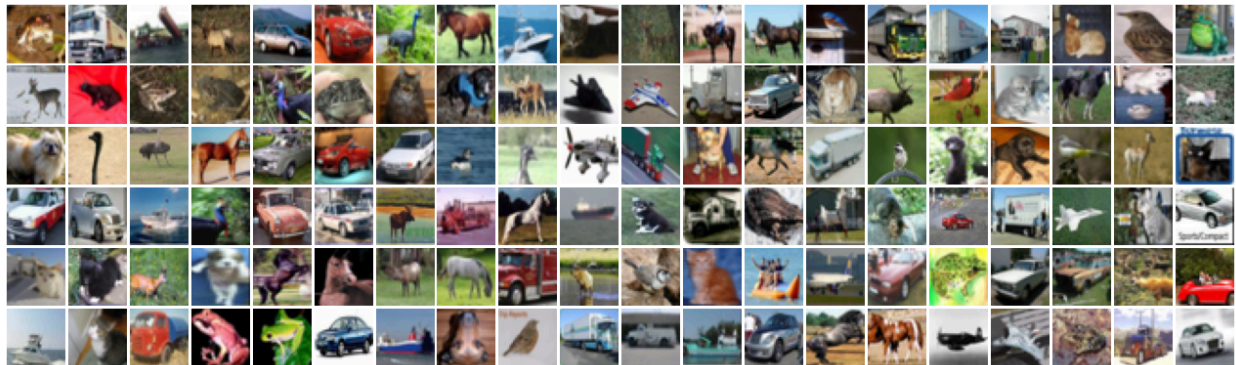


VISION TRANSFORM

BY THE CIFAR 10 DATASET



ABSTRACT

This project implements a Vision Transformer (ViT) for image classification using the CIFAR-10 dataset. This is a type of deep learning model that has shown good results in computer vision tasks.

The code starts by installing the necessary libraries like TensorFlow, Keras and TensorFlow Addons. Then, it loads the CIFAR-10 dataset, defines the model parameters and performs data augmentation. The core of the project is building the ViT model, which involves splitting images into patches, encoding them, applying transformer layers and finally classifying the image.

OBJECTIVE

- Implement a Vision Transformer (ViT) model for image classification. This involves understanding the architecture and building the model using libraries like TensorFlow and Keras.
- Train and evaluate the ViT model on the CIFAR-10 dataset. This includes data preprocessing, augmentation, training, and assessing the model's performance using metrics like accuracy.
- Explore the effectiveness of ViT for image classification on a smaller dataset. The CIFAR-10 dataset, with its limited data, allows us to study how ViT performs in such scenarios.
- Gain practical experience in implementing a relatively new deep learning architecture. ViT is a recent advancement in the field, and this project offers hands-on experience with this architecture.

Introduction

This project implements a Vision Transformer (ViT) for image classification using the CIFAR-10 dataset. This is a type of deep learning model that has shown good results in computer vision tasks.

The code starts by installing the necessary libraries like TensorFlow, Keras and TensorFlow Addons. Then, it loads the CIFAR-10 dataset, defines the model parameters and performs data augmentation. The core of the project is building the ViT model, which involves splitting images into patches, encoding them, applying transformer layers and finally classifying the image.

CIFAR 10

The CIFAR-10 dataset is a collection of small images commonly used to train machine learning and computer vision algorithms. Here's a summary of its key characteristics:

- Number of Images: 60,000
- Image Size: 32x32 pixels
- Color: Color images
- Number of Classes: 10
- Classes: Airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck
- Training/Test Split: 50,000 training images and 10,000 test images

The CIFAR-10 dataset is widely used for its manageable size and diverse set of classes, making it suitable for experimenting with different deep learning models and techniques.

METHODOLOGY

1. Environment Setup:

Install necessary libraries like TensorFlow, Keras, and TensorFlow Addons.

2. Dataset Preparation:

- a. Load the CIFAR-10 dataset using `"keras.datasets.cifar10.load_data()"`.
- b. Normalize pixel values to be in the range [0, 1].

3. Data Augmentation:

- a. Use `keras.Sequential` to define a data augmentation pipeline.
- b. Include random flipping, cropping, and color jittering.

4. Vision Transformer (ViT) Model Implementation:

- **Patching:**

Divide input images into smaller patches.

- **Patch Encoding:**

Flatten patches and project them to a higher dimensional space. Add positional embeddings.

- **Transformer Encoder:**

Utilize multiple transformer layers with multi-head attention and MLP blocks.

- **Classification Head:**

Add an MLP with a final classification layer.

5. Model Training:

- a. Define an optimizer (e.g., AdamW) and learning rate schedule.
- b. Compile the model with an appropriate loss function (e.g., sparse categorical cross-entropy).
- c. Train the model on the training data, potentially using techniques like early stopping and learning rate decay.

6. Model Evaluation:

- a. Evaluate the trained model on the CIFAR-10 test set.
- b. Report metrics such as accuracy, top-5 accuracy, precision, recall, and F1-score.

Algorithm

1. Split Image into Patches:

Divide the input image into fixed-size patches.

