CNN and Transfer Learning

```
data = tf.keras.utils.image_dataset_from_directory(directory = '/content/drive/MyDrive/Colab
Notebooks/dataset',
                                                   color_mode = 'rgb',
                                                   batch_size = 64,
                                                   image_size = (224,224),
                                                   shuffle=True,
                                                   seed = 2022)
```

This code is using the TensorFlow Keras module's **image_dataset_from_directory** function to create an image dataset from a directory containing image files.

Here is a breakdown of the arguments used in the function:

- **directory**: The path to the directory containing the image files.
- **color_mode**: The color mode to use for the images. In this case, it is set to RGB, which means that the images will be loaded in color.
- **batch_size**: The number of images to include in each batch of the dataset.
- **image_size**: The size to which the images will be resized.
- **shuffle**: Whether to shuffle the order of the images in the dataset. In this case, it is set to True.
- **seed**: The random seed used to shuffle the images. In this case, it is set to 2022.

```
labels = np.concatenate([y for x,y in data], axis=0)
```

This line of code is concatenating all the labels (y) from the data list along axis 0 using the NumPy concatenate function. The resulting NumPy array will contain all the labels from all the samples in the data list.

Here's a breakdown of the code:

1. **[y for x,y in data]**: This is a list comprehension that extracts all the labels (y) from the data list. The **for x,y in data** part of the comprehension iterates through each sample in the data list, unpacking the label (y) and ignoring the input (x). The resulting list contains all the labels from all the samples in the data list.
2. **np.concatenate()**: This is a NumPy function that concatenates arrays along a specified axis. In this case, **axis=0** means that the arrays will be concatenated along the first axis, which is the rows.
3. **labels = np.concatenate([y for x,y in data], axis=0)**: This line of code assigns the concatenated array to the variable **labels**.

```
labels = np.concatenate([y for x,y in data], axis=0)
```

Here's a breakdown of the code:

1. `[y for x,y in data]`: This is a list comprehension that extracts all the labels (y) from the data list. The `for x,y in data` part of the comprehension iterates through each sample in the data list, unpacking the label (y) and ignoring the input (x). The resulting list contains all the labels from all the samples in the data list.
2. `np.concatenate()`: This is a NumPy function that concatenates arrays along a specified axis. In this case, `axis=0` means that the arrays will be concatenated along the first axis, which is the rows.
3. `labels = np.concatenate([y for x,y in data], axis=0)`: This line of code assigns the concatenated array to the variable `labels`.

```
values = pd.value_counts(labels)
values = values.sort_index()
```

This code is using the Pandas `value_counts()` function to count the number of occurrences of each unique value in the `labels` array. It then sorts the resulting Series object by the index, which in this case is the unique label values.

Here's a breakdown of the code:

1. `pd.value_counts(labels)`: This Pandas function takes an array or Series object as input and returns a Series object containing the counts of unique values in the input array. In this case, it is applied to the `labels` array, which contains the labels for a set of samples.
2. `values = values.sort_index()`: This line of code sorts the resulting Series object, `values`, by the index using the `sort_index()` method. In this case, since the index is the unique label values, the resulting Series will be sorted by label. The resulting `values` Series contains the count of occurrences of each unique label value, sorted by label.

```
plt.figure(figsize=(12,8))
plt.pie(values,autopct='%1.1f%%', explode = [0.02,0.02,0.02, 0.02], textprops = {"fontsize":15})
plt.legend(labels=data.class_names)
plt.show()
```

This code is creating a pie chart using Matplotlib to visualize the distribution of the labels in the `values` Series object. It sets the figure size, adds a percentage format for the pie chart values, and adds an explode effect to highlight each slice. Additionally, it sets the legend using the `class_names` attribute of a `data` object.

