Assignment 2 – Zero-Shot Image Classification with Transformers

In this assignment, you will apply a pre-trained vision–language transformer (e.g. CLIP) to perform **zero-shot** classification on the Fashion-MNIST dataset—classifying each image without any task-specific training. You will build on the concepts from Assignment 1 by comparing this “off-the-shelf” approach to the CNN you previously trained.

You will:

1. **Load** the Fashion-MNIST images using PyTorch instead of Keras.
2. **Run a zero-shot baseline** with simple text prompts to set a performance reference.
3. **Engineer improved prompts** and measure the resulting accuracy gains.
4. **Visualise image embeddings** with UMAP to inspect class separability.
5. **Conduct one mini-experiment** of your choice.
6. **Summarise findings** and reflect on strengths and weaknesses of zero-shot transformers versus a trained CNN.

1. Loading the Fashion-MNIST Dataset

As in assignment 1, we'll load the Fashion-MNIST dataset, but this time using `torchvision.datasets` to ensure compatibility with the `transformers` library. We will also load our model and processor from the `transformers` library.

The transformers library allows us to use pre-trained models like CLIP, which can perform zero-shot classification by leveraging the text prompts we provide. There are two key objects we will use: the `CLIPModel` for the model itself and the `CLIPProcessor` for preparing our images and text prompts.

Since we are not actually training a model in this assignment, we will set the CLIP model to evaluation mode. If the model is designed to utilize features like dropout or batch normalization, setting it to evaluation mode ensures that these features behave correctly during inference (prediction). Setting the model to evaluaton mode also tells PyTorch that we don't have to compute gradients, which can save memory and speed up inference.

In order to speed up processing, we will also move the model to an "accelerator" if available. This is typically a GPU, but modern MacBooks also have an "Apple Silicon" accelerator that can be used for inference, called MPS (Metal Performance Shaders). If you are using a MacBook with Apple Silicon, you can use the MPS device for faster processing.

```python
from transformers import CLIPModel, CLIPProcessor
import torch
from torchvision import datasets
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix
import numpy as np
```

```python
# Uncomment and run if required
!pip install transformers torchvision torch accelerate
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.53.2)
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-packages (0.21.0+cu124)
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (2.6.0+cu124)
Requirement already satisfied: accelerate in /usr/local/lib/python3.11/dist-packages (1.8.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from transformers) (3.18.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.30.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.3
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (25.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.21.2)
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.5.3)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.11/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.11/dist-packages (from torchvision) (11.2.1)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.11/dist-packages (from torch) (4.14.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch) (2025.3.2)
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch)
```

```
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch) (12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch) (1.3.0
Requirement already satisfied: psutil in /usr/local/lib/python3.11/dist-packages (from accelerate) (5.9.5)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.2 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub<1.0,>=0.30
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests->transformers
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (2.4
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (202
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl (363.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 3.4 MB/s eta 0:00:00
Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (13.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 122.1 MB/s eta 0:00:00
Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (24.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 89.1 MB/s eta 0:00:00
```

```python
# from transformers import CLIPModel, CLIPProcessor
# import torch

clip_model_name = "openai/clip-vit-base-patch32"
clip_model     = CLIPModel.from_pretrained(clip_model_name)
clip_processor = CLIPProcessor.from_pretrained(clip_model_name, use_fast=False)

# Set model to evaluation mode, as we are not training it
clip_model.eval()

# Check for accelerators
device = "cpu" # Default to CPU
if torch.cuda.is_available():
    device = "cuda" # Use GPU if available
elif torch.backends.mps.is_available():
    device = "mps"

clip_model.to(device)

print(f"Using device: {device}")
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it a
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
config.json:  4.19k/? [00:00<00:00, 465kB/s]
pytorch_model.bin: 100%  605M/605M [00:03<00:00, 450MB/s]
preprocessor_config.json: 100%  316/316 [00:00<00:00, 41.0kB/s]
tokenizer_config.json: 100%  592/592 [00:00<00:00, 80.4kB/s]
vocab.json:  862k/? [00:00<00:00, 39.8MB/s]
merges.txt:  525k/? [00:00<00:00, 39.0MB/s]
model.safetensors: 100%  605M/605M [00:02<00:00, 330MB/s]
special_tokens_map.json: 100%  389/389 [00:00<00:00, 54.5kB/s]
tokenizer.json:  2.22M/? [00:00<00:00, 99.4MB/s]
Using device: cuda
```

Now we are ready to load the testing set from Fashion-MNIST. We will use the `torchvision.datasets.FashionMNIST` class to load the dataset. We do not need to apply any transformations to the images, as the `CLIPProcessor` ensures any input images are in the format that the model is trained on.

You should:

- [ ] Use the `torchvision.datasets.FashionMNIST` class to load the *test* split of the dataset. Documentation is available here.
- [ ] Create a PyTorch `DataLoader` to iterate over the dataset in batches. Use a batch size of 16 and set `shuffle=True` to randomise the order of the images. You will also need to supply the provided `collate_clip` function to the `DataLoader collate_fn` argument to ensure the images are processed correctly. Documentation for `DataLoader` is available here.

```python
# from torchvision import datasets
# from torch.utils.data import DataLoader


CLASS_NAMES = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot"
]

def collate_clip(batch):
    imgs, labels = zip(*batch) # Unzip the batch into images and labels
    proc = clip_processor(images=list(imgs),
                        return_tensors="pt",
                        padding=True) # Process images with CLIPProcessor
    # Send pixel_values to GPU/CPU now; labels stay on CPU for metrics
    return proc["pixel_values"].to(device), torch.tensor(labels)

test_dataset = datasets.FashionMNIST(root='FashionMNIST/raw/t10k-images-idx3-ubyte', download=True)
```

```
100%|██████████| 26.4M/26.4M [00:03<00:00, 7.83MB/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 170kB/s]
100%|██████████| 4.42M/4.42M [00:01<00:00, 3.03MB/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 12.3MB/s]
```

```python
print(type(test_dataset))
```

```
<class 'torchvision.datasets.mnist.FashionMNIST'>
```

```
# from torch.utils.data import TensorDataset

# dataset = TensorDataset(test_dataset)
test_loader = DataLoader(
    test_dataset,
    batch_size=16,
    collate_fn=collate_clip,
    # pin_memory=True,
    shuffle=True
    )
```

