Keras Vision Transformer

Status	In progress
Assign	

Material

- Google AI Blog
- Paper
- Youtube Tutorial
- Paper explained

Resources

Resources

Name	Tags	URL
Google AI Blog	Blog	https://ai.googleblog.com/2020/12/transformers-for-image-recognition-at.html
Paper	Paper	https://arxiv.org/abs/2010.11929
Keras Tutorial	Tutorial Video	https://arxiv.org/abs/2010.11929
Paper Explanation	Explanation Youtube	https://www.youtube.com/watch?v=TrdevFK_am4

Implementation Notes

Motivation

To get some experience with **transformer** architecture in vision. Mainly for personal learning.

Overview

How to use Vision Transformer (ViT) for image classification

Main idea:

- Split input into small NxN patches
- Flatten them as a `sequence`
- Treat them as a `token` similar to NLP

Using **tf-addons** mainly to import **AdamW** optimizer.

Dataset

We are using **CIFAR-100** dataset.

```
Train images shape: (50000, 32, 32, 3), Train labels shape: (50000, 1)
Test images shape: (10000, 32, 32, 3), Test labels shape: (10000, 1)
```

Hyperparameters

- From `image_size` and `patch_size`, we get and store number of patches in a variable `num_patches`. It will be used when we define the *positional embeddings*.
    - **Positional embedding** -> where in the original image this patch is located.
- Project patches to `64-dim` feature vectors
    - Concatenated together to form input for the first transformer layer
- Four separate transformations/parameterizations
    - Aggregate the output
- Skip connections are also utilized

Data Augmentation

- To prevent *overfitting*
- In `normalization` we use `adapt` layer
    - The adapt() method
        - Some preprocessing layers have an internal state that can be computed based on a sample of the training data.
        - Crucially, these layers are **non-trainable**. Their state is not set during training; it must be set **before training**, either by initializing them from a pre-computed constant, or by *adapting* them on data.

Custom mlp layer is used to add skip connection.

Implement patch creation as a layer

- Overwriting keras `Layer` object to implement the *patches* layer.
- This is the layer where we take 72x72 image and then transforming it into a grid of 6x6 patches.
- Visualize
    - Unlike regular `plt.imshow`, We will have to loop through these patches, plot them and have them occupy space in `plt.subplot` indexing.

Implement patch encoding layer

We have a projection of these `patches`, and then we are going to add a learnable `embedding` for the position. This is used in transformer architectures because transformers don't have any sense of the original ordering of the sequence. So we are going to be learning this `embedding` as well.

The original “Attention is all you need” paper uses a fixed embedding where you add the `sine` or `cosine` wave.

Learning the embedding is similar to ideas like **word embedding** where we have a table that indexes a vector, that represents each of these **discrete encodings.**

In our case, we have `144` positional embedding because we have `144` patches. Each one of these represents a discrete object like 0, 1, 2 up to 144. Sow we are going to map this into an **embedding table** that transform that 0, 1, 2 into a `vector` or the same projection dimension as we are transforming our `patches`

Build the ViT Model

The Vit model consists of multiple transformer blocks which uses 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.

There can be two ways for **data augmentation** in keras. → Link

Option 1: Make the preprocessing layers part of your model

```
model = tf.keras.Sequential([
  # Add the preprocessing layers you created earlier.
  resize_and_rescale,
  data_augmentation,
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  # Rest of your model.
])
```

There are two important points to be aware of in this case:

- Data augmentation will run on-device, synchronously with the rest of your layers, and benefit from GPU acceleration.
- When you export your model using `model.save`, the preprocessing layers will be saved along with the rest of your model. If you later deploy this model, it will automatically standardize images (according to the configuration of your layers). This can save you from the effort of having to reimplement that logic server-side.

Option 2: Apply the preprocessing layers to your dataset

```
aug_ds = train_ds.map(
  lambda x, y: (resize_and_rescale(x, training=True), y))
```

With this approach, you use `Dataset.map` to create a dataset that yields batches of augmented images. In this case:

- Data augmentation will happen asynchronously on the CPU, and is non-blocking. You can overlap the training of your model on the GPU with data preprocessing, using `Dataset.prefetch`, shown below.
- In this case the preprocessing layers will not be exported with the model when you call `Model.save`. You will need to attach them to your model before saving it or reimplement them server-side. After training, you can attach the preprocessing layers before export.

