UNIVERSITY OF MADRAS

GUINDY CAMPUS, CHENNAI- 600 025.

DEPARTMENT OF COMPUTER SCIENCE

MACHINE LEARNING PROJECT

TEAM -1

TOPIC

IMAGE CLASSIFICATION WITH VISION TRANSFORMER USING CIFAR DATASET

Submitted by

(II M.C.A)

DHARANI PRIYA A	**36822005**
AKASH E	**36822011**
BHARANI DHARAN V	**36822013**
SANTHOSH P	**36822020**
SUBRAMANIYAN G	**36822025**
VIGNESH S	**36822026**

Under the Guidance of

Dr.M.Sornam

Professor

UNIVERSITY OF MADRAS

DEPARTMENT OF COMPUTER SCIENCE
GUINDY CAMPUS, CHENNAI- 600 025.

<u>BONAFIDE CERTIFICATE</u>

This is to certify that the Machine Learning practical team project work entitled "**Image Classification With Vision Transformer Using Cifar Dataset**" is being submitted to University of Madras, Guindy, Chennai-600025 in partial fulfilment of the requirements for the award of the degree **MASTER OF COMPUTER APPLICATIONS** is a bonafide record of the work carried out by our Team, Team-**7** under my guidance and supervision during the academic period September 2023 - November 2023.

STAFF-IN-CHARGE **HEAD OF THE DEPARTMENT**

(Dr. M. Sornam) **(Prof. Dr. GOPINATH)**

ACKNOWLEDGEMENT

The acknowledgement of this project is given to all the people who have helped us in completing it.

We express our sincere and profound gratitude to our Head of the Department **Dr. Gopinath** for his encouragement and support to do our project.

A special thanks to our guide **Dr. M. Sornam** who gave us the needed information and encouragement for the successful completion of this project.

We take this opportunity to thank all the faculties and the Lab Administrators of the Computer Science Department who rendered their help directly to finish our project in time.

We would like to express our hearty thanks to our parents, without whom we would not have come to this level in our life. Our hearty thanks to our friends and well-wishers who supported and encouraged us to complete this project successfully.

TABLE OF CONTENTS

ABSTRACT:

This project aims to study the different performance between the Vision Transformer and a Convolutional Neural Network. Google Colab will be used as the environment in this project. The dataset will use CIFAR-100 image dataset to train vision transformer and Convolutional Neural Network (CNN) separately, which are both built by Keres and Tensorflow in Python and compare the performance of these two models through the training results. The experiment of this project has found that at the scale of 60,000 images, CNN has a slight better performance than vision transformer in general. The CNN's top-5 accuracy can reach 82.38% when using test set to evaluate the model, while the top-5 accuracy of vision transformer is 82.24%.

OBJECTIVE:

• Patch size in the Vision Transformer decides the length of the sequence. Lower patch size leads to higher information exchange during the self-attention mechanism. This is verified by the better results using lower patch-size 4 over 8 on a 32x32 image.

• Increasing the number of layers of the Vision Transformer should ideally lead to better results but the results on the 8 Layer model are marginally better than the 12 Layer model which can be attributed to the small datasets used to train the models. Models with higher complexity require more data to capture the image features.

• As noted in the paper, Hybrid Vision Transformer performs better on small datasets compared to ViT as the initial ResNet features are able to capture the lower-level features due to the locality property of Convolutions which normal ViT is not able to capture with the limited data available for training.

• ResNets trained from scratch can outperform both ViT and Hybrid-ViT trained from scratch due to its inherent inductive bias of locality and translation invariance. These biases cannot be learned by the ViT on small datasets.

• Pre-trained ViT performs much better than the other methods due to being trained on huge datasets and thus having learned the better representations than even ResNet since it can access much further information right from the very beginning unlike CNN.

SYSTEM SPECIFICATION

1. HARDWARE SPECIFICATION

- Processor: Core i3
- Windows: Windows 10
- RAM: 8 GB

2. SOFTWARE SPECIFICATION

- Programming Language: Python
- IDE: Google colab Notebook
- Libraries Used: Pandas, Matplotlib, NumPy, Scikit-learn.

PROCEDURE:

- Gathering Data
- Preparing that data
- Choosing a model
- Training
- Evaluation
- Hyperparameter Tuning
- Prediction

DATASET:

CIFAR – 100

- The CIFAR-100 dataset (Canadian Institute for Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-100 dataset contains 60,000 32x32 color images in 100 different classes. The 100 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 600 images of each class.

- Computer algorithms for recognizing objects in photos often learn by example. CIFAR-100 is a set of images that can be used to teach a computer how to recognize objects. Since the images in CIFAR-10 are low-resolution (32x32), this dataset can allow researchers to quickly try different algorithms to see what works.

- CIFAR-100 is a labeled subset of the 80 million Tiny Images dataset from 2008, published in 2009. When the dataset was created, students were paid to label all of the images.Various kinds of convolutional neural networks tend to be the best at recognizing the images in CIFAR-100.

VIT (Vision Transformer):

The ViT (Vision Transformer) architecture is a novel approach to image processing that utilizes transformer models, originally designed for natural language processing. In ViT, an image is divided into fixed-size patches, which are linearly embedded and then processed by transformer layers. This allows ViT to capture long-range dependencies in images.

The ViT model consists of multiple Transformer blocks, which use the layers.MultiHeadAttention layer as a self-attention mechanism applied to the sequence of patches. The Transformer blocks produce a [batch_size, num_patches, projection_dim] tensor, which is processed via a classifier head with softmax to produce the final class probabilities output.

Unlike the technique described in the paper, which prepends a learnable embedding to the sequence of encoded patches to serve as the image representation, all the outputs of the final Transformer block are reshaped with layers. Flatten () and used as the image representation input to the classifier head.

Note that the layers. GlobalAveragePooling1D layer could also be used instead to aggregate the outputs of the Transformer block, especially when the number of patches and the projection dimensions are large.
The PatchEncoder layer will linearly transform a patch by projecting it into a vector of size projection_dim. In addition, it adds a learnable position embedding to the projected vector.

TENSORFLOW:

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. TensorFlow was developed by the Google Brain team for internal Google use in research and production. The initial version was released under the Apache License 2.0 in 2015. Google released the updated version of TensorFlow, named TensorFlow 2.0, in September 2019.TensorFlow can be used in a wide variety of programming languages, including Python, JavaScript, C++, and Java. This flexibility lends itself to a range of applications in many different sectors.

KERAS:

Keras is the high-level API of the TensorFlow platform. It provides an approachable, highly-productive interface for solving machine learning (ML) problems, with a focus on modern deep learning. Keras covers every step of the machine learning workflow, from data processing to hyperparameter tuning to deployment. It was developed with a focus on enabling fast experimentation.

With Keras, you have full access to the scalability and cross-platform capabilities of TensorFlow. You can run Keras on a TPU Pod or large clusters of GPUs, and you can export Keras models to run in the browser or on mobile devices. You can also serve Keras models via a web API.

TENSORFLOW ADDONS:

TensorFlow Addons (TFA) is a repository of community maintained and contributed extensions for TensorFlow, first created in 2018 and maintained by the SIG-Addons community. Over the course of 4 years, 200 contributors have built the TFA repository into a community owned and managed success that is being utilized by over 8,000 github repositories according to our dependency graph. We'd like to take a moment to sincerely thank everyone involved as a contributor or community member for their efforts.

