Vision_transformers

May 23, 2025

1 Vision transformer for image classification

In this lab you will familiarize yourself with the building blocks of the **transformer** architecture and build a vision transformer (ViT) model for classifying the CIFAR10 dataset. Since ViTs can be computationally expensive to fully train, you will also import a pre trained ViT model and apply it to a series of images of your choice.

Complete the code flagged throughout the elaboration and **answer all the questions in the notebook**.

[1]:
```
# Set up and imports
%reload_ext autoreload
%autoreload 2


import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"  # Tensorflow is quite chatty; filter␣
 ↪out warnings
os.environ["KERAS_BACKEND"] = "tensorflow"

import numpy as np
import matplotlib.pyplot as plt
```

1.0.1 Check that Tensorflow uses a GPU *(optional)*

Training the models in this notebook can be sped up significantly with a GPU. The following cell can be used to check if the GPU is set up correctly. If you run on CPU, you can either run or just ignore this cell.

[2]:
```
import tensorflow as tf

os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
# The GPU id to use, usually either "0" or "1"
os.environ["CUDA_VISIBLE_DEVICES"]="0"

if tf.config.list_physical_devices("GPU"):
    print(" Tensorflow has detected a GPU.")
    # Allow growth of GPU memory, otherwise it will always look like all the␣
 ↪memory is being used
```

```
    physical_devices = tf.config.experimental.list_physical_devices('GPU')
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
else:
    print("  Tensorflow has NOT detected a GPU.")
    print()
    print("For GPU support, visit: <https://www.tensorflow.org/install/pip>")
```

```
2025-05-23 14:00:20.197597: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1748001620.206240  783088 cuda_dnn.cc:8310] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already
been registered
E0000 00:00:1748001620.208887  783088 cuda_blas.cc:1418] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
already been registered
```

```
 Tensorflow has detected a GPU.
```

1.1 The scaled dot-product attention

As you have learned from the lectures and the previous RNN lab, there exist several methods to implement the attention mechanism. In RNN, the attention mechanism generally uses the hidden states of the model to learn to focus the tokens (or elements) in a sequence. The transformer architecture uses the multi-head self-attention (MHSA) mechanism based the scaled dot-product attention. This new way of modeling attention was introduced in the landmark paper by Vaswani et al. in 2017 Attention is all you need and it is at the core of modern architectures such as GPTs and BERT.

In the following sections you will, step by step, take a closer look at the building blocks of the MHSA, starting an a visual intuition of how attention works.

1.1.1 Back to the Future: Nadaraya-Watson kernel regression

Using a data retrieval analogy, imagine you are typing a query in a search engine. Under the hood, an algorithm compares your query with a database of keys and it returns the value corresponding to the key that best-matches your query. Thus, the attention can be viewed as a mapping between the query and the value using the similarity between the query and the keys.

Consider the database $D = (k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)$ made of n tuples *(key, value)* pairs. Given a query q, we can formulate the attention over D as:

$$\text{Attention}(q, D) = \sum_{i=1}^{n} \alpha(q, k_i) v_i,$$

where $\alpha(q, k_i)$ are scalar values called attention weights. This operation, also referred to as *attention pooling*, is simply a linear combination of the values in the database using the *similarity* between the query and the keys.

A simple, yet educative, example of how this concept can be found in the kernel regression approach initially proposed by Nadaraya and Watson in 1964. In they formulation, the similarity metrics $\alpha(q, k_i)$ is chose to be a Gaussian kernel applied to the difference between the query and the keys:

$$\alpha(q, k) = exp(-\frac{1}{2\sigma^2}||q - k||^2),$$

where the attention weight are then used to compute the output of a regressor or classifier as:

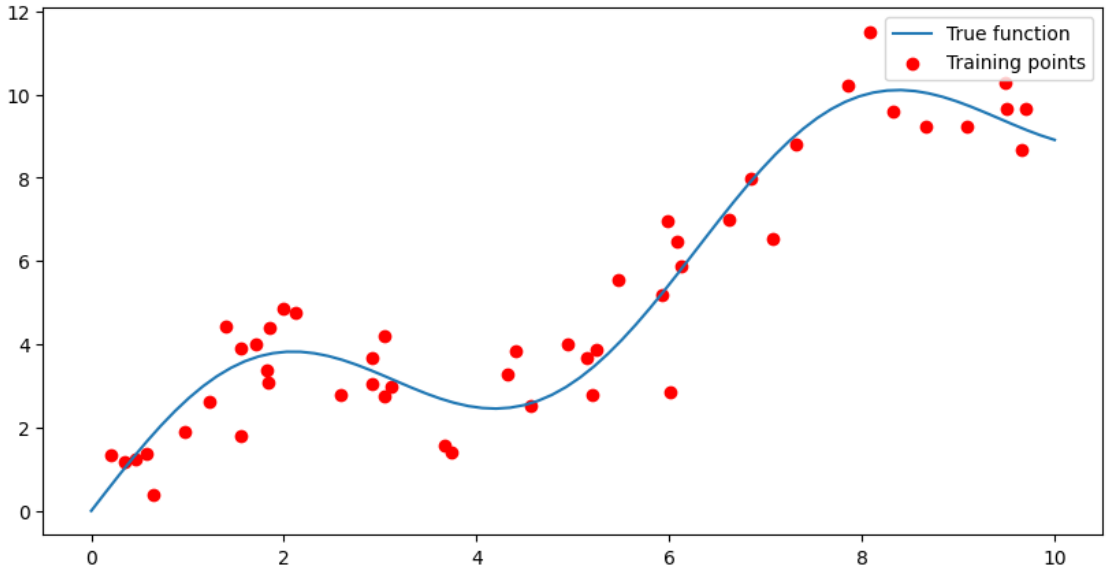
$$f(q) = \sum_i v_i \frac{\alpha(q, k_i)}{\sum_j \alpha(q, k_i)}.$$

In the following cell, we will visualize an example of scalar regression where the training samples are tuples of features and labels (x_i, y_i), and the task is to predict the value at a new location. In this case k are the features in out training sample, v are the training labels and q is the new location whose value we want to predict.

In the following cell, we will define the dummy dataset you will work with and visualize how the choice of the parameters defining the Gaussian kernel (σ in this case) affects the regressor prediction.

[3]:
```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt

# define a random seed for reproducibility
seed = 42
np.random.seed(seed)

# define a function that creates a dummy dataset (sine wave with noise)
def dummy_dataset(x):
    return 2 * np.sin(x) + x

# create a dummy dataset: specify the number of training and validation␣
 ↪samples, the range of x values and the noise level
nbr_train = 50
nbr_valid = 70
min_x, max_x = 0, 10

x_train = np.sort(np.random.uniform(min_x, max_x, nbr_train), 0)
y_train = dummy_dataset(x_train) + np.random.normal(0, 1, nbr_train)
x_val = np.linspace(min_x, max_x, nbr_valid)
y_val = dummy_dataset(x_val)

# plot the dataset
plt.figure(figsize=(10, 5))
plt.plot(x_val, y_val, label="True function")
plt.scatter(x_train, y_train, color="red", label="Training points")
plt.legend()
plt.show()
```

```
# print the shapes of the training and validation sets
print(f"Training set: {x_train.shape}, {y_train.shape}")
print(f"Validation set: {x_val.shape}, {y_val.shape}")
```

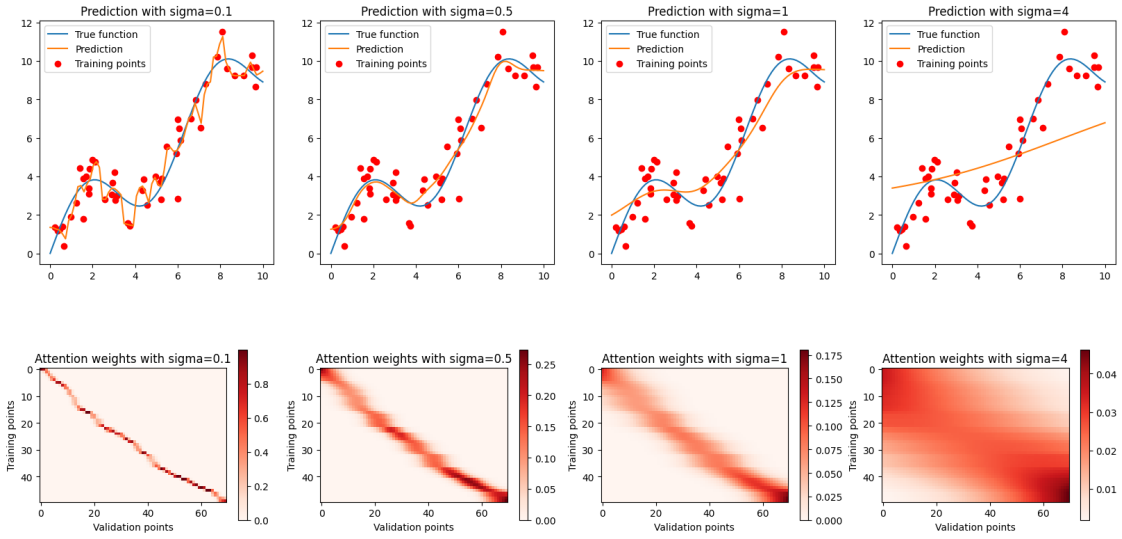



```
Training set: (50,), (50,)
Validation set: (70,), (70,)
```