If your code is correct, the following cell should show the first batch of images from the Fashion-MNIST dataset:

```
# import matplotlib.pyplot as plt

# Display the first batch of images from `test_loader`

def show_batch(loader):
    images, labels = next(iter(loader))
    images = images.cpu()  # Move images to CPU for plotting
    # Renormalize to [0, 1] for visualization
    images = (images - images.min()) / (images.max() - images.min())
    _, axes = plt.subplots(1, len(images), figsize=(15, 5))
    for ax, img, label in zip(axes, images, labels):
        ax.imshow(img.permute(1, 2, 0))
        ax.set_title(CLASS_NAMES[label.item()])
        ax.axis('off')
    plt.show()

show_batch(test_loader)
```

We're now ready to run our zero-shot classification baseline!

Brief Introduction to Zero-Shot Classification

In Assignment 1, we followed the typical machine-learning pipeline: we trained a CNN on the Fashion-MNIST dataset, using labelled examples to update the model's weights. While effective, that approach requires a curated, task-specific training set—a luxury you don't always have in practice.

Zero-shot classification flips the script. A large vision–language model (VLM) such as **CLIP** is first pre-trained on hundreds of millions of image–text pairs scraped from the web. Because it learns *joint* visual–textual embeddings, the model can later solve new tasks simply by "measuring" how similar an image is to a **text prompt** that describes each candidate class—without seeing a single task-labelled example.

How it works

1. Feed an image through CLIP's vision encoder → **image feature**.
2. Feed a textual prompt (e.g. "a photo of a sandal") through CLIP's text encoder → **text feature**.
3. Compute cosine similarity between the image feature and every class's text feature.
4. Pick the class whose prompt is most similar.

For our first attempt, we'll use the bare class names as prompts, e.g.:

- "T-shirt/top"
- "Trouser"

You should:

- ☐ Build embeddings: use the `get_text_embeddings` helper function to create text embeddings for the class names.
- ☐ Run inference: use the `get_image_embeddings` helper function to create image embeddings.
- ☐ Compute cosine similarity: complete and use the `get_cosine_similarity` helper function to compute the cosine similarity between the image and text embeddings.
- ☐ Make predictions: use the `get_predictions` helper function to get the predicted class for each image in the batch.

Note that for normalized vectors like the ones we are using, cosine similarity is equivalent to the dot product. This means we can use the handy formula `cosine_similarity = vector_a @ vector_b.T` to compute the similarity between the image and text embeddings.

```
def get_text_embeddings(class_names: list[str]) -> torch.Tensor:
    """    Get text embeddings for the given class names using CLIP.
    Args:
        class_names (list[str]): List of class names to encode.
    Returns:
        torch.Tensor: Normalized text embeddings for the class names.
    """
    tokenized = clip_processor(text=class_names,
                               padding=True,
                               return_tensors="pt").to(device)

    with torch.no_grad():
        text_embeddings = clip_model.get_text_features(**tokenized)

    text_feats = text_embeddings / text_embeddings.norm(dim=-1, keepdim=True)

    return text_feats

def get_image_embeddings(images: torch.Tensor) -> torch.Tensor:
    """    Get image embeddings for the given images using CLIP.
    Args:
        images (torch.Tensor): Batch of images to encode.
    Returns:
        torch.Tensor: Normalized image embeddings for the images.
    """
    with torch.no_grad():
        image_embeddings = clip_model.get_image_features(pixel_values=images)

    image_feats = image_embeddings / image_embeddings.norm(dim=-1, keepdim=True)

    return image_feats
```

```
# import numpy as np

def get_cosine_similarity(image_feats: torch.Tensor, text_feats: torch.Tensor) -> np.ndarray:
    """
    Compute cosine similarity between image features and text features.
    Args:
        image_feats (torch.Tensor): Image features of shape (N, D).
        text_feats (torch.Tensor): Text features of shape (M, D).
    Returns:
        numpy.ndarray: Cosine similarity matrix of shape (N, M), where N is the number of images and M is the number
    """
    image_feats = image_feats.cpu()  # Ensure image features are on CPU
    text_feats = text_feats.cpu()    # Ensure text features are on CPU

    # Compute cosine similarity, which is the dot product of normalized vectors
    # cosine_sim = CosineSimilarity()
    # return cosine_sim(image_feats, text_feats)
    return image_feats.numpy() @ text_feats.numpy().T

def get_predictions(similarity: np.ndarray) -> np.ndarray:
    """
    Get predictions based on cosine similarity scores.
    Args:
        similarity (numpy.ndarray): Cosine similarity matrix of shape (N, M), where N is the number of images and M
    Returns:
        numpy.ndarray: Predicted class indices for each image, shape (N,).
    """
    # Get the index of the maximum similarity for each image

    return np.argmax(simis, axis=1)
```

With these functions complete, you are ready to run the zero-shot classification baseline. Complete the code to follow these steps:

