Machine Learning with Transformers

Vineet Sai Burugu

08/22/2024

1 Introduction

Transformers have shown remarkable capabilities in a number of areas, including image classification. In this paper, we present a transformer-based classifier to identify handwritten digits from the MNIST dataset.

2 Tasks

2.1 Load the MNIST dataset and preprocess the images and labels

To start, we load the MNIST dataset and preprocess the images and labels. The preprocessing includes converting images to RGB format and applying transformations to prepare the data for the transformer model.

```
# Install necessary packages
!pip install datasets transformers torch evaluate
!pip install --upgrade torch torchvision torchaudio
!pip install scikit-learn

# Import necessary libraries
from huggingface_hub import login
import torch
import numpy as np
from transformers import ViTForImageClassification,
   ViTImageProcessor, TrainingArguments, Trainer,
   TrainerCallback
from datasets import load_dataset, DatasetDict
import matplotlib.pyplot as plt
```

```python
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Login to Hugging Face
login(token='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')

# Set device to GPU if available, else CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load and split MNIST dataset
dataset = load_dataset("ylecun/mnist")
labels = list(set(dataset['train']['label']))  # Get unique labels
splitDataset = dataset['train'].train_test_split(test_size=0.2)  # Split into train/eval
# Create dataset dict
ourDataset = DatasetDict({
    'train': splitDataset['train'],
    'validation': splitDataset['test'],
    'test': dataset['test']
})

# Load ViT image processor
processor = ViTImageProcessor.from_pretrained('google/vit-base-patch16-224-in21k')

# Preprocess and transform a batch of images
def transformBatch(batch):
    convertedImages = []  # List for converted images
    for image in batch['image']:
        convertedImages.append(image.convert('RGB'))  # Convert to RGB
    batch['image'] = convertedImages  # Replace with converted images
    inputs = processor(batch['image'], return_tensors='pt')  # Process images with ViT
    # Assign labels
    inputs['labels'] = []
    for y in batch['label']:
        inputs['labels'].append(y)
    return inputs  # Return processed inputs

processedDataset = ourDataset.with_transform(transformBatch)  # Apply transformBatch
```

```python

# Collate a batch into a format for training
def collateBatch(batch):
    pixelValuesList = []  # List for pixel values
    labelsList = []  # List for labels
    for item in batch:
        pixelValuesList.append(item['pixel_values'])  # Add pixel values
        labelsList.append(item['labels'])  # Add labels
    return {
        'pixel_values': torch.stack(pixelValuesList),
        'labels': torch.tensor(labelsList)
    }
```

2.2 Define a transformer model suitable for image classification

Next, we define a Vision Transformer (ViT) model for image classification. This model will be fine-tuned on the MNIST dataset.

```python
# Load ViT model for image classification
model = ViTForImageClassification.from_pretrained(
    'google/vit-base-patch16-224',
    num_labels=10,  # Output labels (10 for MNIST)
    ignore_mismatched_sizes=True  # Ignore size mismatches
)
model
#model returns the text below
ViTForImageClassification(
  (vit): ViTModel(
    (embeddings): ViTEmbeddings(
      (patch_embeddings): ViTPatchEmbeddings(
        (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): ViTEncoder(
      (layer): ModuleList(
        (0-11): 12 x ViTLayer(
          (attention): ViTSdpaAttention(
            (attention): ViTSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
```

```
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
            (output): ViTSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
          (intermediate): ViTIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): ViTOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (dropout): Dropout(p=0.0, inplace=False)
          )
          (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (classifier): Linear(in_features=768, out_features=10, bias=True)
)

# Compute accuracy metrics
def computeMetrics(evalPreds):
    logits, labels = evalPreds  # Get logits and labels
    predictions = np.argmax(logits, axis=1)  # Predict labels with argmax
    # Calculate accuracy manually
    correct_predictions = np.sum(predictions == labels)  # Count correct predictions
    total_predictions = len(labels)  # Total number of labels
```

```
    accuracy = correct_predictions / total_predictions  # Calculate accuracy
    return {'accuracy': accuracy} # Return accuracy as a dictionary

# Freeze layers
for name, param in model.named_parameters():
    if not name.startswith('classifier'):
        param.requires_grad = False  # Freeze parameter
```