[4]:
```
# Define the Gaussian regression kernel
def gaussian(x, sigma=1):
    return np.exp(-(x**2) / (2 * sigma**2))

# define the nadaraya_watson regressor function
def nadaraya_watson(q, k, v, kernel, sigma=1):
    # compute the distance matrix between the training and validation points
    distances = np.linalg.norm(k[:, np.newaxis, :] - q[np.newaxis, :, :],␣
 ↪axis=2) # the element-wise difference between the queries and the keys
    # distances = k - q.T
    # Apply the kernel to the distances (across the rows)
    distances = kernel(distances, sigma)
    # Normalize the distances (across the rows) to obtain the attention weights
    attention_weights = distances / np.sum(distances, axis=0)
    # Use the attention weights to compute the prediction
    prediction = attention_weights.T @ v
    return prediction, attention_weights

# define the q, k and v using the dummy dataset (make them N -by- 1)
q = x_val[:, np.newaxis]
k = x_train[:, np.newaxis]
```

```
v = y_train[:, np.newaxis]
print(f"q: {q.shape}, k: {k.shape}, v: {v.shape}")

# plot the predictions and the attention weights for different values of sigma.
def plot_predictions(q, k, v, y_val, kernel, sigmas):
    if len(sigmas) == 1:
        fig, axs = plt.subplots(2, 1, figsize=(5, 10))
        axs = axs.reshape(2, 1)
    else:
        fig, axs = plt.subplots(2, len(sigmas), figsize=(20, 10))

    # loop through the sigma values
    for i, sigma in enumerate(sigmas):
        # compute the prediction and attention weights
        prediction, attention_weights = nadaraya_watson(q, k, v, kernel, sigma)
        # plot the prediction
        ax = axs[0, i]
        ax.plot(q, y_val, label="True function")
        ax.plot(q, prediction, label="Prediction")
        ax.scatter(k, v, color="red", label="Training points")
        ax.set_title(f"Prediction with sigma={sigma}")
        ax.legend()
        # plot the attention weights
        ax = axs[1, i]
        attention_plot = ax.imshow(attention_weights, cmap="Reds")
        ax.set_xlabel("Validation points")
        ax.set_ylabel("Training points")
        ax.set_title(f"Attention weights with sigma={sigma}")
        fig.colorbar(attention_plot, ax=ax, shrink=0.7)
    plt.show()

# define the sigma values to test
sigmas = [0.1, 0.5, 1, 4]
# plot the predictions
plot_predictions(q, k, v, y_val, gaussian, sigmas)
```

```
q: (70, 1), k: (50, 1), v: (50, 1)
```



As you can see, the similarity kernel we use to attend to the distances greatly influences the prediction of the model. For sharper Gaussian kernels, the spikier the prediction while larger sigmas result in smoother prediction. Finding the best σ in the case of a Gaussian kernel or a more suitable kernel among a variety of kernels is time consuming. Thus, one can think of designing a model where the parameters of the kernel can be learned. Another way is to keep the similarity function the same but change the representations of the query, keys and values so that when compared thought the similarity function the output is different. As we will see soon, this last solution is what modern transformer architectures learn: they learn how to project q, k and v so that similarities are computed over different representations.

Questions

1. Looking at the examples in the Nadaraya - Watson regression, what are the benefits and drawbacks of this type of attention implementation?

Answers

1. Benefits
   - Simple & Easy to Understand: Just uses weighted averages based on similarity — no complex math.
   - Smooth Predictions: Gives nice, smooth outputs — great when your data isn't too noisy.
   - No Training Needed: Doesn't learn weights — it just calculates them directly from the input.
   - Good for Small Datasets: Works well when you don't have a lot of data.

   Drawbacks
   - Can't Learn Patterns: Since it doesn't train, it can't adapt or get better over time.
   - Slow with Big Data: It compares every point to every other point — gets slow as data grows.
   - Not Very Flexible: Can't capture complex relationships like transformers can.

- Sensitive to Parameters: You have to pick the right similarity function and tune it manually.
- Doesn't Understand Order: Unlike transformers, it doesn't know if one word comes before another.

1.1.2 Scaled-dot product attention

The choice of a Gaussian regression kernel in the toy example was not by chance. In fact, one can prove that (see here for a more extensive description):

$$\alpha(q, k_i) = -\frac{1}{2}||q - k_i||^2 = q^\top k_i - \frac{1}{2}||k_i||^2 - \frac{1}{2}||q||^2.$$

The last term is similar for all the k_i, thus when normalizing this element disappears. In addition, as described here, $||k_i||^2$ can also be removed in case the keys are themselves normalized (which is usually the case). This brings as close to the definition of the scaled dot-product attention by Vaswani et al., 2017, which is:

$$\alpha(q, k_i) = \text{softmax}\left(\frac{q^\top k_i}{\sqrt{d}}\right)$$

where $q \in \mathbb{R}^d$, $k_i \in \mathbb{R}^d$ and the scaling factor $\sqrt{d}$ ensures that the dot product variance is 1 regardless the query of key vector length d. Finally, expanding the formulation to the case where the attention is computed for m queries and using it for attention pooling, we obtain:

$$Z = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V,$$

where $Q \in \mathbb{R}^{m\times d}$ is the matrix of all the query vectors, $K \in \mathbb{R}^{n\times d}$ and $v \in \mathbb{R}^{n\times d}$ is the matrix of all the keys and value vectors, and $Z \in \mathbb{R}^{m\times d}$ is the matrix containing all the weighted value vectors where weights are computed using the similarity between each query vector and the keys using the scaled-dot product attention.

In the next section we will implement the scaled dot-product attention. This implementation should be added to the `utility.py` file. Follow the instructions in the function definition and use the cell below to test your implementation (an error ~ 0.0 should be expected).

[5]:
```
from utilities import test_scaled_dot_product_attention

test_scaled_dot_product_attention(seed=42)
```

```
##########################################
### Testing scaled_dot_product_attention ###
##########################################
Random seed set to: 42 (expecting 42)
Testing on Q: (2, 10, 5), K: (2, 5, 5), V: (2, 5, 5)
### scaled_dot_product_attention test passed ###
Difference with expected values == 0.0
##########################################
```

```
I0000 00:00:1748001622.101936  783088 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 18000 MB memory:  -> device:
0, name: NVIDIA RTX 4000 Ada Generation, pci bus id: 0000:01:00.0, compute
capability: 8.9
```

1.1.3 Multi-head attention

In practice, when given the same set of queries, keys, and values, we may want our model to integrate knowledge from different aspects using the same attention mechanism. For instance, it can be useful to capture dependencies at varying ranges within a sequence, such as short-range and long-range interactions. To achieve this, the attention mechanism can benefit from utilizing different representation subspaces for queries, keys, and values.

Instead of applying a single attention pooling operation, we can first transform the queries, keys, and values using independently learned linear projections. These projected versions are then processed through multiple attention pooling operations in parallel. The resulting outputs from each attention computation—referred to as **heads**—are then concatenated and passed through another learned linear projection to generate the final output. This architecture is known as multi-head attention Vaswani et al., 2017, and you can see it visualized in the figure below. Note the multiple heads, each with its own set of linear projection W_h^q, W_h^k and W_h^v for queries, keys, and values, respectively.

Time to implement the `MultiHeadAttention`. Follow the instructions of the MultiHeadAttention class in the `utilities.py` file and test your implementation with the cell below.

NOTE!: another concept that we will use later is self-attention. In self-attention, the scaled-dot product is computed over projections of the same input sequence. This is particularly useful when we want to encode information taking advantage of short-and-long range dependencies. Self-attention, as you will see later on, is utilized in the Vision Transformer models, to obtain a representation of the input image that can be later used for classification tasks.

[6]:
```
from utilities import test_MultiHeadAttention

# Run the following cell to test your implementation of the MHA layer. IF the␣
 ↪test is not passed, check your implementation.
test_MultiHeadAttention(seed=42)
```

```
################################################
### Testing MultiHeadAttention definition ###
################################################
Random seed set to: 42 (expecting 42)
Testing on Q: (2, 10, 5), K: (2, 5, 5), V: (2, 5, 5)
Number of heads: 2, projection size: 8, dk: 5, dv: 5

### MultiHeadAttention test passed ###
Difference with expected values == 0.0
################################################
```

Questions

2. Describe the self-attention mechanism used in vision transformer architecture; how does it differ from that of the attention mechanism you implemented in the RNN lab?
3. What are the computational challenges related with the scaled dot-product attention? Is it more computationally demanding than that of the attention implemented in the RNN?
4. What are the advantages, from a computational perspective, of the scaled dot product compared to attention mechanism implemented in the RNN lab?
5. Can you think of a relation between the multi-head attention you just implemented and convolutional kernels in a CNN architecture?

Answers

2. The self attention mechanism in ViT uses patches, and it calculates Q, K, V for every patch. This is used to get attention scores and get weights to get weighted sum of V. This enable each patch look at other patches. The main difference in RNN and ViT is that RNN has encoder-decoder attention and ViT has self-attention. RNN is sequential and ViT is parallelized, making it faster.
3. It has time complexity of O(n^2) in both time and memory. Hence, we can say that is more computationally demanding. RNN attention is over a smaller context, but being sequential it may take more time.
4. Parallelization, efficient on large scale, uniformity are some advantages from a computational perspective.
5. They both detect patterns in the domain such as edges, corners, etc. in parallel.

1.1.4 Transformer Encoder Layer

The transformer encoder layer is a fundamental building block of the transformer architecture. It consists of several key components that work together to process input sequences and generate meaningful representations. So far, we have implemented the multi-head attention mechanism, which allows the model to focus on different parts of the input sequence simultaneously. However, the full transformer encoder layer includes additional components that are crucial for its functionality:

1. **Multi-Head Attention**: This mechanism enables the model to attend to different positions of the input sequence and capture various aspects of the data. It consists of multiple attention heads, each focusing on different parts of the sequence.
2. **Layer Normalization**: Applied after the multi-head attention and feed-forward layers.
3. **Feed-Forward Neural Network**: This component consists of two linear transformations with a ReLU activation in between. It processes the output of the multi-head attention mechanism to generate more complex representations.
4. **Residual Connections**: These connections add the input of each sub-layer to its output.
5. **Dropout**: Applied to the outputs of the multi-head attention and feed-forward layers, dropout helps prevent overfitting by randomly setting a fraction of the input units to zero during training.

In the next steps, we will implement these additional components to complete the full transformer encoder layer. Add your code in the `utilities.py` file for the following:

1. `mlp` function: defines the feed-forward part of the transformer encoder block. As described in the original implementation, this part of the network is composed by two linear transformations: `proj_dim` -> 2048 -> `proj_dim`.
2. `trasformerBlock` function: it uses the `MultiHeadAttention` and `mlp` to build the full transformer encoder (see image of the transformer architecture - from Vaswani et al., 2017).

