```
In [9]:
import tensorflow as tf
from tensorflow.keras import datasets, models, layers
from tensorflow.keras.layers import Dropout
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, Image
import pandas as pd
```

Functions

```
In [2]:
###Parsing Functions###

#Per TensorFlow documentation;
#Parse the TFRecords for emmbeded features


# Create a dictionary of features
features = {
    'height': tf.io.FixedLenFeature([], tf.int64),
    'width': tf.io.FixedLenFeature([], tf.int64),
    'depth': tf.io.FixedLenFeature([], tf.int64),
    'label': tf.io.FixedLenFeature([], tf.int64),
    'image_raw': tf.io.FixedLenFeature([], tf.string),
}

# Use dictionary to parse data and decode raw image data
def _parse_image_function(example_proto):
    parsed_data = tf.io.parse_single_example(example_proto, features)
    parsed_data['image_raw'] = tf.io.decode_jpeg(parsed_data['image_raw'], channels = 3)
    parsed_data['image_raw'] = tf.reshape(parsed_data['image_raw'], [parsed_data['height'], parsed_data['width'], parsed_data['depth']])
    return parsed_data



###PreProcessing###


#Normalize pixel values and return the image and label
def preprocess(features):
    image = tf.cast(features['image_raw'], tf.float32) / 255.0  # normalize to [0,1] range
    label = tf.cast(features['label'], tf.int32)
    return image, label
```

Notes to remember

-after parsing you need to decode the data, and the decoding needs to be in line with the encoding of the original data
-whenever decoding, the data needs to be reshaped
-its easier to resize at the beginning before writing to TFRecord
-this all seems obvious now that I'm writing it out

Load, Parse, Preprocess Data

> The original files were resized and written to TFRecords in the TFRecord_writer module

```
In [3]:
#Load TFRecords
train_raw = tf.data.TFRecordDataset('train.tfrecords')
test_raw = tf.data.TFRecordDataset('test.tfrecords')
valid_raw = tf.data.TFRecordDataset('validate.tfrecords')

#Parse the TFRecords
train_parsed = train_raw.map(_parse_image_function)
test_parsed = test_raw.map(_parse_image_function)
valid_parsed = valid_raw.map(_parse_image_function)
```

```
In [ ]:
#Display sample of images


for features in valid_parsed:
    image = features['image_raw']
    label = features['label']
    plt.figure()
    plt.imshow(image)
    plt.title(f'Label: {label}')
    plt.show()
```

```
In [4]:
#Preprocess data
train_dataset = train_parsed.map(preprocess)
test_dataset = test_parsed.map(preprocess)
valid_dataset = valid_parsed.map(preprocess)
```

```
In [5]:
#Batch and Shuffle

BATCH_SIZE = 32
train_dataset = train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
valid_dataset = valid_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
```

```
In [6]:
for images, labels in train_dataset.take(1):
    print(images.shape)
    print(labels.shape)
```

```
(32, 64, 64, 3)
(32,)
```

Feature Extraction

```
In [7]:
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64,64, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

Dense Layers

```
In [13]:
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(layers.Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(layers.Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(layers.Dense(4))
```

Training

```
In [14]:
optimizer = Adam(learning_rate=0.0001)

model.compile(optimizer=optimizer,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_dataset, epochs=20, validation_data=valid_dataset)
```