2.3 Train the transformer model on the MNIST training dataset

We set up the training configuration and train the model using the MNIST training dataset.

```
# Custom callback to log training metrics
class MetricsLogger(TrainerCallback):
    def __init__(self):
        super().__init__()
        self.trainLosses = []  # Store training losses
        self.valLosses = []  # Store validation losses
        self.valAccuracies = []  # Store validation accuracies
        self.currentEpochTrainLosses = []  # Store current epoch's training losses

    def on_log(self, args, state, control, logs=None, **kwargs):
        if logs is not None:
            if 'loss' in logs:
                self.currentEpochTrainLosses.append(logs['loss'])  # Log train loss
            if 'eval_loss' in logs:
                self.valLosses.append(logs['eval_loss'])  # Log val loss
            if 'eval_accuracy' in logs:
                self.valAccuracies.append(logs['eval_accuracy'])  # Log val accuracy

    def on_epoch_end(self, args, state, control, **kwargs):
        if self.currentEpochTrainLosses:
            avgTrainLoss = np.mean(self.currentEpochTrainLosses)  # Avg train loss for
```

```
epoch
            self.trainLosses.append(avgTrainLoss)  # Log avg train loss
            self.currentEpochTrainLosses = []  # Reset for next epoch

    def reset(self):
        self.trainLosses = []
        self.valLosses = []
        self.valAccuracies = []
        self.currentEpochTrainLosses = []

# Set up training arguments
trainingArgs = TrainingArguments(
    output_dir="./vit-base-mnist",  # Directory for checkpoints
    per_device_train_batch_size=32,  # Training batch size
    evaluation_strategy="epoch",  # Evaluate at each epoch end
    save_strategy="epoch",  # Save at each epoch end
    logging_steps=200,  # Log every 200 steps
    num_train_epochs=5,  # Number of epochs
    learning_rate=2e-4,  # Optimizer learning rate
    save_total_limit=3,  # Max number of checkpoints
    remove_unused_columns=False,  # Keep all dataset columns
    report_to='tensorboard',  # Report to TensorBoard
    load_best_model_at_end=True,  # Load best model at end
)

# Initialize MetricsLogger callback
metricsLogger = MetricsLogger()
metricsLogger.reset()

# Create Trainer instance
trainer = Trainer(
    model=model,
    args=trainingArgs,
    data_collator=collateBatch,
    compute_metrics=computeMetrics,
    train_dataset=processedDataset['train'],
    eval_dataset=processedDataset['validation'],
    tokenizer=processor,
    callbacks=[MetricsLogger()]  # Add custom callback
)

```

```
# Train the model
trainer.train()
```

2.4 Evaluate the transformer model on the MNIST test dataset

Finally, we evaluate the trained model on the MNIST test dataset and analyze its performance.

```
# Plot validation accuracies over epochs
plt.plot(metricsLogger.valAccuracies, '-x')  # Plot validation accuracies
plt.xlabel('Epoch')  # X-axis label
plt.ylabel('Validation Accuracy')  # Y-axis label
plt.title('Validation Accuracy vs. Epochs')  # Title
plt.show()  # Show plot

# Plot training and validation losses over epochs
plt.plot(metricsLogger.valLosses, '-x')  # Plot validation losses
plt.xlabel('Epoch')  # X-axis label
plt.ylabel('Validation Loss')  # Y-axis label
plt.title('Validation Loss vs. Epochs')  # Title
plt.show()  # Show plot

# Predict test labels, compute confusion matrix, and plot heatmap
predictionsOutput = trainer.predict(processedDataset['test'])  # Predict on test dataset
testPreds = np.argmax(predictionsOutput.predictions, axis=1)  # Get predicted labels
testLabels = predictionsOutput.label_ids  # Get true labels
confMatrix = confusion_matrix(testLabels, testPreds)  # Compute confusion matrix
plt.figure(figsize=(10, 7))  # Set figure size
sns.heatmap(confMatrix, annot=True, fmt='d', cmap='Blues', xticklabels=range(10), yticklabels=range(10))  # Heatmap of confusion matrix
plt.xlabel('Predicted Label')  # X-axis label
plt.ylabel('True Label')  # Y-axis label
plt.title('Confusion Matrix')  # Plot title
plt.show()  # Show plot
```

3 Discussion

3.1 Compare the performance of the transformer model with traditional convolutional neural networks (CNNs) typically used for MNIST classification

For the MNIST dataset, I designed a conventional convolutional neural network. Installing the essential packages and importing the relevant libraries are the first things we do. In the first job, we establish DataLoader objects for training, validation, and testing; we also define transformations for data augmentation, load the MNIST dataset for training and testing, and divide the training dataset into training and validation sets. We define the CNN class with various layers in the following challenge. In the next step, we initialize lists to hold epoch-wise data, define the loss function and optimizer, set the device to GPU if available, otherwise to CPU, and begin the training process. The last process involves testing the model using the test data, calculating the accuracy of the test, plotting the accuracy and loss with time, computing the confusion matrix, and finally plotting the confusion matrix.

