

Advanced Applications in Computer Vision: Finetuning Image Classification Models with Transformers

Topic Areas: Pre-Training Transformers Vision Transformers (ViTs) Image Classification Image Augmentation Image Segmentation Attention Mechanisms Classification Models

Matthew Harper
University of Houston Victoria

Abstract: Human vision is largely considered as the highest input and processing capacity and contributes substantially to our understanding of the world, and the interaction with other entities. Moreover, approximately one third of the human brain is allocated for vision and image recognition and processing tasks such as object detection and depth and distance inference [3]. Similarly, in computer vision applications, a digitized input image is captured by the computer and converted into an internal representation suitable for processing and interpretation. Typical applications of computer vision include edge detection, pattern and object recognition, the extraction of depth information, and motion detection [1,4]. Unlike classical CNN model methods, computer vision with Vision Transformer models exhibits enhanced performance and ability in image segmentation, better scalability and integration into natural language processing models, superior long-term dependencies with the use of attention mechanisms, and ability to integrate into generative models [1].

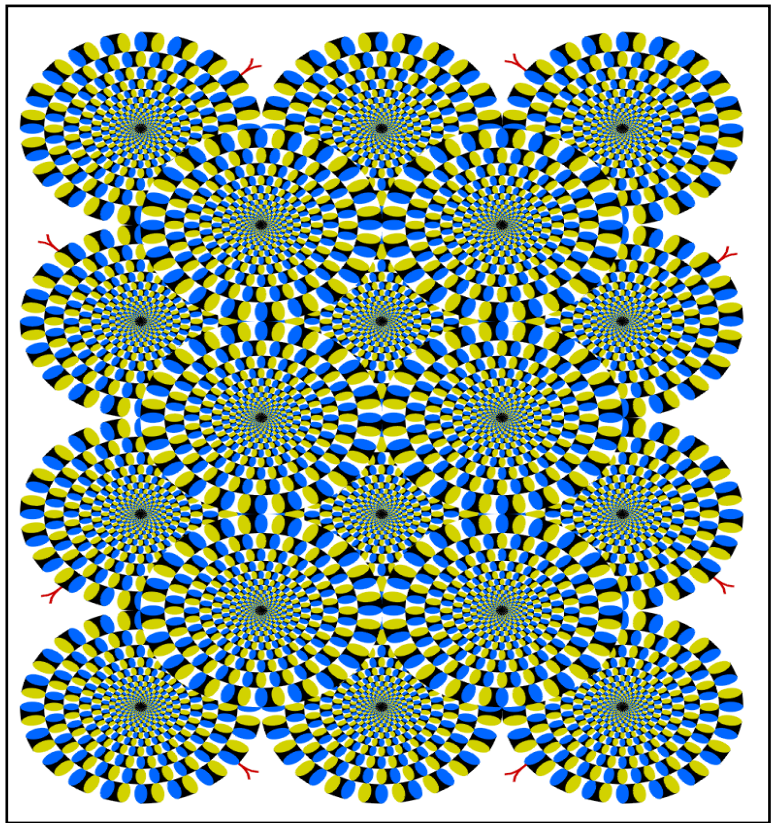


Figure 1 – An example of the visual effect of illusory motion with Akiyoshi Kitaoka's rotating snake illusions in which suggest that both ocular motion and the information processing of the cerebral cortex are responsible for the perception of illusory motion ([Hisakata and Murakami, 2008](#); [Kuriki et al., 2008](#); [Ashida et al., 2012](#)) [5].

Introduction

Computer vision, the ability of machines to understand and interpret visual data, has evolved significantly since its inception in the late 1950s. Early research focused on understanding how the human brain processes visual information and developing algorithms to detect basic shapes like edges and lines [2].

Key advancements included the development of computer image scanning technology, the ability to convert 2D images into 3D forms, and the introduction of optical character recognition (OCR) and intelligent character recognition (ICR) for text processing [2].

In the 1980s, researchers began exploring hierarchical approaches to vision and developing neural networks for pattern recognition. The 2000s saw a shift towards object recognition and the standardization of visual data sets. Deep learning models, such as AlexNet, introduced in 2012, have dramatically improved the accuracy of image recognition tasks [2].

Today modern vision models have been introduced to methods and applications involving the use of vision transformers which enable superior performance, particularly for long-term dependencies, and are highly scalable and able to be integrated into existing natural language systems [1].

Even today, computer vision is a rapidly advancing field that bridges the gap between visual perception and digital analysis. Computer vision is a field within artificial intelligence that uses machine learning and deep neural networks to train computers and systems to derive meaningful information from digital images, videos, and visual information [1, 2]. Computer vision attempts to mimic human vision, except humans have a head start. Human sight has the advantage of lifetimes of context to train how to tell objects apart, how far away they are, whether they are moving or something is wrong with an image [2].

Computer vision trains machines to perform these functions, but it must do it in much less time with cameras, data and algorithms rather than retinas, optic nerves and a visual cortex. Because a system trained to inspect products or watch a production asset can analyze thousands of products or processes a minute, noticing imperceptible defects or issues, it can quickly surpass human capabilities [2].

