - 2s 24ms/step - loss: 0.8274 - accuracy: 0.5870 -
val_loss: 0.7912 - val_accuracy: 0.5000
Epoch 2/30
 97/100 [============================>.] - ETA: 0s - loss: 0.6601 - accuracy: 0.6624
Epoch 00002: val_loss improved from 0.79115 to 0.75663, saving model to ./train/cat_dog_cnn/cp-
0002.ckpt
100/100 [==============================] - 2s 23ms/step - loss: 0.6586 - accuracy: 0.6615 -
val_loss: 0.7566 - val_accuracy: 0.5060
Epoch 3/30
100/100 [==============================] - ETA: 0s - loss: 0.5615 - accuracy: 0.7130
Epoch 00003: val_loss improved from 0.75663 to 0.73090, saving model to ./train/cat_dog_cnn/cp-
0003.ckpt
100/100 [==============================] - 2s 22ms/step - loss: 0.5615 - accuracy: 0.7130 -
val_loss: 0.7309 - val_accuracy: 0.5750
Epoch 4/30
100/100 [==============================] - ETA: 0s - loss: 0.5408 - accuracy: 0.7335
Epoch 00004: val_loss improved from 0.73090 to 0.57448, saving model to ./train/cat_dog_cnn/cp-
0004.ckpt
100/100 [==============================] - 2s 23ms/step - loss: 0.5408 - accuracy: 0.7335 -
val_loss: 0.5745 - val_accuracy: 0.7060
Epoch 5/30
100/100 [==============================] - ETA: 0s - loss: 0.4738 - accuracy: 0.7655
Epoch 00005: val_loss did not improve from 0.57448
```

```
100/100 [==============================] - 2s 20ms/step - loss: 0.4738 - accuracy: 0.7655 -
val_loss: 0.5784 - val_accuracy: 0.6970
Epoch 6/30
 99/100 [============================>.] - ETA: 0s - loss: 0.4863 - accuracy: 0.7682
Epoch 00006: val_loss improved from 0.57448 to 0.53626, saving model to ./train/cat_dog_cnn/cp-
0006.ckpt
100/100 [==============================] - 2s 24ms/step - loss: 0.4867 - accuracy: 0.7680 -
val_loss: 0.5363 - val_accuracy: 0.7300
Epoch 7/30
...
Epoch 30/30
100/100 [==============================] - ETA: 0s - loss: 0.2182 - accuracy: 0.9115
Epoch 00030: val_loss did not improve from 0.43556
100/100 [==============================] - 2s 20ms/step - loss: 0.2182 - accuracy: 0.9115 -
val_loss: 0.4576 - val_accuracy: 0.8130
```

1.4 Model Evaluation

After training, we can get 85% of training accuracy, and 78% of validation accuracy. But the important thing is that can we use this model for inference. And that's why we split raw data with training and test data. Test set must be unknown or unseen data for the training model. First, let's evaluate our trained model.

```
model.evaluate(test_ds, steps=test_len / 20)
```

```
100/100 [==============================] - 1s 6ms/step - loss: 0.4753 - accuracy: 0.8050

[0.47534096240997314, 0.8050000071525574]
```

The accuracy of test dataset is quite lower than training/validation accuracy. Because our model is not perfect model for classification, and test dataset may be the unknown data for trained model. Maybe the class distribution is different from training set and test set (or imbalance of class)

We used callback function for saving best performance model's weight. If we can redefine our structure of model, we can load the weight in that model. So we don't need retrain the model while using many hours.

```
model_inf = tf.keras.Sequential(name='Cat_Dog_CNN_load')

model_inf.add(Conv(filters=32, kernel_size=(3, 3)))
model_inf.add(Conv(filters=64, kernel_size=(3, 3)))
model_inf.add(Conv(filters=128, kernel_size=(3, 3)))
model_inf.add(Conv(filters=128, kernel_size=(3, 3)))

model_inf.add(tf.keras.layers.Flatten())
model_inf.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
model_inf.add(tf.keras.layers.Dense(units=2, activation=tf.keras.activations.softmax))

model_inf.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4),
                  loss = tf.keras.losses.SparseCategoricalCrossentropy(),
                  metrics = ['accuracy'])

for images, labels in train_ds.take(1):
    outputs = model_inf(images, training=False)

