Report on Alzheimer MRI Image Classification Task

Author: Maria Magdalena Michail

Date: 07/01/2025

Class: Deep Learning

Course Name: Big Data Engineering and Data Science

Institution: Athens Tech College

Table of contents

Objective

The primary aim of this project was to design and implement a machine learning model capable of accurately classifying MRI images into four dementia levels:

- Mild Demented
- Moderate Demented
- Non-Demented
- Very Mild Demented

Dataset Overview

The dataset used for this task consisted of MRI brain scans, organized into training and test sets:

- **Training Set**: 5,120 images
- **Test Set**: 1,280 images

Each image was labeled according to one of the four dementia levels. The dataset was sourced from the HuggingFace Dataset Repository and provided in Parquet format.

Implementation Steps

Data Loading and Preparation

```
urltest = "https://huggingface.co/datasets/Falah/Alzheimer_MRI/resolve/main/data/test-00000-of-00001-44110b9df98c5585.parquet"
urltrain = "https://huggingface.co/datasets/Falah/Alzheimer_MRI/resolve/main/data/train-00000-of-00001-c08a401c53fe5312.parquet"
```

These lines define the URLs for downloading the training and test datasets from HuggingFace.

urltest: URL pointing to the test dataset in Parquet format.

urltrain: URL pointing to the training dataset in Parquet format.

```
output_file_test = "test_data.parquet"
output_file_train = "train_data.parquet"
```

Specifies the local file names where the datasets will be saved after downloading.

output_file_test: Name of the file to save the test dataset locally.

output_file_train: Name of the file to save the training dataset locally.

```
if not os.path.exists(output_file_train):
    print("Downloading the dataset")
    response = requests.get(urltrain)
    with open(output_file_train, 'wb') as f:
        f.write(response.content)
    print("Download complete")
else:
    print("File already exists")
```

Checks if the training dataset file already exists locally; if not, downloads it.

1. os.path.exists(output_file_train): Checks if the file output_file_train exists in the current directory.
2. If the file does not exist:
    - Sends a GET request using requests.get(urltrain) to download the file.
    - Opens the file output_file_train in write-binary mode ('wb') and writes the content of the downloaded file (response.content) to it.
3. If the file already exists:
    - Prints "File already exists.", skipping the download.

```
if not os.path.exists(output_file_test):
    print("Downloading the dataset")
    response = requests.get(urltest)
    with open(output_file_test, 'wb') as f:
        f.write(response.content)
    print("Download complete")
else:
    print("File already exists")
```

Repeats the same process as above, but for the test dataset file (output_file_test).

Downloads the test dataset if it is not already available locally and saves it as test_data.parquet.

```
data_train = pd.read_parquet(output_file_train)
data_test = pd.read_parquet(output_file_test)
```

Loads the datasets into memory as pandas DataFrames using pd.read_parquet.

1. data_train: Reads the Parquet file output_file_train (training data) into a DataFrame.
2. data_test: Reads the Parquet file output_file_test (test data) into a DataFrame.

Dataset Class:

The class AlzheimerDataset(Dataset) is used to handle and preprocess the Alzheimer MRI dataset. The dataset is stored in a non-standard format, byte strings.

AlzheimerDataset customizes how data is accessed and prepared for the model:

- Reads image data from byte strings.
- Converts byte strings into RGB image format.
- Retrieves corresponding labels.

```
class AlzheimerDataset(Dataset):
    def __init__(self, dataframe, processor):
        self.data = dataframe
        self.processor = processor
```

Initializes the dataset.

dataframe: A Pandas DataFrame containing the dataset (images and labels).

processor: A Hugging Face image processor used for preprocessing images.

```
def __len__(self):
    return len(self.data)
```

Returns the total number of samples in the dataset (self.data). The __len__ method is a required method because the class inherits from torch.utils.data.Dataset.

PyTorch relies on this method to determine the total size of the dataset, which is critical for:

- Shuffling during training.
- Iterating through the dataset in batches.
- Knowing when to stop during training epochs.

```python
def __getitem__(self, idx):
    image_data = self.data.iloc[idx]['image']['bytes']
    label = self.data.iloc[idx]['label']
    image = Image.open(BytesIO(image_data)).convert("RGB")
    encoding = self.processor(images=image, return_tensors="pt")
    return {
        "pixel_values": encoding["pixel_values"].squeeze(0),
        "labels": torch.tensor(label, dtype=torch.long)
    }
```

Retrieves a single item (image and label) from the dataset at the specified index (idx). This is another required method. It allows the DataLoader to:

- Retrieve individual samples using their index.
- Apply batching logic by combining multiple items fetched using __getitem__.

__getitem__ method explained:

Extract Image Data:

- self.data.iloc[idx]['image']['bytes']: Accesses the image's byte data.

Extract Label:

- self.data.iloc[idx]['label']: Retrieves the corresponding label.

Load Image:

- Image.open(BytesIO(image_data)).convert("RGB"): Converts the image byte data into an image and ensures it's in RGB format.

Process Image:

- self.processor(images=image, return_tensors="pt"): Applies preprocessing to the image, returning it as a PyTorch tensor.
- squeeze(0): Removes the batch dimension.

Return Dictionary:

- pixel_values: Processed image tensor.

- labels: The label as a PyTorch tensor (torch.tensor).

Model Selection

I experimented with 4 different models to find which one has the best performance on the Alzheimer's MRI dataset.