Epoch	Training Loss	Validation Loss	Accuracy
1	0.3204	0.2960	0.9220
2	0.2335	0.2229	0.9385
3	0.1979	0.1968	0.9461
4	0.1868	0.1842	0.9489
5	0.1849	0.1811	0.9503

Table 1: Performance of Vision Transformer (VIT) model

Epoch	Training Loss	Validation Loss	Accuracy
1	0.8500	0.2990	0.8979
2	0.3229	0.1593	0.9522
3	0.2344	0.1165	0.9630
4	0.1884	0.0938	0.9701
5	0.1621	0.0798	0.9735

Table 2: Performance of CNN model

Comparison between models

1. The Vision Transformer (VIT) achieves an accuracy of 0.95, whereas the Convolutional Neural Network (CNN) reaches a higher accuracy of 0.97. This indicates that the CNN is more effective at classification than the VIT.

2. The training loss for the VIT decreases from 0.32 to 0.18, while the CNN's training loss drops from 0.85 to 0.16. This suggests that the CNN learns more efficiently than the VIT.

3. For validation loss, the VIT shows a reduction from 0.30 to 0.18, whereas the CNN's validation loss decreases from 0.30 to 0.08. This demonstrates that the CNN generalizes better compared to the VIT.

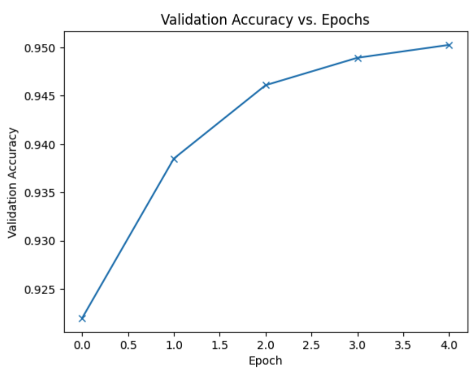

Figure 1: VIT Validation Accuracy

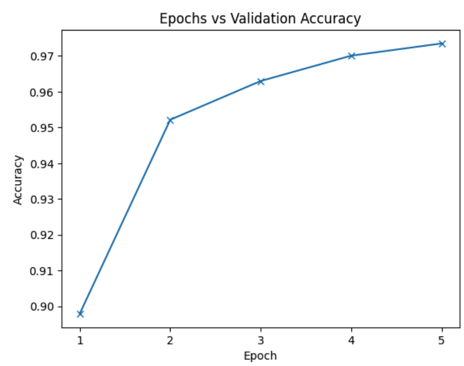

Figure 2: CNN Validation Accuracy

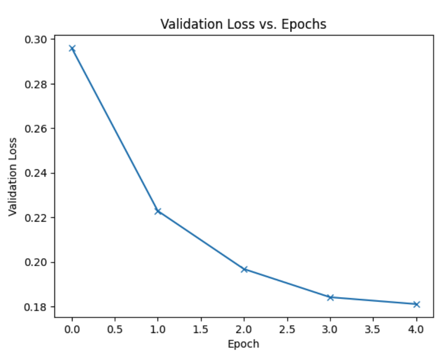

Figure 3: VIT Validation Loss

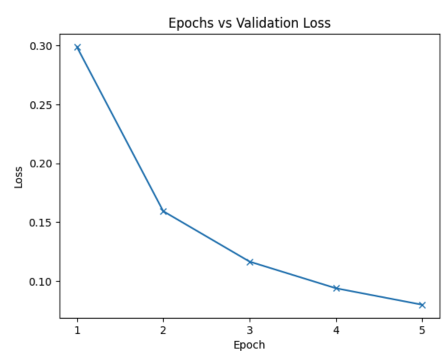

Figure 4: CNN Validation Loss

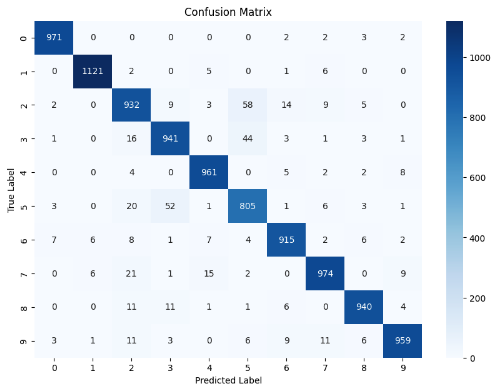

Figure 5: VIT Confusion Matrix

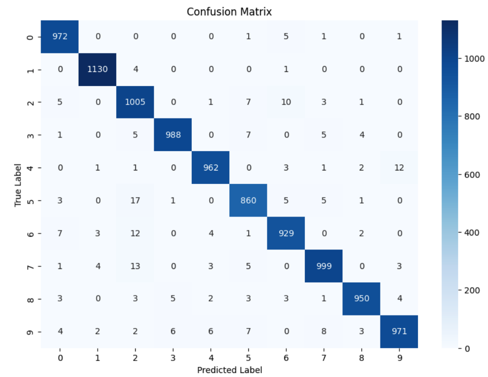

Figure 6: CNN Confusion Matrix

3.2 Highlight the strengths and weaknesses of using a transformer model for this type of task.

Strengths

1. Due to their ability to interpret images in their entirety, Visual Transformers capture context between different portions of an image, allowing the model to effectively identify subtle relationships.

2. They use patch embeddings that allocate weights to various visual segments, or "patches," enabling the model to understand the essence of each portion of the image and quickly find relationships.

3. Transformers utilize self-attention, which assigns similarity scores between different portions of the image using the patch embeddings.

Weaknesses

1. Due to the fact that transformers look at an image as a whole, they are very demanding on the device they train on. CNN look at localized regions, so less computational resources are required.

2. By finding unique peculiarities in the training dataset, transformers run the risk of over fitting their data, meaning they fit well on the training data, but not for the test data.

3.3 Reflect on the challenges you faced during the implementation and training of the transformer model

Finding the Right Training Arguments Because the program takes a long time to run (almost an hour or two), making modifications to the training arguments was often difficult. These training arguments include factors like batch size, logging steps, and learning rate. Initially, I had set the batch size to 64, which was the same as in my CNN model, to ensure a fair comparison between the CNN and ViT models. However, I noticed that the program's performance wasn't as good. I experimented with different batch sizes—both increasing and decreasing it. Decreasing the batch size actually resulted in a more accurate model.