Here's a breakdown of the code:

1. `plt.figure(figsize=(12,8))`: This sets the figure size of the plot to 12 inches by 8 inches using the `figure()` function of Matplotlib.
2. `plt.pie(values,autopct='%1.1f%%', explode = [0.02,0.02,0.02, 0.02], textprops = {"fontsize":15})`: This creates a pie chart using the `pie()` function of Matplotlib. The `values` Series object is used as input, and the `autopct` parameter is set to display values as percentages with one decimal place. The `explode` parameter creates an "exploded" effect by separating each slice from the center of the pie chart by a small amount, with the `explode` list specifying the amount of separation for each slice. Finally, the `textprops` parameter is used to set the font size of the labels on the pie chart.
3. `plt.legend(labels=data.class_names)`: This sets the legend for the plot using the `legend()` function of Matplotlib. The `labels` parameter is set to the `class_names` attribute of a `data` object, which presumably contains a list of class names corresponding to the labels.
4. `plt.show()`: This displays the plot.

```
data_iterator = data.as_numpy_iterator()
batch = data_iterator.next()
```

These two lines of code create a data iterator from a Pandas DataFrame and then get the next batch of data from the iterator. The **batch** variable will contain the next batch of data in the form of a NumPy array.

Here's a breakdown of the code:

1. **data**: This is a Pandas DataFrame object that presumably contains the input data and labels for a machine learning task.
2. **data.as_numpy_iterator()**: This method returns a data iterator that yields each row of the DataFrame as a NumPy array. This is useful for iterating through large datasets that do not fit into memory, as it allows for processing one batch of data at a time.
3. **data_iterator**: This variable is assigned to the data iterator returned by **data.as_numpy_iterator()**. It can be used in a **for** loop to iterate through the rows of the DataFrame one batch at a time, and each batch will be returned as a NumPy array.
4. **data_iterator.next()**: This method gets the next batch of data from the iterator. Each batch will be returned as a NumPy array.
5. **batch**: This variable is assigned to the next batch of data returned by **data_iterator.next()**. It will contain the next batch of data in the form of a NumPy array. This batch can then be used as input to a machine learning model or for further processing.

```
batch[0].shape
```

1. **batch[0].shape**: This retrieves the shape of the first element in the batch as a tuple. The shape of the data is important for ensuring that it is compatible with a machine learning model, and for reshaping the data if necessary. The shape tuple will have one element for each dimension of the data, such as **(batch_size, num_features)** for a batch of input data with **batch_size** examples and **num_features** features.

```
data = data.map(lambda x, y: (x/255, y))
```

This code maps a lambda function to each element of a TensorFlow dataset, where the lambda function normalizes the input data by dividing it by 255. This is a common preprocessing step for image data, where the pixel values are typically integers between 0 and 255.

Here's a breakdown of the code:

1. **data**: This is a TensorFlow dataset object that presumably contains the input data and labels for a machine learning task.
2. **data.map()**: This method applies a function to each element of the dataset, returning a new dataset with the transformed elements.
3. **lambda x, y: (x/255, y)**: This lambda function takes two arguments, **x** and **y**, which represent the input data and labels, respectively. It then normalizes the input data by dividing it by 255, and returns a tuple containing the normalized input data and the original labels.

4. **(x/255, y)**: This is the output of the lambda function. The input data **x** is divided element-wise by 255 to normalize it, and the original labels **y** are returned unchanged.
5. The normalized dataset is returned and can be used for further processing, such as training a machine learning model.

```
sample = data.as_numpy_iterator().next()
```

This code gets the first batch of data from a TensorFlow dataset as a NumPy array. The **sample** variable will contain the first batch of data in the form of a NumPy array.

Here's a breakdown of the code:

1. **data**: This is a TensorFlow dataset object that presumably contains the input data and labels for a machine learning task.
2. **data.as_numpy_iterator()**: This method returns a data iterator that yields each element of the dataset as a NumPy array. This is useful for iterating through datasets in a format that can be easily processed by NumPy functions.
3. **data.as_numpy_iterator().next()**: This code gets the first batch of data from the data iterator. Each batch will be returned as a NumPy array.
4. **sample**: This variable is assigned to the first batch of data returned by **data.as_numpy_iterator().next()**. It will contain the first batch of data in the form of a NumPy array. This batch can then be used as input to a machine learning model or for further processing.

```
train_size = int(0.7 * len(data)) +1
val_size = int(0.2 * len(data))
test_size = int(0.1 * len(data))
```

This code calculates the size of the training, validation, and test sets as a percentage of the total size of the data.

Here's a breakdown of the code:

1. **len(data)**: This returns the total number of elements in the **data** object, which is assumed to be a Python list or NumPy array.
2. **0.7 * len(data)**: This calculates 70% of the total size of the data.
3. **int(0.7 * len(data)) + 1**: This rounds up the size of the training set to the nearest integer and adds 1. This ensures that the training set will contain at least one example.
4. **0.2 * len(data)**: This calculates 20% of the total size of the data, which will be used as the validation set size.
5. **0.1 * len(data)**: This calculates 10% of the total size of the data, which will be used as the test set size.