- [ ] Build text embeddings for the class names using the `get_text_embeddings` function (this only needs to be done once).
- [ ] For each batch of images:
    - [ ] Get image embeddings using the `get_image_embeddings` function.
    - [ ] Compute cosine similarity between the image and text embeddings using the `get_cosine_similarity` function.
    - [ ] Save the predictions so that we can build a confusion matrix later.
- [ ] Report the accuracy of the predictions and the confusion matrix using the `accuracy_score` and `confusion_matrix` functions from `sklearn.metrics`.

```
# from sklearn.metrics import accuracy_score, confusion_matrix

y_true, y_pred = [], []


for pixel_values, labels in test_loader:
    # Report the accuracy of the predictions
    text_embes = get_text_embeddings(CLASS_NAMES)
    image_embes = get_image_embeddings(pixel_values)
    simis = get_cosine_similarity(image_embes, text_embes)
    preds = get_predictions(simis)
    y_true.append(labels.cpu().numpy())
    y_pred.append(preds)
```

```
_y_true = np.array(y_true).flatten()
_y_pred = np.array(y_pred).flatten()

print('Accuracy score: {}'.format(accuracy_score(_y_true, _y_pred)))
```

```
Accuracy score: 0.6299833333333333
```

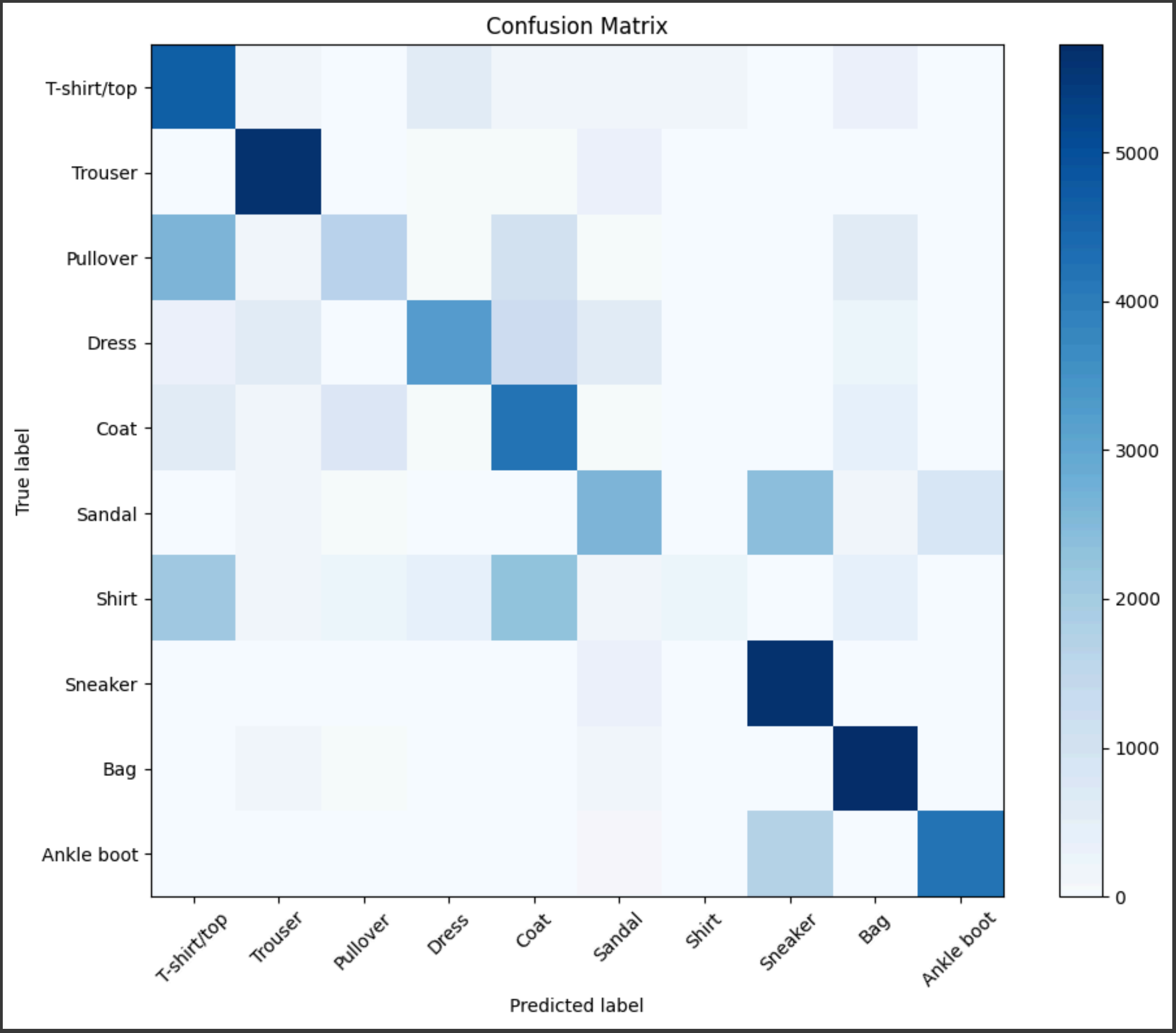
Reflection: Consider the results. How does the performance of this zero-shot baseline compare to the CNN you trained in Assignment 1? What are the strengths and weaknesses of this approach?