As a result, by training computers and systems to derive meaningful information from digital images, videos, and other visual inputs, computer vision can be applied to many fields and tasks such as object recognition or image classification among others. The process on how a computer interprets images is similar to how human vision works, however, humans have an inherent ability to see and interpret their surroundings, while computer vision requires a combination of algorithms and machine learning to interpret visual data [1].

At its core, computer vision seeks to automate tasks that the human visual system can do, such as object recognition, image classification, facial recognition, and more. However, beyond mere recognition, it also involves the ability to analyze, process, and even generate images, pushing the boundaries of what machines can do with visual data. Where computer vision applications struggled in the past, advances have been made with the use and implementation of CNN based computer vision models, and today with the modern applications of transformers [1].

Applications

Computer vision has been implemented in many fields and applications and use cases continue to grow as technology and the quality of image recognition and processing improves. A few of the wide range of applications and industries include the following [1]:

- Image Recognition and Classification: The ability to identify objects, people, and anomalies within an image which is foundational for many computer vision applications.
- Object Detection: where an object can be located within an image or video stream and is often utilized for autonomous systems, surveillance, and security.
- Image Segmentation: The process of dividing an image into patches or segments to isolate specific elements like cars in a parking lot, faces in a crowd, or medical image processing to locate and distinguish tumors.
- Facial Recognition: Computer vision is often utilized for facial recognition applications in the recognition and detection and verification of human faces used in security, surveillance, social media, and customer service.
- Optical Character Recognition (OCR): Extracting text from images, used in digitizing documents or reading license plates.
- 3D Vision: Understanding depth and dimensions from visual streams and images is important for humanoid robotics and machines as well as applications in augmented reality (AR).

Computer Vision using Transformers

Modern computer vision concepts and models are transitioning from traditional computer vision methods using CNN models in favor of vision transformers which feature the use of self-attention mechanisms to image and process data in a manner which is highly scalable, integrates well into NLP models, and performs well with long-term dependencies [1].

ViT Data Pipeline

The key concept with ViTs is that the transformer utilizes attention mechanisms to image rather than using convolutional layers to process images. This involves dividing an image into patches similar to how NLP divides sentences into tokens, which can be processed and analyzed within the transformer architecture.

The ViT data processing pipeline includes the following methods [1]:

- Image Patches – input image is divided into fixed sized patches
- Patch Embedding – patches are flattened and embedded into a vector similar to tokenizing text strings.
- Positional Encoding – Positional encodings are added to the embeddings to retain important information.
- Transformer Processing - Embeddings are then processed with multiple layers of transformers where attention mechanisms detect important features within the image patch based on the relevance to the tasks and effect on the output.
- Classification -The output is typically a classification token that is used to predict labels for the image and can be used for object detection.

Image Augmentation

Image augmentation is a powerful image processing technique used to artificially increase the size and diversity of a training dataset by generating modified versions of images in the dataset. Its main purpose is to increase the size and data diversity of existing training data which is often difficult to collect and expensive to store [1].

Image augmentation is often implemented by processing the original images with transformations such as rotations, scaling, flipping, and more (cropping, translation, shearing, color jittering). These transformations create new training examples, which help improve the robustness and generalization of machine learning models. An essential strategy in computer vision that enhances model performance by artificially expanding the size and diversity of a training dataset. Moreover, by introducing variations in the data, image augmentation helps prevent the effects of model overfitting, improves model generalization, and enhances a model's ability to handle unseen data [1].

Image Segmentation

Image segmentation is a computer vision technique which involves dividing an image into separate regions called segments. Moreover, with segmentation, each segment represents a different part of the image. This is particularly useful for image processing tasks where it is important to understand the structure of the image at very fine granularity such as the pixel level. This approach is similar to a divide and conquer method which features two main types which include semantic and instance segmentation [1].

Semantic Segmentation: Semantic segmentation is an image segmentation method in which each pixel in the image is classified into a category, and all pixels belonging to a category are assigned a label.

Instance Segmentation: Instance segmentation is an image segmentation method which not only labels each pixel but also distinguishes between different instances of the same category. To accomplish this, instance segmentation combines object detection with semantic segmentation.

3D Vision and Depth Estimation

3D vision is a fundamental aspect of computer vision which enables machines to perceive depth and understand spatial arrangements of objects in a three-dimensional perspective. This capability is similar to human binocular vision, which utilizes the slightly different perspectives captured by each eye to create a sense of depth. Moreover, by using various 3D and depth perception techniques, machines and computers can disseminate and estimate distances between objects and the camera or within an image or video stream [1].