By splitting the data into training, validation, and test sets, we can evaluate the performance of machine learning models on data that they haven't seen during training. The training set is used to fit the model, the validation set is used to tune model hyperparameters and assess performance during training, and the test set is used to evaluate the final performance of the model.

```
test_set = {"images":np.empty((0,224,224,3)), "labels":np.empty(0)}
```

```
while True:
    try:
        batch = test_iter.next()
        test_set['images'] = np.concatenate((test_set['images'], batch[0]))
        test_set['labels'] = np.concatenate((test_set['labels'], batch[1]))
    except:
        break
```

The code block you provided creates a dictionary `test_set` with two keys: `images` and `labels`. The values associated with these keys are initially set to empty arrays using `np.empty()`.

A while loop is then used to iterate over the test data generator (`test_iter`) using the `next()` method. At each iteration, the code attempts to retrieve the next batch of data from `test_iter` and concatenate the images and labels with the existing arrays in `test_set`. This is done using the `np.concatenate()` function, which is used to join arrays along a specified axis.

The `try-except` block is used to catch the `StopIteration` exception, which is raised by the `next()` method when there is no more data to retrieve from the generator. When the exception is caught, the loop breaks and the `test_set` dictionary is returned.

Overall, the purpose of this code is to create a test set consisting of images and labels, by concatenating the batches of data retrieved from the test data generator.

```
y_true = test_set['labels']
```

The line `y_true = test_set['labels']` assigns the labels of the test set to the variable `y_true`. This is assuming that the `test_set` dictionary was previously created and populated with data, as described in the code block you provided in your previous message.

By retrieving the labels from `test_set`, this line of code allows you to compare the model's predicted labels with the true labels of the test set, and evaluate the performance of the model.

```
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
def plot_his(history):
    plt.figure(figsize=(15,12))
    metrics = ['accuracy', 'loss']
    for i, metric in enumerate(metrics):
        plt.subplot(220+1+i)
        plt.plot(history.epoch, history.history[metric], color=colors[0], label='Train')
        plt.plot(history.epoch, history.history['val_'+metric],
                 color=colors[1], linestyle="--", label='Val')
        plt.xlabel('Epoch')
        plt.ylabel(metric)
        plt.legend()
    plt.show()
```

This is a function for plotting the training and validation loss/accuracy over epochs of a neural network. The `history` argument is the object returned by the `fit` method of a Keras model, which contains the training and validation metrics

for each epoch.

The function creates a plot with two subplots: one for the training and validation accuracy, and one for the training and validation loss. The x-axis of each subplot represents the epoch number, and the y-axis represents the corresponding metric (accuracy or loss). The plot shows the evolution of the metrics over the training epochs, with separate lines for the training and validation sets.

```
model = Sequential([
        Conv2D(filters = 64, kernel_size=3, activation = 'relu',padding='same', input_shape=
(224,224,3)),
        Conv2D(filters = 64, kernel_size=3, activation = 'relu',padding='same'),
        BatchNormalization(),
        MaxPool2D(2),
        Dropout(0.3),

        Conv2D(filters = 128, kernel_size=3,padding='same', activation = 'relu',),
        Conv2D(filters = 128, kernel_size=3,padding='same', activation = 'relu',),
        BatchNormalization(),
        MaxPool2D(2),
        Dropout(0.3),

        Conv2D(filters = 128, kernel_size=3,padding='same', activation = 'relu',),
        Conv2D(filters = 128, kernel_size=3,padding='same', activation = 'relu',),
        BatchNormalization(),
        Conv2D(filters = 128, kernel_size=3,padding='same', activation = 'relu',),
        BatchNormalization(),
        MaxPool2D(2),
        Dropout(0.3),

        Flatten(),
        Dense(64, activation = 'relu'),
        Dropout(0.3),
        BatchNormalization(),
        Dense(128, activation = 'relu'),
        Dropout(0.3),
        BatchNormalization(),
        Dense(4, activation='softmax')
    ])
    model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
optimizer='adam', metrics=['accuracy'])
    return model
```

This code defines a convolutional neural network (CNN) using the Keras Sequential API for classifying images into one of four classes. The network consists of 3 convolutional blocks, followed by two fully connected (dense) layers, and a final output layer with softmax activation.

