```python
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory
print("Hello World")
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session

import matplotlib.pyplot as plt
import numpy as np
import PIL as image_lib
import tensorflow as tf
import pathlib
from datasets import load_dataset
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.optimizers import Adam
```

Import dataset

```python
# dataset = load_dataset('alkzar90/NIH-Chest-X-ray-dataset', 'image-classification', trust_remote_code=True)
ds = load_dataset("keremberke/chest-xray-classification", name="full")
example = ds['train'][0]

print(ds)
print(example)

train_dataset = ds['train']
validation_dataset = ds['validation']
test_dataset = ds["test"]
```

```python
print(train_dataset)
print(type(train_dataset))
print(train_dataset.features)
print("Size in bytes:", train_dataset.dataset_size)
print()
print(test_dataset)

img_height = 240
img_width = 240
batch_size = 32
num_classes = 2  # Adjust this based on your dataset

'''
Function to preprocess the images themselves by:
1) Turning them into numpy arrays
2) Resizing the images down to 240x240 pixels
3) using the specific resnet preprocess_input function on the final
image
'''
def preprocess_function(example):
    # Access the in-memory image data
    image = example['image']

    # Convert the image to a tensor
    image = tf.convert_to_tensor(np.array(image))

    # Resize the image
    image = tf.image.resize(image, [img_height, img_width])

    # Preprocess the image for ResNet50
    image = tf.keras.applications.resnet50.preprocess_input(image)

    # Convert label to categorical
    label = to_categorical(example['labels'], num_classes=num_classes)

    return (image, label)

'''
Converts data set into specifically a tensor flow batched dataset
which is required to train a model using the tensor flow API
'''
def to_tf_dataset(dataset, batch_size):
    tf_dataset = tf.data.Dataset.from_generator(
        lambda: (preprocess_function(example) for example in dataset),
        output_signature=(
            tf.TensorSpec(shape=(img_height, img_width, 3),
dtype=tf.float32),
            tf.TensorSpec(shape=(num_classes,), dtype=tf.float32)
        )
    )
```

```
    return
tf_dataset.shuffle(buffer_size=len(dataset)).batch(batch_size).repeat(
)

train_tf_dataset = to_tf_dataset(train_dataset, batch_size)
validation_tf_dataset = to_tf_dataset(validation_dataset, batch_size)
test_tf_dataset = to_tf_dataset(test_dataset, batch_size)

print(type(train_tf_dataset))
# prev: <_BatchDataset element_spec=(TensorSpec(shape=(None, 240, 240,
3), dtype=tf.float32, name=None), TensorSpec(shape=(None, 2),
dtype=tf.float32, name=None))>
# curr: <_BatchDataset element_spec=(TensorSpec(shape=(None, 240, 240,
3), dtype=tf.float32, name=None), TensorSpec(shape=(None, 2),
dtype=tf.float32, name=None))>

print(train_tf_dataset)

model = Sequential()
pretrained_model = tf.keras.applications.ResNet50(
    include_top=False, # Allow adding input and outputs for custom
problem
    input_shape=(img_height,img_width,3), # This is the shape of our
images (not sure what the 3 is though)
    pooling="avg",
    classes=2,
    weights="imagenet"
)

for layer in pretrained_model.layers:
    layer.trainable=False

model.add(pretrained_model)

model.add(Flatten()) # Transforms input layer into 1-D array, allows
resent output to be feed to our fully connect FFNN that gets added
next
model.add(Dense(512, activation='relu')) # Fully connected layer of
512 likely because ResNet returns a 512 size vector
model.add(Dense(2, activation="softmax")) # Adds a final output layer
with 5 nodes for each class and softmax to get a final classifier

steps_per_epoch = len(train_dataset) // batch_size
validation_steps = len(validation_dataset) // batch_size

optimizer=Adam() # learning rate is 0.001
loss_function = "categorical_crossentropy"
metrics=["accuracy"]
model.compile(optimizer=optimizer, loss=loss_function,
metrics=metrics) # configures model for training
epochs = 5
```

```
history = model.fit(train_tf_dataset,
validation_data=validation_tf_dataset, epochs=epochs,
steps_per_epoch=steps_per_epoch, validation_steps=validation_steps) #
train the model for a fixed number of epochs

plt.figure(figsize=(8,8)) # Set our graph sizes to 8x8 for latter
epoch_range = range(epochs)
plt.plot(epoch_range, history.history['accuracy'], label="Training
Accuracy")
plt.plot(epoch_range, history.history['val_accuracy'],
label="Validation Accuracy")
plt.axis(ymin=0.4,ymax=1)
plt.grid()
plt.title('Large Fully Trained Model Accuracy using 5 epochs')

plt.ylabel('Accuracy')

plt.xlabel('Epochs')

plt.legend(['train', 'validation'])

print(history.history["val_accuracy"][-1]) # Prints the final accuray
of our model on the dev set
```