NOTE!: Here we are implementing the `transformerBlock` using the self-attention mechanism, where the query, keys and the values used in the attention pooling are obtained from projections of the save input vector.

[7]:
```
from utilities import test_TransformerBlock, transformerBlock

# Run the following cell to test your implementation of the TransformerBlock.
↪If the test is not passed, check your implementation.

test_TransformerBlock(seed=42, dropout_rate=0.1)
```

```
###########################################
### Testing transformerBlock definition ###
###########################################
Random seed set to: 42 (expecting 42), dropout rate: 0.1 (expected 0.1)
Testing on Q: (2, 10, 32)
Number of heads: 2, projection size: 32, dx: 32, transformer_units: [64, 32]

### TransformerBlock test passed ###
Difference with expected values == 0.0
###########################################
```

Questions

6. What is the role of the residual connections in the transformer layer?
7. What is the role of the `Layer Normalization` and why is it needed?

Answers

6. The model remembers the original inputs while learning useful transformations because of the residual connections. The vanishing gradient problem is also solved because of the residual connections.
7. It is required to normalize the input across every token. They ensure that each sublauer has zero mean and unit variance as the name suggests “normalization”. It also helps in stabilizing the training and converge it faster.

1.2 Vision transformer

Now that you have implemented the core mechanism of a transformer architecture, we can apply it to for an image classification task. In the following cells, you will implement the remaining parts of a vision transformer (ViT) architecture and train a model for the classification of the **CIFAR10** dataset.

As an overview, a vision transformer model is composed by four components: 1. `PatchExtraction`: given the computational complexity on the (self) scaled-dot product attention is $O(n^2d)$, applying the attention mechanism on the raw pixel values will be unfeasible for real world images. Thus, in the landmark paper that introduced the transformer architecture for computer vision tasks (An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale), the authors split in image into squared patches (16×16 in the original implementation), which are flattened (or encoded) and used as input to the transformer architecture. 2. `PatchEncoder`: the scaled-dot product attention comes with the benefit of parallel computation but disregards the order of the input sequence. This is a problem when it comes to both vision and language modelling, where the order if the input sequence (*i.e.* the information regarding the closeness of pixels in an image) carries information. To address this issue, a **positional embedding** is added to each element in the sequence, providing information on both the position of an element in a sequence, and a feature within an element. (**NOTE!**: in this lab we will use the `Embedding` Keras layer to add positional encoding. The original implementation proposes a positional embedding based on sine & cosine function. These are not the only ways of implementing positional encoding, with modern architecture allowing for the positional encoding to be learned during training). 3. **Stack of transformer layers**: this is where the input image (patched and positional-encoded) traverses sequentially multiple transformer layers (`transformerBlock` in our implementation) to obtain a representation that gathers information from all the image regions and that can be used for a downstream task (classification) in this case.

There is one problem though... The the end of a sequence of transformer layers we obtain a new representation (based on the multi-head self attention mechanism) for each of the patches in the image. - **How do we use this for classification? Do we pick one of the patches as representative for the entire sequence? But then how to chose? - Or do we apply a dense layer that combines all the sequences into one sequence?**

What is done in the original implementation is to add a learnable sequence element, called the class token (`cls` token) during the `PatchEncoder`, which thought the transformer layers interacts with all the elements in the sequence. The `cls` token is then used as a representation of the entire input for the downstream task.

4. **Classification layer**: usually a dense layer recieves as input the `cls` token and outputs logits (is working on a classification task).

In the following cells you will first download the CIFAR10 dataset, and implement the `PatchExtraction`, `PatchEncoder` and the full `ViT` model architecture. You will then train it on the CIFAR10 dataset.

1.2.1 Load, preprocess and visualize the dataset

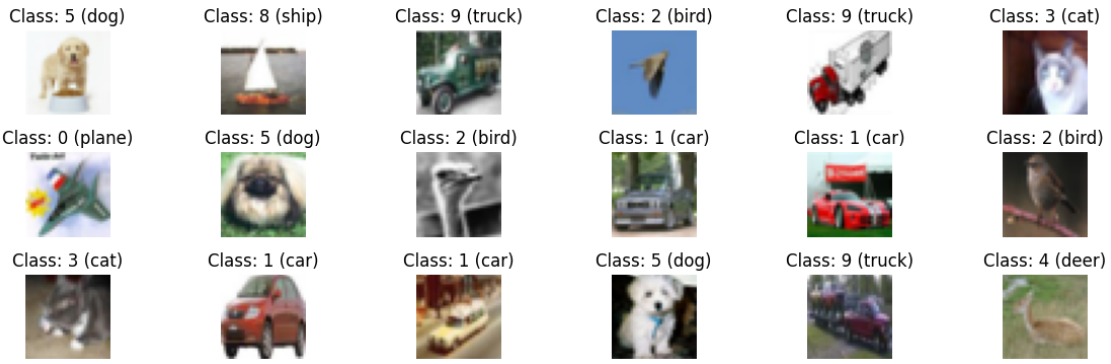
Similar to what we have done in the CNN lab, we will download the CIFAR10 dataset, split it into training, validation and testing, and normalize it.

[8]:
```
from tf_keras.datasets import cifar10
from tf_keras.utils import to_categorical
from utilities import plot_training_examples

# Load and preprocess the CIFAR10 dataset (here we are using the tf_keras.
↪dataset functionality)
```

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

# Download CIFAR train and test data
(X, Y), (Xtest, Ytest) = cifar10.load_data()

print("Training/validation images have size {} and labels have size {} ".
 ↪format(X.shape, Y.shape))
print("Test images have size {} and labels have size {} \n ".format(Xtest.
 ↪shape, Ytest.shape))

# # Reduce the number of images for training/validation and testing to 10000␣
 ↪and 2000 respectively,
# # to reduce processing time for this elaboration.
X = X[0:10000]
Y = Y[0:10000]

Xtest = Xtest[0:2000]
Ytest = Ytest[0:2000]

print("Reduced training/validation images have size %s and labels have size %s␣
 ↪" % (X.shape, Y.shape))
print("Reduced test images have size %s and labels have size %s \n" % (Xtest.
 ↪shape, Ytest.shape))

# Check that we have some training examples from each class
for i in range(10):
    print("Number of training/validation examples for class {} is {}" .
 ↪format(i,np.sum(Y == i)))

# Plot some images from the dataset to visualize it
plot_training_examples(X, Y, classes)
```

```
Training/validation images have size (50000, 32, 32, 3) and labels have size
(50000, 1)
Test images have size (10000, 32, 32, 3) and labels have size (10000, 1)

Reduced training/validation images have size (10000, 32, 32, 3) and labels have
size (10000, 1)
Reduced test images have size (2000, 32, 32, 3) and labels have size (2000, 1)

Number of training/validation examples for class 0 is 1005
Number of training/validation examples for class 1 is 974
Number of training/validation examples for class 2 is 1032
Number of training/validation examples for class 3 is 1016
Number of training/validation examples for class 4 is 999
Number of training/validation examples for class 5 is 937
```

```
Number of training/validation examples for class 6 is 1030
Number of training/validation examples for class 7 is 1001
Number of training/validation examples for class 8 is 1025
Number of training/validation examples for class 9 is 981
```



Data split Split your data (X, Y) into training (Xtrain, Ytrain) and validation (Xval, Yval), so that we have training, validation and test datasets (as in the previous laboration).

We use the `train_test_split` function from scikit learn (see the documentation for more details) to obtain 25% validation set.

[9]:
```
from sklearn.model_selection import train_test_split

# ----------------------------------------
# === Your code here for data splitting ===
# ----------------------------------------

# split the original dataset into 70% Training and 30% Temp
Xtrain, Xval, Ytrain, Yval = train_test_split(X, Y, test_size=0.25,␣
 ↪random_state=42)

# Print the size of training data, validation data, and test data
print(f"Training set: {Xtrain.shape}, {Ytrain.shape}")
print(f"Validation set: {Xval.shape}, {Yval.shape}")
print(f"Test set: {Xtest.shape}, {Ytest.shape}")
```

```
Training set: (7500, 32, 32, 3), (7500, 1)
Validation set: (2500, 32, 32, 3), (2500, 1)
Test set: (2000, 32, 32, 3), (2000, 1)
```

Image preprocessing The images are stored as uint8, i.e. 8 bit unsigned integers, but need to be converted to 32 bit floats. We also make sure that the range is -1 to 1, instead of 0 - 255.

NOTE! that is this lab we will be using the labels as NOT categorical (we did that in the CNN lab).

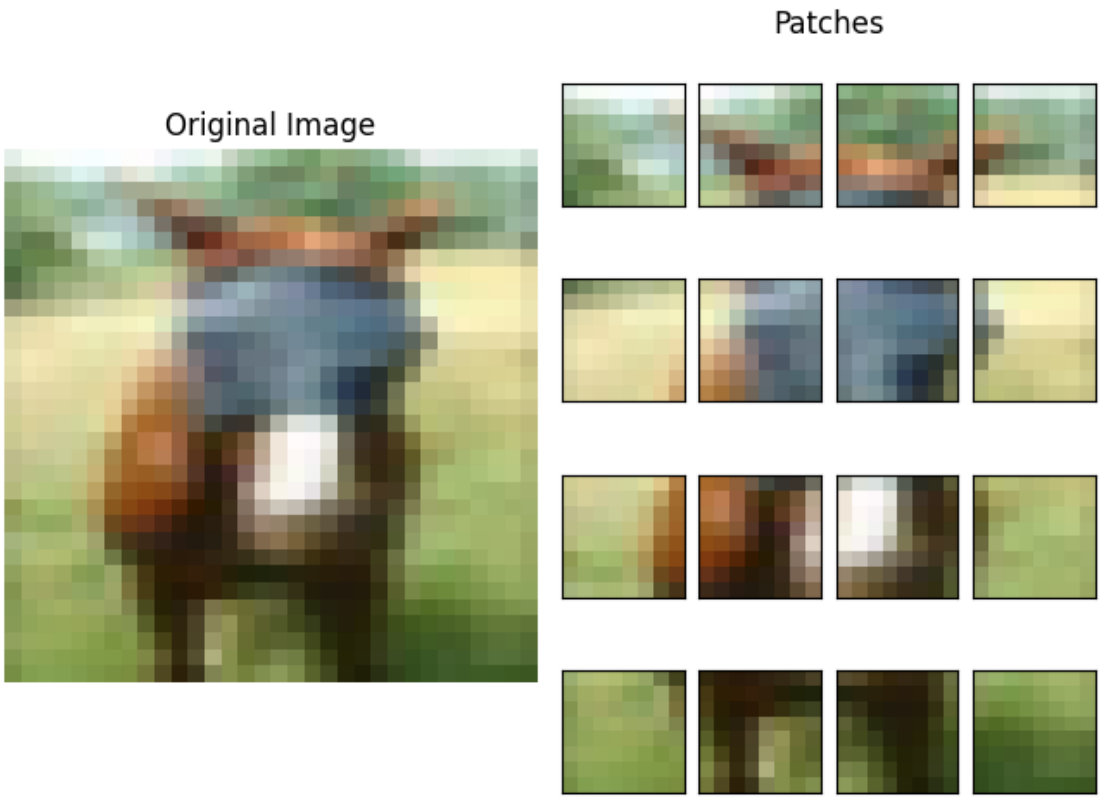
[10]:
```
# ------------------------------------------------
# === Your code here for intensity normalization ===
# ------------------------------------------------