Depth estimation is an important skill for machines to utilize for tasks such as navigation, obstacle avoidance, scene reconstruction, and image generation. In addition, by understanding the distance to objects relative to machines or other objects, computer vision systems can determine how to interact and adjust to objects in the physical world with spatial arrangement. The two primary techniques for depth estimation include stereo vision and LiDAR or Light Detection and Ranging as follows [1]:

Stereo Vision:

- Simulates human binocular vision using two cameras.
- Compares images to calculate disparity and estimate depth.
- Widely used in autonomous vehicles, drones, and robotics.
- Advantages - low-cost, passive, suitable for many applications.

LiDAR:

- Uses laser pulses to measure distances to objects.
- Creates detailed 3D maps.
- Used in autonomous vehicles, surveying, and geospatial analysis.
- Advantages – Highly accurate, robust spatial inference for real-world applications.

Applications for 3D and depth estimation include autonomous driving, collision avoidance, robotics, drones, and augmented reality just to name a few [1].

Lab 4 introduces the student to introductory concepts in fine-tuning pre-trained image classification models with transformers.

Module 4: Lab 4 – How to Fine-tune a Model for Text Classification

Code Reference: Huggingface. (n.d.). notebooks/examples/image_classification.ipynb at main · huggingface/notebooks. GitHub.

https://github.com/huggingface/notebooks/blob/main/examples/image_classification.ipynb

The process for the Transformer Classification:

Step 1 – Load the Python Libraries datasets, evaluate, transforms et.al.

Step 2 – Upload EuroSAT dataset

Step 3 – Import Metrics_List and Accuracy for Model evaluation

Step 4 – Load the Vision Transformer for train and validation

Step 5 – Define the Image Augmentation

Step 6 – Train the Model

Step 7 – Fine-Tune the Model

Step 8 – Model Inference

Step 7 – Model Inference with Pipeline API

Course: COSC 6370 Advanced Topics in AI

Student: Matthew Harper

Assignment: Lab 4 - Image Classification with Transformers

Reference: https://github.com/huggingface/notebooks/blob/main/examples/image_classification-tf.ipynb

Step 1 – Load the Python Libraries datasets, transformers, and evaluate

The first of the lab is to import the required libraries.

```
!pip install datasets==2.18.0
```

```
[ ] !pip install evaluate
```

```
[ ] !pip install transformers
```

Assign the Pre-trained model that we will fine-tune to our dataset with batch size 32.

```
[9] model_checkpoint = "microsoft/swin-tiny-patch4-window7-224" # pre-trained model from which to fine-tune  
    batch_size = 32 # batch size for training and evaluation
```

Step 2 - Next we need to install the notebook login feature for hugging face and create user account.

The screenshot shows the Hugging Face user profile for 'matthew harper' (username: mattwharper). The 'Access Tokens' section is active, displaying a table of user access tokens. The table has columns for Name, Value, Last Refreshed Date, Last Used Date, and Permissions. Two tokens are listed: 'Image_Classification' and 'COSC_6370', both with 'FINEGRAINED' permissions. A '+ Create new token' button is visible in the top right of the section.

Name	Value	Last Refreshed Date	Last Used Date	Permissions
Image_Classification	hf_...lmyJ	1 minute ago	-	FINEGRAINED
COSC_6370	hf_...DkIN	9 days ago	9 days ago	FINEGRAINED

```
[10] from huggingface_hub import notebook_login
```

```
notebook_login()
```



Token is valid (permission: fineGrained).

Your token has been saved in your configured git credential helpers (store).

Your token has been saved to /root/.cache/huggingface/token

Login successful

Install the git-lfs which is a GIT extension for versioning large files and training data.

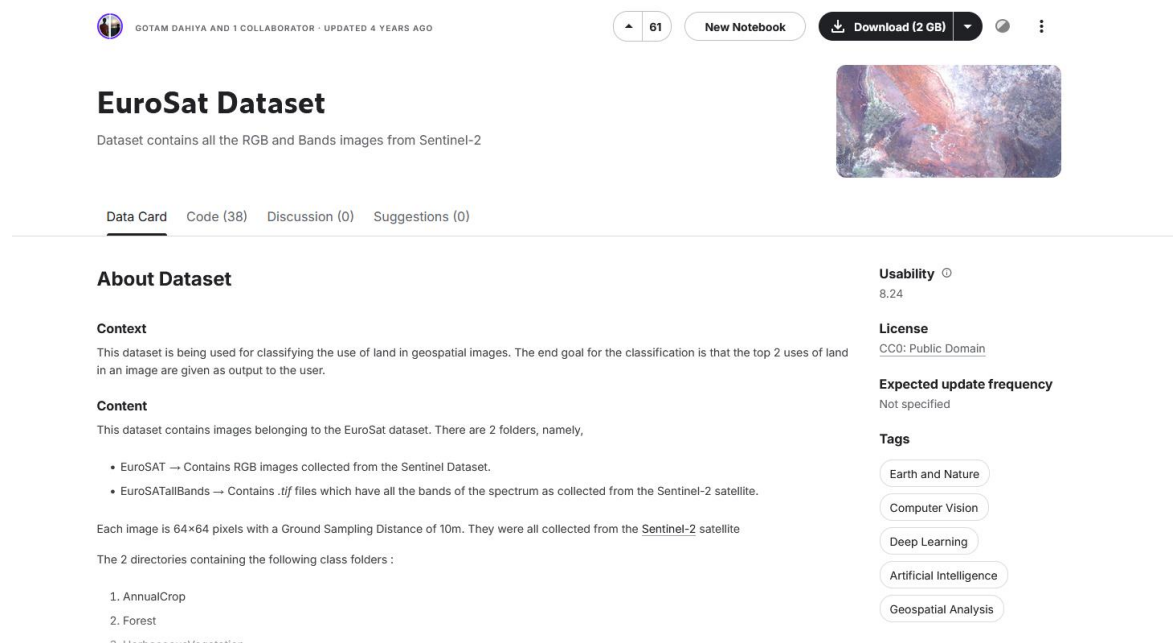
```
[ ] %%capture
!git lfs install
!git config --global credential.helper store
```

