Candicate Name: Nguyen Quoc Khanh

Email: nqkdeveloper@gmail.com

Exercise 1: – Image classification

Requirements

This is the instruction for AI intern assignment at Golden Owl. You will build a simple Image Classification model base on Teachable Machine or Tensorflow.

You will do "Classify dog and cat images". For example:

Image	Prediction
	Dog
	Cat

In this exercise, you need to write a program in Python to classify dog and cat images. For dataset that you can collect on internet

You can use teachable machine/your computer for training model

DEMO

Requirements

- Your program gets the input as image file and print the output.

- Describe briefly your solution in a report (i.e: your pipeline, algorithms that we use, remaining problem as well as ideas for improvements,...).
- Your solution will be evaluated on our private dataset. So, you need to submit your source and the report.
- Do not forget to write a README.md (i.e: environment requirements, how to run) to guide us to reproduce your code easily.

Dataset: [Kaggle Cats and Dogs Dataset](#) . The training archive contains 25,000 images of dogs and cats.

cats_vs_dogs

- **Description**:

A large set of images of cats and dogs. There are 1738 corrupted images that are dropped.

- **Additional Documentation**: Explore on Papers With Code ↗
- **Homepage**: https://www.microsoft.com/en-us/download/details.aspx?id=54765
- **Source code**: `tfds.image_classification.CatsVsDogs`
- **Versions**:
  - `4.0.0` (default): New split API (https://tensorflow.org/datasets/splits)
- **Download size**: `786.67 MiB`
- **Dataset size**: `689.64 MiB`
- **Auto-cached** (documentation): No
- **Splits**:

I will skip the data visualization part because the data is clean and perfect in everything (except corrupt image), and will focus on the requirements of the assignment.

Remove Corrupt Images

1. If the image opening process raises an exception or throws an error (e.g., IOError, SyntaxError, or UserWarning), it indicates that the image is corrupt.
2. Remove or exclude the corrupt image from further processing or analysis.

```
data_path = "/kaggle/working/dataset/PetImages"
categories = os.listdir(data_path)

all_data = []
for category in categories:
    category_path = os.path.join(data_path, category)
    filelist = os.listdir(category_path)
    data = {"imgpath": [], "labels": []}

    for file in filelist:
        fpath = os.path.join(category_path, file)
        try:
            with warnings.catch_warnings():
                warnings.filterwarnings("error")
                with Image.open(fpath) as img:
                    img.verify()
                    data["imgpath"].append(fpath)
                    data["labels"].append(category)
        except (IOError, SyntaxError, UserWarning) as e:
            continue

    all_data.append(data.copy())
```

Train-test-validation ratios

The train-test-validation split ratios would be approximately as follows:

- Train set: 80% of the total dataset.
- Validation set: 16% (20% of the remaining data after the train set).
- Test set: 4% (10% of the remaining data after the train and validation sets).

```
for category in categories:
    category_data = df[df['labels'] == category]
    num_samples = len(category_data)
    num_train_samples = int(0.8 * num_samples)  # 80% of data for training
    num_valid_samples = num_samples - num_train_samples  # 20% of data for validation

    train_samples = category_data.sample(n=num_train_samples, random_state=42)
    valid_samples = category_data.drop(train_samples.index)

    train_df = pd.concat([train_df, train_samples], ignore_index=True)
    valid_df = pd.concat([valid_df, valid_samples], ignore_index=True)
```