Comment:

> - The CNN model using class-based labeled data performs better than the zero-shot baseline model.
>
>   CNN: `test_acc: 0.919`
>
>   Zero-shot: `acc: 0.630`
> - The above might be due to the fact that the data in the CNN is labelled, in this sense, there's less room for mistakenly classify images
> - The main weakness I identify while using the zero-show approach is the quality/detail of the text associated to the image. Shallow "descriptions" might lead to two images being identified as the same object
> - A strength of this method is the fact that the images don't need to be labelled, which can be resource intensive

```
# # Report the confusion matrix
def plot_confusion_matrix(y_true, y_pred, class_names):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)

    # Loop over data dimensions and create text annotations.
    for i in range(len(class_names)):
        for j in range(len(class_names)):
            text = plt.text(j, i, cm[i, j],
                           ha="center", va="center", color="black")

    plt.title('Confusion Matrix')
    plt.colorbar()
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()
```

```
plot_confusion_matrix(_y_true, _y_pred, CLASS_NAMES)
```



Improving Zero-Shot Classification with Prompt Engineering

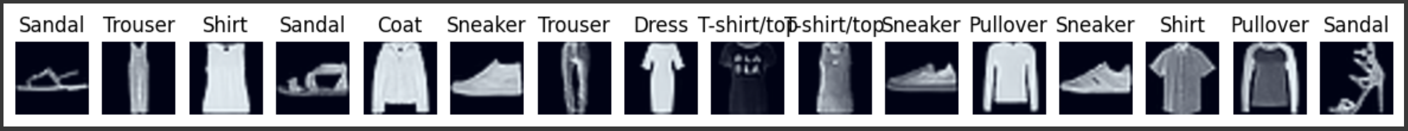
In the previous section, we directly used the class names as text prompts for zero-shot classification. However, we can often improve performance by crafting more descriptive prompts that better capture the visual characteristics of each class. For example, instead of just "T-shirt/top", we could use "a photo of a T-shirt" or "a photo of a top". This additional context can help the model make more accurate predictions.

In this section, we will experiment with more detailed prompts for each class to see if we can improve the zero-shot classification performance. You should:

- ☐ Create a list of improved prompts for each class. For example, instead of just "T-shirt/top", you could use "a photo of a T-shirt" or "a photo of a top".
- ☐ Use the `get_text_embeddings` function to create text embeddings for the improved prompts.
- ☐ Run the zero-shot classification baseline again using the improved prompts and report the accuracy and confusion matrix.

Note: Take advantage of the confusion matrix above. If two classes are often confused, consider how you might improve the prompts to help the model distinguish them better.

The aim for this section is for you to improve the performance of the model. However, if you find that the performance does not improve significantly, you can still reflect on the process and consider how you might further refine the prompts with more effort.

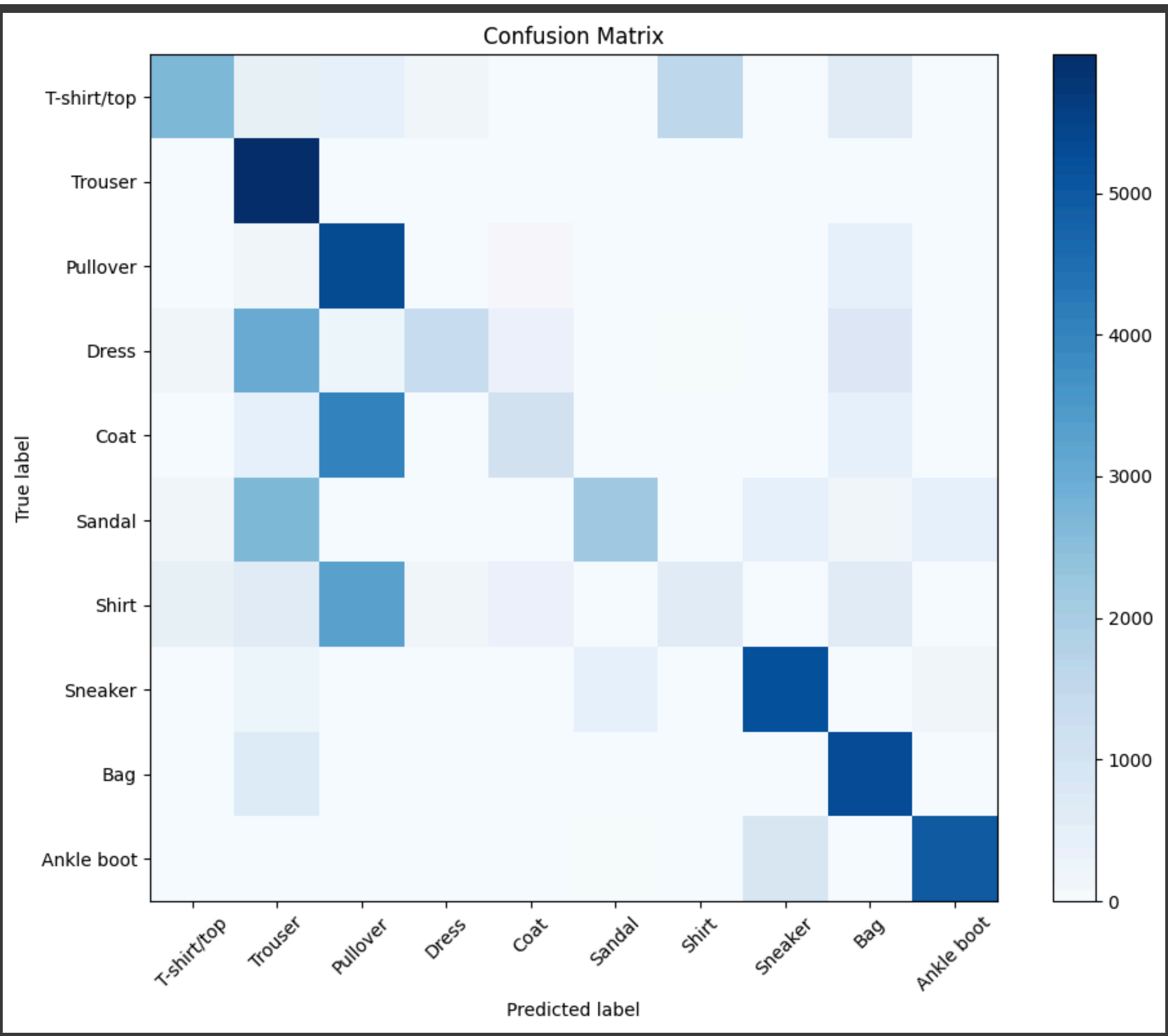


```
CLASS_NAMES_2 = [
    "A women's T-shirt/top with text on it",
    "A photo of long Trousers",
    "Pullover with long sleeves",
    "Dress",
    "Coat",
    "Sandal with crossed straps",
    "Shirt with short sleeves",
    "Sneaker with laces and details on the side",
    "Bag with long straps and squared-shape",
    "An Ankle boot without laces"
]


y_true, y_pred = [], []


for pixel_values, labels in test_loader:
    # Report the accuracy of the predictions
    text_embes = get_text_embeddings(CLASS_NAMES_2)
    image_embes = get_image_embeddings(pixel_values)
    simis = get_cosine_similarity(image_embes, text_embes)
    preds = get_predictions(simis)
    y_true.append(labels.cpu().numpy())
    y_pred.append(preds)
```

```
_y_true = np.array(y_true).flatten()
_y_pred = np.array(y_pred).flatten()

print('Accuracy score: {}'.format(accuracy_score(_y_true, _y_pred)))
```

```
Accuracy score: 0.5775333333333333
```

```
plot_confusion_matrix(_y_true, _y_pred, CLASS_NAMES)
```