Step 2 - Upload the Dataset

Now we need to load the dataset. To do this we first need to have the EuroSAT dataset zip folder. I had an issue with downloading this from colab but was able to simply download the dataset zip and load it in the content folder.

<https://madm.dfki.de/files/sentinel/EuroSAT.zip>

I also located the dataset in Kaggle below.



EuroSat Dataset

Dataset contains all the RGB and Bands images from Sentinel-2

About Dataset

Context

This dataset is being used for classifying the use of land in geospatial images. The end goal for the classification is that the top 2 uses of land in an image are given as output to the user.

Content

This dataset contains images belonging to the EuroSat dataset. There are 2 folders, namely,

- EuroSAT → Contains RGB images collected from the Sentinel Dataset.
- EuroSATallBands → Contains .tif files which have all the bands of the spectrum as collected from the Sentinel-2 satellite.

Each image is 64×64 pixels with a Ground Sampling Distance of 10m. They were all collected from the Sentinel-2 satellite

The 2 directories containing the following class folders :

1. AnnualCrop
2. Forest
3. HerbaceousVegetation

Usability 8.24

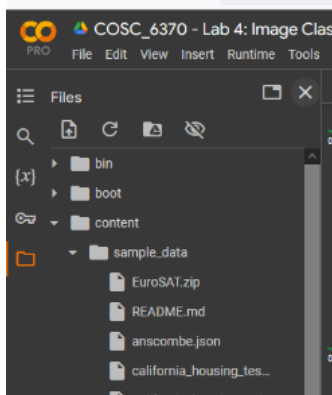
License CC0: Public Domain

Expected update frequency Not specified

Tags

- Earth and Nature
- Computer Vision
- Deep Learning
- Artificial Intelligence
- Geospatial Analysis

<https://www.kaggle.com/datasets/apollo2506/eurosat-dataset>



Once the dataset is upload to the contents drive folder in colab you can right click and select copy path. Then copy and paste the dataset folder location in quotations below.

```
] from datasets import load_dataset

# load a custom dataset from local/remote files or folders using the ImageFolder feature

dataset = load_dataset("imagefolder", data_files="/content/sample_data/EuroSAT.zip")
```

Step 3 – Import List Metrics and Assign Accuracy

Next we want to import the metrics utilized to determine the accuracy. We will import accuracy in a later step but I also wanted to determine how many and what metrics list items were available. According to the routine below there is 281 different metrics available.

```
from datasets import list_metrics

metrics_list = list_metrics()

for i in metrics_list:

    print(i)

#print the total number of metrics available from datasets
print("total number of metrics is: ", len(metrics_list))
```



```
↗ accuracy
bertscore
bleu
bleurt
brier_score
cer
character
charcut_mt
chrif
yongting/average_precision_score
youssef101/accuracy
youssef101/f1
yqsong/execution_accuracy
yulong-me/y1_metric
yuyijiong/quad_match_score
yzha/ctc_eval
zbeloki/m2
zsqr/ter
total number of metrics is: 281

[ ] metric = load_metric("accuracy")
↗ /usr/local/lib/python3.10/dist-packages/datasets/load.py:756: FutureWarning: The repository for accuracy contains
You can avoid this message in future by passing the argument `trust_remote_code=True`.
Passing `trust_remote_code=True` will be mandatory to load this metric from the next major release of `datasets`.
warnings.warn(
<
```

Verify Dataset is loaded properly. Observe the number of images with labels = 27000.

```
dataset
↗ DatasetDict({
  train: Dataset({
    features: ['image', 'label'],
    num_rows: 27000
  })
})
```

Load and example dataset image with features. This will display the labeled features within the dataset. Features include names=['AnnualCrop', 'Forest', 'HerbaceousVegetation', 'Highway', 'Industrial', 'Pasture', 'PermanentCrop', 'Residential', 'River', 'SeaLake']

Then display the image and resize for better view.