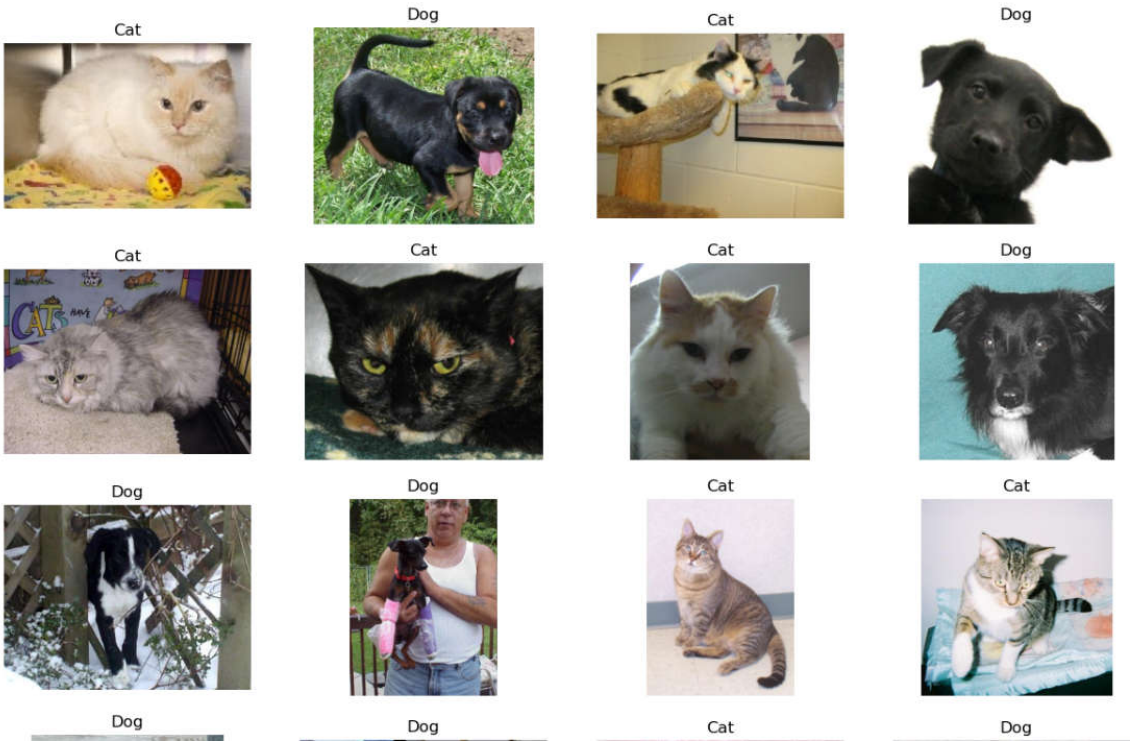
```
valid_df , test_df = train_test_split(valid_df ,  train_size= 0.90 , shuffle=True, random_state=124)
valid_df = valid_df.reset_index(drop=True)
test_df = test_df.reset_index(drop=True)
```

To summarize:

- The training set contains 19,997 samples.
- The validation set contains 4,500 samples.
- The test set contains 500 samples.

Preprocessing data

Sample data



Looking at a few random photos in the directory, we can see that the photos are color and have different shapes and sizes. We can also see photes where the cat/dog is barely visible and another that has two cats/dogs. This suggests that any classifier fit on this problem will have to be robust.

If we want to load all of the images into memory, we can estimate that it would require about 12 gigabytes of RAM.

That is 25,000 images with 200x200x3 pixels each, or 3,000,000,000 32-bit pixel values.

We could load all of the images, reshape them, and store them as a single NumPy array. This could fit into RAM on many modern machines, but not all, especially if we only have 8 gigabytes to work with.

But I have more than 12 gigabytes of RAM on Kaggle Notebook (19.5GB) so I load the images progressively using the **Keras ImageDataGenerator** class and flow_from_directory() API. This will be slower to execute but will run on more machines.

```
generator = ImageDataGenerator(
    preprocessing_function = tf.keras.applications.efficientnet.preprocess_input,
)
```

```
%%time

BATCH_SIZE = 25
IMAGE_SIZE = (224, 224)

# Split the data into three categories.
train_images = generator.flow_from_dataframe(
    dataframe=train_df,
    x_col='imgpath',
    y_col='labels',
    target_size=IMAGE_SIZE,
    color_mode='rgb',
    class_mode='categorical',
    batch_size=BATCH_SIZE,
    shuffle=True,
    seed=42,
)

val_images = generator.flow_from_dataframe(
    dataframe=valid_df,
    x_col='imgpath',
    y_col='labels',
    target_size=IMAGE_SIZE,
    color_mode='rgb',
    class_mode='categorical',
    batch_size=BATCH_SIZE,
    shuffle=False
)

test_images = generator.flow_from_dataframe(
    dataframe=test_df,
    x_col='imgpath',
    y_col='labels',
    target_size=IMAGE_SIZE,
    color_mode='rgb',
    class_mode='categorical',
    batch_size=BATCH_SIZE,
    shuffle=False
)
```

When the dataset include images with various size, we need to resize them into a shared size.

EfficientNet-B7 model architecture was originally designed and trained with a larger input size of 600x600 pixels to achieve optimal performance. However, it is possible to resize my input images to a smaller size such as 224x224 pixels if needed.

Resizing the input images to a smaller size can be beneficial in scenarios where I have memory constraints or require faster inference times..

I also maintained the aspect ratio to avoid distortions by using various image processing libraries such as PIL (Python Imaging Library) or OpenCV to resize images to the desired dimensions (e.g., 224x224 pixels) before feeding them into the EfficientNet-B7 model for inference.

Method

I have tried many different approaches such as Feature Extraction, Build CNN from Scratch, Fine Tuned,.. and the results are almost perfect (accuracy 99%) with the transfer learning approach with the models trained before.