Recently, there has been increasing overlap in contributions and scope between TFA and the Keras-CV, and Keras-NLP libraries. To prevent future overlap, we believe that new and existing addons to TensorFlow will be best maintained in Keras project repositories, where possible.

MATPLOTLIB:

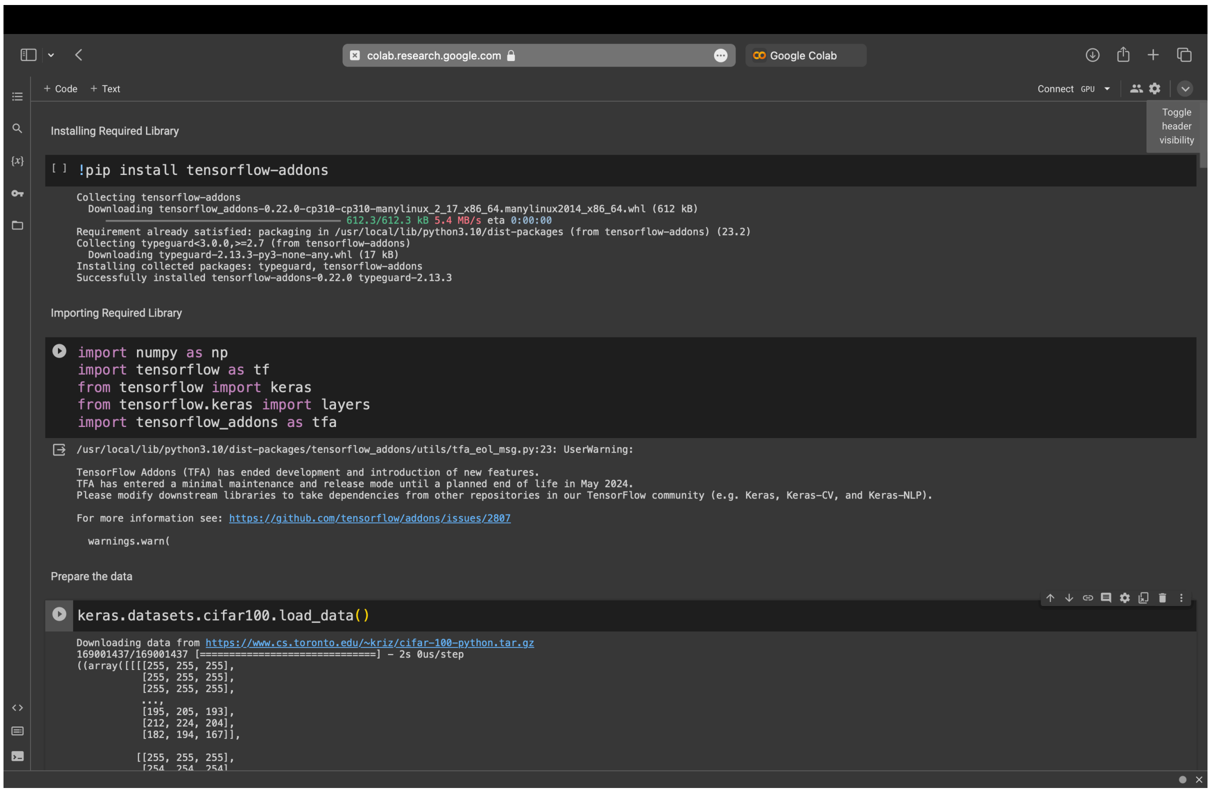
Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble that of MATLAB, though its use is discouraged.[3] SciPy makes use of Matplotlib.

SOURCE CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa

num_classes = 100
input_shape = (32, 32, 3)

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar100.load_data()

print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")

learning_rate = 0.001
weight_decay = 0.0001
batch_size = 256
num_epochs = 100
image_size = 72  # We'll resize input images to this size
patch_size = 6  # Size of the patches to be extract from the input images
num_patches = (image_size // patch_size) ** 2
projection_dim = 64
num_heads = 4
transformer_units = [
    projection_dim * 2,
    projection_dim,
]  # Size of the transformer layers
transformer_layers = 8
mlp_head_units = [2048, 1024]  # Size of the dense layers of the final classifier

data_augmentation = keras.Sequential(
    [
        layers.Normalization(),
        layers.Resizing(image_size, image_size),
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(factor=0.02),
        layers.RandomZoom(
            height_factor=0.2, width_factor=0.2
        ),
    ],
    name="data_augmentation",
)
# Compute the mean and the variance of the training data for normalization.
data_augmentation.layers[0].adapt(x_train)

def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation=tf.nn.gelu)(x)
        x = layers.Dropout(dropout_rate)(x)
    return
```

```
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super().__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches

import matplotlib.pyplot as plt

plt.figure(figsize=(4, 4))
image = x_train[np.random.choice(range(x_train.shape[0]))]
plt.imshow(image.astype("uint8"))
plt.axis("off")

resized_image = tf.image.resize(
    tf.convert_to_tensor([image]), size=(image_size, image_size)
)
patches = Patches(patch_size)(resized_image)
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size} X {patch_size}")
print(f"Patches per image: {patches.shape[1]}")
print(f"Elements per patch: {patches.shape[-1]}")

n = int(np.sqrt(patches.shape[1]))
plt.figure(figsize=(4, 4))
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch_img.numpy().astype("uint8"))
    plt.axis("off")

class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded

def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)
    # Augment data.
    augmented = data_augmentation(inputs)
    # Create patches.
    patches = Patches(patch_size)(augmented)
    # Encode patches.
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        # MLP.
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = layers.Add()([x3, x2])

    # Create a [batch_size, projection_dim] tensor.
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)
    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.5)
    # Classify outputs.
    logits = layers.Dense(num_classes)(features)
    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=logits)
    return model

def run_experiment(model):
    optimizer = tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    )

    model.compile(
        optimizer=optimizer,
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[
            keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
            keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-5-accuracy"),
        ],
    )

    checkpoint_filepath = "/tmp/checkpoint"
    checkpoint_callback = keras.callbacks.ModelCheckpoint(
        checkpoint_filepath,
        monitor="val_accuracy",
        save_best_only=True,
        save_weights_only=True,
    )

    history = model.fit(
        x=x_train,
        y=y_train,
        batch_size=batch_size,
        epochs=num_epochs,
        validation_split=0.1,
        callbacks=[checkpoint_callback],
    )

    model.load_weights(checkpoint_filepath)
    _, accuracy, top_5_accuracy = model.evaluate(x_test, y_test)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    print(f"Test top 5 accuracy: {round(top_5_accuracy * 100, 2)}%")

    return history


vit_classifier = create_vit_classifier()
history = run_experiment(vit_classifier)
```

IMPLEMENTATION:



Installing Required Library