```
Epoch 1/20
124/124 [==============================] - 11s 80ms/step - loss: 1.3933 - accuracy: 0.2589 - val_loss: 1.2281 - val_accuracy: 0.5957
Epoch 2/20
124/124 [==============================] - 10s 80ms/step - loss: 1.1623 - accuracy: 0.4480 - val_loss: 1.0443 - val_accuracy: 0.6017
Epoch 3/20
124/124 [==============================] - 10s 78ms/step - loss: 0.9776 - accuracy: 0.3914 - val_loss: 0.8242 - val_accuracy: 0.4681
Epoch 4/20
124/124 [==============================] - 10s 78ms/step - loss: 0.9053 - accuracy: 0.3662 - val_loss: 0.8035 - val_accuracy: 0.4835
Epoch 5/20
124/124 [==============================] - 10s 78ms/step - loss: 0.8533 - accuracy: 0.3939 - val_loss: 0.8334 - val_accuracy: 0.4941
Epoch 6/20
124/124 [==============================] - 10s 78ms/step - loss: 0.8539 - accuracy: 0.3779 - val_loss: 0.7793 - val_accuracy: 0.4823
Epoch 7/20
124/124 [==============================] - 10s 79ms/step - loss: 0.8374 - accuracy: 0.4069 - val_loss: 0.7849 - val_accuracy: 0.4976
Epoch 8/20
124/124 [==============================] - 10s 79ms/step - loss: 0.8355 - accuracy: 0.3769 - val_loss: 0.7750 - val_accuracy: 0.4988
Epoch 9/20
124/124 [==============================] - 10s 78ms/step - loss: 0.8132 - accuracy: 0.4071 - val_loss: 0.7862 - val_accuracy: 0.5118
Epoch 10/20
124/124 [==============================] - 10s 78ms/step - loss: 0.8129 - accuracy: 0.4117 - val_loss: 0.7647 - val_accuracy: 0.5106
Epoch 11/20
124/124 [==============================] - 10s 79ms/step - loss: 0.7981 - accuracy: 0.4269 - val_loss: 0.7809 - val_accuracy: 0.5331
Epoch 12/20
124/124 [==============================] - 10s 79ms/step - loss: 0.8053 - accuracy: 0.4150 - val_loss: 0.7632 - val_accuracy: 0.5934
Epoch 13/20
124/124 [==============================] - 10s 79ms/step - loss: 0.7966 - accuracy: 0.4820 - val_loss: 0.8951 - val_accuracy: 0.5024
Epoch 14/20
124/124 [==============================] - 10s 79ms/step - loss: 0.7731 - accuracy: 0.4919 - val_loss: 0.8192 - val_accuracy: 0.4764
Epoch 15/20
124/124 [==============================] - 10s 79ms/step - loss: 0.8210 - accuracy: 0.3596 - val_loss: 0.8568 - val_accuracy: 0.6017
Epoch 16/20
124/124 [==============================] - 10s 79ms/step - loss: 0.7607 - accuracy: 0.4997 - val_loss: 0.7391 - val_accuracy: 0.5567
Epoch 17/20
124/124 [==============================] - 10s 80ms/step - loss: 0.7583 - accuracy: 0.5140 - val_loss: 0.7451 - val_accuracy: 0.4976
Epoch 18/20
124/124 [==============================] - 10s 79ms/step - loss: 0.6895 - accuracy: 0.6023 - val_loss: 0.7664 - val_accuracy: 0.5473
Epoch 19/20
124/124 [==============================] - 10s 82ms/step - loss: 0.7427 - accuracy: 0.5467 - val_loss: 0.7072 - val_accuracy: 0.5520
Epoch 20/20
124/124 [==============================] - 10s 79ms/step - loss: 0.6325 - accuracy: 0.6525 - val_loss: 0.6787 - val_accuracy: 0.6217
```

```
In [15]:
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 32)        896

 max_pooling2d (MaxPooling2  (None, 31, 31, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 29, 29, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 14, 14, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 12, 12, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 6, 6, 128)         0
 g2D)

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 512)               2359808

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 256)               131328

 dropout_1 (Dropout)         (None, 256)               0

 dense_2 (Dense)             (None, 128)               32896

 dropout_2 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 4)                 516

 flatten_1 (Flatten)         (None, 4)                 0

 dense_4 (Dense)             (None, 512)               2560

 dropout_3 (Dropout)         (None, 512)               0

 dense_5 (Dense)             (None, 256)               131328

 dropout_4 (Dropout)         (None, 256)               0

 dense_6 (Dense)             (None, 128)               32896

 dropout_5 (Dropout)         (None, 128)               0

 dense_7 (Dense)             (None, 4)                 516

=================================================================
Total params: 2785096 (10.62 MB)
Trainable params: 2785096 (10.62 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
In [16]:
test_loss, test_acc = model.evaluate(test_dataset, verbose=2)
print(test_acc)
```

```
27/27 - 0s - loss: 0.6501 - accuracy: 0.6379 - 475ms/epoch - 18ms/step
0.6378698348999023
```

Notes on hyperparameter tuning

```
Data Augmentation: You can use data augmentation techniques to artificially increase the size of your dataset and improve generalization.

More Complex Model: The current model might not be complex enough to capture the underlying patterns in the data. You could experiment with adding more convolutional
layers, or increasing the number of filters in the existing layers.

Regularization: You could use regularization techniques, like dropout or L2 regularization, to prevent overfitting.

Learning Rate: Adjusting the learning rate can sometimes significantly improve performance. You could experiment with different learning rates to see if this improves your
results.

Early Stopping: Use early stopping to halt the training when the validation loss stops decreasing.

Data Inspection: Inspect your data to ensure that the labels are correct, that the data is properly cleaned and preprocessed, and that your train and validation splits are
representative of the overall distribution.
```

```
In [17]:
#predictions

predictions = model.predict(test_dataset)
predictions
```

```
27/27 [==============================] - 1s 17ms/step
Out[17]:
array([[ 2.0478764 ,  2.8070717 , -3.2854788 , -2.4127898 ],
       [ 2.036663  ,  0.15213811, -1.830981  , -0.5043692 ],
       [ 1.6113943 , -0.290832  , -1.2608676 , -0.1360637 ],
       ...,
       [-2.2221727 , -3.8131573 ,  2.2766795 ,  2.8448625 ],
       [-2.152085  , -3.5070183 ,  2.2761571 ,  2.4220438 ],
       [-2.153783  , -3.5097346 ,  2.27708   ,  2.4247603 ]],
      dtype=float32)
```

```
In [ ]:
predicted_labels = np.argmax(predictions, axis=1)
actual_labels = []
for img, label in test_dataset:
    actual_labels.extend(label.numpy())

for i in range(len(predicted_labels)):
    print(f"Predicted: {predicted_labels[i]}, Actual: {actual_labels[i]}")
```

```
In [ ]:
```