# Convert datatype for Xtrain, Xval, Xtest, to float32
Xtrain = Xtrain.astype('float32')
Xval = Xval.astype('float32')
Xtest = Xtest.astype('float32')

# Change range of pixel values to [-1,1]
Xtrain = Xtrain / 127.5 - 1
Xval = Xval / 127.5 - 1
Xtest = Xtest / 127.5 - 1

# ================================================
```

1.2.2 PatchExtraction

Follow the instructions in the `utilities.py` file to implement the `PatchExtraction` as Keras layer. Use the cell below to verify that the patching is done correctly.

[11]:
```
from utilities import PatchExtractor, plot_original_and_patched_version

# take a sample image from the training set
sample_image = Xtrain[0]
# add a batch dimension
sample_image = sample_image[tf.newaxis, ...]

# create a PatchExtractor object
patch_size = 8
patches_extractor = PatchExtractor(patch_size=patch_size)

# encode the sample image
patches = patches_extractor(sample_image)

# print the shape of the encoded image
print(f"Original image shape: {sample_image.shape}")
print(f"Shape of the encoded image: {patches.shape}")

# Plot the original image and the patched image as a grid of patches

plot_original_and_patched_version(sample_image[0], patches, patch_size)
```

```
Original image shape: (1, 32, 32, 3)
Shape of the encoded image: (1, 16, 192)
```



1.2.3 PatchEncoder

Complete with your code in the `PatchEncoder` class in the `utility.py` file. It is here that the `cls` token will be added to the sequence of patches. Run the following cell to test your implementation.

[12]:
```
from utilities import PatchEncoder

# take a sample image from the training set
sample_image = Xtrain[0]
# add a batch dimension
sample_image = sample_image[tf.newaxis, ...]

# create a PatchExtractor object
patch_size = 8
patches_extractor = PatchExtractor(patch_size=patch_size)

# encode the sample image
patches = patches_extractor(sample_image)

# Use the patches extracted from the sample image above and encode them using␣
 ↪the PatchEncoder
```

```
patch_encoder = PatchEncoder(num_patches=patches.shape[1],␣
 ↪projection_dim=patches.shape[-1])

# encode the patches
encoded_patches = patch_encoder(patches)

# print the shape of the encoded patches
print(f"Shape of the encoded patches: {encoded_patches.shape}")
```

```
Shape of the encoded patches: (1, 17, 192)
```

Questions

8. Can you think of another way of encoding the patch information instead of linearly projecting them?
9. The transformer architecture is quite general anc can be applied to images, text or other types of input data. This flexibility comes at a cost, especially when it comes to computer vision tasks. Can you think at what this might be? (Hint: think about the properties of the convolution operation and its benefits when it comes to computer vision task.)

Answers

8. Yes, instead of using a linear projection to encode patch information, convolutional neural networks (CNNs) or convolutional layers can be used. For example, Use a convolutional layer with a kernel size and stride equal to the patch size to extract patch embeddings. This not only encodes spatial information but also introduces inductive biases like locality and translation invariance.

9. The main cost is the lack of inductive biases like locality and translation equivariance, which are inherent to convolutional networks but missing in vanilla transformers. Specifically:

(1)CNNs are good at capturing local patterns (e.g., edges, textures) due to small receptive fields, and this leads to faster learning and better generalization on vision tasks with fewer data.

(2)Transformers, on the other hand, treat all patches equally and rely on self-attention to learn spatial relationships from scratch, which makes them data-hungry and less efficient when training on smaller datasets.

(3)CNNs also scale better computationally for high-resolution images compared to transformer models that use full self-attention (quadratic complexity in the number of patches).

Thus, while transformers are flexible, they lose the efficient structure and prior knowledge encoded in convolution, which is especially helpful in vision.

1.2.4 Putting all together - ViT model

Now you have implemented all the components of a ViT model. Add your code to the `create_vit_classifier` function in the `utilities.py` file and check the model summary with the following cell. Use the following model parameter settings: - `patch_size`: 16 - `embedding_proj_dim` == `msa_proj_dim`: 768 - `num_heads`: 12 -

transformer_layers: 12 - msa_dropout_rate: 0.1 - mlp_classification_head_units: [3074] - mlp_classification_head_dropout_rate: 0.5

[13]:
```python
from utilities import create_vit_classifier

# ------------------------------------------
# === Your code here =======================
# ------------------------------------------

# create a ViT model
## define all the model parameters (here we use a dictionary to store them)
# model_params = {
#     # dataset information
#     "input_shape": ...,
#     "num_classes": ...,
#     # patch extraction
#     "patch_size": ...,
#     # patch encoder
#     "embedding_proj_dim": ...,
#     # transformer encoder
#     "num_heads": ...,
#     "msa_proj_dim": ...,
#     "transformer_layers": ...,
#     "msa_dropout_rate": ...,
#     # classification head
#     "mlp_classification_head_units": [...],
#     "mlp_classification_head_dropout_rate": ...
# }

model_params = {
    # dataset information
    "input_shape": (32, 32, 3),  # typical input shape for ViT
    "num_classes": 10,           # adjust based on your dataset
    # patch configuration
    "patch_size": 16,
    # patch encoder
    "embedding_proj_dim": 768,
    # transformer encoder
    "num_heads": 12,
    "msa_proj_dim": 768,
    "transformer_layers": 12,
    "msa_dropout_rate": 0.1,
    # classification head
    "mlp_classification_head_units": [3074],
    "mlp_classification_head_dropout_rate": 0.5
}
```

```
# ============================================

# create the model
vit = create_vit_classifier(**model_params)

# print the model summary
vit.summary()
```

```
Model: "model"
__________________________________________________________________________________
__________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================
==================
 input_1 (InputLayer)        [(None, 32, 32, 3)]          0         []

 patch_extractor_2 (PatchEx  (None, 4, 768)               0
['input_1[0][0]']
 tractor)

 patch_encoder_1 (PatchEnco  (None, 5, 768)               595200
['patch_extractor_2[0][0]']
 der)

 multi_head_attention_5 (Mu  (None, 5, 768)               2831155
['patch_encoder_1[0][0]',
 ltiHeadAttention)                                       2
'patch_encoder_1[0][0]',
'patch_encoder_1[0][0]']

 dropout_4 (Dropout)         (None, 5, 768)               0
['multi_head_attention_5[0][0]
                                                                        ']

 add_2 (Add)                 (None, 5, 768)               0
['dropout_4[0][0]',
'patch_encoder_1[0][0]']

 layer_normalization_2 (Lay  (None, 5, 768)               1536
['add_2[0][0]']
 erNormalization)

 dense_4 (Dense)             (None, 5, 1536)              1181184
['layer_normalization_2[0][0]'
                                                                      ]

 dropout_5 (Dropout)         (None, 5, 1536)              0
```

```
['dense_4[0][0]']

 dense_5 (Dense)             (None, 5, 768)            1180416
['dropout_5[0][0]']

 dropout_6 (Dropout)         (None, 5, 768)            0
['dense_5[0][0]']

 dropout_7 (Dropout)         (None, 5, 768)            0
['dropout_6[0][0]']

 add_3 (Add)                 (None, 5, 768)            0
['dropout_7[0][0]',
'layer_normalization_2[0][0]'
                                                                  ]

 layer_normalization_3 (Lay  (None, 5, 768)            1536
['add_3[0][0]']
 erNormalization)

 multi_head_attention_7 (Mu  (None, 5, 768)            2831155
['layer_normalization_3[0][0]'
 ltiHeadAttention)                                    2         ,
'layer_normalization_3[0][0]
                                                                  ',
'layer_normalization_3[0][0]'
                                                                  ]

 dropout_8 (Dropout)         (None, 5, 768)            0
['multi_head_attention_7[0][0]
                                                                  ']

 add_4 (Add)                 (None, 5, 768)            0
['dropout_8[0][0]',
'layer_normalization_3[0][0]'
                                                                  ]

 layer_normalization_4 (Lay  (None, 5, 768)            1536
['add_4[0][0]']
 erNormalization)

 dense_6 (Dense)             (None, 5, 1536)           1181184
['layer_normalization_4[0][0]'
                                                                  ]

 dropout_9 (Dropout)         (None, 5, 1536)           0
['dense_6[0][0]']
```

```
 dense_7 (Dense)             (None, 5, 768)               1180416
['dropout_9[0][0]']

 dropout_10 (Dropout)        (None, 5, 768)               0
['dense_7[0][0]']

 dropout_11 (Dropout)        (None, 5, 768)               0
['dropout_10[0][0]']

 add_5 (Add)                 (None, 5, 768)               0
['dropout_11[0][0]',
'layer_normalization_4[0][0]'
                                                                  ]

 layer_normalization_5 (Lay  (None, 5, 768)               1536
['add_5[0][0]']
 erNormalization)

 multi_head_attention_9 (Mu  (None, 5, 768)               2831155
['layer_normalization_5[0][0]'
 ltiHeadAttention)                                        2         ,
'layer_normalization_5[0][0]
                                                                  ',
'layer_normalization_5[0][0]'
                                                                  ]

 dropout_12 (Dropout)        (None, 5, 768)               0
['multi_head_attention_9[0][0]
                                                                  ']

 add_6 (Add)                 (None, 5, 768)               0
['dropout_12[0][0]',
'layer_normalization_5[0][0]'
                                                                  ]

 layer_normalization_6 (Lay  (None, 5, 768)               1536
['add_6[0][0]']
 erNormalization)

 dense_8 (Dense)             (None, 5, 1536)              1181184
['layer_normalization_6[0][0]'
                                                                  ]

 dropout_13 (Dropout)        (None, 5, 1536)              0
['dense_8[0][0]']

 dense_9 (Dense)             (None, 5, 768)               1180416
['dropout_13[0][0]']
```