Apply the preprocessing layers to the datasets

```
batch_size = 32
AUTOTUNE = tf.data.AUTOTUNE

def prepare(ds, shuffle=False, augment=False):
  # Resize and rescale all datasets.
  ds = ds.map(lambda x, y: (resize_and_rescale(x), y),
              num_parallel_calls=AUTOTUNE)

  if shuffle:
    ds = ds.shuffle(1000)

  # Batch all datasets.
  ds = ds.batch(batch_size)

  # Use data augmentation only on the training set.
  if augment:
    ds = ds.map(lambda x, y: (data_augmentation(x, training=True), y),
                num_parallel_calls=AUTOTUNE)

  # Use buffered prefetching on all datasets.
  return ds.prefetch(buffer_size=AUTOTUNE)
```

```
train_ds = prepare(train_ds, shuffle=True, augment=True)
val_ds = prepare(val_ds)
test_ds = prepare(test_ds)
```

Split into `patches` and also add the `embedding` layer. Then add the loop of **transformer layers.** Then finally take the representation and flatten it and get the final results.

Compile, train and evaluate the model

Using the `AdamW` optimizer from **tf-addons**.

`from_logits=true` because we did not apply *softmax* in the architecture. This option will automatically apply *softmax* to the `logits.`

Using `SparseTopKCategoricalAccuracy` to see that at least the prediction is in top-5 labels. This gives us more information about the performance of the model.

We can also add other callback options like *EarlyStopping, LearningRateScheduler* and so on. A list of various callback option is available here.

Plotting the final `vit_classifier`. Check model summary as well.

For plotting function, need to install:

```
pip install pydot
sudo apt-get install graphviz
```

Results

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 CIFAR-100 dataset, as **ResNet50V2** trained from scratch on the same data can achieve 67% accuracy.

Note that the state of the art results reported in this 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`
- 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 parameter such as the learning rate schedule, optimizer, weight decay, etc. In practice, it is recommended to fine-tune a ViT model that was pre-trained using a large, high-resolution dataset.

Testing

Calculate test accuracy. We can use `np.count_nonzero(y_pred == y_test)/ len(y_test)` for it or we can use `accuracy_score` from **sklearn**

Visualizing predictions

List of all the **CIFAR-100** labels from issue:

```
str_labels = ['apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee',
              'beetle', 'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus',
              'butterfly', 'camel', 'can', 'castle', 'caterpillar', 'cattle',
              'chair', 'chimpanzee', 'clock', 'cloud', 'cockroach', 'couch',
              'crab', 'crocodile', 'cup', 'dinosaur', 'dolphin', 'elephant',
              'flatfish', 'forest', 'fox', 'girl', 'hamster', 'house', 'kangaroo',
              'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion', 'lizard',
              'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse',
              'mushroom', 'oak_tree', 'orange', 'orchid', 'otter', 'palm_tree',
              'pear', 'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy',
              'porcupine', 'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket',
              'rose', 'sea', 'seal', 'shark', 'shrew', 'skunk', 'skyscraper', 'snail',
              'snake', 'spider', 'squirrel', 'streetcar', 'sunflower', 'sweet_pepper',
              'table', 'tank', 'telephone', 'television', 'tiger', 'tractor', 'train',
              'trout', 'tulip', 'turtle', 'wardrobe', 'whale', 'willow_tree', 'wolf',
              'woman', 'worm']
```

Plot some images with their **ground-truth** and **predicted** labels. Helpful link.

Issue

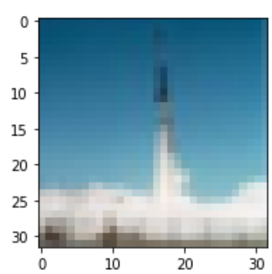
Currently, images are not matching with the labels. Need to fix!

I faced some issue in displaying results. My **ground-truth** labels are not in line with the displayed images. So need to check if there is an issue with `str_labels` list or the issue is in my plotting function.

```
idx = 13

plt.rcParams["figure.figsize"] = (3,3)
plt.imshow(x_test[idx])
print("Ground-truth label and class: ", y_test[idx], "->", str_labels[y_test[i]])
print("Predicted label and class: ", y_pred[idx], "->", str_labels[y_pred[i]])
```

```
Ground-truth label and class:  69 -> rocket
Predicted label and class:  69 -> rocket
```



Results show that ground labels are in accordance with `str_labels` list. So the issue is in my plotting function. I copied it from another source. Need to write my own function now.

Fixing the function

There was indexing issue. It is working now.

```
# Plot some predictions
plt.rcParams["figure.figsize"] = (15,15)

num_row = 4
num_col = 4

image_id = np.random.randint(0, len(x_test), num_row * num_col)
fig, axes = plt.subplots(num_row, num_col)

for i in range(0, num_row):
    for j in range(0, num_col):
        k = (i * num_col) + j
        axes[i,j].imshow(x_test[image_id[k]])
        axes[i,j].set_title(f"True: {str_labels[y_test[image_id[k]]]}, \nPredicted: {str_labels[y_pred[image_id[k]]]}", fontsize=14)
        axes[i,j].axis('off')
        fig.suptitle("Images with True and Predicted Labels", fontsize=18)

plt.show()
```