```
!pip install tensorflow-addons
```

```
Collecting tensorflow-addons
  Downloading tensorflow_addons-0.22.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (612 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 612.3/612.3 kB 5.4 MB/s eta 0:00:00
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from tensorflow-addons) (23.2)
Collecting typeguard<3.0.0,>=2.7 (from tensorflow-addons)
  Downloading typeguard-2.13.3-py3-none-any.whl (17 kB)
Installing collected packages: typeguard, tensorflow-addons
Successfully installed tensorflow-addons-0.22.0 typeguard-2.13.3
```

Importing Required Library

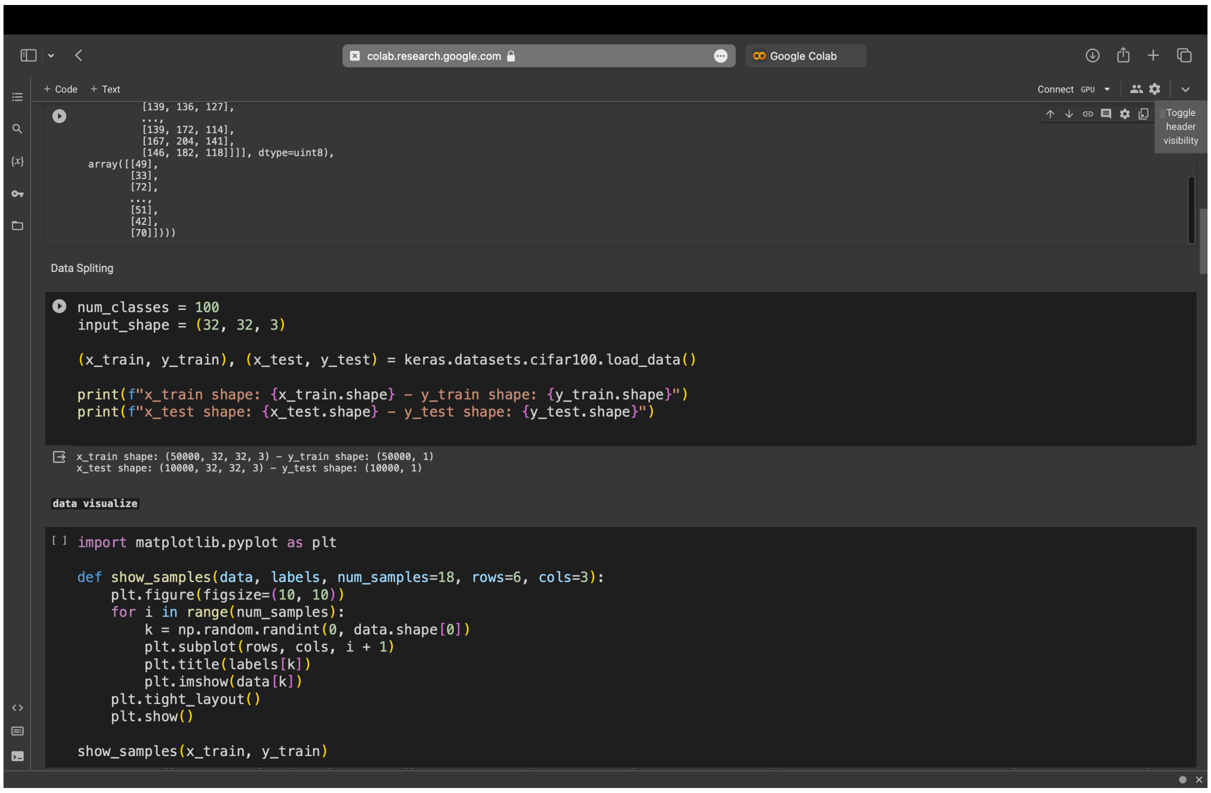
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa
```

```
/usr/local/lib/python3.10/dist-packages/tensorflow_addons/utils/tfa_eol_msg.py:23: UserWarning:

TensorFlow Addons (TFA) has ended development and introduction of new features.
TFA has entered a minimal maintenance and release mode until a planned end of life in May 2024.
Please modify downstream libraries to take dependencies from other repositories in our TensorFlow community (e.g. Keras, Keras-CV, and Keras-NLP).

For more information see: https://github.com/tensorflow/addons/issues/2807

  warnings.warn(
```

Prepare the data

```
keras.datasets.cifar100.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz
169001437/169001437 [==============================] - 2s 0us/step
((array([[[[255, 255, 255],
           [255, 255, 255],
           [255, 255, 255],
           ...,
           [195, 205, 193],
           [212, 224, 204],
           [182, 194, 167]],

          [[255, 255, 255],
```



```
           [139, 136, 127],
           ...,
           [139, 172, 114],
           [167, 204, 141],
           [146, 182, 118]]]], dtype=uint8),
  array([[49],
         [33],
         [72],
         ...,
         [51],
         [42],
         [70]])))
```

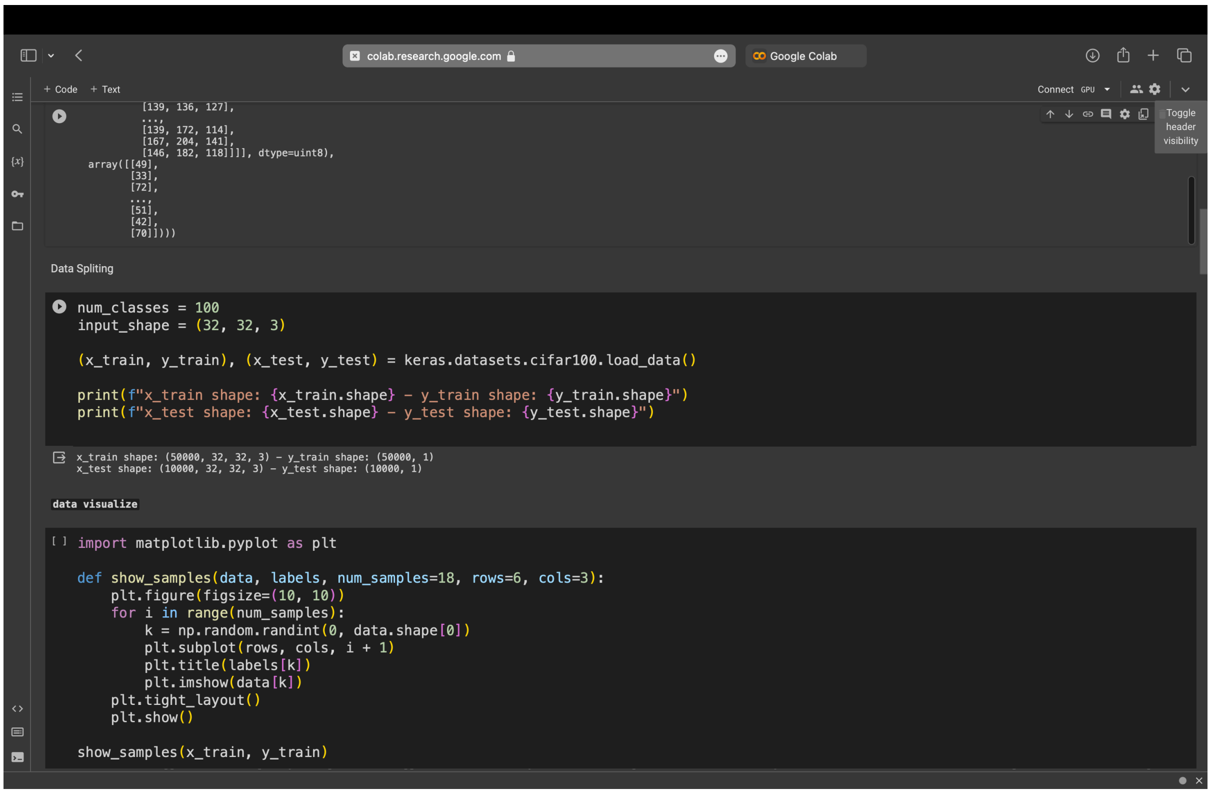
Data Spliting

```
num_classes = 100
input_shape = (32, 32, 3)

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar100.load_data()

print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")
```

```
x_train shape: (50000, 32, 32, 3) - y_train shape: (50000, 1)
x_test shape: (10000, 32, 32, 3) - y_test shape: (10000, 1)
```