# Load model
model_inf.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
```

```
model_inf.evaluate(test_ds, steps=test_len / 20)
```

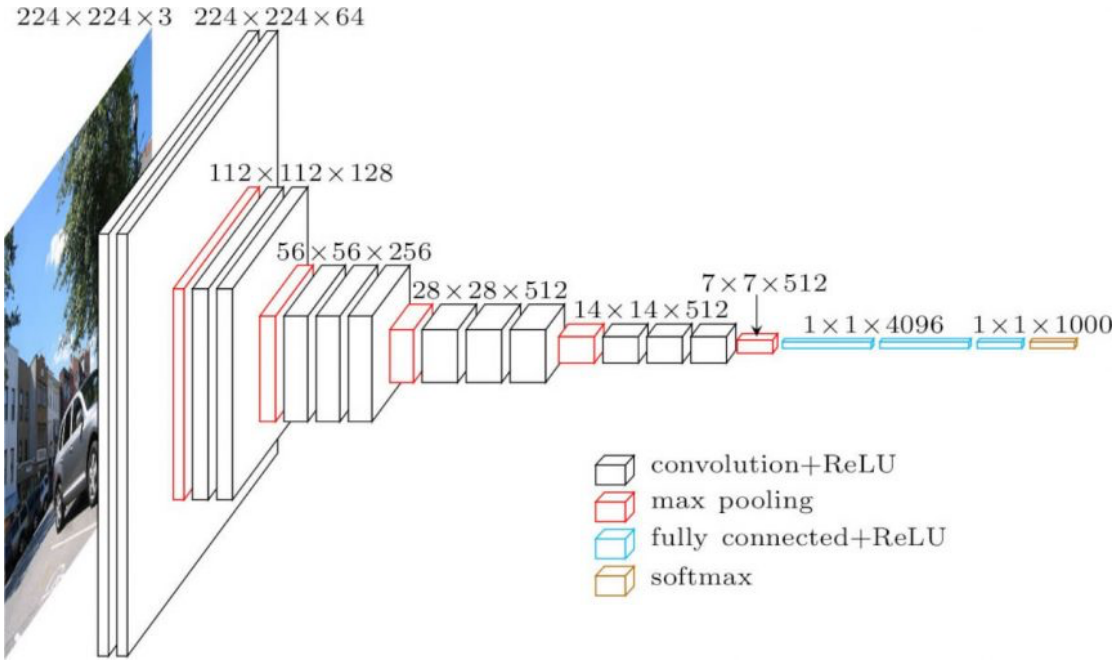
```
100/100 [==============================] - 1s 7ms/step - loss: 0.4560 - accuracy: 0.8095

[0.4560146927833557, 0.809499979019165]
```

Almost same result as you saw before. And we can also visualize our performance with images. Here, we can display the correct result with blue color, incorrect result with red color in title.

```
test_batch_size = 25

for images, labels in test_ds.take(1):
    predictions = model_inf(images)

images = images[:test_batch_size]
labels = labels[:test_batch_size]
predictions = predictions[:test_batch_size]

labels_map = {0: 'cat', 1: 'dog'}

# 시각화
fig = plt.figure(figsize=(10, 10))
for i, (px, py, y_pred) in enumerate(zip(images, labels, predictions)):
    p = fig.add_subplot(5, 5, i+1)
    if np.argmax(y_pred.numpy()) == py.numpy():
        p.set_title("{}".format(labels_map[py.numpy()]), color='blue')
    else:
        p.set_title("{}/{}".format(labels_map[np.argmax(y_pred.numpy())],
                                   labels_map[py.numpy()]), color='red')
    p.imshow(px.numpy()*0.5+0.5)
    p.axis('off')
```

2 Transfer Learning

We built our CNN model for cat-dog classification. But we can make our model with state-of-the-art approach. We will build our model with VGG16.



VGG16 was introduced in the ILSVRC 2014, we can borrow the structure of model as an convolutional layer like our Conv layer.

```
conv_vgg16 = tf.keras.applications.VGG16(weights='imagenet',
                                         include_top=False,
                                         input_shape=(150, 150, 3))
```

Actually, this model is trained with 1000 classes included in ImageNet dataset. But in our task, our task is to classify only 2 classes, not 1000. So it is not required whole layers. And in Deep Neural Network, the layer close to input usually does extract the general features. and the layer close to output usually extract the specific feature of class.

For transfer learning, we can borrow the general extraction layer, and add it as a convolution layer. Then we can add dense layer for the output to can get 2 labels, same as before. So we can see the argument include_top in VGG16. That means that we notice that only use specific layer, not whole layers.

So our implementation will be like this,