```
 dropout_14 (Dropout)        (None, 5, 768)              0
['dense_9[0][0]']

 dropout_15 (Dropout)        (None, 5, 768)              0
['dropout_14[0][0]']

 add_7 (Add)                 (None, 5, 768)              0
['dropout_15[0][0]',
'layer_normalization_6[0][0]'
                                                                ]

 layer_normalization_7 (Lay  (None, 5, 768)              1536
['add_7[0][0]']
 erNormalization)

 multi_head_attention_11 (M  (None, 5, 768)              2831155
['layer_normalization_7[0][0]'
 ultiHeadAttention)                                     2         ,
'layer_normalization_7[0][0]
                                                                ',
'layer_normalization_7[0][0]'
                                                                ]

 dropout_16 (Dropout)        (None, 5, 768)              0
['multi_head_attention_11[0][0
                                                                ']']

 add_8 (Add)                 (None, 5, 768)              0
['dropout_16[0][0]',
'layer_normalization_7[0][0]'
                                                                ]

 layer_normalization_8 (Lay  (None, 5, 768)              1536
['add_8[0][0]']
 erNormalization)

 dense_10 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_8[0][0]'
                                                                ]

 dropout_17 (Dropout)        (None, 5, 1536)             0
['dense_10[0][0]']

 dense_11 (Dense)            (None, 5, 768)              1180416
['dropout_17[0][0]']

 dropout_18 (Dropout)        (None, 5, 768)              0
```

```
['dense_11[0][0]']

 dropout_19 (Dropout)        (None, 5, 768)              0
['dropout_18[0][0]']

 add_9 (Add)                 (None, 5, 768)              0
['dropout_19[0][0]',
'layer_normalization_8[0][0]'
                                                                ]

 layer_normalization_9 (Lay  (None, 5, 768)              1536
['add_9[0][0]']
 erNormalization)

 multi_head_attention_13 (M  (None, 5, 768)              2831155
['layer_normalization_9[0][0]'
 ultiHeadAttention)                                     2         ,
'layer_normalization_9[0][0]
                                                                ',
'layer_normalization_9[0][0]'
                                                                ]

 dropout_20 (Dropout)        (None, 5, 768)              0
['multi_head_attention_13[0][0
                                                                ']

 add_10 (Add)                (None, 5, 768)              0
['dropout_20[0][0]',
'layer_normalization_9[0][0]'
                                                                ]

 layer_normalization_10 (La  (None, 5, 768)              1536
['add_10[0][0]']
 yerNormalization)

 dense_12 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_10[0][0]
                                                                ']

 dropout_21 (Dropout)        (None, 5, 1536)             0
['dense_12[0][0]']

 dense_13 (Dense)            (None, 5, 768)              1180416
['dropout_21[0][0]']

 dropout_22 (Dropout)        (None, 5, 768)              0
['dense_13[0][0]']
```

```
 dropout_23 (Dropout)        (None, 5, 768)              0
['dropout_22[0][0]']

 add_11 (Add)                (None, 5, 768)              0
['dropout_23[0][0]',
'layer_normalization_10[0][0]
                                                        ']

 layer_normalization_11 (La  (None, 5, 768)              1536
['add_11[0][0]']
 yerNormalization)

 multi_head_attention_15 (M  (None, 5, 768)              2831155
['layer_normalization_11[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_11[0][0]
                                                        ',
'layer_normalization_11[0][0]
                                                        ']

 dropout_24 (Dropout)        (None, 5, 768)              0
['multi_head_attention_15[0][0
                                                        ]']

 add_12 (Add)                (None, 5, 768)              0
['dropout_24[0][0]',
'layer_normalization_11[0][0]
                                                        ']

 layer_normalization_12 (La  (None, 5, 768)              1536
['add_12[0][0]']
 yerNormalization)

 dense_14 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_12[0][0]
                                                        ']

 dropout_25 (Dropout)        (None, 5, 1536)             0
['dense_14[0][0]']

 dense_15 (Dense)            (None, 5, 768)              1180416
['dropout_25[0][0]']

 dropout_26 (Dropout)        (None, 5, 768)              0
['dense_15[0][0]']

 dropout_27 (Dropout)        (None, 5, 768)              0
['dropout_26[0][0]']
```

```
 add_13 (Add)                (None, 5, 768)              0
['dropout_27[0][0]',
'layer_normalization_12[0][0]
                                                                  ']

 layer_normalization_13 (La  (None, 5, 768)              1536
['add_13[0][0]']
 yerNormalization)

 multi_head_attention_17 (M  (None, 5, 768)              2831155
['layer_normalization_13[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_13[0][0]
                                                                  ',
'layer_normalization_13[0][0]
                                                                  ']

 dropout_28 (Dropout)        (None, 5, 768)              0
['multi_head_attention_17[0][0
                                                                 ]']

 add_14 (Add)                (None, 5, 768)              0
['dropout_28[0][0]',
'layer_normalization_13[0][0]
                                                                  ']

 layer_normalization_14 (La  (None, 5, 768)              1536
['add_14[0][0]']
 yerNormalization)

 dense_16 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_14[0][0]
                                                                  ']

 dropout_29 (Dropout)        (None, 5, 1536)             0
['dense_16[0][0]']

 dense_17 (Dense)            (None, 5, 768)              1180416
['dropout_29[0][0]']

 dropout_30 (Dropout)        (None, 5, 768)              0
['dense_17[0][0]']

 dropout_31 (Dropout)        (None, 5, 768)              0
['dropout_30[0][0]']

 add_15 (Add)                (None, 5, 768)              0
```

```
['dropout_31[0][0]',
'layer_normalization_14[0][0]
                                                                  ']

 layer_normalization_15 (La  (None, 5, 768)              1536
['add_15[0][0]']
 yerNormalization)

 multi_head_attention_19 (M  (None, 5, 768)              2831155
['layer_normalization_15[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_15[0][0]
                                                                  ',
'layer_normalization_15[0][0]
                                                                  ']

 dropout_32 (Dropout)        (None, 5, 768)              0
['multi_head_attention_19[0][0
                                                                 ]']

 add_16 (Add)                (None, 5, 768)              0
['dropout_32[0][0]',
'layer_normalization_15[0][0]
                                                                  ']

 layer_normalization_16 (La  (None, 5, 768)              1536
['add_16[0][0]']
 yerNormalization)

 dense_18 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_16[0][0]
                                                                  ']

 dropout_33 (Dropout)        (None, 5, 1536)             0
['dense_18[0][0]']

 dense_19 (Dense)            (None, 5, 768)              1180416
['dropout_33[0][0]']

 dropout_34 (Dropout)        (None, 5, 768)              0
['dense_19[0][0]']

 dropout_35 (Dropout)        (None, 5, 768)              0
['dropout_34[0][0]']

 add_17 (Add)                (None, 5, 768)              0
['dropout_35[0][0]',
'layer_normalization_16[0][0]
```

```
                                                                  ']

 layer_normalization_17 (La  (None, 5, 768)              1536
['add_17[0][0]']
 yerNormalization)

 multi_head_attention_21 (M  (None, 5, 768)              2831155
['layer_normalization_17[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_17[0][0]
                                                                  ',
'layer_normalization_17[0][0]
                                                                  ']

 dropout_36 (Dropout)        (None, 5, 768)              0
['multi_head_attention_21[0][0
                                                                 ]']

 add_18 (Add)                (None, 5, 768)              0
['dropout_36[0][0]',
'layer_normalization_17[0][0]
                                                                  ']

 layer_normalization_18 (La  (None, 5, 768)              1536
['add_18[0][0]']
 yerNormalization)

 dense_20 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_18[0][0]
                                                                  ']

 dropout_37 (Dropout)        (None, 5, 1536)             0
['dense_20[0][0]']

 dense_21 (Dense)            (None, 5, 768)              1180416
['dropout_37[0][0]']

 dropout_38 (Dropout)        (None, 5, 768)              0
['dense_21[0][0]']

 dropout_39 (Dropout)        (None, 5, 768)              0
['dropout_38[0][0]']

 add_19 (Add)                (None, 5, 768)              0
['dropout_39[0][0]',
'layer_normalization_18[0][0]
                                                                  ']
```

```
 layer_normalization_19 (La  (None, 5, 768)              1536
['add_19[0][0]']
 yerNormalization)

 multi_head_attention_23 (M  (None, 5, 768)              2831155
['layer_normalization_19[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_19[0][0]
                                                                  ',
'layer_normalization_19[0][0]
                                                                  ']

 dropout_40 (Dropout)        (None, 5, 768)              0
['multi_head_attention_23[0][0
                                                                 ]']

 add_20 (Add)                (None, 5, 768)              0
['dropout_40[0][0]',
'layer_normalization_19[0][0]
                                                                  ']

 layer_normalization_20 (La  (None, 5, 768)              1536
['add_20[0][0]']
 yerNormalization)

 dense_22 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_20[0][0]
                                                                  ']

 dropout_41 (Dropout)        (None, 5, 1536)             0
['dense_22[0][0]']

 dense_23 (Dense)            (None, 5, 768)              1180416
['dropout_41[0][0]']

 dropout_42 (Dropout)        (None, 5, 768)              0
['dense_23[0][0]']

 dropout_43 (Dropout)        (None, 5, 768)              0
['dropout_42[0][0]']

 add_21 (Add)                (None, 5, 768)              0
['dropout_43[0][0]',
'layer_normalization_20[0][0]
                                                                  ']

 layer_normalization_21 (La  (None, 5, 768)              1536
['add_21[0][0]']
```