Test accuracy on the seperate test set now

```
steps = len(test_dataset) // 32
test_loss, test_accuracy = model.evaluate(test_tf_dataset,
steps=steps)

print(test_accuracy)
bigModelTestAccuracy = test_accuracy
```

Show images with labels and predictions

```
class_names = ["normal", "pnemonia"]
plt.figure(figsize=(10,10)) # Set the size of the images we are
generating
for images, labels in test_tf_dataset.take(1):
    predictions = model.predict(images)
    for var in range(6):
        # modelPrediction = demo_resnet_model.predict(images[var])
        classIdx = np.argmax(labels[var].numpy()) # Get the class
number for the predicted flower. Use argmax because of the final layer
using softmax
        className = class_names[classIdx]
        predictedClassName =
class_names[(np.argmax(predictions[var]))]
        ax = plt.subplot(3, 3, var + 1)
        plt.imshow(images[var].numpy().astype("uint8"))
```

```
        plt.text(x=0, y=1.05, s=f"Predicted: {predictedClassName}")
        plt.title(label=className)
        plt.axis("off")

model.save("/kaggle/working/ResNet50_smallDataset.keras")

# # This breaks right now because of something with the Flatten() call 
being wrong
# loaded_model = 
load_model("/kaggle/working/ResNet50_smallDataset.keras")
```

Now try to train the model using only a subset of the training data

```
import random

# Assuming train_dataset is your original dataset
num_rows = 32 # Similar size as few shot set

# Set a random seed for reproducibility
random.seed(12)

# Select 1000 random indices from the original dataset
indices = random.sample(range(len(train_dataset)), num_rows)

# Use the select method to create a new dataset with the selected 
indices
subset_dataset = train_dataset.select(indices)
```