Load the training feature labels into the dataset and determine what type or what label does our example image have? Label Id [0] = 'AnnualCrop'


```
[ ] example = dataset["train"][10]
example

{ 'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=64x64>,
  'label': 0 }
```

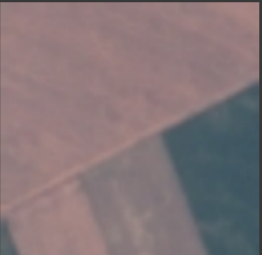
```
[ ] dataset["train"].features

{ 'image': Image(decode=True, id=None),
  'label': ClassLabel(names=[ 'AnnualCrop', 'Forest', 'HerbaceousVegetation', 'Highway', 'Industrial', 'Pasture', 'PermanentCrop', 'Residential', 'Road', 'Shrubland', 'Water']) }
```

```
[ ] example['image']


```

```
example['image'].resize((200, 200))


```

```
[ ] example['label']

0
```

```
[ ] dataset["train"].features["label"]

ClassLabel(names=[ 'AnnualCrop', 'Forest', 'HerbaceousVegetation', 'Highway', 'Industrial', 'Pasture', 'PermanentCrop', 'Residential', 'Road', 'Shrubland', 'Water'])
```

```
[ ] labels = dataset["train"].features["label"].names
label2id, id2label = dict(), dict()
for i, label in enumerate(labels):
    label2id[label] = i
    id2label[i] = label

id2label[0]
```

```
'AnnualCrop'
```

Step 4 - Processing the Data with a Vision Transformer (ViT)

Next we load the Vision Transformer from AutoImageProcessor and call the `image_processor` to display the attributes.

Below we can see that `normalize`, `rescale`, and `resize` is defaulted to the “True” value and is resizing to a 2D tensor image of size 224x224.

```
[ ] from transformers import AutoImageProcessor

image_processor = AutoImageProcessor.from_pretrained(model_checkpoint)
image_processor
```

```
⇒ ViTImageProcessor {
  "do_normalize": true,
  "do_rescale": true,
  "do_resize": true,
  "image_mean": [
    0.485,
    0.456,
    0.406
  ],
  "image_processor_type": "ViTImageProcessor",
  "image_std": [
    0.229,
    0.224,
    0.225
  ],
  "resample": 3,
  "rescale_factor": 0.00392156862745098,
  "size": {
    "height": 224,
    "width": 224
  }
}
```

Optional Step: This next part is optional but displays 10 images that are resized which gives an example of the EuroSAT dataset images that we will be attempting to perform image classification on.

```

import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

n = 10
fig, ax = plt.subplots(2, n, figsize=(20, 5), dpi=80, sharex=True, sharey=True)

for i in range(n):
    orig_img = dataset['train'][i]['image']

    #proc_img = train_transforms(orig_img)

    #the method below works better for resizing
    proc_img = orig_img.resize((200, 200))
    orig_img = np.array(orig_img.convert("RGB"))

    # In order to plot and easy compare the images,
    # we denormalise and rescale here so that pixel values
    # are between [0, 255] and reorder to be HWC
    #proc_img = process_for_plotting(proc_img)

    ax[0][i].imshow(orig_img)
    ax[1][i].imshow(proc_img)
    ax[0][i].axis('off')
    ax[1][i].axis('off')

```



Step 5 – Define Train & Validate and Create Image Augmentation

Using Torchvision load the following libraries CenterCrop, Compose, Normalize, RandomHorizontalFlip, RandomResizedCrop, Resize, ToTensor.

Then define two functions to train and validate the model. To assist with overfitting and training the model we use random horizontal flip and random resized as image augmentation techniques.

```

from torchvision.transforms import (
    CenterCrop,
    Compose,
    Normalize,
    RandomHorizontalFlip,
    RandomResizedCrop,
    Resize,
    ToTensor,
)

normalize = Normalize(mean=image_processor.image_mean, std=image_processor.image_std)
if "height" in image_processor.size:
    size = (image_processor.size["height"], image_processor.size["width"])
    crop_size = size
    max_size = None
elif "shortest_edge" in image_processor.size:
    size = image_processor.size["shortest_edge"]
    crop_size = (size, size)
    max_size = image_processor.size.get("longest_edge")

train_transforms = Compose(
    [
        RandomResizedCrop(crop_size),
        RandomHorizontalFlip(),
        ToTensor(),
        normalize,
    ]
)

val_transforms = Compose(
    [
        Resize(size),
        CenterCrop(crop_size),
        ToTensor(),
        normalize,
    ]
)

def preprocess_train(example_batch):
    """Apply train_transforms across a batch."""
    example_batch["pixel_values"] = [
        train_transforms(image.convert("RGB")) for image in example_batch["image"]
    ]
    return example_batch

def preprocess_val(example_batch):
    """Apply val_transforms across a batch."""
    example_batch["pixel_values"] = [val_transforms(image.convert("RGB")) for image in example_batch["image"]]
    return example_batch

```

Split the dataset into train and test batches with test = 10% of dataset.

```

[ ] ## split up training into training + validation
    splits = dataset["train"].train_test_split(test_size=0.1)
    train_ds = splits['train']
    val_ds = splits['test']

```

Use the `set_transform` method to apply the `train_ds` and `val_ds` functions above when the images are loaded in RAM.