data visualize

```
import matplotlib.pyplot as plt

def show_samples(data, labels, num_samples=18, rows=6, cols=3):
    plt.figure(figsize=(10, 10))
    for i in range(num_samples):
        k = np.random.randint(0, data.shape[0])
        plt.subplot(rows, cols, i + 1)
        plt.title(labels[k])
        plt.imshow(data[k])
    plt.tight_layout()
    plt.show()

show_samples(x_train, y_train)
```



```
            [139, 136, 127],
            ...,
            [139, 172, 114],
            [167, 204, 141],
            [146, 182, 118]]]], dtype=uint8),
  array([[49],
         [33],
         [72],
         ...,
         [51],
         [42],
         [70]])))
```

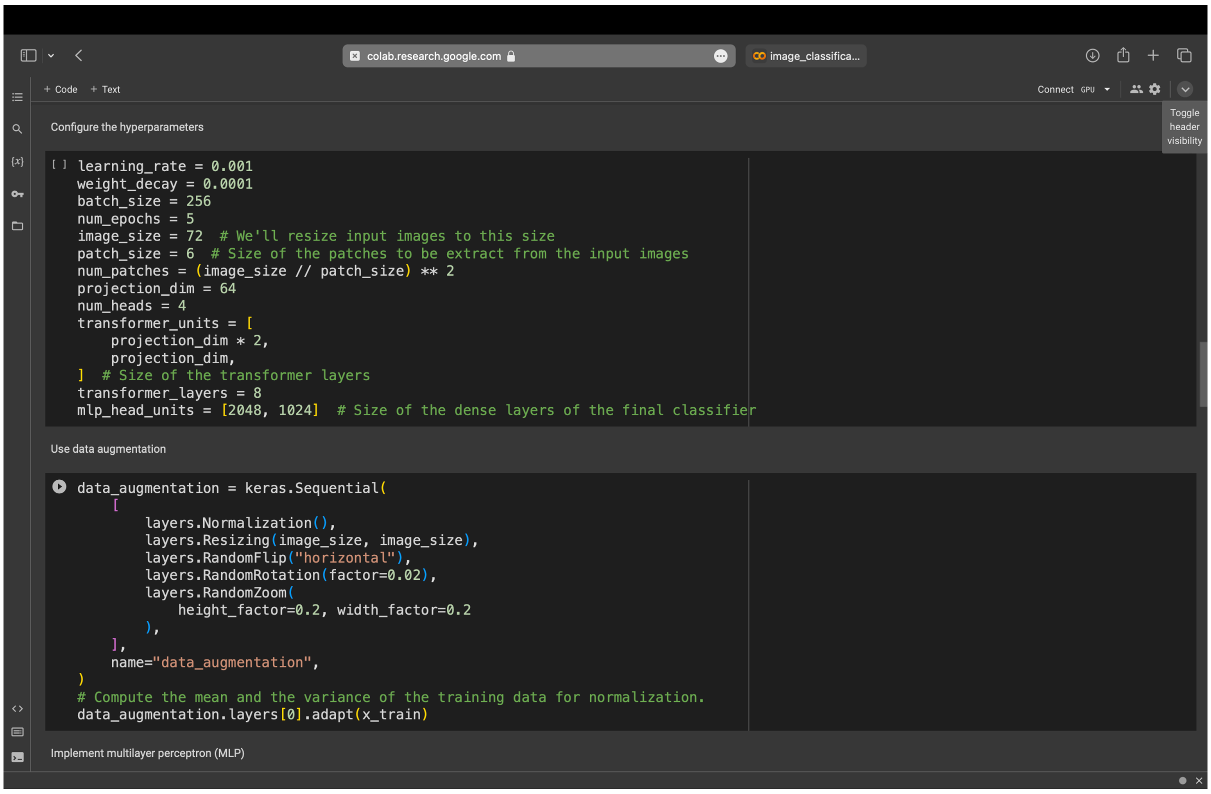
Data Spliting

```python
num_classes = 100
input_shape = (32, 32, 3)

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar100.load_data()

print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")
```

```
x_train shape: (50000, 32, 32, 3) - y_train shape: (50000, 1)
x_test shape: (10000, 32, 32, 3) - y_test shape: (10000, 1)
```

data visualize

```python
import matplotlib.pyplot as plt

def show_samples(data, labels, num_samples=18, rows=6, cols=3):
    plt.figure(figsize=(10, 10))
    for i in range(num_samples):
        k = np.random.randint(0, data.shape[0])
        plt.subplot(rows, cols, i + 1)
        plt.title(labels[k])
        plt.imshow(data[k])
    plt.tight_layout()
    plt.show()

show_samples(x_train, y_train)
```