A pre-trained model is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. I either use the pretrained model as is or use transfer learning to customize this model to a given task.

The intuition behind transfer learning for image classification is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model of the visual world. I can then take advantage of these learned feature maps without having to start from scratch by training a large model on a large dataset.

My pipline:

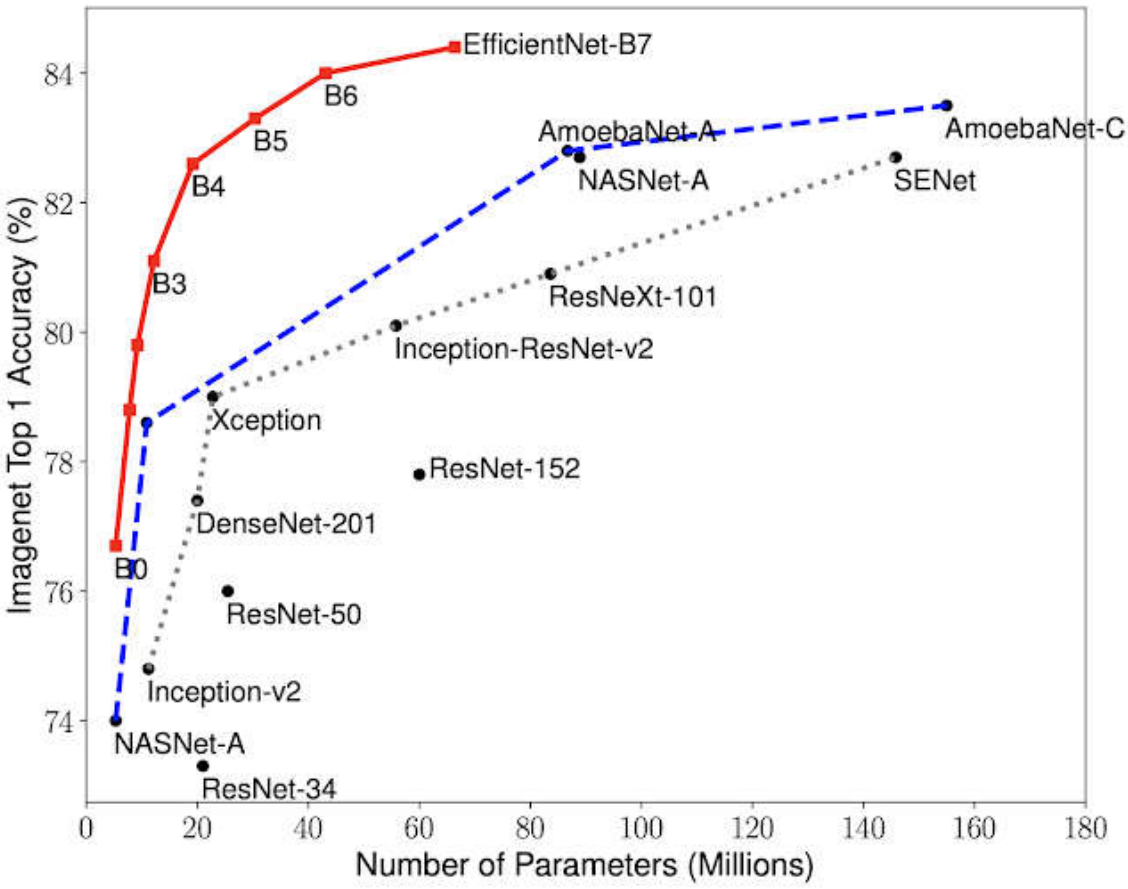
- Build an input pipeline, in this case using Keras ImageDataGenerator
- Compose the model
- Load in the pretrained base model (and pretrained weights)
- Stack the classification layers on top
- Train the model

- Evaluate model

In my solution, i will use **EfficientNet** (B7) architure. **EfficientNet** is a powerful deep-learning architecture that has achieved state-of-the-art performance on several image recognition tasks. Its unique combination of NAS and model scaling has allowed EfficientNet to optimize its architecture to achieve superior performance over other popular transfer learning networks.

EfficientNet Performance

Google AI Blog: “We have compared our EfficientNets with other existing CNNs on ImageNet. In general, the EfficientNet models achieve both higher accuracy and better efficiency over existing CNNs, reducing parameter size and FLOPS by an order of magnitude. For example, in the high-accuracy regime, our EfficientNet-B7 reaches state-of-the-art 84.4% top-1 / 97.1% top-5 accuracy on ImageNet, while being 8.4x smaller and 6.1x faster on CPU inference than the previous Gpipe. Compared with the widely used ResNet-50, our EfficientNet-B4 uses similar FLOPS, while improving the top-1 accuracy from 76.3% of ResNet-50 to 82.6% (+6.3%).”