Here's a breakdown of the code:

1. `Sequential()`: This initializes a sequential model object in Keras, which allows you to stack layers on top of each other in a linear fashion.
2. `Conv2D(filters = 64, kernel_size=3, activation = 'relu',padding='same', input_shape=(224,224,3))`: This adds a 2D convolutional layer to the model with 64 filters, a kernel size of 3x3, ReLU activation, same padding, and an input shape of 224x224x3 (assuming RGB images).
3. `Conv2D(filters = 64, kernel_size=3, activation = 'relu',padding='same')`: This adds a second 2D convolutional layer to the model with the same hyperparameters as the first layer.
4. `BatchNormalization()`: This adds a batch normalization layer to the model, which normalizes the inputs to the next layer by subtracting the mean and dividing by the standard deviation of the inputs in each batch.
5. `MaxPool2D(2)`: This adds a max pooling layer to the model with a pool size of 2x2, which reduces the spatial dimensions of the output feature maps by taking the maximum value in each 2x2 block.
6. `Dropout(0.3)`: This adds a dropout layer to the model with a dropout rate of 0.3, which randomly drops out 30% of the inputs to the layer during training to prevent overfitting.
7. The next three blocks repeat the pattern of convolutional layer, convolutional layer, batch normalization, max pooling, and dropout with increasing filter size.
8. `Flatten()`: This flattens the output of the previous layer into a 1D vector, which can be fed into a fully connected layer.
9. `Dense(64, activation = 'relu')`: This adds a fully connected (dense) layer with 64 neurons and ReLU activation.
10. `Dropout(0.3)`: This adds another dropout layer with a dropout rate of 0.3.
11. `BatchNormalization()`: This adds another batch normalization layer.
12. `Dense(128, activation = 'relu')`: This adds another dense layer with 128 neurons and ReLU activation.
13. `Dropout(0.3)`: This adds another dropout layer with a dropout rate of 0.3.
14. `BatchNormalization()`: This adds another batch normalization layer.
15. `Dense(4, activation='softmax')`: This adds the output layer with 4 neurons and softmax activation, which outputs a probability distribution over the 4 classes.
16. `model.compile()`: This compiles the model with the specified loss function, optimizer, and evaluation metric.

This model uses convolutional layers to extract features from the input images and fully connected layers to classify the images based on those features. Dropout and batch normalization are used to prevent overfitting and stabilize the learning process.

`padding='same'`

It is a parameter used in convolutional layers in neural networks. It refers to the technique of adding additional padding to the input data before applying the convolutional filter. The padding ensures that the output of the convolutional operation has the same spatial dimensions as the input data. Specifically, with `padding='same'`, the padding size is determined such that the output feature map has the same height and width as the input feature map, given a stride of 1. If the stride is greater than 1, then the padding is adjusted accordingly. This is useful for preventing information loss at the edges of the input data.

`activation='relu'`

It is a parameter used in activation layers in neural networks. It stands for "rectified linear unit" and is a popular activation function used in deep learning models. The ReLU function is defined as:

```
f(x) = max(0, x)
```

ReLU activation is commonly used in convolutional neural networks (CNNs) for image classification tasks, as it has been shown to provide better performance than other activation functions like sigmoid and tanh. It helps to improve the model's ability to learn complex features by introducing nonlinearity into the network.

```
y_pred = np.argmax(model.predict(test_set['images']), 1)
```

This line of code predicts the class labels for the test set images using the trained model.

`model.predict(test_set['images'])` returns an array of shape (num_samples, num_classes), where each element in the array represents the predicted probability of the corresponding sample belonging to each of the classes.

`np.argmax` returns the indices of the maximum values along an axis. Here, **`axis=1`** means that for each sample, the index of the class with the highest predicted probability is returned.