Turn the dataset into a tensorflow compatitable object and resize images

```
subset_tf_dataset = to_tf_dataset(subset_dataset, batch_size)

img_height, img_width = 240, 240
model2 = Sequential()
pretrained_model = tf.keras.applications.ResNet50(
    include_top=False, # Allow adding input and outputs for custom 
problem
    input_shape=(img_height,img_width,3), # This is the shape of our 
images (not sure what the 3 is though)
    pooling="avg",
    classes=2,
    weights="imagenet"
)

for layer in pretrained_model.layers:
    layer.trainable=False

model2.add(pretrained_model)
```

```
model2.add(Flatten()) # Transforms input layer into 1-D array, allows
resent output to be feed to our fully connect FFNN that gets added
next
model2.add(Dense(512, activation='relu')) # Fully connected layer of
512 likely because ResNet returns a 512 size vector
# model2.add(Dense(512, activation='leaky_relu'))
model2.add(Dense(2, activation="softmax")) # Adds a final output layer
with 5 nodes for each class and softmax to get a final classifier

steps_per_epoch = len(subset_dataset) // batch_size
validation_steps = len(validation_dataset) // batch_size
optimizer=Adam() # learning rate is 0.001
loss_function = "categorical_crossentropy"
metrics=["accuracy"]
model2.compile(optimizer=optimizer, loss=loss_function,
metrics=metrics) # configures model for training
epochs = 5
history2 = model2.fit(subset_tf_dataset,
validation_data=validation_tf_dataset, epochs=epochs,
steps_per_epoch=steps_per_epoch, validation_steps=validation_steps) #
train the model for a fixed number of epochs

plt.figure(figsize=(8,8)) # Set our graph sizes to 8x8 for latter
epoch_range = range(epochs)
plt.plot(epoch_range, history2.history['accuracy'], label="Training
Accuracy")
plt.plot(epoch_range, history2.history['val_accuracy'],
label="Validation Accuracy")
plt.axis(ymin=0.4,ymax=1)
plt.grid()
plt.title('Naievely Trained Model Accuracy with 5 epochs')

plt.ylabel('Accuracy')

plt.xlabel('Epochs')

plt.legend(['train', 'validation'])

steps = len(test_dataset) // 32
test_loss, test_accuracy = model2.evaluate(test_tf_dataset,
steps=steps)

print(test_accuracy)
naieveSubsetTestAccuracy = test_accuracy
```

Model with smaller subset of training data gets ~94% accuracy

Changes needed for the large multi labeled multi class dataset

```
# model.add(pretrained_model)
# model.add(Flatten())
# model.add(Dense(512, activation='relu'))
# model.add(Dense(15, activation="sigmoid"))

# optimizer = Adam()
# loss_function = "binary_crossentropy"
# metrics = ["accuracy"]
# model.compile(optimizer=optimizer, loss=loss_function,
metrics=metrics)
```