2. Patch Embedding:

- a. Flatten each patch into a 1D vector.
- b. Apply a linear transformation to map the vector to a higher-dimensional embedding space.
- c. Add positional embeddings to encode the location of each patch within the original image.

3. Transformer Encoder:

a. Layer Normalization:

Normalize the input embeddings.

b. Multi-Head Self-Attention:

Calculate attention weights to capture relationships between different patches.

c. Add & Norm:

Add the attention output to the input embeddings and perform layer normalization.

d. Feed Forward Network:

Apply a fully connected network to each embedding.


e. Add & Norm:

Add the feed forward output to the previous output and perform layer normalization.

4. Transformer Encoder:


- Extract the embedding corresponding to a special classification token (added during embedding).
- Pass this embedding through a Multilayer Perceptron (MLP) head.
- Apply a softmax function to obtain class probabilities.

CODE

```
 # prompt: install tensorflow 2.8.0, keras=2.8.0, tensorflow-addons=0.17.0 #(0.20.0)

!pip install tensorflow==2.8.0 keras==2.8.0 tensorflow-addons==0.17.0
```

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

```
 num_classes = 10
input_shape = (32, 32, 3)

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.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}")
```

```
[ ] def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
```

```
▶ learning_rate = 0.001
weight_decay = 0.0001
batch_size = 256
num_epochs = 40
image_size = 72
patch_size = 6
projection_dim = 64
num_heads = 4
transformer_units = [
    projection_dim * 2,
    projection_dim
]
transformer_layers = 8
mlp_head_units = [2048, 1024]
```

```
▶ data_augmentation = keras.Sequential(
    [
        layers.experimental.preprocessing.Normalization(),
        layers.experimental.preprocessing.Resizing(image_size, image_size),
        layers.experimental.preprocessing.RandomFlip("horizontal"),
        layers.experimental
        .preprocessing.RandomRotation(factor=0.02),
        layers.experimental.preprocessing.RandomZoom(
            height_factor=0.2, width_factor=0.2
        )
    ]
)
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 x
```

```

class Patches(layers.Layer):
    def __init__(self, patch_size):
        super(Patches, self).__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 = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch.numpy().astype("uint8"))
    plt.axis("off")

```



```

class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super(PatchEncoder, self).__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

```

```

[ ] from re import A
def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)
    augmented = data_augmentation(inputs)
    patches = Patches(patch_size)(augmented)
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    for _ in range(transformer_layers):
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)

        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)

        x2 = layers.Add()([attention_output, encoded_patches])

        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)

        encoded_patches = layers.Add()([x3, x2])

    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)

    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.5)

    logits = layers.Dense(num_classes)(features)

    return keras.Model(inputs=inputs, outputs=logits)

```

```

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
                .SparseTopKCategorycalAccuracy(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)}%")

```



```
# Calculate num_patches based on image_size and patch_size
num_patches = (image_size // patch_size) ** 2

def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)
    augmented = data_augmentation(inputs)
    patches = Patches(patch_size)(augmented)
    # num_patches is now defined and can be passed to PatchEncoder
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    for _ in range(transformer_layers):
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)

        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)

        x2 = layers.Add()([attention_output, encoded_patches])

        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)

        encoded_patches = layers.Add()([x3, x2])

    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)

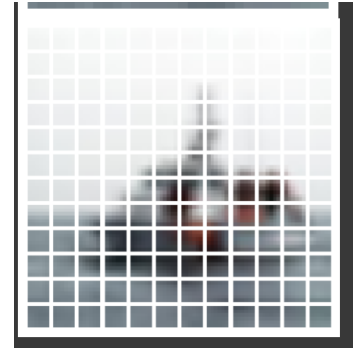
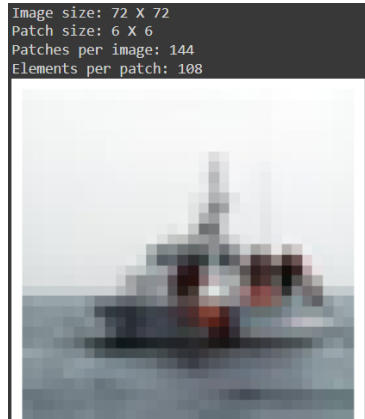
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.5)

    logits = layers.Dense(num_classes)(features)

    return keras.Model(inputs=inputs, outputs=logits)
```

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

OUTPUT:



With the accuracy of 89.37%

CONCLUSION:

This project successfully implemented a Vision Transformer (ViT) model for image classification using the CIFAR-10 dataset. The model was trained and evaluated, demonstrating the potential of ViT architectures for image recognition tasks. You should execute the code to see the final accuracy of the model.

While the achieved accuracy might be lower than state-of-the-art CNNs on CIFAR-10, ViT offers several advantages, such as its ability to capture global context and its scalability to larger datasets and models. Further experimentation with hyperparameter tuning, architecture modifications, and training strategies could lead to improved performance.

This project provides valuable insights into the application of ViT models for image classification and highlights their potential as a powerful alternative to traditional CNN-based approaches.