Metrics Logger for the Graphs Originally, there was no way to keep track of metrics like validation loss and training loss because the HuggingFace interface didn't allow me to store that data in a list or anything similar. The only place these metrics were displayed was in the table that showed up during training. To resolve this, I created a MetricsLogger callback to store these variables in a list. I then used this data to plot validation graphs.

Problems Plotting the Confusion Matrix I initially struggled with plotting the confusion matrix and wasn't sure how to approach it. In addition, I wasn't sure how to make predictions from the model. After some research on Seaborn, I learned that it can be used to draw a heatmap for the confusion matrix, and to get all the predictions, I simply called the trainer to predict based on the test dataset.

3.4 Consider the potential improvements or alternative approaches that could enhance the performance of the transformer classifier on the MNIST dataset

Data Augmentation As of this moment, my code doesn't use data augmentation although it could be beneficial. In data augmentation, flipping, rotation, scaling, and other transformations are applied to data (not just images). Transformers require vast amounts of data for high accuracy, and augmentation helps to combat this problem by artificially creating additional data. More data generally correlates with better performance for many models. It also helps with overfitting, which occurs when our model performs very well on the training data but not on the test data. This indicates that the model is not generalized enough for all data, which can hinder model performance.

Additional Metrics The metric analyzed in this model is accuracy, which allows us to see how "correct" our model is, in other words, how true our predictions were. Despite this, there are other metrics that allow for a better assessment of our model. Precision provides the portion of all positive classifications that turn out to actually be positive. Recall gives the portion of all actual positive classifications that are classified as actual positives. The F1-score calculates the average of both precision and recall.

Alternative Models The model in this current project is a pre-trained Vision Transformer that has been adjusted for the MNIST database. If we wanted better accuracy and speed, other transformer models that could have been used include the Swin Transformer, DeiT (Data Efficient Image Transformer), CvT (Convolutional Vision Transformer), and more. The paper CvT: Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows introduces the Swin Transformer, which has a hierarchical vision Transformer architecture using shifted windows, enabling efficient computation and strong performance across various vision tasks. Additionally, we could have created a Vision Transformer from scratch with just PyTorch. This approach allows for more customization but would likely result in lower accuracy compared to both the CNN and pre-trained Vision Transformer, and the training speed would be considerably slower, potentially taking many hours.

4 Code

In this study, we have provided detailed analyses and implementations for both the Vision Transformer (ViT) and the Convolutional Neural Network (CNN) models. The corresponding Jupyter Notebooks can be accessed via the following links:

Vision Transformer (ViT) ViT Notebook (if you decide to run the code, here is token you can use: ”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”)

Convolutional Neural Network (CNN) CNN Notebook