Therefore, **`y_pred`** will be an array of the same length as **`test_set['labels']`**, where each element is the predicted class label for the corresponding image in **`test_set['images']`**.

```
effnet = EfficientNetB3(include_top=False, weights="imagenet",input_shape=(224,224,3), pooling='max')
    effnet.trainable=False

    for layer in effnet.layers[83:]:
      layer.trainable=True

    x = effnet.output
    x = BatchNormalization()(x)
    x = Dense(1024, kernel_regularizer = regularizers.l2(l =
0.016),activity_regularizer=regularizers.l1(0.006),
                    bias_regularizer=regularizers.l1(0.006) ,activation='relu')(x)
    x = Dropout(rate=.45, seed=2022)(x)
    output=Dense(4, activation='softmax')(x)

    model= tf.keras.Model(inputs=effnet.input, outputs=output)
    model.compile(optimizer = 'adamax', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

This code defines a function that creates a Keras model based on the EfficientNetB3 architecture with some modifications:

- The top layers (the part that maps the output of the network to the desired output shape) are not included, because they will be replaced with a custom output layer.
- The weights are initialized to the pre-trained weights on the ImageNet dataset.
- The input shape of the model is (224,224,3) because this is the size of the input images that will be fed into the model.
- The **`trainable`** attribute of the EfficientNetB3 model is set to **`False`**, meaning that the weights of the model will not be updated during training.

- The last 83 layers of the EfficientNetB3 model are set to be trainable, so that they can be fine-tuned on the specific task at hand.
- The output of the EfficientNetB3 model is fed into a batch normalization layer, followed by a fully connected (dense) layer with 1024 units and ReLU activation.
- A dropout layer is added after the dense layer to prevent overfitting.
- Finally, a dense output layer with 4 units and softmax activation is added, to classify the input images into one of the 4 classes.
- The model is compiled with the **adamax** optimizer and **sparse_categorical_crossentropy** loss, and **accuracy** metric is used to evaluate the performance of the model.

```
effnet = EfficientNetB3(include_top=False, weights="imagenet",input_shape=(224,224,3), pooling='max')
    effnet.trainable=False
```

This code snippet loads the pre-trained EfficientNetB3 model with the following configurations:

- **include_top=False** - This means that the final fully connected layer of the pre-trained model is not included, allowing us to add our own layers on top of the pre-trained model for transfer learning.
- **weights="imagenet"** - This means that the pre-trained weights for the model are initialized using the ImageNet dataset, which is a large dataset of labeled images.
- **input_shape=(224,224,3)** - This specifies the input shape of the model. EfficientNetB3 takes inputs of size 224x224 with 3 channels for red, green, and blue.
- **pooling='max'** - This sets the type of pooling to use after the convolutional layers. In this case, a max pooling layer is used to reduce the dimensionality of the feature maps.

After loading the pre-trained model, **effnet.trainable=False** is used to freeze all the layers in the model so that their weights are not updated during training. This is done to preserve the pre-trained weights and allow the model to use the pre-trained features as a starting point for transfer learning. By freezing the layers, we can speed up the training process and prevent the model from overfitting to the new dataset.

```
for layer in effnet.layers[83:]:
      layer.trainable=True
```

The code snippet you provided is setting all layers of an object named **effnet** starting from the 84th layer (index 83) to be trainable.

The **trainable** attribute of a layer in a neural network determines whether its weights can be updated during training. When a layer is trainable, the optimizer can adjust its weights based on the loss calculated during the forward and backward pass of the network.

By setting all layers starting from the 84th layer to be trainable, the intention is to fine-tune the weights of those layers based on the specific dataset being used for training. This is often done when using a pre-trained model, such as EfficientNet, and adapting it to a new task.

Note that changing the **trainable** attribute only affects the layers going forward from that point in the network. In other words, any layers before the 84th layer in **effnet** will remain frozen and not updated during training.

```
x = Dense(1024, kernel_regularizer = regularizers.l2(l =
0.016),activity_regularizer=regularizers.l1(0.006),
                    bias_regularizer=regularizers.l1(0.006) ,activation='relu')(x)
```

This code snippet creates a dense layer with 1024 units and applies three different regularization techniques to the layer's kernel (weights), activity, and bias.

Regularization is a technique used to prevent overfitting in a neural network. Overfitting occurs when a model becomes too complex and starts to fit the noise in the training data, rather than the underlying patterns. Regularization techniques aim to simplify the model or constrain its weights, so it can generalize better to unseen data.

In this code, the **kernel_regularizer** parameter applies L2 regularization to the layer's kernel with a regularization strength of **0.016**. L2 regularization is a type of weight decay that adds a penalty to the loss function proportional to the square of the weights. This encourages the network to learn smaller weight values, which can lead to a more robust model.

The **activity_regularizer** parameter applies L1 regularization to the layer's output (i.e., the activation), with a regularization strength of **0.006**. L1 regularization adds a penalty to the loss function proportional to the absolute value of the weights. This encourages the network to learn sparse representations, which can improve generalization and interpretability.

Finally, the **bias_regularizer** parameter applies L1 regularization to the layer's bias with a regularization strength of **0.006**. This is similar to the **activity_regularizer**, but it applies to the bias term instead of the activation.

The **activation** parameter sets the activation function for the layer to be **relu**, which is a popular non-linear activation function that introduces non-linearity to the model and helps to improve its representational power.