```
model_vgg = tf.keras.Sequential(name='Cat_Dog_CNN_VGG16')
model_vgg.add(conv_vgg16)
model_vgg.add(tf.keras.layers.Flatten())
model_vgg.add(tf.keras.layers.Dense(units=256, activation=tf.keras.activations.relu))
model_vgg.add(tf.keras.layers.Dense(units=2, activation=tf.keras.activations.softmax))
```

We can check our model structure,

```
for variable in model_vgg.trainable_variables:
    print(variable.name)
```

```
block1_conv1/kernel:0
block1_conv1/bias:0
block1_conv2/kernel:0
block1_conv2/bias:0
block2_conv1/kernel:0
block2_conv1/bias:0
block2_conv2/kernel:0
block2_conv2/bias:0
block3_conv1/kernel:0
block3_conv1/bias:0
block3_conv2/kernel:0
block3_conv2/bias:0
block3_conv3/kernel:0
block3_conv3/bias:0
block4_conv1/kernel:0
block4_conv1/bias:0
block4_conv2/kernel:0
block4_conv2/bias:0
block4_conv3/kernel:0
block4_conv3/bias:0
block5_conv1/kernel:0
block5_conv1/bias:0
block5_conv2/kernel:0
block5_conv2/bias:0
block5_conv3/kernel:0
...
dense_6/kernel:0
dense_6/bias:0
dense_7/kernel:0
dense_7/bias:0
```

We need to train the Dense layer for classifying cats and dogs. This kind of process is called fine-tuning. To do this, we need to change the trainable property of each layers.

```
conv_vgg16.trainable = True

set_trainable = False

for layer in conv_vgg16.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

After that, we found out that trainable variable of our model is smaller than before.

```
for variable in model_vgg.trainable_variables:
    print(variable.name)
```

```
block5_conv1/kernel:0
block5_conv1/bias:0
block5_conv2/kernel:0
block5_conv2/bias:0
block5_conv3/kernel:0
block5_conv3/bias:0
dense_6/kernel:0
dense_6/bias:0
dense_7/kernel:0
dense_7/bias:0
```

2.1 Train and Evaluation

Same processes are processed here, training and evaluation. We expect our accuracy is much higher than our built model.

```
model_vgg.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4),
                  loss = tf.keras.losses.SparseCategoricalCrossentropy(),
                  metrics = ['accuracy'])
```

```
checkpoint_path = "./train/exp_pre_trained/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)
```

```
model_vgg.fit(train_ds, steps_per_epoch = train_len / 20,
              validation_data=val_ds,
              validation_steps= val_len / 20,
              epochs= 5,
              callbacks= [cp_callback])
```

```
model_vgg.evaluate(test_ds, steps=test_len / 20)
```

```
100/100 [==============================] - 3s 35ms/step - loss: 0.1691 - accuracy: 0.9400

[0.1691393107175827, 0.9399999976158142]
```

The validation accuracy and test accuracy is much higher than our model. And we can also reduce our model training epoch, and get that result.

4 Summary

In this project, we build cat-dot classifier with CNN model. To overcome the limitation of dataset amount, we apply data augmentation, and re-generate the dataset. After that, we made dataset pipeline for memory efficiency.

We improve our model with transfer learning. So we borrow the convolution layer of VGG16 (the winner of ILSVRC 2014) and fune-tuned some layers. After training, we get almost 94% classification accuracy from our model.

References

[1] Z. Raduly, C. Sulyok, Z. Vad´ aszi, and A. Z´ olde, “Dog breed identification¨ using deep learning,” in 2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY), 2018, pp. 000 271–000 276.

[2] P. Borwarnginn, W. Kusakunniran, S. Karnjanapreechakorn, and K. Thongkanchorn, “Knowing your dog breed: Identifying a dog breed with deep learning,” International Journal of Automation and Computing, vol. 18, no. 1, pp. 45–54, Feb 2021. [Online]. Available: https://doi.org/10.1007/s11633-020-1261-0

[3] Y. Lee, "Image Classification with Artificial Intelligence: Cats vs Dogs," 2021 2nd International Conference on Computing and Data Science (CDS), 2021, pp. 437-441, doi: 10.1109/CDS52072.2021.00081.

[4] J Deng, W Dong, R Socher et al., "Imagenet: A large-scale hierarchical image database[C]", 009 IEEE conference on computer vision and pattern recognition. Ieee, pp. 248-255, 2009.

[5] L Deng, "The mnist database of handwritten digit images for machine learning research [best of the web][J]", IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141-142, 2012.

[6] O Kouropteva, O Okun and M Pietikäinen, "Classification of handwritten digits using supervised locally linear embedding algorithm and support vector machine[C]", ESANN, pp. 229-234, 2003.