Model Size vs. Accuracy Comparison. EfficientNet-B0 is the baseline network developed by AutoML MNAS, while Efficient-B1 to B7 are obtained by scaling up the baseline network. In particular, our EfficientNet-B7 achieves new state-of-the-art 84.4% top-1 / 97.1% top-5 accuracy, while being 8.4x smaller than the best existing CNN.

In my solution, I will try two ways to customize a pretrained model:

1. **Feature Extraction**: Use the representations learned by a previous network to extract meaningful features from new samples. I simply add a new classifier, which will be trained from scratch, on top of the pretrained model so that I can repurpose the feature maps learned previously for the dataset.

   I don't need to (re)train the entire model. The base convolutional network already contains features that are generically useful for

classifying pictures. However, the final, classification part of the pretrained model is specific to the original classification task, and subsequently specific to the set of classes on which the model was trained.

2. Fine-Tuning: Unfreeze a few of the top layers of a frozen model base and jointly train both the newly-added classifier layers and the last layers of the base model. This allows us to "fine-tune" the higher-order feature representations in the base model in order to make them more relevant for the specific task.

Create the base model from the pre-trained convnets

```
# Load the pretained model
pretrained_model = tf.keras.applications.EfficientNetB7(
    input_shape=(224, 224, 3),
    include_top=False, # we don't need a pre-trained top layer (output layer)
    weights='imagenet',
    pooling='max'
)
```

You will create the base model from the **EfficientNetB7** model This is pre-trained on the ImageNet dataset, a large dataset consisting of 1.4M images and 1000 classes. ImageNet is a research training dataset with a wide variety of categories like jackfruit and syringe. This base of knowledge will help me classify cats and dogs from our specific dataset.

First, I need to pick which layer of **EfficientNetB7** I will use for feature extraction. The very last classification layer (on "top", as most diagrams of machine learning models go from bottom to top) is not very useful. Instead, I will follow the common practice to depend on the very last layer before the flatten operation. This layer is called the "bottleneck layer". The bottleneck layer features retain more generality as compared to the final/top layer.

First, instantiate a **EfficientNetB7** model pre-loaded with weights trained on ImageNet. By specifying the **include_top=False** argument, I

load a network that doesn't include the classification layers at the top, which is ideal for feature extraction.

Feature extraction

In this step, I will freeze the convolutional base created from the previous step and to use as a feature extractor. Additionally, you add a classifier on top of it and train the top-level classifier.

```
# Freezing the layers of a pretrained neural network
for i, layer in enumerate(pretrained_model.layers):
    pretrained_model.layers[i].trainable = False
```

When you unfreeze a model that contains BatchNormalization layers in order to do fine-tuning, you should keep the BatchNormalization layers in inference mode by passing training = False when calling the base model. Otherwise, the updates applied to the non-trainable weights will destroy what the model has learned.

Build a model by chaining together the data augmentation, rescaling, base_model and feature extractor layers using the Keras Functional API. As previously mentioned, use training=False as my model contains a BatchNormalization layer.

```
inputs = layers.Input(shape = (224,224,3), name='inputLayer')
x = augment(inputs)
pretrain_out = pretrained_model(x, training = False)
x = layers.Dense(256)(pretrain_out)
x = layers.Activation(activation="relu")(x)
x = BatchNormalization()(x)
x = layers.Dropout(0.3)(x)
x = layers.Dense(num_classes)(x)
outputs = layers.Activation(activation="softmax", dtype=tf.float32, name='activationLayer')(x)
model = Model(inputs=inputs, outputs=outputs)
```

Here, I used**data augmentation**

When I don't have a large image dataset (about 20k for training), it's a good practice to artificially introduce sample diversity by applying random, yet realistic, transformations to the training images, such as rotation and horizontal flipping. This helps expose the model to different aspects of the training data and reduce overfitting.

```
# Data Augmentation Step
augment = tf.keras.Sequential([
  layers.experimental.preprocessing.RandomFlip("horizontal"),
  layers.experimental.preprocessing.RandomRotation(0.1),
  layers.experimental.preprocessing.RandomZoom(0.1),
  layers.experimental.preprocessing.RandomContrast(0.1),
], name='AugmentationLayer')
```

Compile the model

Compile the model before training it. As before, I used class_mode='categorical' so the loss='categorical_crossentropy' and metric for evaluate is accuracy.

```
model.compile(
    optimizer=Adam(0.0005),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Summary model

```
print(model.summary())
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (InputLayer)     [(None, 224, 224, 3)]     0