Reflection: How did your detailed prompts affect the zero-shot classification performance? Did you see a significant improvement compared to the baseline? What insights did you gain about the model's understanding of the classes? Do you think that with more effort you could further improve the performance? If so, how?

Comments:

> - Changing the prompts had a noticeable impact on the performance of the predictions, although not in the desired direction. The accuracy of the model decreased with respect to the original prompts
> - Classes with similar features like `Shirt` and `Pullover` are misclassified and their prompts required better engineering
> - My guess is that by adding more context to similar classes might improve the performance of the model.

Visualizing Image Embeddings with UMAP

To better understand how the model perceives the different classes, we can visualize the image embeddings using UMAP (Uniform Manifold Approximation and Projection). UMAP is a dimensionality reduction technique that helps us see how similar or dissimilar the embeddings are in a lower-dimensional space.

By visualizing the embeddings, we can gain insights into how well the model can distinguish certain images, even without considering the text prompts. This can help us identify clusters of similar images and see if there are any overlaps between classes.

You should:

- ☐ Use the `get_image_embeddings` function to get the image embeddings for the entire test set.
- ☐ Use UMAP to reduce the dimensionality of the image embeddings to 2D.
- ☐ Plot the 2D embeddings, coloring each point by its true class label.

You may need to install the `umap-learn` library if you haven't already. You can do this by running `pip install umap-learn`.

```
# Uncomment the following line to install UMAP if you haven't already
!pip install umap-learn
```

```
Requirement already satisfied: umap-learn in /usr/local/lib/python3.11/dist-packages (0.5.9.post2)
Requirement already satisfied: numpy>=1.23 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (2.0.2)
Requirement already satisfied: scipy>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (1.15.3)
Requirement already satisfied: scikit-learn>=1.6 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (1.6.1)
Requirement already satisfied: numba>=0.51.2 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (0.60.0)
Requirement already satisfied: pynndescent>=0.5 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (0.5.13)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from umap-learn) (4.67.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.11/dist-packages (from numba>=0.51.2->umap-le
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.11/dist-packages (from pynndescent>=0.5->umap-learn) (1.5.1
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.6->umap-lear
```

```
from umap import UMAP

# ------------------------------------------------------------
# 1. Collect image embeddings
# ------------------------------------------------------------
all_img_emb = []
all_labels  = []

for pixel_values, labels in test_loader:
    all_labels.append(labels)
    all_img_emb.append(get_image_embeddings(pixel_values).cpu().numpy())

_all_labels = np.array(all_labels).flatten()
```

```
np.vstack(all_img_emb).shape
```

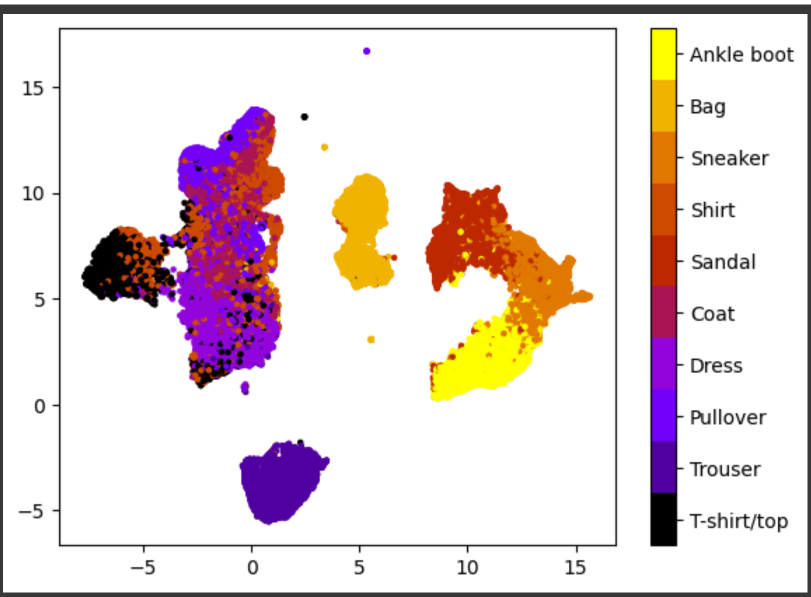
```
(60000, 512)
```

```
# ------------------------------------------------------------
# 2. Fit UMAP
# ------------------------------------------------------------
umap_model = UMAP(n_neighbors=5)
embeddings_umap = umap_model.fit_transform(np.vstack(all_img_emb))
```

```
embeddings_umap[0]
```

```
array([-1.2456169,  7.523645 ], dtype=float32)
```

```
# ------------------------------------------------------------
# 3. Plot coloured by ground-truth label
# ------------------------------------------------------------
colors = dict(zip(CLASS_NAMES, list(range(16))))

fig, ax = plt.subplots()
_scatterp = ax.scatter(embeddings_umap[:, 0], embeddings_umap[:, 1], s=5, c=all_labels, cmap='gnuplot')
plt.colorbar(_scatterp, boundaries=np.arange(11)-0.5).set_ticks(np.arange(10), labels=colors.keys())
plt.show();
```



The UMAP embeddings allow us to see how separable or non-separable different classes are with our specific model. If two specific images are very similar, then they will be placed near each other on this graph.