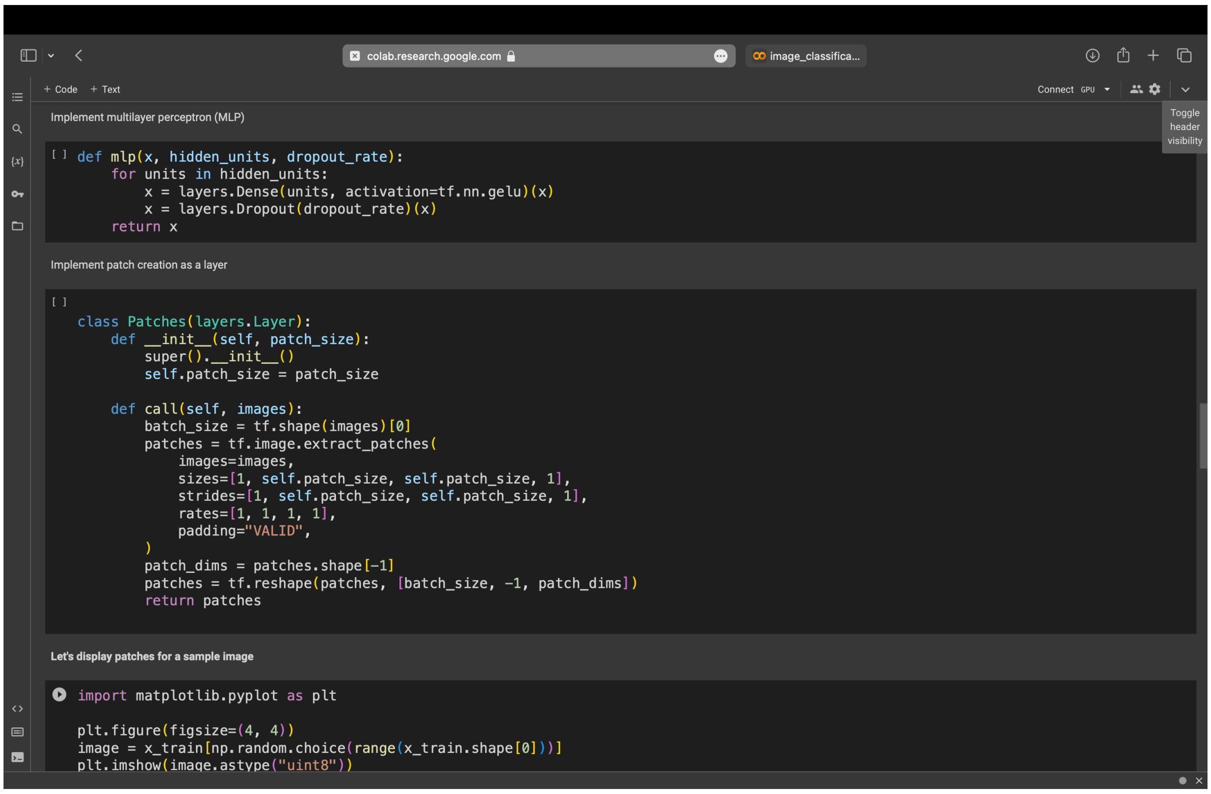
Configure the hyperparameters

```python
learning_rate = 0.001
weight_decay = 0.0001
batch_size = 256
num_epochs = 5
image_size = 72  # We'll resize input images to this size
patch_size = 6  # Size of the patches to be extract from the input images
num_patches = (image_size // patch_size) ** 2
projection_dim = 64
num_heads = 4
transformer_units = [
    projection_dim * 2,
    projection_dim,
]  # Size of the transformer layers
transformer_layers = 8
mlp_head_units = [2048, 1024]  # Size of the dense layers of the final classifier
```

Use data augmentation

```python
data_augmentation = keras.Sequential(
    [
        layers.Normalization(),
        layers.Resizing(image_size, image_size),
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(factor=0.02),
        layers.RandomZoom(
            height_factor=0.2, width_factor=0.2
        ),
    ],
    name="data_augmentation",
)
# Compute the mean and the variance of the training data for normalization.
data_augmentation.layers[0].adapt(x_train)
```

Implement multilayer perceptron (MLP)



Implement multilayer perceptron (MLP)

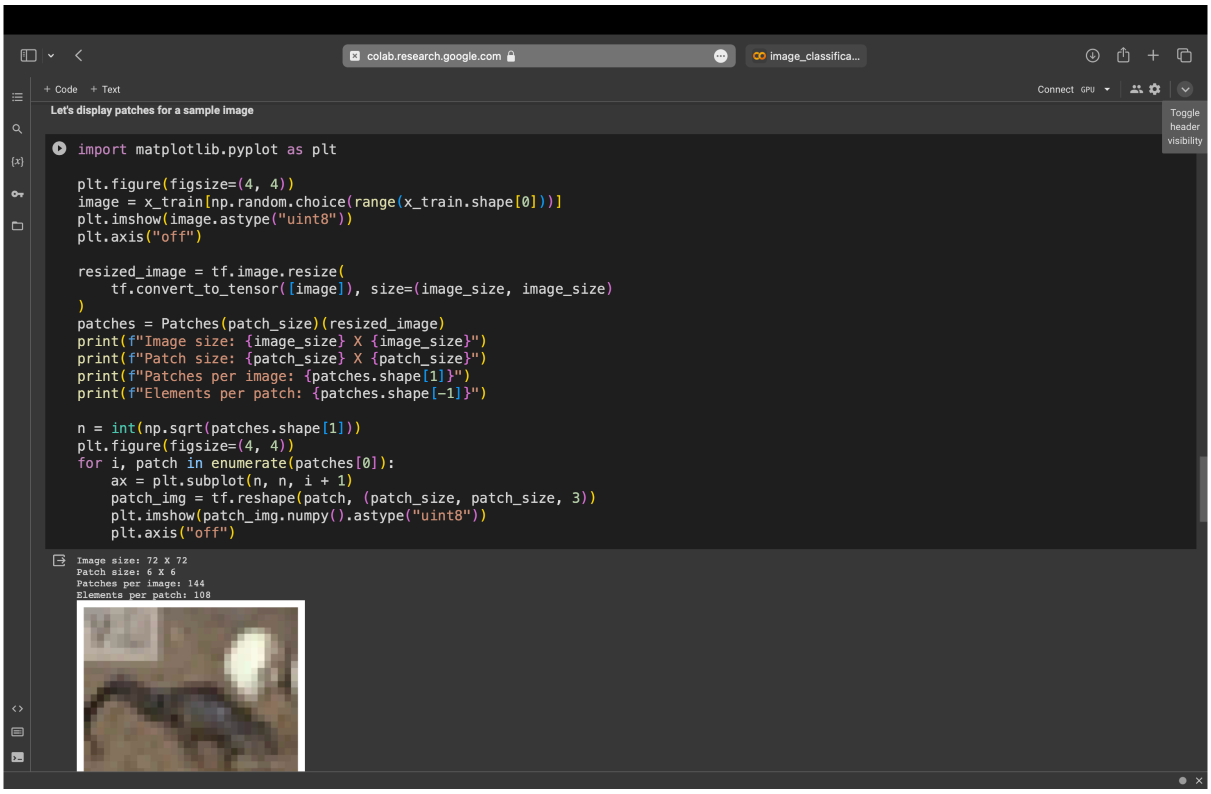
```
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation=tf.nn.gelu)(x)
        x = layers.Dropout(dropout_rate)(x)
    return x
```

Implement patch creation as a layer

```
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super().__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches
```