 AugmentationLayer (Sequenti  (None, 224, 224, 3)      0
 al)

 efficientnetb7 (Functional)  (None, 2560)             64097687

 dense (Dense)               (None, 256)               655616

 activation (Activation)     (None, 256)               0

 batch_normalization (BatchN  (None, 256)              1024
 ormalization)

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 2)                 514

 activationLayer (Activation  (None, 2)                0
 )

=================================================================
Total params: 64,754,841
Trainable params: 656,642
Non-trainable params: 64,098,199
_________________________________________________________________
```

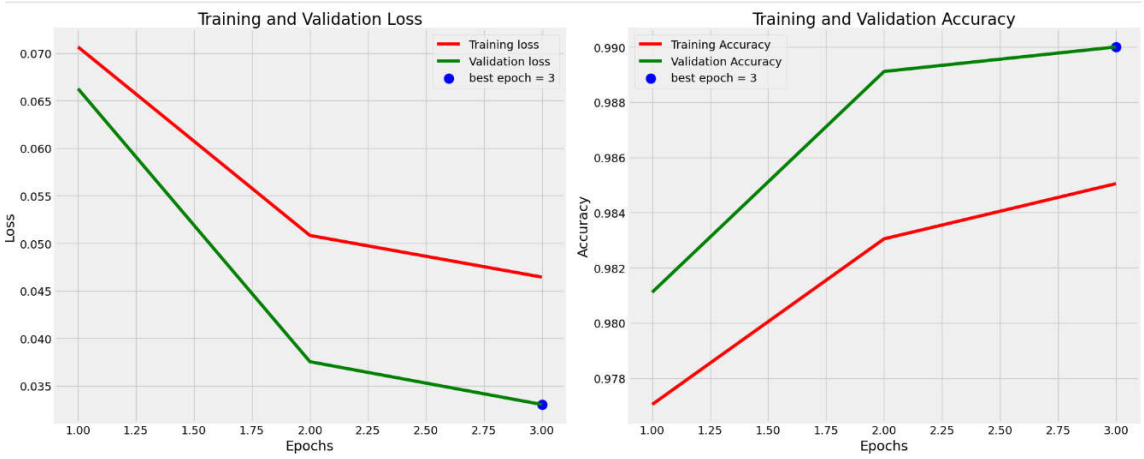
The ~65 million parameters in **EfficientNetB7** are frozen, but there are 656 thousand *trainable* parameters in the Dense layer. These are divided between two tf.Variable objects, the weights and biases.

Train the model

After training for 3 epochs, I saw ~99% accuracy on the validation set.

```
Epoch 1/3
800/800 [==============================] - 217s 231ms/step - loss: 0.0707 - accuracy: 0.9770 - val_loss: 0.0663 - val_accuracy: 0.9811 - lr: 5.0000e-04
Epoch 2/3
800/800 [==============================] - 179s 224ms/step - loss: 0.0508 - accuracy: 0.9830 - val_loss: 0.0375 - val_accuracy: 0.9891 - lr: 5.0000e-04
Epoch 3/3
800/800 [==============================] - 179s 224ms/step - loss: 0.0464 - accuracy: 0.9850 - val_loss: 0.0330 - val_accuracy: 0.9900 - lr: 5.0000e-04
```

Learning curves



Fine Tuning

In the feature extraction experiment, I were only training a few layers on top of an **EfficientNetB7** base model. The weights of the pre-trained network were **not** updated during training.

One way to increase performance even further is to train (or "fine-tune") the weights of the top layers of the pre-trained model alongside the training of the classifier I added. The training process will force the weights to be tuned from generic feature maps to features associated specifically with the dataset.

Un-freeze the top layers of the model

All I need to do is unfreeze the base_model and set the bottom layers to be un-trainable. Then, I recompile the model (necessary for these changes to take effect), and resume training.

```
# Enable pre-trained model training
pretrained_model.trainable = True

# Set BatchNormalization layers as non-trainable
for layer in pretrained_model.layers:
    if isinstance(layer, layers.BatchNormalization):
        layer.trainable = False
```

```
# Save the model weights
# model.save_weights('./checkpoints/my_checkpoint')
# Save the entire model
model.save('best_model.h5')
```

```
input_1 True
rescaling True
normalization True
rescaling_1 True
stem_conv_pad True
stem_conv True
stem_bn False
stem_activation True
block1a_dwconv True
block1a_bn False
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (InputLayer)     [(None, 224, 224, 3)]     0

 AugmentationLayer (Sequenti  (None, None, None, None)  0
 al)

 efficientnetb7 (Functional)  (None, 2560)             64097687

 dense (Dense)               (None, 256)               655616

 activation (Activation)     (None, 256)               0

 batch_normalization (BatchN  (None, 256)              1024
 ormalization)