Reflection: Do you notice any challenges in distinguishing images based on this figure? Are there any types of clothing in the dataset which the model has no trouble distinguishing from the others?

Comments:

> Yes, the above supports that some classes are challenging to differentiate from each other. For example, the pair `T-shirt/top` / `Shirt`
>
> It seems that similar objects, for example, all tops are hard to properly clssify. Something that does not happen in the case of the class `Trouser`, which has its own well-defined blob

Mini-Experiment

In this section, you will conduct a mini-experiment of your choice to further explore the capabilities of zero-shot classification with transformers. This can be anything you'd like, but here are some ideas to get you started.

A. Alternative Model

So far we have been utilizing OpenAI's CLIP model for zero-shot classification. However, there are many other vision–language models available in the `transformers` library that you can experiment with. For example, there are larger CLIP models such as clip-vit-large-patch14, and open-source versions such as laion/CLIP-ViT-B-32-laion2B-s34B-b79K. You can also search huggingface here to find other models that might be suitable for zero-shot classification.

You can try using a different model to see if it improves the zero-shot classification performance. You should:

- [ ] Load a different model and processor from the `transformers` library.
- [ ] Run the zero-shot classification baseline with the new model and report the accuracy and confusion matrix.
- [ ] Reflect on the performance of the new model compared to the original CLIP model
    - How does the new model perform compared to the original CLIP model?
    - Do you notice any differences in the types of errors made by the new model?

B. Multiple-Description Classification

Another interesting experiment is to explore multiple-description classification. *This involves providing multiple text prompts for each class, allowing the model to choose the most relevant one. For example, instead of just "T-shirt/top", you could provide "a photo of a T-shirt", "a photo of a top", and "a photo of a shirt". This can help the model better understand the class and increases the likelihood of a correct prediction. You should:

- ☐ Create a list of multiple prompts for each class.
- ☐ Use the `get_text_embeddings` function to create text embeddings for the multiple prompts.
- ☐ Run the zero-shot classification baseline again using the multiple prompts and report the accuracy and confusion matrix.
- ☐ Consider the model to be correct if it guesses *any* of the prompts belonging to the correct class.

C. Top-K Classification

In some classification tasks, it can be useful to consider if the right answer is among the top K (e.g. top 3) predictions. This can be particularly useful in cases where the model is uncertain or when there are multiple similar classes. You should:

- ☐ Modify the `get_predictions` function to return the top K predictions for each image.
- ☐ Modify the accuracy calculation to consider the model correct if the true class is among the top K predictions.
- ☐ Report the accuracy and confusion matrix for the top K predictions. Report at least two different values of K (e.g. K=2 and K=4).

D. Other Ideas

You are welcome to come up with your own mini-experiment! Explain your idea in the report and implement it. Did it work as you expected? What did you learn from it?

Selection: `C. Top-K Classification`

```
def get_predictions(similarity: np.ndarray, k: int = 3) -> np.ndarray:
    """
    Get predictions based on cosine similarity scores.
    Args:
        similarity (numpy.ndarray): Cosine similarity matrix of shape (N, M), where N is the number of images and M
    Returns:
        numpy.ndarray: top 3 predicted class indices for each image, shape (N,).
    """
    # Get the index of the top three similarity for each image
    return np.argsort(simis, axis=1)[:, -k:]
```