Semantic Segmentation - As an example, call the `train_ds[0]` image after processing to see the tensor values for each pixel.

```
[ ] train_ds.set_transform(preprocess_train)
    val_ds.set_transform(preprocess_val)
```

```
▶ train_ds[0]
```

```
{'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=64x64>,
 'label': 0,
 'pixel_values': tensor([[2.2147, 2.2147, 2.2147, ..., 2.2147, 2.2147, 2.2147],
 [2.2147, 2.2147, 2.2147, ..., 2.2147, 2.2147, 2.2147],
 [2.2147, 2.2147, 2.2147, ..., 2.2147, 2.2147, 2.2147],
 ...,
 [2.2489, 2.2489, 2.2489, ..., 2.2489, 2.2489, 2.2489],
 [2.2489, 2.2489, 2.2489, ..., 2.2489, 2.2489, 2.2489],
 [2.2489, 2.2489, 2.2489, ..., 2.2489, 2.2489, 2.2489]],
 [[1.1506, 1.1506, 1.1506, ..., 1.2031, 1.2031, 1.2031],
 [1.1506, 1.1506, 1.1506, ..., 1.2031, 1.2031, 1.2031],
 [1.1506, 1.1506, 1.1506, ..., 1.2031, 1.2031, 1.2031],
 ...,
 [1.2206, 1.2206, 1.2206, ..., 1.1506, 1.1506, 1.1506],
 [1.2206, 1.2206, 1.2206, ..., 1.1506, 1.1506, 1.1506],
 [1.2206, 1.2206, 1.2206, ..., 1.1506, 1.1506, 1.1506]],
 [[1.0191, 1.0191, 1.0191, ..., 1.0191, 1.0191, 1.0191],
 [1.0191, 1.0191, 1.0191, ..., 1.0191, 1.0191, 1.0191],
 [1.0191, 1.0191, 1.0191, ..., 1.0191, 1.0191, 1.0191],
 ...,
 [1.1062, 1.1062, 1.1062, ..., 1.0365, 1.0365, 1.0365],
 [1.1062, 1.1062, 1.1062, ..., 1.0365, 1.0365, 1.0365],
 [1.1062, 1.1062, 1.1062, ..., 1.0365, 1.0365, 1.0365]]])}
```

Step 6 – Training the model

For image classification we will use the transformers built-in function class object `AutoModelForImageClassification`.

Use the `from_pretrained` method to download and cache the weights. Also pass `label2id`, and `id2label` with the pretrained model we assigned earlier with `model_checkpoint` here. This will make sure a custom classification head will be created.

```
from transformers import AutoModelForImageClassification, TrainingArguments, Trainer

model = AutoModelForImageClassification.from_pretrained(
    model_checkpoint,
    label2id=label2id,
    id2label=id2label,
    ignore_mismatched_sizes = True, # provide this in case you're planning to fine-tune an already fine-tuned checkpoint
)

Some weights of SwinForImageClassification were not initialized from the model checkpoint at microsoft/swin-tiny-patch4-window7-224:
- classifier.bias: found shape torch.Size([1000]) in the checkpoint and torch.Size([10]) in the model instantiated
- classifier.weight: found shape torch.Size([1000, 768]) in the checkpoint and torch.Size([10, 768]) in the model instantiated
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

Next we need to instantiate a Trainer, we will need to define the training configuration and the evaluation metric. The most important is the [TrainingArguments](#), which is a class that contains all the attributes to customize the training. It requires one folder name, which will be used to save the checkpoints of the model.

Most of the training arguments are pretty self-explanatory, but one that is quite important here is `remove_unused_columns=False`. This one will drop any features not used by the model's call function. By default it's True because usually it's ideal to drop unused feature columns, making it easier to unpack inputs into the model's call function. But, in our case, we need the unused features ('image' in particular) in order to create 'pixel_values' [6].

```
[ ] model_name = model_checkpoint.split("/")[-1]

args = TrainingArguments(
    f"{model_name}-finetuned-eurosat",
    remove_unused_columns=False,
    evaluation_strategy = "epoch",
    save_strategy = "epoch",
    learning_rate=5e-5,
    per_device_train_batch_size=batch_size,
    gradient_accumulation_steps=4,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=3,
    warmup_ratio=0.1,
    logging_steps=10,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    push_to_hub=True,
)
```

Next we define a function to compute the metrics for accuracy using the `argmax` method and define a collate function which is used to batch examples together with the 2 keys of pixel values and labels.