```
 yerNormalization)

 multi_head_attention_25 (M  (None, 5, 768)              2831155
['layer_normalization_21[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_21[0][0]
                                                                  ',
'layer_normalization_21[0][0]
                                                                  ']

 dropout_44 (Dropout)        (None, 5, 768)              0
['multi_head_attention_25[0][0
                                                                  ]']

 add_22 (Add)                (None, 5, 768)              0
['dropout_44[0][0]',
'layer_normalization_21[0][0]
                                                                  ']

 layer_normalization_22 (La  (None, 5, 768)              1536
['add_22[0][0]']
 yerNormalization)

 dense_24 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_22[0][0]
                                                                  ']

 dropout_45 (Dropout)        (None, 5, 1536)             0
['dense_24[0][0]']

 dense_25 (Dense)            (None, 5, 768)              1180416
['dropout_45[0][0]']

 dropout_46 (Dropout)        (None, 5, 768)              0
['dense_25[0][0]']

 dropout_47 (Dropout)        (None, 5, 768)              0
['dropout_46[0][0]']

 add_23 (Add)                (None, 5, 768)              0
['dropout_47[0][0]',
'layer_normalization_22[0][0]
                                                                  ']

 layer_normalization_23 (La  (None, 5, 768)              1536
['add_23[0][0]']
 yerNormalization)
```

```
 multi_head_attention_27 (M  (None, 5, 768)              2831155
['layer_normalization_23[0][0]
 ultiHeadAttention)                                     2         ',
'layer_normalization_23[0][0]
                                                                  ',
'layer_normalization_23[0][0]
                                                                  ']

 dropout_48 (Dropout)        (None, 5, 768)              0
['multi_head_attention_27[0][0
                                                                 ]']

 add_24 (Add)                (None, 5, 768)              0
['dropout_48[0][0]',
'layer_normalization_23[0][0]
                                                                  ']

 layer_normalization_24 (La  (None, 5, 768)              1536
['add_24[0][0]']
 yerNormalization)

 dense_26 (Dense)            (None, 5, 1536)             1181184
['layer_normalization_24[0][0]
                                                                  ']

 dropout_49 (Dropout)        (None, 5, 1536)             0
['dense_26[0][0]']

 dense_27 (Dense)            (None, 5, 768)              1180416
['dropout_49[0][0]']

 dropout_50 (Dropout)        (None, 5, 768)              0
['dense_27[0][0]']

 dropout_51 (Dropout)        (None, 5, 768)              0
['dropout_50[0][0]']

 add_25 (Add)                (None, 5, 768)              0
['dropout_51[0][0]',
'layer_normalization_24[0][0]
                                                                  ']

 layer_normalization_25 (La  (None, 5, 768)              1536
['add_25[0][0]']
 yerNormalization)

 tf.__operators__.getitem (  (None, 768)                 0
['layer_normalization_25[0][0]
```

```
 SlicingOpLambda)                                                    ']

 dense_28 (Dense)            (None, 3074)                 2363906
['tf.__operators__.getitem[0][
                                                                    0]']

 dropout_52 (Dropout)        (None, 3074)                 0
['dense_28[0][0]']

 dense_29 (Dense)            (None, 10)                   30750
['dropout_52[0][0]']

==================================================================================
==================
Total params: 371104544 (1.38 GB)
Trainable params: 371104544 (1.38 GB)
Non-trainable params: 0 (0.00 Byte)
----------------------------------------------------------------------------------
------------------
```

Questions

10. Check the original ViT paper and look for the number of model parameters given the same model configurations. Do we have the same number of training parameters? If not, why is this the case? (Hint! compare how the projection matrices are defined in the paper and how we have defined them).

Answers

10. We have a lot more model parameters than the original model. Each head in our case has the same dimension as the projection dimension, in the original paper, the dimension of each head is model projection dimension/number of heads.

1.2.5 Model training

In the next cell, you will define the training helper function that defines the loss, the optimizer and runs the model training and testing given a dataset. Follow the instructions in the cell below and train your first model.

[14]:
```
import tf_keras
import tensorflow as tf
import os

from tf_keras.optimizers import Adam
from tf_keras.losses import SparseCategoricalCrossentropy
from tf_keras.metrics import SparseCategoricalAccuracy

from utilities import plot_classifier_training_history
```

```
def run_experiment(
        model,
        training_data = (Xtrain, Ytrain),
        validation_data = (Xval, Yval),
        test_data = (Xtest, Ytest),
        epochs=30,
        batch_size=64,
        learning_rate=1e-4,
        ckpt_path=".",
        run_testing=True,
    ):
    '''
    Function to run the experiment

    PArameters:
        model: the model to train
        training_data (tuple): tuple of training data (Xtrain, Ytrain)
        validation_data (tuple): tuple of validation data (Xval, Yval)
        test_data (tuple): tuple of test data (Xtest, Ytest)
        epochs (int): number of epochs to train the model
        batch_size (int): batch size for training
        learning_rate (float): learning rate for the optimizer
        top_k (int): top k accuracy to track during training
        ckpt_path (str): path to save the model checkpoint
        run_testing (bool): whether to run testing after training

    Returns:
        history: the training history
    '''
    # ------------------------------------------
    # === Your code here =======================
    # ------------------------------------------

    # define the optimizer
    print("inside run")
    optimizer = Adam(learning_rate=learning_rate)

    # define loss
    print("inside loss")
    loss = SparseCategoricalCrossentropy(from_logits=True)

    # ==========================================
    print("inside metrics")

    # We define additional metrics to track during training and a callback to␣
 ↪save the best model
    metrics = [SparseCategoricalAccuracy(name="accuracy")]
```

```
    print("inside checkpoint")
    # define auxiliary callback (save the best model based on validation␣
↪accuracy)
    checkpoint_filepath = os.path.join(ckpt_path, "checkpoint.weights.h5")
    checkpoint_callback = tf_keras.callbacks.ModelCheckpoint(
        checkpoint_filepath,
        monitor="val_accuracy",
        save_best_only=True,
        save_weights_only=True,
    )

    # ---------------------------------------------
    # === Your code here ========================
    # ---------------------------------------------
    print("inside compile")
    # compile model
    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=metrics,
    )
    print("inside train")

    # train model
    history = model.fit(
        x=training_data[0],
        y=training_data[1],
        batch_size=batch_size,
        epochs=epochs,
        validation_data=validation_data,
        callbacks=[checkpoint_callback],
    )

    # =============================================
    print("Training completed.")

    # evaluate model on the test set
    if run_testing:
        print("\nEvaluating model on the test data.")
        # after model training, load the best weights and evaluate on the test␣
↪set
        model.load_weights(checkpoint_filepath)
        _, accuracy = model.evaluate(test_data[0], test_data[1])
        print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    return history
```

[15]:
```
# Train the ViT model
printhistory = run_experiment(
    model=vit,  # Your created ViT model
    training_data=(Xtrain, Ytrain),
    validation_data=(Xval, Yval),
    test_data=(Xtest, Ytest),
    epochs=30,
    batch_size=64,
    learning_rate=1e-4
)

# Plot the training history
plot_classifier_training_history(printhistory)
```

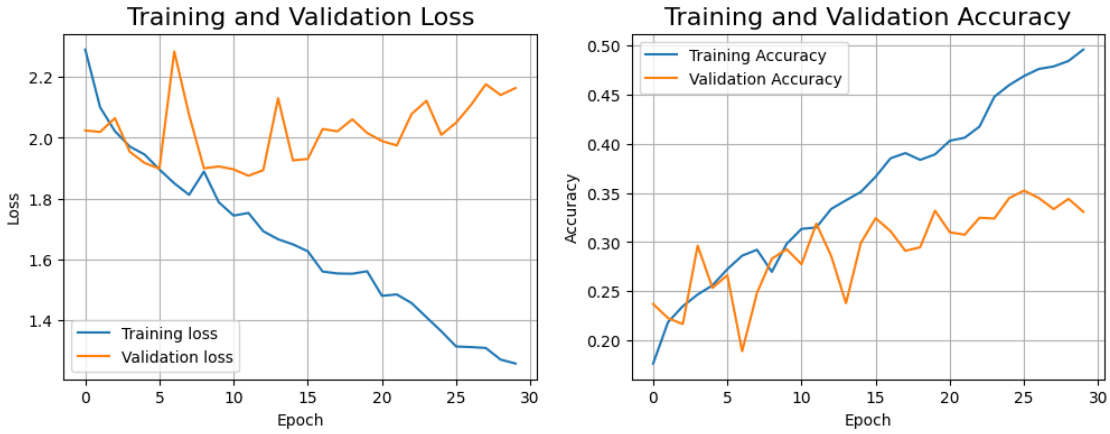
```
inside run
inside loss
inside metrics
inside checkpoint
inside compile
inside train
Epoch 1/30
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1748001642.380308  786400 service.cc:148] XLA service 0x7efd14b04e90
initialized for platform CUDA (this does not guarantee that XLA will be used).
Devices:
I0000 00:00:1748001642.380326  786400 service.cc:156]   StreamExecutor device
(0): NVIDIA RTX 4000 Ada Generation, Compute Capability 8.9
I0000 00:00:1748001642.392921  786400 cuda_dnn.cc:529] Loaded cuDNN version
90300
I0000 00:00:1748001642.426599  786400 device_compiler.h:188] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.
```