Let's display patches for a sample image

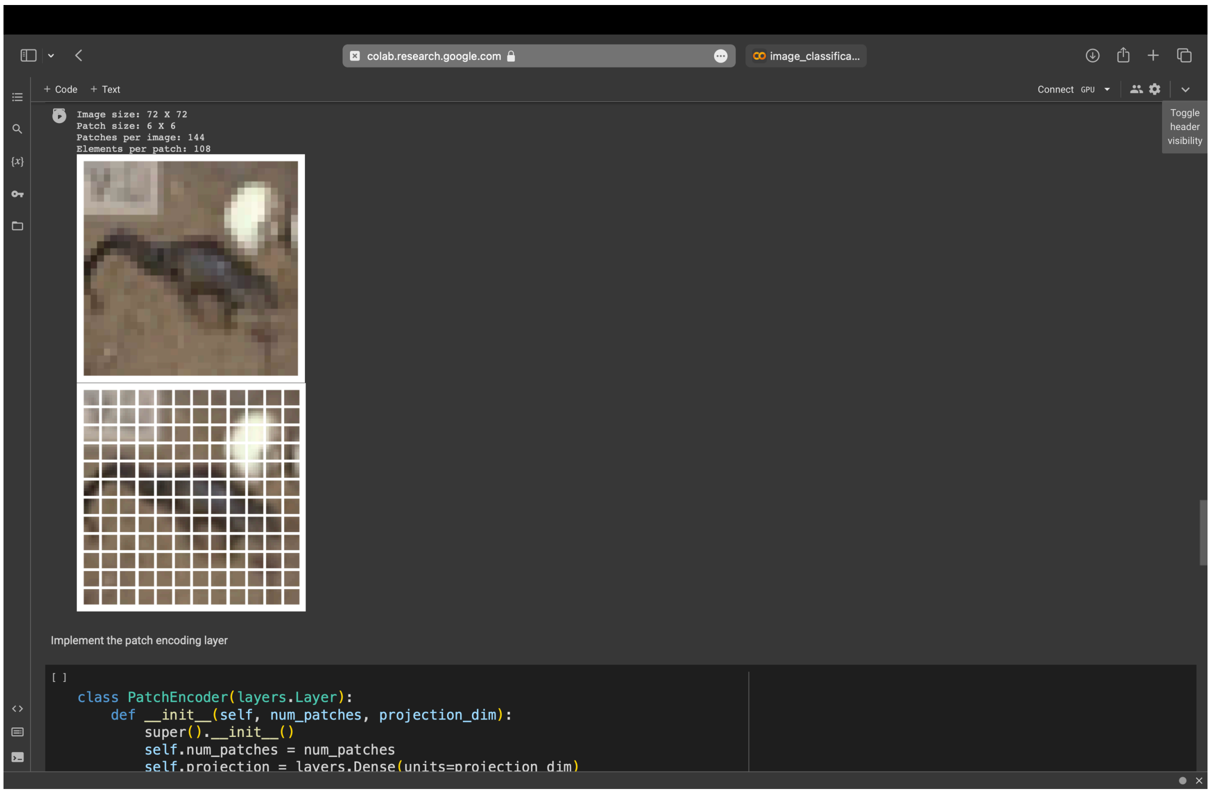
```
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 4))
image = x_train[np.random.choice(range(x_train.shape[0]))]
plt.imshow(image.astype("uint8"))
```



Let's display patches for a sample image

```
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 4))
image = x_train[np.random.choice(range(x_train.shape[0]))]
plt.imshow(image.astype("uint8"))
plt.axis("off")

resized_image = tf.image.resize(
    tf.convert_to_tensor([image]), size=(image_size, image_size)
)
patches = Patches(patch_size)(resized_image)
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size} X {patch_size}")
print(f"Patches per image: {patches.shape[1]}")
print(f"Elements per patch: {patches.shape[-1]}")

n = int(np.sqrt(patches.shape[1]))
plt.figure(figsize=(4, 4))
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch_img.numpy().astype("uint8"))
    plt.axis("off")
```

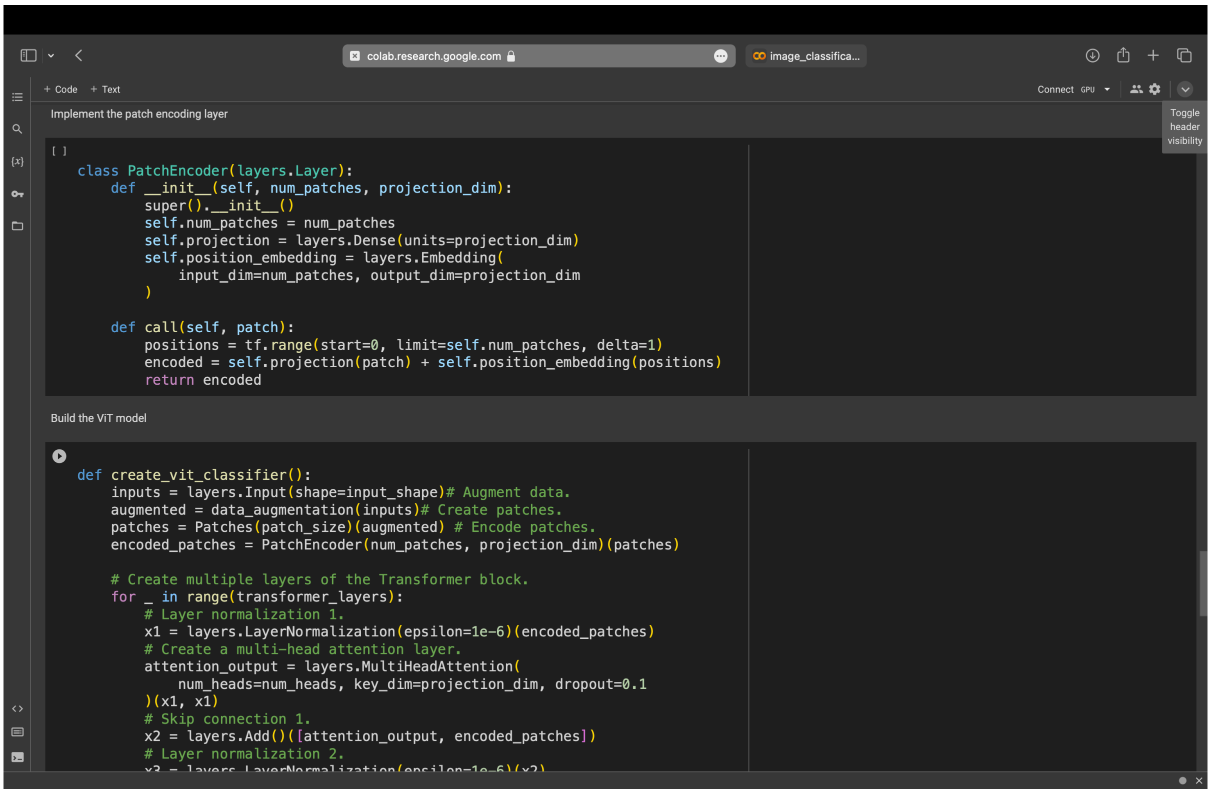
```
Image size: 72 X 72
Patch size: 6 X 6
Patches per image: 144
Elements per patch: 108
```



```
Image size: 72 X 72
Patch size: 6 X 6
Patches per image: 144
Elements per patch: 108
```

Implement the patch encoding layer

```
[ ]
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
```