Try to do sample choosing policy on small data set

```
import tensorflow as tf
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Flatten, Dense, Input
from tensorflow.keras.applications import ResNet50
import numpy as np

# Feature extractor using pretrained ResNet50
def create_feature_extractor(input_shape):
    base_model = ResNet50(
        include_top=False,
        input_shape=input_shape,
        pooling="avg",
        weights="imagenet"
    )
    for layer in base_model.layers:
        layer.trainable = False

    inputs = Input(shape=input_shape)
    x = base_model(inputs)
    outputs = Dense(2, activation='softmax')(x)
    # outputs = Dense(256, activation='relu')(x)  # Adding a trainable
layer
    return Model(inputs, outputs)

# Create feature extractor model
few_shot_model = create_feature_extractor((240, 240, 3))

# Function to compute class prototypes
def compute_prototypes(support_set_features, support_set_labels,
num_classes):
    prototypes = []
    for cls in range(num_classes):
        cls_features = support_set_features[support_set_labels == cls]
```

```
        cls_prototype = tf.reduce_mean(cls_features, axis=0)
        prototypes.append(cls_prototype)
    return tf.stack(prototypes)

import tensorflow as tf

# Assuming your dataset is already defined and named
'train_tf_dataset'
# Dataset example structure: <_RepeatDataset
element_spec=(TensorSpec(shape=(None, 240, 240, 3), dtype=tf.float32,
name=None), TensorSpec(shape=(None, 2), dtype=tf.float32, name=None))>

def split_support_query(dataset, support_size, query_size):
    support_set_images = []
    support_set_labels = []
    query_set_images = []
    query_set_labels = []

    for images, labels in dataset.take(1):  # Take the first batch for
simplicity
        # Flatten the batch dimension
        images = tf.reshape(images, [-1, 240, 240, 3])
        labels = tf.reshape(labels, [-1, 2])

        # Support set
        support_set_images.append(images[:support_size])
        support_set_labels.append(labels[:support_size])

        # Query set
        query_set_images.append(images[support_size:support_size +
query_size])
        query_set_labels.append(labels[support_size:support_size +
query_size])

    support_set_images = tf.concat(support_set_images, axis=0)
    support_set_labels = tf.concat(support_set_labels, axis=0)
    query_set_images = tf.concat(query_set_images, axis=0)
    query_set_labels = tf.concat(query_set_labels, axis=0)

    return (support_set_images, support_set_labels),
(query_set_images, query_set_labels)

# Define the number of examples
support_size = 22  # Number of examples per class for the support set
query_size = 10     # Number of examples per class for the query set

# Split the dataset into support and query sets
(support_set_images, support_set_labels), (query_set_images,
query_set_labels) = split_support_query(train_tf_dataset,
support_size, query_size)
```

```
# Convert labels from one-hot to class indices
# TODO: Look into using softmax here
support_set_labels = tf.argmax(support_set_labels, axis=1)
query_set_labels = tf.argmax(query_set_labels, axis=1)

print(len(query_set_labels))

# Loss function for few-shot learning
def prototype_loss(prototypes, query_features, query_labels,
num_classes):
    distances = tf.norm(tf.expand_dims(query_features, 1) -
tf.expand_dims(prototypes, 0), axis=2)
    logits = -distances
    return
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=q
uery_labels, logits=logits))

# Training step
def train_step(feature_extractor, support_set_images,
support_set_labels, query_set_images, query_set_labels, optimizer):
    with tf.GradientTape() as tape:
        support_set_features = feature_extractor(support_set_images,
training=True)
        query_set_features = feature_extractor(query_set_images,
training=True)

        prototypes = compute_prototypes(support_set_features,
support_set_labels, num_classes=2)
        loss = prototype_loss(prototypes, query_set_features,
query_set_labels, num_classes=2)

    gradients = tape.gradient(loss,
feature_extractor.trainable_variables)
    optimizer.apply_gradients(zip(gradients,
feature_extractor.trainable_variables))
    return loss

# Training loop
optimizer = tf.keras.optimizers.Adam()

for epoch in range(30):  # Number of epochs
    loss = train_step(
        few_shot_model,
        support_set_images,
        support_set_labels,
        query_set_images,
        query_set_labels,
        optimizer
```

```
    )
    print(f"Epoch {epoch+1}, Loss: {loss.numpy()}")

# Evaluate the model on the validation dataset
steps = len(test_dataset) // batch_size
few_shot_model.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
val_loss, val_accuracy = few_shot_model.evaluate(test_tf_dataset,
steps=steps)

# plt.figure(figsize=(8,8)) # Set our graph sizes to 8x8 for latter
# epoch_range = range(epochs)
# plt.plot(epoch_range, history.history['accuracy'], label="Training
Accuracy")
# plt.plot(epoch_range, history.history['val_accuracy'],
label="Validation Accuracy")
# plt.axis(ymin=0.4,ymax=1)
# plt.grid()
# plt.title('Model Accuracy')

# plt.ylabel('Accuracy')

# plt.xlabel('Epochs')

# plt.legend(['train', 'validation'])

print(f"Validation Loss: {val_loss}, Validation Accuracy:
{val_accuracy}")
fewShotTestAccuracy = val_accuracy

import matplotlib.pyplot as plt

# Data for the bar chart
models = ['Big Model', 'Naive Subset', 'Few-Shot']
accuracies = [bigModelTestAccuracy * 100, naieveSubsetTestAccuracy *
100, fewShotTestAccuracy * 100]

# Create the bar chart
plt.figure(figsize=(8, 6))
bars = plt.bar(models, accuracies, color=['blue', 'green', 'red'])

# Add title and labels
plt.title('Model Test Accuracies')
plt.xlabel('Model Type')
plt.ylabel('Accuracy (%)')

# Add accuracy values on top of each bar
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval + 1, f'{yval}%',
ha='center', va='bottom')
```

```
# Display the plot
plt.ylim(0, 100)  # Ensure y-axis starts from 0 to 100 to match the percentage range
plt.show()
```