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 2)                 514

 activationLayer (Activation  (None, 2)                0
 )

=================================================================
Total params: 64,754,841
Trainable params: 64,132,882
Non-trainable params: 621,959
```

Because I trained to convergence earlier (99%) , this step will improve my accuracy by a few percentage points.

Evaluation and prediction

Loss and acccuracy

```
# Evaluate the model on test data
results = model.evaluate(test_images, verbose=0)

# Print the test loss
print("Test Loss: {:.5f}".format(results[0]))

# Print the test accuracy as a percentage
print("Test Accuracy: {:.2f}%".format(results[1] * 100))
```

```
Test Loss: 0.01065
Test Accuracy: 99.80%
```

The provided test results indicate a very impressive performance for the model.

1. Test Loss: 0.01065 - The test loss represents the average loss value of the model's predictions compared to the true labels in the test set. In this case, the test loss value of 0.01065 suggests that, on average, the model's predictions were very close to the true labels, indicating a low level of error.
2. Test Accuracy: 99.80% - The test accuracy represents the percentage of correctly predicted labels out of all the test samples. In this case, the test accuracy of 99.80% indicates that the model achieved a very high level of accuracy in predicting the labels of the test data. It correctly classified 99.80% of the test samples.

Overall, the test results indicate that the model performed exceptionally well, achieving both a low test loss and a high test accuracy. It demonstrates that the model's predictions were highly accurate and reliable for the given task or dataset.

F1 Score / Recall / Precision

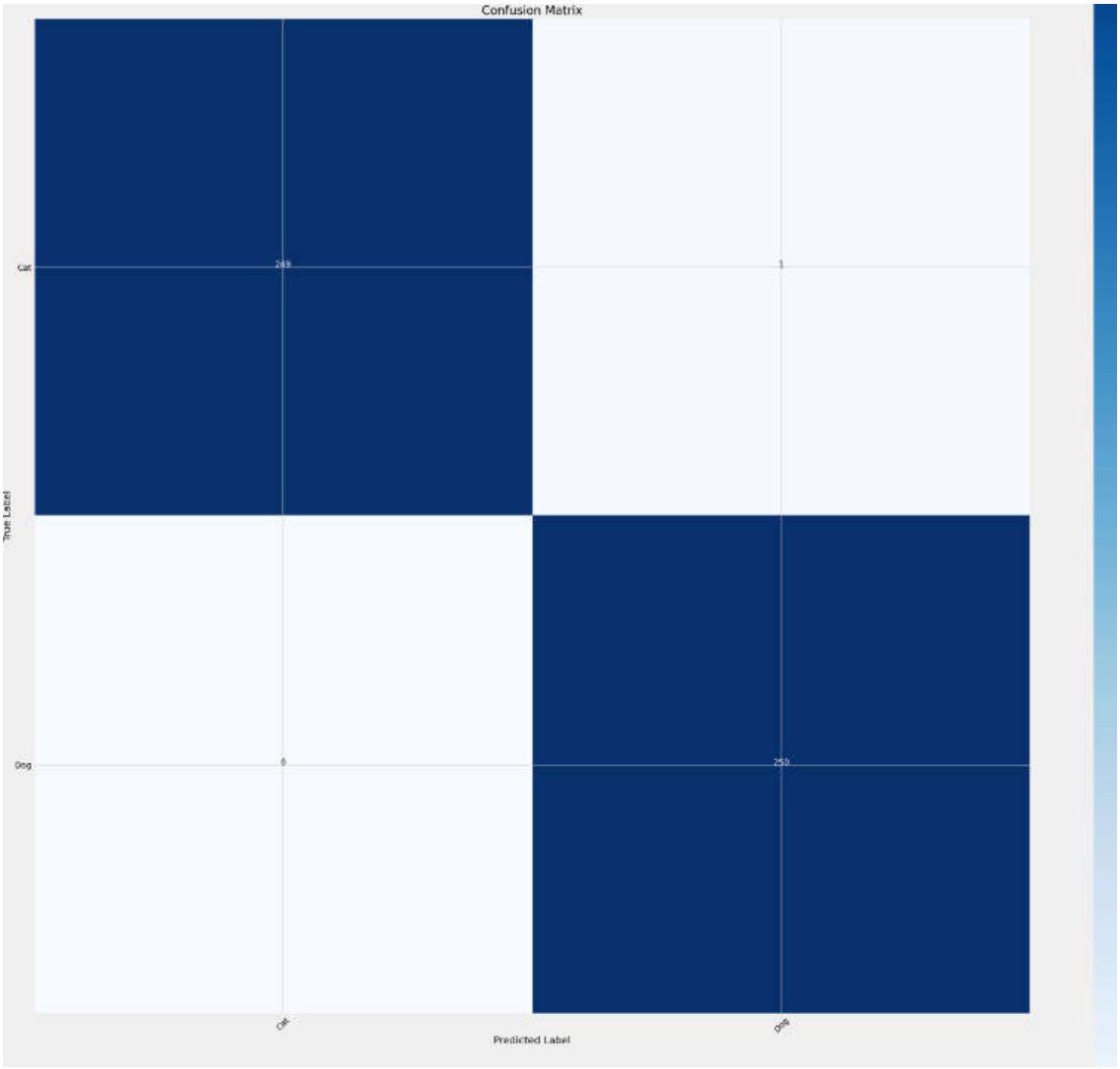
```
# Get the true labels and predicted labels
y_true = test_images.classes
y_pred = np.argmax(model.predict(test_images), axis=1)