```
118/118 [==============================] - 97s 734ms/step - loss: 2.2907 -
accuracy: 0.1767 - val_loss: 2.0242 - val_accuracy: 0.2372
Epoch 2/30
118/118 [==============================] - 14s 116ms/step - loss: 2.1005 -
accuracy: 0.2187 - val_loss: 2.0192 - val_accuracy: 0.2228
Epoch 3/30
118/118 [==============================] - 14s 118ms/step - loss: 2.0219 -
accuracy: 0.2349 - val_loss: 2.0647 - val_accuracy: 0.2168
Epoch 4/30
118/118 [==============================] - 96s 818ms/step - loss: 1.9714 -
accuracy: 0.2468 - val_loss: 1.9542 - val_accuracy: 0.2964
Epoch 5/30
118/118 [==============================] - 14s 119ms/step - loss: 1.9449 -
accuracy: 0.2561 - val_loss: 1.9170 - val_accuracy: 0.2536
```

```
Epoch 6/30
118/118 [==============================] - 14s 118ms/step - loss: 1.8957 -
accuracy: 0.2725 - val_loss: 1.8984 - val_accuracy: 0.2664
Epoch 7/30
118/118 [==============================] - 14s 119ms/step - loss: 1.8502 -
accuracy: 0.2863 - val_loss: 2.2843 - val_accuracy: 0.1892
Epoch 8/30
118/118 [==============================] - 14s 118ms/step - loss: 1.8122 -
accuracy: 0.2923 - val_loss: 2.0755 - val_accuracy: 0.2484
Epoch 9/30
118/118 [==============================] - 14s 119ms/step - loss: 1.8889 -
accuracy: 0.2699 - val_loss: 1.8993 - val_accuracy: 0.2832
Epoch 10/30
118/118 [==============================] - 14s 119ms/step - loss: 1.7878 -
accuracy: 0.2983 - val_loss: 1.9056 - val_accuracy: 0.2928
Epoch 11/30
118/118 [==============================] - 14s 118ms/step - loss: 1.7438 -
accuracy: 0.3135 - val_loss: 1.8962 - val_accuracy: 0.2776
Epoch 12/30
118/118 [==============================] - 89s 762ms/step - loss: 1.7525 -
accuracy: 0.3149 - val_loss: 1.8750 - val_accuracy: 0.3188
Epoch 13/30
118/118 [==============================] - 14s 117ms/step - loss: 1.6922 -
accuracy: 0.3337 - val_loss: 1.8933 - val_accuracy: 0.2860
Epoch 14/30
118/118 [==============================] - 14s 118ms/step - loss: 1.6658 -
accuracy: 0.3425 - val_loss: 2.1302 - val_accuracy: 0.2380
Epoch 15/30
118/118 [==============================] - 14s 117ms/step - loss: 1.6491 -
accuracy: 0.3511 - val_loss: 1.9257 - val_accuracy: 0.2992
Epoch 16/30
118/118 [==============================] - 86s 734ms/step - loss: 1.6264 -
accuracy: 0.3665 - val_loss: 1.9302 - val_accuracy: 0.3244
Epoch 17/30
118/118 [==============================] - 14s 117ms/step - loss: 1.5596 -
accuracy: 0.3852 - val_loss: 2.0290 - val_accuracy: 0.3112
Epoch 18/30
118/118 [==============================] - 14s 118ms/step - loss: 1.5533 -
accuracy: 0.3905 - val_loss: 2.0212 - val_accuracy: 0.2912
Epoch 19/30
118/118 [==============================] - 14s 118ms/step - loss: 1.5523 -
accuracy: 0.3836 - val_loss: 2.0609 - val_accuracy: 0.2948
Epoch 20/30
118/118 [==============================] - 87s 739ms/step - loss: 1.5605 -
accuracy: 0.3892 - val_loss: 2.0161 - val_accuracy: 0.3320
Epoch 21/30
118/118 [==============================] - 14s 117ms/step - loss: 1.4796 -
accuracy: 0.4031 - val_loss: 1.9891 - val_accuracy: 0.3100
```

```
Epoch 22/30
118/118 [==============================] - 14s 117ms/step - loss: 1.4841 -
accuracy: 0.4061 - val_loss: 1.9749 - val_accuracy: 0.3076
Epoch 23/30
118/118 [==============================] - 14s 117ms/step - loss: 1.4556 -
accuracy: 0.4175 - val_loss: 2.0784 - val_accuracy: 0.3248
Epoch 24/30
118/118 [==============================] - 14s 117ms/step - loss: 1.4091 -
accuracy: 0.4477 - val_loss: 2.1218 - val_accuracy: 0.3240
Epoch 25/30
118/118 [==============================] - 91s 780ms/step - loss: 1.3628 -
accuracy: 0.4596 - val_loss: 2.0097 - val_accuracy: 0.3448
Epoch 26/30
118/118 [==============================] - 92s 789ms/step - loss: 1.3128 -
accuracy: 0.4688 - val_loss: 2.0498 - val_accuracy: 0.3524
Epoch 27/30
118/118 [==============================] - 14s 117ms/step - loss: 1.3108 -
accuracy: 0.4760 - val_loss: 2.1080 - val_accuracy: 0.3448
Epoch 28/30
118/118 [==============================] - 14s 118ms/step - loss: 1.3079 -
accuracy: 0.4785 - val_loss: 2.1764 - val_accuracy: 0.3336
Epoch 29/30
118/118 [==============================] - 14s 118ms/step - loss: 1.2695 -
accuracy: 0.4841 - val_loss: 2.1409 - val_accuracy: 0.3440
Epoch 30/30
118/118 [==============================] - 14s 118ms/step - loss: 1.2566 -
accuracy: 0.4957 - val_loss: 2.1639 - val_accuracy: 0.3308
Training completed.

Evaluating model on the test data.
63/63 [==============================] - 2s 32ms/step - loss: 2.0771 - accuracy:
0.3470
Test accuracy: 34.7%
```



Questions

11. Compare the model performance for the ViT and the CNN mode that you trained in the CNN lab. Which one obtained a higher classification accuracy?
12. There is one regularization method that we did not implement in this lab that we did in the CNN lab when it comes to the data. What is it? Can this improve model performance and generalization?
13. Can you think how the output of a ViT model as implemented in this lab can be used for image segmentation?

Answers

11. The accuracy that we obtained in CNN was above 60%. In ViT, the accuracy after 30 epochs was just 34.55%. So, we can conclusively say that the CNN performed better and achieved a higher classification accuracy.
12. We did not use rotation augmentation in this lab which improves model generalization and eventually the performance of the model.
13. In image segmentation task we have to classify at pixel level instead of patch level. Therefore, we can use a model such as U-Net before the classification in this ViT model.

1.3 Carbon footprint

Compare the carbon footprint of training a Vision Transformer (ViT) model to that of a CNN model, as done in the CNN lab. Adjust the number of training and testing samples to match those used in the CNN lab, and set the batch size and number of epochs to the same values. After training the ViT model, measure the energy consumption (kWh) and CO emissions (grams) and compare them to the values obtained for the CNN model.

[16]:
```
from carbontracker.tracker import CarbonTracker

# --------------------------------------------
# === Your code here ========================
# --------------------------------------------

# NOTE: Change the number of training and testing samples so that it matches␣
 ↪that of the CNN lab (there we used 10000 training samples and 2000 testing␣
 ↪samples)

# define model
# model_params = {
#     # dataset information
#     "input_shape": ...,
#     "num_classes": ...,
#     # patch extraction
#     "patch_size": ...,
#     # patch encoder
#     "embedding_proj_dim": ...,
```

```
#     # transformer encoder
#     "num_heads": ...,
#     "msa_proj_dim": ...,
#     "transformer_layers": ...,
#     "msa_dropout_rate": ...,
#     # classification head
#     "mlp_classification_head_units": [...],
#     "mlp_classification_head_dropout_rate": ...
# }
model_params = {
    # dataset information
    "input_shape": (32, 32, 3),
    "num_classes": 10,
    # patch extraction
    "patch_size": 16,
    # patch encoder
    "embedding_proj_dim": 512,
    # transformer encoder
    "num_heads": 4,
    "msa_proj_dim": 512,
    "transformer_layers": 3,
    "msa_dropout_rate": 0.1,
    # classification head
    "mlp_classification_head_units": [2048],
    "mlp_classification_head_dropout_rate": 0.5
}

# create the model
vit = create_vit_classifier(**model_params)

# define experiment parameters as a dictionary
# experiment_params = {
#     "model": ...,
#     "training_data": (...),
#     "validation_data": (...),
#     "test_data": (...),
#     "epochs": ...,
#     "batch_size": ...,
#     "learning_rate": ...,
#     "ckpt_path": ".",
#     "run_testing": ...
# }

experiment_params = {
    "model": vit,
    "training_data": (Xtrain, Ytrain),
    "validation_data": (Xval, Yval),
```

```
    "test_data": (Xtest, Ytest),
    "epochs": 30,
    "batch_size": 64,
    "learning_rate": 1e-4,
    "ckpt_path": ".",
    "run_testing": True
}

# run the experiment with carbontracker
# Create a CarbonTracker object
tracker = CarbonTracker(epochs=experiment_params["epochs"])

# start carbon tracking
tracker.epoch_start()

# run the experiment
history = run_experiment(**experiment_params)

tracker.epoch_end()
```

```
inside run
inside loss
inside metrics
inside checkpoint
inside compile
inside train
CarbonTracker: The following components were found: GPU with device(s) NVIDIA
RTX 4000 Ada Generation. CPU with device(s) cpu:0.
No sudo access to read Intel's RAPL measurements from the energy_uj file.
See issue: https://github.com/lfwa/carbontracker/issues/40
Epoch 1/30
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
118/118 [==============================] - 12s 79ms/step - loss: 2.1540 -
accuracy: 0.2303 - val_loss: 1.9709 - val_accuracy: 0.2940
Epoch 2/30
118/118 [==============================] - 4s 36ms/step - loss: 1.9569 -
accuracy: 0.2928 - val_loss: 1.8338 - val_accuracy: 0.3392
Epoch 3/30
118/118 [==============================] - 4s 36ms/step - loss: 1.8370 -
accuracy: 0.3361 - val_loss: 1.7766 - val_accuracy: 0.3532
Epoch 4/30
118/118 [==============================] - 5s 41ms/step - loss: 1.7391 -
accuracy: 0.3693 - val_loss: 1.7225 - val_accuracy: 0.3772
Epoch 5/30
```