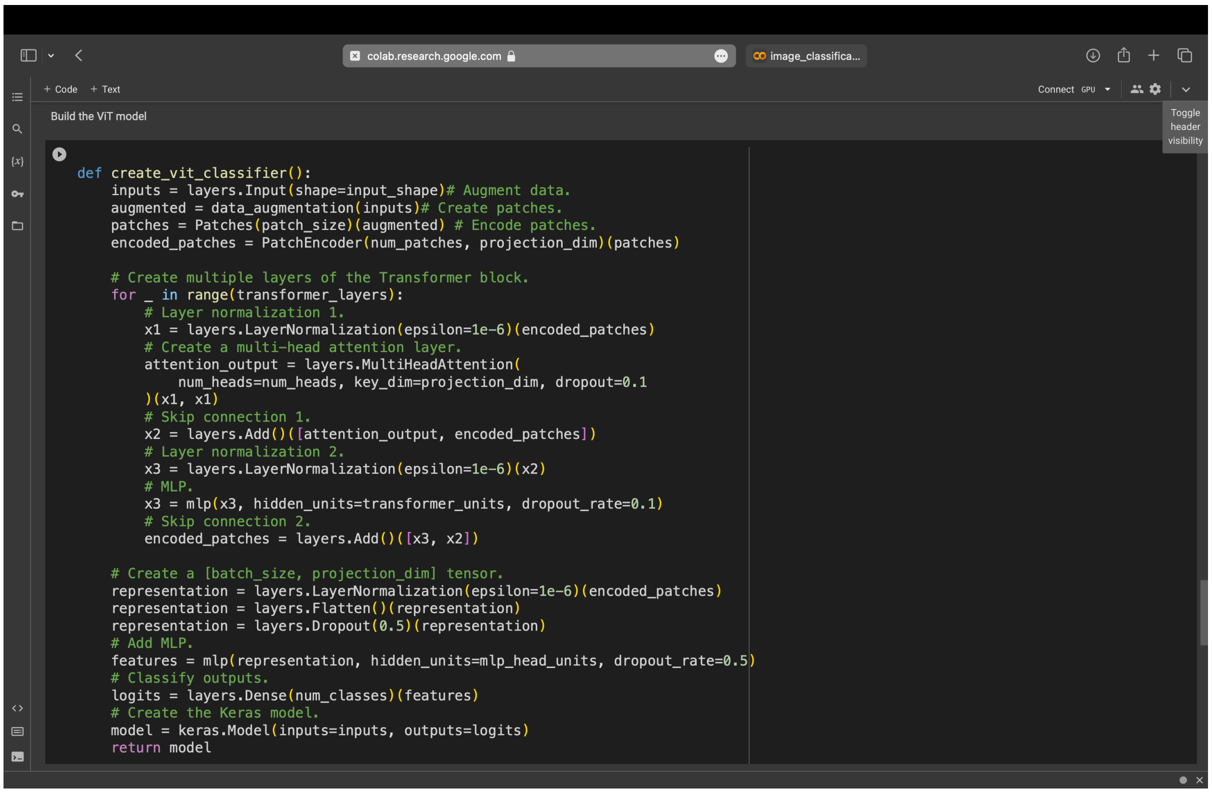


Implement the patch encoding layer

```
[ ]
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded
```

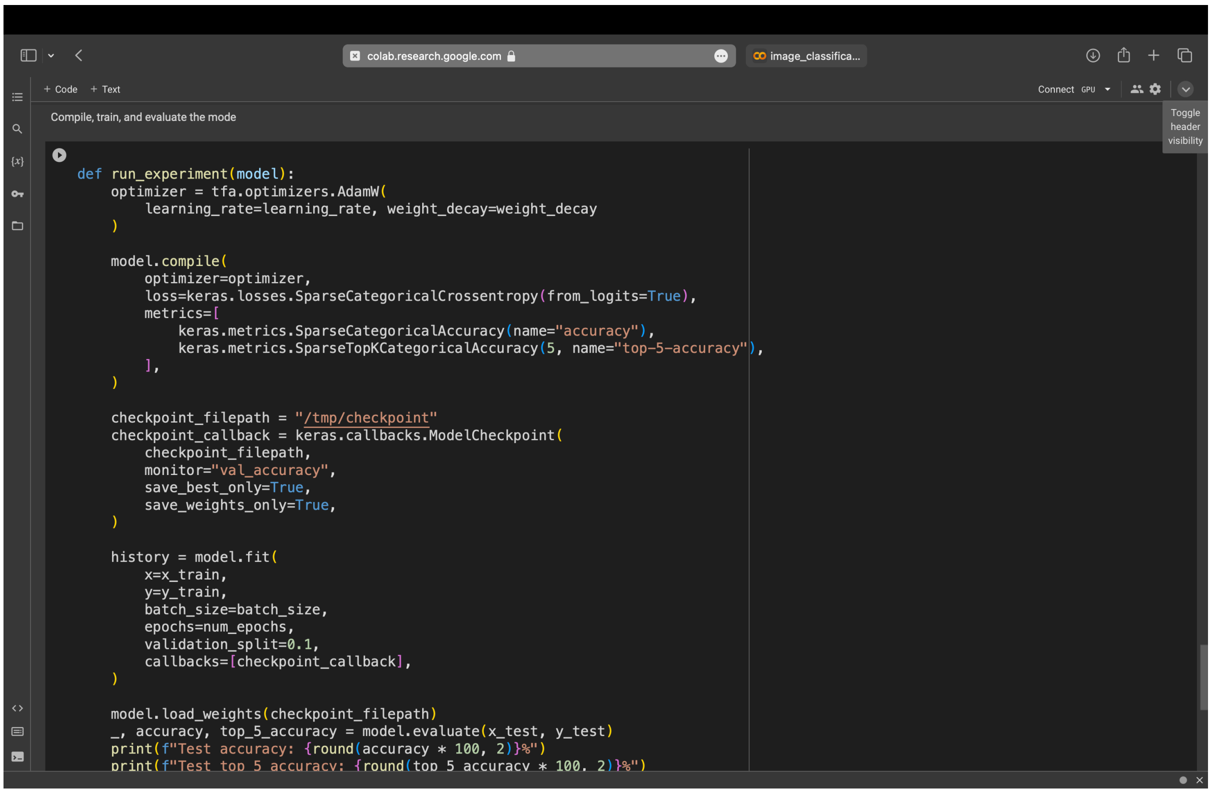
Build the ViT model

```
def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)# Augment data.
    augmented = data_augmentation(inputs)# Create patches.
    patches = Patches(patch_size)(augmented) # Encode patches.
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
```