# Calculate the F1 score
f1 = f1_score(y_true, y_pred, average='macro')

# Print the F1 score
print("F1 Score:", f1)

# Print the classification report
print(classification_report(y_true, y_pred, target_names=test_images.class_indices.keys()))
```

```
20/20 [==============================] - 9s 174ms/step
F1 Score: 0.997999991999968
              precision    recall  f1-score   support

         Cat       1.00      1.00      1.00       250
         Dog       1.00      1.00      1.00       250

    accuracy                           1.00       500
   macro avg       1.00      1.00      1.00       500
weighted avg       1.00      1.00      1.00       500
```

1. F1 Score: 0.997999991999968 - The F1 score is a metric that combines precision and recall into a single value. In this case, the F1 score of 0.997999991999968 suggests that the model achieved very high precision and recall, indicating an excellent overall performance.
2. Precision, Recall, and F1-Score: The precision, recall, and F1-score values are reported for each class (Cat and Dog). The precision represents the proportion of correctly predicted positive instances (True Positives) out of all instances predicted as positive. The recall represents the proportion of correctly predicted positive instances out of all actual positive instances. The F1-score is the harmonic mean of precision and recall.
    - For both the Cat and Dog classes, the precision, recall, and F1-score values are all 1.00, indicating perfect predictions for both classes. This means that all Cat and Dog samples were classified correctly.

3. Accuracy: The overall accuracy of the model is reported as 1.00 (100%). This means that the model correctly classified all 500 samples in the test set.

Confusion Matrix



In a perfect scenario where there are no misclassifications, the confusion matrix would show diagonal elements with all values being the same and non-diagonal elements being zero.

[[250, 0],

[0, 250]]

Testing

	Image Index	Test Labels	Test Classes	Prediction Labels	Prediction Classes	Path	Prediction Probability
0	0	1	Dog	1	Dog	/kaggle/working/dataset/PetImages/Dog/11009.jpg	0.999963
1	1	0	Cat	0	Cat	/kaggle/working/dataset/PetImages/Cat/870.jpg	0.999982
2	2	1	Dog	1	Dog	/kaggle/working/dataset/PetImages/Dog/9274.jpg	0.999836
3	3	0	Cat	0	Cat	/kaggle/working/dataset/PetImages/Cat/5128.jpg	0.999528
4	4	0	Cat	0	Cat	/kaggle/working/dataset/PetImages/Cat/1602.jpg	0.999742
5	5	0	Cat	0	Cat	/kaggle/working/dataset/PetImages/Cat/3616.jpg	0.999856
6	6	1	Dog	1	Dog	/kaggle/working/dataset/PetImages/Dog/1757.jpg	0.999945
7	7	0	Cat	0	Cat	/kaggle/working/dataset/PetImages/Cat/8977.jpg	0.999789
8	8	1	Dog	1	Dog	/kaggle/working/dataset/PetImages/Dog/6607.jpg	0.999957
9	9	1	Dog	1	Dog	/kaggle/working/dataset/PetImages/Dog/3033.jpg	0.999943

The testing results show a high level of accuracy and consistency in the model's predictions. Each row in the table represents a test sample, and the columns display the corresponding information.

A few general ideas for further improvement:

- **Increase dataset size**: Consider expanding your dataset with more diverse examples of dogs and cats. This can help improve the model's ability to generalize and make accurate predictions on unseen data.
- **Hyperparameter tuning**: Experiment with different hyperparameter settings such as learning rate, batch size, and optimizer. Hyperparameter tuning can help optimize the model's performance by finding the best configuration for your specific dataset and problem.
- **Ensemble methods**: Consider using ensemble methods such as model averaging or majority voting. Ensemble methods combine predictions from multiple models to improve overall accuracy and robustness.
- **Regularization techniques**: Apply regularization techniques like dropout or L1/L2 regularization to prevent overfitting and improve generalization.

Excercise 2: Text To Speech Ideas Report

Requirements