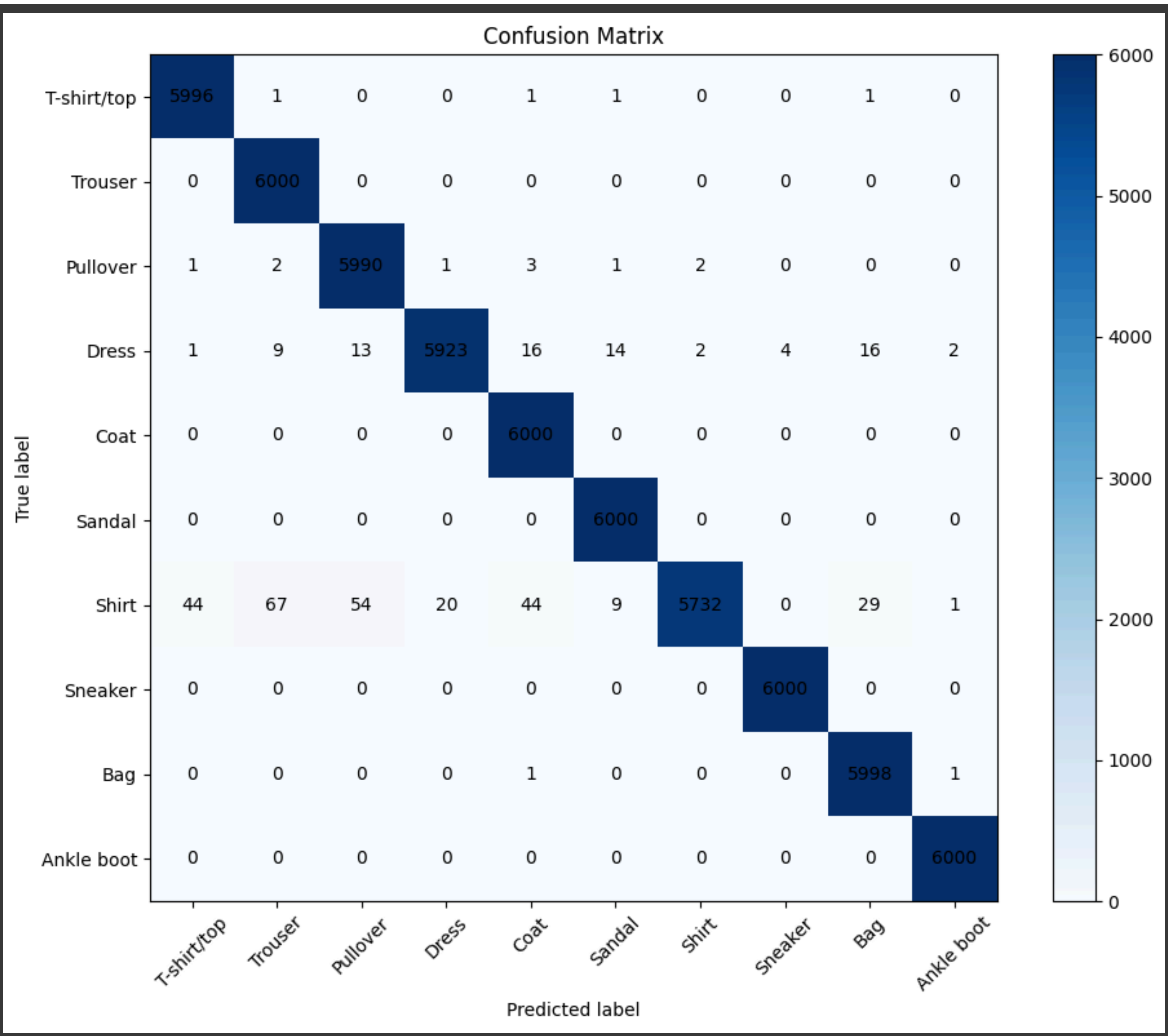
Using top 3

```
y_true, y_pred = [], []

for pixel_values, labels in test_loader:
    # Report the accuracy of the predictions
    text_embes = get_text_embeddings(CLASS_NAMES)
    image_embes = get_image_embeddings(pixel_values)
    simis = get_cosine_similarity(image_embes, text_embes)

    # get preds
    _preds = get_predictions(simis)
    is_true = np.isin(labels, _preds)  # check if y_true is in top 3
    preds = np.zeros(_preds.shape[0])  # create "empty" array
    # print(_preds)
    # print(_preds[:, 0].shape)
    preds[~is_true] = _preds[:, 0][~is_true]  # populate empty array with prediction if y_true not in top 3
    preds[is_true] = labels[is_true]  # populate empty array with y_true if y_true in top 3

    y_true.append(labels.cpu().numpy())
    y_pred.append(preds)
```

```
_y_true = np.array(y_true).flatten()
_y_pred = np.array(y_pred).flatten()

print('Accuracy score: {}'.format(accuracy_score(_y_true, _y_pred)))
```

```
Accuracy score: 0.9939833333333333
```

```
plot_confusion_matrix(_y_true, _y_pred, CLASS_NAMES)
```



	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	5996	1	0	0	1	1	0	0	1	0
Trouser	0	6000	0	0	0	0	0	0	0	0
Pullover	1	2	5990	1	3	1	2	0	0	0
Dress	1	9	13	5923	16	14	2	4	16	2
Coat	0	0	0	0	6000	0	0	0	0	0
Sandal	0	0	0	0	0	6000	0	0	0	0
Shirt	44	67	54	20	44	9	5732	0	29	1
Sneaker	0	0	0	0	0	0	0	6000	0	0
Bag	0	0	0	0	1	0	0	0	5998	1
Ankle boot	0	0	0	0	0	0	0	0	0	6000