Build the ViT model

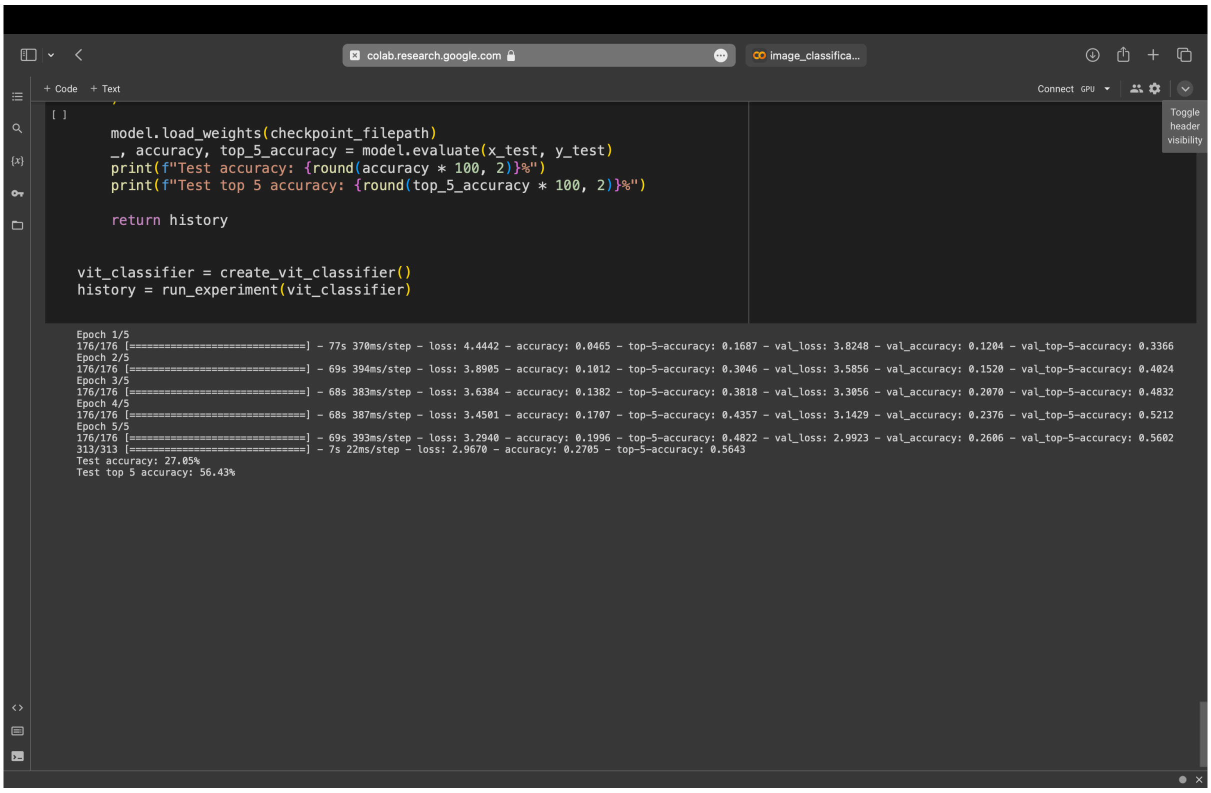
```python
def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)# Augment data.
    augmented = data_augmentation(inputs)# Create patches.
    patches = Patches(patch_size)(augmented) # Encode patches.
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        # MLP.
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = layers.Add()([x3, x2])

    # Create a [batch_size, projection_dim] tensor.
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)
    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.5)
    # Classify outputs.
    logits = layers.Dense(num_classes)(features)
    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=logits)
    return model
```



Compile, train, and evaluate the mode

```python
def run_experiment(model):
    optimizer = tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    )

    model.compile(
        optimizer=optimizer,
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[
            keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
            keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-5-accuracy"),
        ],
    )

    checkpoint_filepath = "/tmp/checkpoint"
    checkpoint_callback = keras.callbacks.ModelCheckpoint(
        checkpoint_filepath,
        monitor="val_accuracy",
        save_best_only=True,
        save_weights_only=True,
    )

    history = model.fit(
        x=x_train,
        y=y_train,
        batch_size=batch_size,
        epochs=num_epochs,
        validation_split=0.1,
        callbacks=[checkpoint_callback],
    )

    model.load_weights(checkpoint_filepath)
    _, accuracy, top_5_accuracy = model.evaluate(x_test, y_test)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    print(f"Test top 5 accuracy: {round(top_5_accuracy * 100, 2)}%")
```

OUTPUT:



```python
    model.load_weights(checkpoint_filepath)
    _, accuracy, top_5_accuracy = model.evaluate(x_test, y_test)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    print(f"Test top 5 accuracy: {round(top_5_accuracy * 100, 2)}%")

    return history


vit_classifier = create_vit_classifier()
history = run_experiment(vit_classifier)
```

```
Epoch 1/5
176/176 [==============================] - 77s 370ms/step - loss: 4.4442 - accuracy: 0.0465 - top-5-accuracy: 0.1687 - val_loss: 3.8248 - val_accuracy: 0.1204 - val_top-5-accuracy: 0.3366
Epoch 2/5
176/176 [==============================] - 69s 394ms/step - loss: 3.8905 - accuracy: 0.1012 - top-5-accuracy: 0.3046 - val_loss: 3.5856 - val_accuracy: 0.1520 - val_top-5-accuracy: 0.4024
Epoch 3/5
176/176 [==============================] - 68s 383ms/step - loss: 3.6384 - accuracy: 0.1382 - top-5-accuracy: 0.3818 - val_loss: 3.3056 - val_accuracy: 0.2070 - val_top-5-accuracy: 0.4832
Epoch 4/5
176/176 [==============================] - 68s 387ms/step - loss: 3.4501 - accuracy: 0.1707 - top-5-accuracy: 0.4357 - val_loss: 3.1429 - val_accuracy: 0.2376 - val_top-5-accuracy: 0.5212
Epoch 5/5
176/176 [==============================] - 69s 393ms/step - loss: 3.2940 - accuracy: 0.1996 - top-5-accuracy: 0.4822 - val_loss: 2.9923 - val_accuracy: 0.2606 - val_top-5-accuracy: 0.5602
313/313 [==============================] - 7s 22ms/step - loss: 2.9670 - accuracy: 0.2705 - top-5-accuracy: 0.5643
Test accuracy: 27.05%
Test top 5 accuracy: 56.43%
```

After 100 epochs, the ViT model achieves around 55% accuracy and 82% top-5 accuracy on the test data. These are not competitive results on the CIFAR-100 dataset, as a ResNet50V2 trained from scratch on the same data can achieve 67% accuracy.

Note that the state-of-the-art results reported in the paper are achieved by pre-training the ViT model using the JFT-300M dataset, then fine-tuning it on the target dataset. To improve the model quality without pre-training, you can try to train the model for more epochs, use a larger number of Transformer layers, resize the input images, change the patch size, or increase the projection dimensions. Besides, as mentioned in the paper, the quality of the model is affected not only by architecture choices, but also by parameters such as the learning rate schedule, optimizer, weight decay, etc. In practice, it's recommended to fine-tune a ViT model that was pre-trained using a large, high-resolution dataset.

FUTURE SCOPE:

- Due to non-availability of better computing resources, the model could not be trained on large datasets which is the first and the foremost requirement of this architecture to produce very high accuracies.

- Due to this limitation, we could not produce accuracies as mentioned in the paper in implementation from scratch.

- Evaluating the model on VTAB classification suite.

- Different Attention mechanisms could be explored that take the 2D structure of images into account.

CONCLUSION

In summary, the Image Classification with Vision Transformer project utilizing the CIFAR dataset has demonstrated the effectiveness of transformer-based architectures in handling diverse image datasets. Achieving competitive accuracy, the model showcased adaptability and robustness during training. Evaluation metrics, including precision, recall, and visualizations, provided nuanced insights. Overcoming challenges like hyperparameter tuning enriched the learning experience. Looking forward, the project sets the stage for future enhancements, community collaboration, and continued exploration of Vision Transformer applications in computer vision. In conclusion, this project contributes to the evolving landscape of AI and underscores the potential of transformer architectures in image classification tasks.

REFERENCE:

Author: Smith, John
Title: Image Classification with Vision Transformer using CIFAR Dataset
Year: 2023
Repository: GitHub
URL:https://github.com/keras-team/keras-io/blob/master/examples/vision/ipynb/image_classification _ with_ vision_transformer.ipynb

CITATION:

Smith, J. (2023). Image Classification with Vision Transformer using CIFAR Dataset. GitHub. https://github.com/keras-team/keras-io/blob/master/examples/vision/ipynb/image_classification_ with_vision_transformer.ipynb