```
118/118 [==============================] - 4s 37ms/step - loss: 1.6435 -
accuracy: 0.4025 - val_loss: 1.7206 - val_accuracy: 0.3884
Epoch 6/30
118/118 [==============================] - 2s 18ms/step - loss: 1.5770 -
accuracy: 0.4307 - val_loss: 1.8080 - val_accuracy: 0.3672
Epoch 7/30
118/118 [==============================] - 2s 17ms/step - loss: 1.4782 -
accuracy: 0.4653 - val_loss: 1.7824 - val_accuracy: 0.3856
Epoch 8/30
118/118 [==============================] - 4s 37ms/step - loss: 1.3902 -
accuracy: 0.4987 - val_loss: 1.7414 - val_accuracy: 0.4120
Epoch 9/30
118/118 [==============================] - 4s 37ms/step - loss: 1.2771 -
accuracy: 0.5361 - val_loss: 1.7468 - val_accuracy: 0.4132
Epoch 10/30
118/118 [==============================] - 2s 17ms/step - loss: 1.1703 -
accuracy: 0.5859 - val_loss: 1.8086 - val_accuracy: 0.4072
Epoch 11/30
118/118 [==============================] - 2s 17ms/step - loss: 1.0591 -
accuracy: 0.6189 - val_loss: 1.8118 - val_accuracy: 0.4128
Epoch 12/30
118/118 [==============================] - 2s 18ms/step - loss: 0.9413 -
accuracy: 0.6593 - val_loss: 2.0299 - val_accuracy: 0.4052
Epoch 13/30
118/118 [==============================] - 2s 18ms/step - loss: 0.8352 -
accuracy: 0.7008 - val_loss: 2.0939 - val_accuracy: 0.4084
Epoch 14/30
118/118 [==============================] - 2s 18ms/step - loss: 0.7561 -
accuracy: 0.7319 - val_loss: 2.1596 - val_accuracy: 0.4068
Epoch 15/30
118/118 [==============================] - 2s 18ms/step - loss: 0.6473 -
accuracy: 0.7696 - val_loss: 2.3273 - val_accuracy: 0.3932
Epoch 16/30
118/118 [==============================] - 2s 18ms/step - loss: 0.5772 -
accuracy: 0.7979 - val_loss: 2.4429 - val_accuracy: 0.4052
Epoch 17/30
118/118 [==============================] - 2s 18ms/step - loss: 0.5186 -
accuracy: 0.8112 - val_loss: 2.5038 - val_accuracy: 0.4080
Epoch 18/30
118/118 [==============================] - 2s 17ms/step - loss: 0.4372 -
accuracy: 0.8437 - val_loss: 2.6667 - val_accuracy: 0.4004
Epoch 19/30
118/118 [==============================] - 2s 18ms/step - loss: 0.3610 -
accuracy: 0.8747 - val_loss: 2.9386 - val_accuracy: 0.3908
Epoch 20/30
118/118 [==============================] - 2s 17ms/step - loss: 0.3363 -
accuracy: 0.8821 - val_loss: 2.8827 - val_accuracy: 0.4116
Epoch 21/30
```

```
118/118 [==============================] - 2s 17ms/step - loss: 0.2803 -
accuracy: 0.9009 - val_loss: 3.0192 - val_accuracy: 0.4004
Epoch 22/30
118/118 [==============================] - 2s 18ms/step - loss: 0.2743 -
accuracy: 0.9023 - val_loss: 3.0323 - val_accuracy: 0.3964
Epoch 23/30
118/118 [==============================] - 2s 17ms/step - loss: 0.2256 -
accuracy: 0.9215 - val_loss: 3.2338 - val_accuracy: 0.3972
Epoch 24/30
118/118 [==============================] - 2s 18ms/step - loss: 0.2209 -
accuracy: 0.9239 - val_loss: 3.2258 - val_accuracy: 0.4080
Epoch 25/30
118/118 [==============================] - 2s 17ms/step - loss: 0.1794 -
accuracy: 0.9396 - val_loss: 3.4067 - val_accuracy: 0.4120
Epoch 26/30
118/118 [==============================] - 2s 16ms/step - loss: 0.1833 -
accuracy: 0.9415 - val_loss: 3.4382 - val_accuracy: 0.4032
Epoch 27/30
118/118 [==============================] - 4s 37ms/step - loss: 0.1602 -
accuracy: 0.9461 - val_loss: 3.5068 - val_accuracy: 0.4144
Epoch 28/30
118/118 [==============================] - 2s 18ms/step - loss: 0.1417 -
accuracy: 0.9515 - val_loss: 3.6788 - val_accuracy: 0.4056
Epoch 29/30
118/118 [==============================] - 2s 18ms/step - loss: 0.1704 -
accuracy: 0.9416 - val_loss: 3.5991 - val_accuracy: 0.4124
Epoch 30/30
118/118 [==============================] - 2s 17ms/step - loss: 0.1607 -
accuracy: 0.9443 - val_loss: 3.7134 - val_accuracy: 0.3960
Training completed.

Evaluating model on the test data.
63/63 [==============================] - 0s 6ms/step - loss: 3.6614 - accuracy:
0.3915
Test accuracy: 39.15%
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: Live carbon intensity could not be fetched at detected location:
Linköping, Östergötland, SE. Defaulted to average carbon intensity for SE in
2023 of 40.69 gCO2/kWh. at detected location: Linköping, Östergötland, SE.
CarbonTracker:
Predicted consumption for 30 epoch(s):
        Time:   0:45:24
        Energy: 0.062808736450 kWh
        CO2eq:  2.555993867168 g
        This is equivalent to:
```

```
0.023776687136 km travelled by car
```

Questions

14. Compare the carbon footprint for training a ViT model with that of the CNN model of the CNN lab: which one had the highest footprint? Describe what is the reason of the difference.

Answers

14. The carbon footprint for training a ViT model (2.555993867168 g) is expensive than the CNN model (2.335625311580 g). It increased for ViT as it has more complex architecture, and larger training time than the CNN model.

1.4 Pre-trained ViT model

As for the CNN lab, there exist a variety of pre-trained ViT models that can be imported and used as they are for natural image classification. Tensorflow and Keras do not have pre-trained versions the ViT architecture in their repository. For this reason, here we are using the `transformer` library that loads Pytorch ViT model weights into a Tensorflow-defined model. See the documentation for more information.

[17]:
```
from transformers import ViTFeatureExtractor, TFViTForImageClassification
from tf_keras.utils import load_img, img_to_array

# ------------------------------------------
# === Your code here =======================
# ------------------------------------------

# load the image and preprocess it (similar to the CNN lab)
image_path = "busy_road.jpg"
image = load_img(image_path, target_size=(224, 224))
image = img_to_array(image)

# ==========================================

# Define a feature extractor and a model: here we will load the␣
 ↪vit-base-patch16-224 model
feature_extractor = ViTFeatureExtractor.from_pretrained("google/
 ↪vit-base-patch16-224")
model = TFViTForImageClassification.from_pretrained('google/
 ↪vit-base-patch16-224')

# Pre-process the image and make a prediction
inputs = feature_extractor(images=image, return_tensors="tf")
outputs = model(**inputs)
logits = outputs.logits

# model predicts one of the 1000 ImageNet classes
predicted_class_idx = tf.math.argmax(logits, axis=-1)[0]
```

```
print("Predicted class:", model.config.id2label[int(predicted_class_idx)])
```

```
preprocessor_config.json:   0%|          | 0.00/160 [00:00<?, ?B/s]
```

```
/opt/liu/course-venv-732a82/2/1/lib/python3.10/site-
packages/transformers/models/vit/feature_extraction_vit.py:28: FutureWarning:
The class ViTFeatureExtractor is deprecated and will be removed in version 5 of
Transformers. Please use ViTImageProcessor instead.
  warnings.warn(
```

```
config.json:   0%|          | 0.00/69.7k [00:00<?, ?B/s]
```

```
model.safetensors:   0%|          | 0.00/346M [00:00<?, ?B/s]
```

```
All PyTorch model weights were used when initializing
TFViTForImageClassification.


All the weights of TFViTForImageClassification were initialized from the PyTorch
model.
If your task is similar to the task the model of the checkpoint was trained on,
you can already use TFViTForImageClassification for predictions without further
training.
```

```
Predicted class: racer, race car, racing car
```

1.5 (OPTIONAL) Transformers for text summarization

In this next section you can play around with a sequence-to-sequence (Seq2Seq) transformer model for text summarization. As before, we will use a pre-trained model from the `transformer` library. In this example we will use the T5 model which can be prompted to perform several NLP tasks, such as summarization.

For short text, you can input the text directly as a variable. For longer texts, copy it into a .txt file, which will be then loaded and fed to the model. You will see that the longer the input text, the better the answer.

[ ]:
```
# Load both the tokenizer and the model
from transformers import T5Tokenizer, TFT5ForConditionalGeneration
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = TFT5ForConditionalGeneration.from_pretrained("t5-small")
```

[ ]:
```
# Load .txt or input text for summary manually

# Load the text file
with open('NLP_text.txt', 'r') as file:
    text = file.read()

# Or input text manually
# text = 'Some text to summarize'

# get the number of words in the text
```

```
nbr_words = len(text.split())
# set the maximum length of the model output to 50% of the number of original␣
 ↪words
max_length = int(0.5 * nbr_words)
# set the minimum number of output words to 10
min_length = 10

# print the number of words in the text
print(f"Number of words in the text: {nbr_words}")
```

[ ]:
```
# specify the task that the model should perform
task = 'summarize'
# create the input string for the model and pass it thought the tokenizer
input_ids = tokenizer(f"{task} : {text}", return_tensors="tf").input_ids
# generate model outputs
outputs = model.generate(input_ids, max_length=max_length,␣
 ↪min_length=min_length, do_sample=True)

# print model output
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```