Using top 2

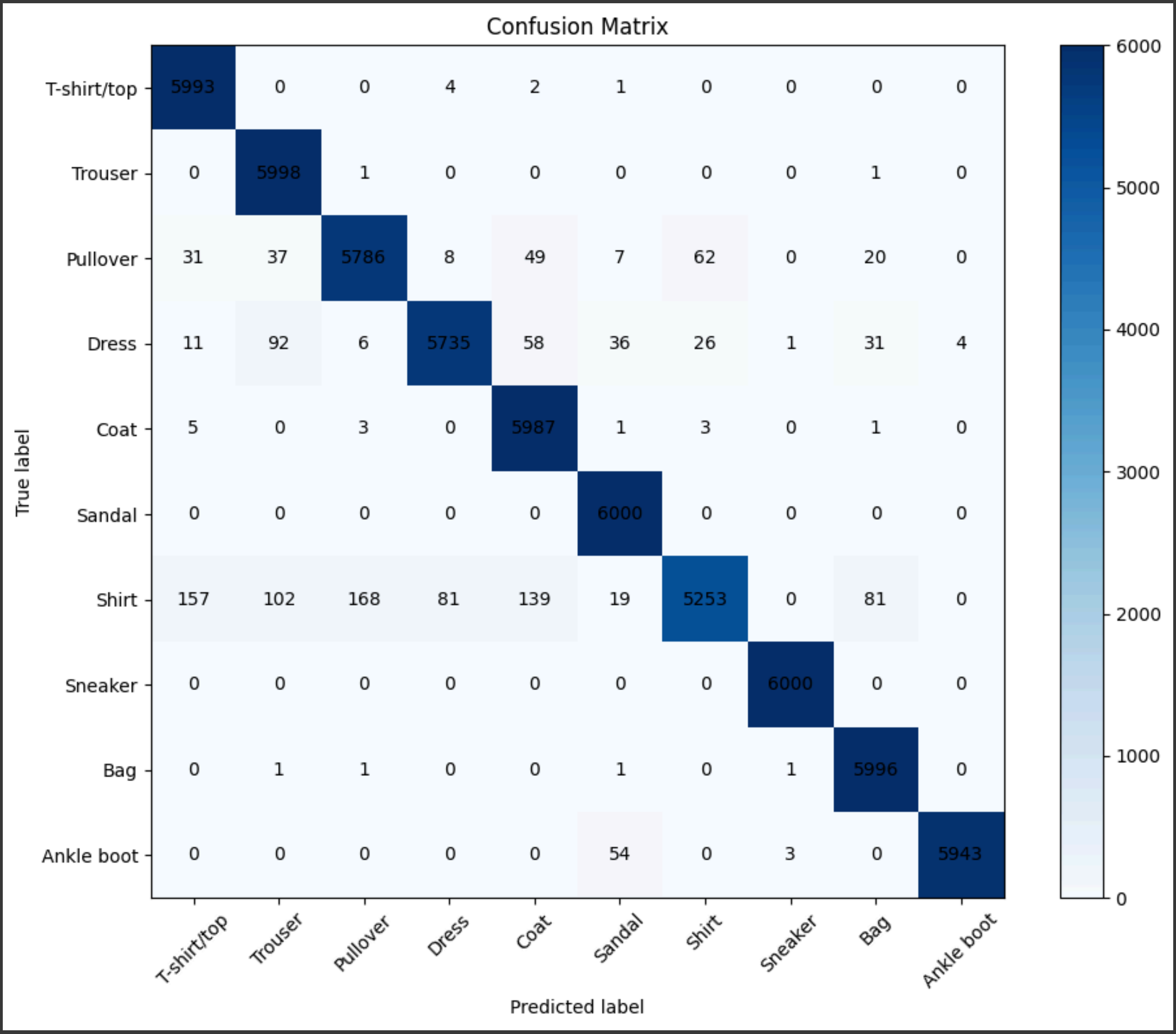
```
y_true, y_pred = [], []

for pixel_values, labels in test_loader:
    # Report the accuracy of the predictions
    text_embes = get_text_embeddings(CLASS_NAMES)
    image_embes = get_image_embeddings(pixel_values)
    simis = get_cosine_similarity(image_embes, text_embes)

    # get preds
    _preds = get_predictions(simis, 2)
    is_true = np.isin(labels, _preds)  # check if y_true is in top 3
    preds = np.zeros(_preds.shape[0])  # create "empty" array
    # print(_preds)
    # print(_preds[:, 0].shape)
    preds[~is_true] = _preds[:, 0][~is_true]  # populate empty array with prediction if y_true not in top 3
    preds[is_true] = labels[is_true]  # populate empty array with y_true if y_true in top 3

    y_true.append(labels.cpu().numpy())
    y_pred.append(preds)
```

```
_y_true = np.array(y_true).flatten()
_y_pred = np.array(y_pred).flatten()
```

```
print('Accuracy score: {}'.format(accuracy_score(_y_true, _y_pred)))
```

```
Accuracy score: 0.9781833333333333
```

```
plot_confusion_matrix(_y_true, _y_pred, CLASS_NAMES)
```



Short Report

In this section, you will write a short report summarizing your findings from the mini-experiment. The report should include the following sections:

- **Introduction**: Briefly describe the mini-experiment you conducted and its objectives.
- **Methodology**: Explain the steps you took to conduct the experiment, including any modifications you made to the code or model.
- **Results**: Present the results of your experiment.
- **Discussion**: Reflect on the performance of the model and the implications of your findings. Consider the strengths and weaknesses of zero-shot transformers versus a trained CNN.

Report

Introduction

> Based on the idea that not only the largest cosine similarity score can be used to obtain the predicted class, but the top k cosine similarities, I tried improving the performance of the model by using the `Top-K Classification` With this approach,

> the model not has a larger tolerance and its accuracy is improved. To examine the effect of k, two different k's were tested, 2 and 3.

Methodology

> 1. The `get_predictions` function was modified.
> 2. Once the predictions were obtained, aditional treatment to the results was performed:

```
For a single example:
a. get_predictions --> predictions  # indices of top k cosine similarities
b. if true_class \in predictions:   # check if y_true is in top k
c.  prediction = true_class
d. else:
e.  return prediction
```

For the above, matrix operations using numpy were performed along the `predictions` rows

Results

> The accuracy of the model was improved, from 0.58-0.63 to 0.98.

Discussion

> Considering the top 3 cosine similarities instead of the max value, improved the performance of the model. This approach does not consider whether or not the second before last cosine similarity is closer to the max(predictions). Accounting for such conditions could make the implementation more robust and granular.

🚨 **Please review our Assignment Submission Guide** 🚨 for detailed instructions on how to format, branch, and submit your work. Following these guidelines is crucial for your submissions to be evaluated correctly.

Submission Parameters:

- Submission Due Date: `23:59 PM - 18/07/2025`
- The branch name for your repo should be: `assignment-2`
- What to submit for this assignment:
  - This Jupyter Notebook (assignment_2.ipynb)
  - The Lab 3 notebook (labs/lab_3.ipynb)
  - The Lab 4 notebook (labs/lab_4.ipynb)
  - The Lab 5 notebook (labs/lab_5.ipynb)
  - The Lab 6 notebook (labs/lab_6.ipynb)
- What the pull request link should look like for this assignment: `https://github.com/<your_github_username>/deep_learning/pull/<pr_id>`
- Open a private window in your browser. Copy and paste the link to your pull request into the address bar. Make sure you can see your pull request properly. This helps the technical facilitator and learning support staff review your submission easily. Checklist:
  - ☐ Created a branch with the correct naming convention