Text to speech (TTS) is a technology that converts text into spoken audio. It can read aloud PDFs, websites, and books using natural AI voices.. Text-to-speech (TTS) technology can be helpful for anyone who needs to access written content in an auditory format, and it can provide a more inclusive and accessible way of communication for many people.

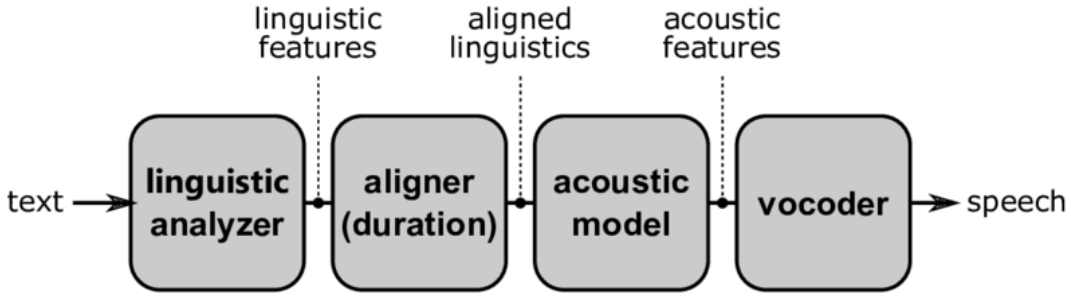
In this exercise, you are requested to propose your ideas for how to build TTS model (Vietnamese)

Requirements

- Proposes your ideas and describes them in a report (i.e: your pipeline, algorithms,...).
- Try to analyze the problem that we can face and how to solve them.

Text-to-speech (TTS) technology is widely used to convert written text into spoken audio. In this ideas report, the task requires me propose a pipeline and algorithms for building a TTS model specifically designed for the Vietnamese language. The goal is to develop a system that can accurately convert Vietnamese text into natural-sounding speech, improving accessibility and inclusivity for individuals who prefer auditory content.

Pipeline:



1. **Data Collection:**
    - Collect a large corpus of Vietnamese text data from various sources, including books, news articles, websites, and other relevant texts.
    - Ensure the data covers a wide range of topics and represents the diverse linguistic characteristics of Vietnamese.
2. **Text Preprocessing:**
    - Clean the collected text data by removing unnecessary symbols, punctuation, and formatting inconsistencies.
    - Perform word segmentation to separate words in Vietnamese sentences, as Vietnamese is a tonal language.
3. **Phoneme Extraction:**
    - Convert the preprocessed Vietnamese text into a sequence of phonemes, which are the basic units of speech sounds in the Vietnamese language.
    - Utilize a Vietnamese phonetic dictionary or a rule-based approach to map Vietnamese characters to corresponding phonemes.
4. **Acoustic Model:**

- Train an acoustic model using deep learning techniques such as Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), or Transformer models.
- Use the preprocessed text and corresponding phoneme sequences as input to the model.
- Train the acoustic model to predict the acoustic features of the speech, such as mel-spectrograms or linear spectrograms.

5. **Waveform Generation:**
    - Use a vocoder, such as WaveNet or Griffin-Lim, to convert the predicted acoustic features into a waveform that represents the actual speech signal.
    - The vocoder should produce high-quality and natural-sounding speech.
6. **Post-processing:**
    - Apply post-processing techniques to enhance the synthesized speech, such as prosody modification, emphasis, or adjusting the speaking rate.
    - Improve the output quality by smoothing transitions between phonemes and reducing artifacts.

Challenges and Solutions:

1. **Data Availability**: The availability of a large and diverse Vietnamese text corpus might be a challenge. One solution is to leverage existing publicly available Vietnamese text datasets and consider utilizing data augmentation techniques to increase the dataset size.

2. **Word Segmentation**: Vietnamese is a tonal language, and accurate word segmentation is crucial. Advanced word segmentation algorithms specifically designed for Vietnamese, such as Maximum Matching, Longest Matching, or Conditional Random Fields (CRF), can be employed to address this challenge.
3. **Pronunciation Variations**: Vietnamese has a complex phonetic system with many pronunciation variations. It is important to handle these variations by incorporating a comprehensive Vietnamese phonetic dictionary and considering context-based rules for phoneme mapping.
4. **Naturalness of Speech**: Generating high-quality and natural-sounding speech is a crucial aspect. State-of-the-art models like Tacotron, FastSpeech, or Parallel WaveGAN can be explored, and a large and diverse Vietnamese speech dataset can be used for training the models.