```
    x = Dropout(rate=.45, seed=2022)(x)
     output=Dense(4, activation='softmax')(x)
```

This code snippet adds a dropout layer to the network and then creates an output layer with a softmax activation function.

The **Dropout** layer is a regularization technique that randomly drops out (sets to zero) a proportion of the input units during training. The **rate** parameter specifies the fraction of the input units to drop out, in this case, 45%. Dropout helps to prevent overfitting by forcing the network to learn more robust features that are not dependent on the presence of any single input unit.

The **seed** parameter sets the random seed used by the dropout layer. This ensures that the same set of units are dropped out for each forward pass during training, making the results more reproducible.

After the dropout layer, a **Dense** layer with 4 units is created, representing the 4 possible output classes. The **softmax** activation function is used to convert the output of the network into a probability distribution over the output classes. The softmax function ensures that the output values are non-negative and sum to 1, making it suitable for multi-class classification problems. The class with the highest probability is chosen as the predicted class.

```
optimizer='adamax'
```

Changing the **optimizer** parameter to **'adamax'** specifies a different optimization algorithm to use during training.

The Adamax optimizer is a variant of the Adam optimizer that has been shown to be particularly effective for training deep neural networks with sparse gradients. It is similar to Adam but uses the L-infinity norm to normalize the learning

rate, which can help improve stability and convergence.

Note that changing the optimizer can affect the performance of the model during training and evaluation, so it's often a good idea to try different optimizers and hyperparameters to find the best combination for your specific problem.

```
model.compile(optimizer = 'adamax', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

The code snippet you provided shows the compilation of a neural network model using the Keras API.

optimizer='adamax' specifies the optimizer used during the training of the model. In this case, it is the "Adamax" optimizer, which is a variant of the Adam optimizer that is designed to be more robust to the presence of sparse gradients in the data.

loss='sparse_categorical_crossentropy' specifies the loss function used to optimize the model. In this case, it is the "sparse categorical cross-entropy" loss function, which is typically used for multi-class classification problems where the target labels are integers.

metrics=['accuracy'] specifies the evaluation metric used to monitor the performance of the model during training and evaluation. In this case, it is the "accuracy" metric, which is a common metric used to evaluate classification models.

Once the model has been compiled, it is ready for training on a dataset using the **fit** method.

```
mc = ModelCheckpoint(filepath='/content/drive/MyDrive/Project',
                     verbose=1,
                     save_best_only = True)
```

This code snippet creates a **ModelCheckpoint** callback to save the best weights of a model during training.

The **ModelCheckpoint** callback is a way to save the model weights during training, which allows us to use the best model for making predictions on new data. The **filepath** parameter specifies the location where the weights will be saved. In this case, the weights will be saved to **/content/drive/MyDrive/Project**.

The **verbose** parameter controls the amount of output to be displayed during training. A value of 1 means that progress messages will be displayed on the screen.

The **save_best_only** parameter ensures that only the best weights are saved. If set to **True**, the weights will only be saved if the monitored metric (e.g. validation accuracy) improves. If set to **False**, the weights will be saved every time the model improves during training, regardless of whether it is the best improvement or not. In this case, **save_best_only=True** means that only the best weights will be saved.

```
early_stop = callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=10,
        verbose=1,
        mode="max",
        restore_best_weights=True,
     )
```

This code snippet creates an EarlyStopping callback to monitor the validation accuracy of a model during training.

The **EarlyStopping** callback is a technique used to prevent overfitting and improve the efficiency of model training. It monitors a specified metric (in this case, **val_accuracy**) on a validation set and stops the training if the metric stops improving for a specified number of epochs (**patience**).

The **monitor** parameter specifies the metric to monitor, which is the validation accuracy in this case. The **patience** parameter specifies the number of epochs to wait before stopping the training if the monitored metric does not improve. In this case, training will stop if the validation accuracy does not improve for 10 epochs.