```
[ ] import numpy as np

# the compute_metrics function takes a Named Tuple as input:
# predictions, which are the logits of the model as Numpy arrays,
# and label_ids, which are the ground-truth labels as Numpy arrays.
def compute_metrics(eval_pred):
    """Computes accuracy on a batch of predictions"""
    predictions = np.argmax(eval_pred.predictions, axis=1)
    return metric.compute(predictions=predictions, references=eval_pred.label_ids)
```

```
[ ] import torch

def collate_fn(examples):
    pixel_values = torch.stack([example["pixel_values"] for example in examples])
    labels = torch.tensor([example["label"] for example in examples])
    return {"pixel_values": pixel_values, "labels": labels}
```

Next we assign the trainer with our Trainer function with the datasets (train_ds and val_ds) our tokenizer, compute_metrics and collator we just created.

```
[ ] trainer = Trainer(
    model,
    args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    tokenizer=image_processor,
    compute_metrics=compute_metrics,
    data_collator=collate_fn,
)
```

Step 7 – Fine-Tune the Model

Next we fine tune the model by calling the train method.

Overall we can observe that we had 3 epochs, 570 optimization steps, and 24300 training images.

```
[ ] train_results = trainer.train()
# rest is optional but nice to have
trainer.save_model()
trainer.log_metrics("train", train_results.metrics)
trainer.save_metrics("train", train_results.metrics)
trainer.save_state()
```



[570/570 06:30, Epoch 3/3]

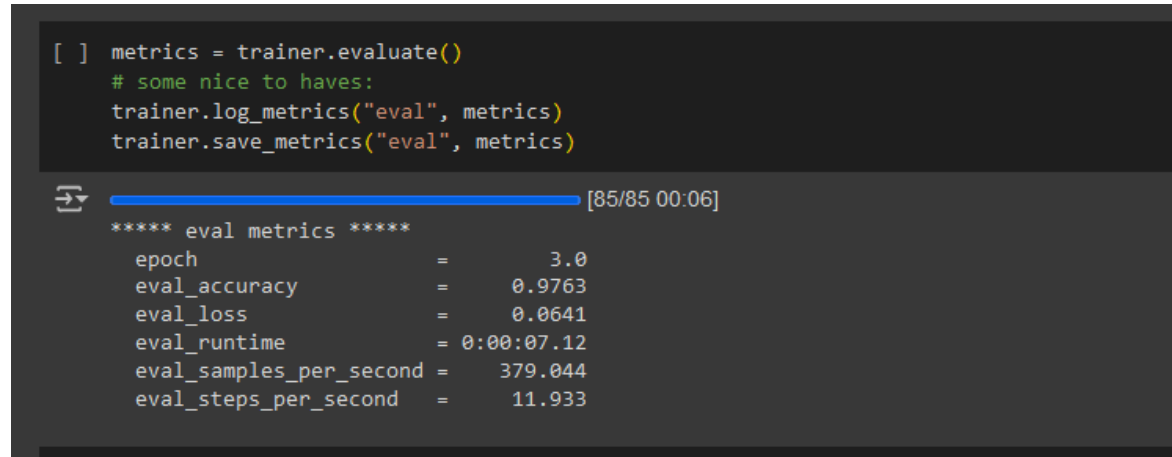
Epoch	Training Loss	Validation Loss	Accuracy
1	0.266800	0.116698	0.962963
2	0.179400	0.083736	0.971111
3	0.153900	0.064117	0.976296

***** train metrics *****

```
epoch = 3.0
total_flos = 1687935228GF
train_loss = 0.3485
train_runtime = 0:06:31.36
train_samples_per_second = 186.272
train_steps_per_second = 1.456
```


Next we can check with the evaluate method that the Trainer loaded the best model, the total runtime and the accuracy of the best model of 97.6%.

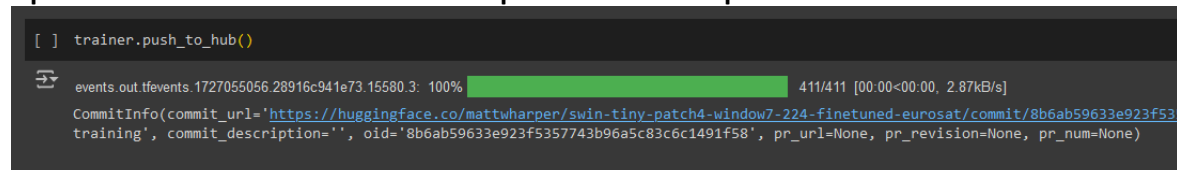
```
[ ] metrics = trainer.evaluate()
# some nice to haves:
trainer.log_metrics("eval", metrics)
trainer.save_metrics("eval", metrics)
```



```
***** eval metrics *****
epoch                =          3.0
eval_accuracy         =         0.9763
eval_loss             =         0.0641
eval_runtime          =        0:00:07.12
eval_samples_per_second =       379.044
eval_steps_per_second  =        11.933
```

Upload the trained model to the Hub as part of our Data Pipeline as follows.

```
[ ] trainer.push_to_hub()
```



```
events.out.tfevents.1727055056.28916c941e73.15580.3: 100% 411/411 [00:00<00:00, 2.87kB/s]
CommitInfo(commit_url='https://huggingface.co/mattwharper/swin-tiny-patch4-window7-224-finetuned-eurostat/commit/8b6ab59633e923f5357743b96a5c83c6c1491f58', commit_description='', oid='8b6ab59633e923f5357743b96a5c83c6c1491f58', pr_url=None, pr_revision=None, pr_num=None)
```

Step 8 – Model Inference

Now its time to try out our model with real world data on image that we will load from a hugging face repository at the following url = 'https://huggingface.co/nielsr/convnext-tiny-finetuned-eurostat/resolve/main/forest.png'

This image will be used to test our fine-tune model.

Also, display the image so we know it loaded properly with the image statement

```
[ ] from PIL import Image
import requests

url = 'https://huggingface.co/nielsr/convnext-tiny-finetuned-eurostat/resolve/main/forest.png'
image = Image.open(requests.get(url, stream=True).raw)
image
```



Load the image processor and process the image with the model from the hub (from the previous step above). Then prepare and encode the image for the model with the torch.

```
[ ] from transformers import AutoModelForImageClassification, AutoImageProcessor

repo_name = "nielsr/swin-tiny-patch4-window7-224-finetuned-eurosat"

image_processor = AutoImageProcessor.from_pretrained(repo_name)
model = AutoModelForImageClassification.from_pretrained(repo_name)
```

```
preprocessor_config.json: 100% ██████████ 240/240 [00:00<00:00, 19.1kB/s]
config.json: 100% ██████████ 1.26k/1.26k [00:00<00:00, 111kB/s]
pytorch_model.bin: 100% ██████████ 110M/110M [00:02<00:00, 57.9MB/s]
```

```
[ ] # prepare image for the model
encoding = image_processor(image.convert("RGB"), return_tensors="pt")
print(encoding.pixel_values.shape)

torch.Size([1, 3, 224, 224])
```

```
[ ] import torch

# forward pass
with torch.no_grad():
    outputs = model(**encoding)
    logits = outputs.logits
```

Call the predicted class argument using the argmax method to retrieve the models predicted label for the image which has the correct image classification as “Forest”.

```
[ ] predicted_class_idx = logits.argmax(-1).item()
    print("Predicted class:", model.config.id2label[predicted_class_idx])
```

→ Predicted class: Forest

Step 9 – Pipeline Method

Using a Pipeline API method we can perform the previous steps to quickly perform inference with any model on the hub using the pipelined API method.

```
[ ] from transformers import pipeline

pipe = pipeline("image-classification", "nielsr/swin-tiny-patch4-window7-224-finetuned-eurosat")
```

```
[ ] pipe(image)
```

→

```
[{'label': 'Forest', 'score': 0.9807635545730591},
{'label': 'HerbaceousVegetation', 'score': 0.9139886498451233},
{'label': 'Pasture', 'score': 0.8830826878547668},
{'label': 'Highway', 'score': 0.5106503367424011},
{'label': 'Residential', 'score': 0.5033761262893677}]
```

```
[ ] pipe = pipeline("image-classification",
                    model=model,
                    feature_extractor=image_processor)
```

→ Hardware accelerator e.g. GPU is available in the environment, but no `device` argument is

```
[ ] pipe(image)
```

→

```
[{'label': 'Forest', 'score': 0.9807635545730591},
{'label': 'HerbaceousVegetation', 'score': 0.9139886498451233},
{'label': 'Pasture', 'score': 0.8830826878547668},
{'label': 'Highway', 'score': 0.5106503367424011},
{'label': 'Residential', 'score': 0.5033761262893677}]
```

Conclusion

Overall, this exercise provides students with a comprehensive introduction and tutorial for deep learning model development using *Transformers* for advanced natural language processing concepts. The lab provided students with an introductory to pre-trained large language models which can be tuned for use in specific tasks and domains with computer vision applications. Results of the lab also demonstrated the effectiveness of a transformers ability for image classification and which achieved a metric score of 97.6% during training which was aided by image augmentation methods of flipping and resizing.

References

- [1] Gohel, Hardik. LLMs and Fine-Tuning using PEFTs. COSC 6370 – Advanced Topics in AI Module 1-8.
<https://hpedsu.uh.edu/people/hardik-gohel>Links to an external site.
- [2] IBM. (2024, August 13). Computer Vision. IBM What is Computer Visions.
<https://www.ibm.com/topics/computer-vision>
- [3] Advanced Computer Vision. (n.d.). <https://16820advancedcv.github.io/pages/assignments.html>
- [4] Franz J. Kurfess, Artificial Intelligence in Encyclopedia of Physical Science and Technology (2003)
- [5] Watanabe, E., Kitaoka, A., Sakamoto, K., Yasugi, M., & Tanaka, K. (2018). Illusory motion reproduced by deep neural networks trained for prediction. *Frontiers in Psychology*, 9.
<https://doi.org/10.3389/fpsyg.2018.00345>
- [6] Huggingface. (n.d.-b). notebooks/examples/image_classification.ipynb at main · huggingface/notebooks. GitHub.
https://github.com/huggingface/notebooks/blob/main/examples/image_classification.ipynb