The **verbose** parameter controls the amount of output to be displayed during training. A value of 1 means that progress messages will be displayed on the screen.

The **mode** parameter specifies whether to maximize or minimize the monitored metric. Since we want to maximize the validation accuracy, **mode="max"** is set.

The **restore_best_weights** parameter restores the weights of the model to the point where the monitored metric was the highest. This means that we can use the model with the best validation accuracy, rather than the final model, for making predictions on new data.

```
%%writefile app.py
import streamlit as st
import tensorflow as tf

st.set_option('deprecation.showfileUploaderEncoding',False)
@st.cache(allow_output_mutation=True)
def load_model():
    model = tf.keras.models.load_model('/content/TL.hdf5')
    return model
model = load_model()

st.write("""
         # Eye Disease Detection
         """
        )

file = st.file_uploader("Please upload the image of eye",type=["jpg","png"])
#import cv2
from PIL import Image,ImageOps
import numpy as np

def import_and_predict(img,model):
    size = (224,224)
    image = ImageOps.fit(img,size,Image.ANTIALIAS)
    img = np.asarray(image)
    img_reshape = img[np.newaxis,...]
    prediction = model.predict(img_reshape)

    return prediction
```

```
if file is None:
    st.text("Please upload an image")
else:
    image = Image.open(file)
    st.image(image,use_column_width=True)
    predictions = import_and_predict(image,model)
    class_names= ['Cataract', 'Diabetic_retinopathy', 'Glaucoma', 'Normal']
    string = "This is image is most likely is : "+class_names[np.argmax(predictions)]
    st.success(string)
```

This is a Streamlit app that allows users to upload an image of an eye and get predictions for the presence of four different eye diseases: cataract, diabetic retinopathy, glaucoma, and normal. The app loads a pre-trained model from a saved HDF5 file and uses it to make predictions on the uploaded image. The **load_model()** function uses Streamlit's caching feature to load the model only once and then reuse it on subsequent app runs. The **import_and_predict()** function preprocesses the uploaded image, resizes it to 224x224 pixels, converts it to a NumPy array, and makes a prediction using the loaded model. Finally, the app displays the uploaded image and the predicted disease class.

```
!ngrok authtoken
```

!ngrok authtoken is a command used to authenticate the ngrok service. ngrok is a service that allows you to expose a web server running on your local machine to the internet. The **authtoken** command is used to provide authentication credentials to the ngrok service.

```
!nohup streamlit run app.py &
```

This command runs the **app.py** file in the background using **nohup** (which allows the command to keep running even after the terminal is closed), and streams the output to a file called **nohup.out**. The **&** at the end of the command puts the command in the background, allowing the user to continue using the terminal.

```
!streamlit run --server.port 80 app.py >/dev/null
```

This command runs the Streamlit app named **app.py** on port 80 and redirects the output to **/dev/null**, which essentially discards the output. This is often done to prevent the output from cluttering the terminal or command prompt. By running the app on port 80, it makes it accessible through a web browser using the public IP address of the server or computer running the app.

```
!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
```

This command is used to download the ngrok binary for Linux on 64-bit machines. Ngrok is a tool that creates a secure tunnel between your local machine and the internet. This is useful for exposing local web servers or applications to the internet for testing or sharing purposes.

```
get_ipython().system_raw('./ngrok http 8501 &')
```

This line of code executes a shell command from within a Jupyter notebook or Google Colab environment. Specifically, it starts an HTTP tunneling service called Ngrok on port 8501, which is where the Streamlit app is running. The "&" symbol at the end of the command allows the tunneling service to run in the background, so that other commands can be executed in the same cell

```
!curl -s http://localhost:4040/api/tunnels | python3 -c \
```

This command uses the **curl** command to make an HTTP request to the ngrok API running on **localhost:4040** to get the list of all active tunnels. The response of the request is piped to the **python3** interpreter, which runs a command-line Python script to extract the public URL of the ngrok tunnel and prints it to the console output. This URL can then be used to access the web application from a remote location.

```
!streamlit run /content/app.py
```

This command runs a Streamlit app located at **/content/app.py**. The app will be served on the default Streamlit port of 8501, unless a different port is specified in the app code. Once the command is executed, the app will be available to access in a web browser at **http://localhost:8501/**.