I chose a diverse set of pre-trained models:

- microsoft/swin-tiny-patch4-window7-224: A Swin Transformer model with hierarchical feature maps.
- facebook/dino-vitb8: A Vision Transformer (ViT) trained with self-supervised learning.
- Falah/Alzheimer_classification_model: A specialized model for Alzheimer's classification.
- google/vit-base-patch16-224: A Vision Transformer with a patch-based input representation.

```
train_dataset = AlzheimerDataset(data_train, processor)
train_dataset_subsample_indices = np.random.choice(len(train_dataset), size=200, replace=False)
train_dataset_subsample = torch.utils.data.Subset(train_dataset, train_dataset_subsample_indices)

test_dataset = AlzheimerDataset(data_test, processor)
test_dataset_subsample_indices = np.random.choice(len(test_dataset), size=50, replace=False)
test_dataset_subsample = torch.utils.data.Subset(test_dataset, test_dataset_subsample_indices)
```

To reduce computation time and make experimentation feasible:

- Training Subsample: 200 samples randomly selected from the training dataset.
- Testing Subsample: 50 samples randomly selected from the test dataset.

```
for param in model.base_model.parameters():
    param.requires_grad = False
```

To reduce training time and computational cost:

- Freeze base model parameters. The base layers are not updated during training.

Training arguments configurations (mostly to reduce training time)**:**

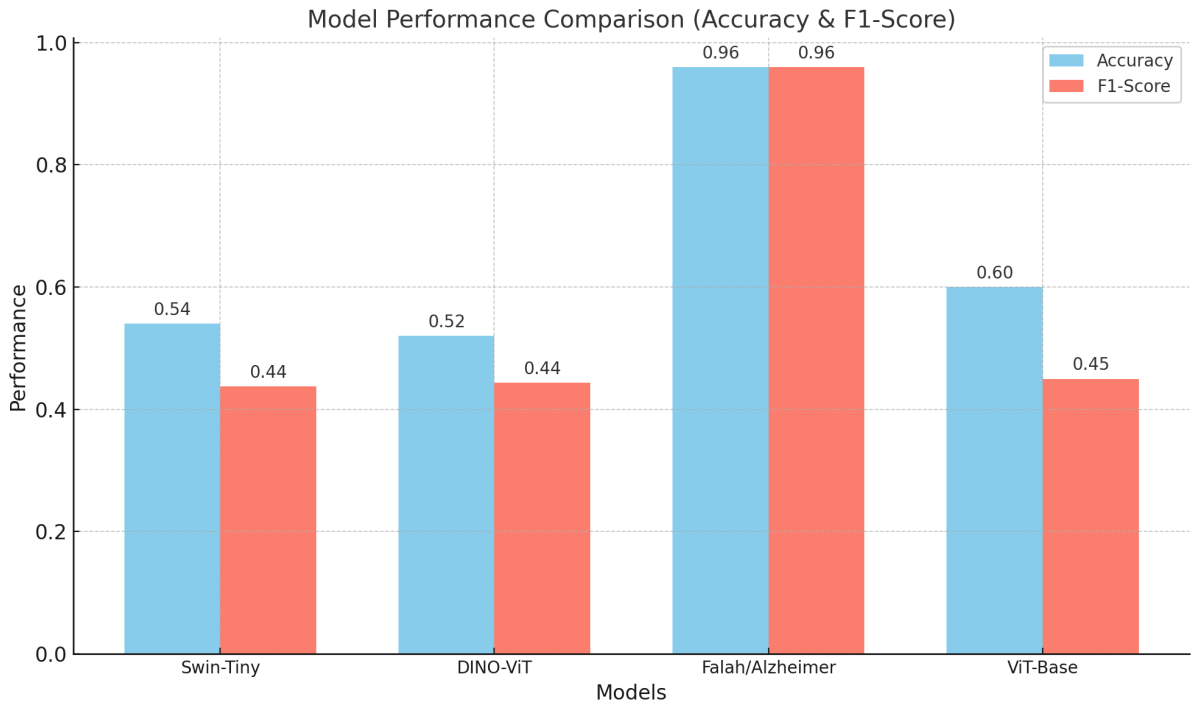
```
training_args = TrainingArguments(
    output_dir=f"./results_{model_name}",
    evaluation_strategy="no",
    save_strategy="no",
    learning_rate=5e-5,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=1,
    weight_decay=0.01,
    logging_dir=f"./logs_{model_name}",
    logging_steps=20,
    save_total_limit=1,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    fp16=True,
    gradient_accumulation_steps=2,
)
```

- evaluation_strategy="no": Disables automatic evaluation during training.
- num_train_epochs=1: Number of training epochs (1 in this case for quick experimentation).
- fp16=True: Enables mixed-precision training to reduce memory usage and speed up training.
- per_device_train_batch_size=4: Number of samples per batch during training. This is to not crash due to using all available RAM.
- gradient_accumulation_steps=2: Accumulates gradients over 2 steps before performing a backward pass, effectively doubling the batch size.

```
def compute_metrics(pred):
    labels = pred.label_ids
    preds = np.argmax(pred.predictions, axis=1)
    acc = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds, average="weighted")
    return {"accuracy": acc, "f1": f1}
```

Used to evaluate the different models in accuracy and f1-score.

```
metrics = trainer.evaluate(test_dataset_subsample)
print(f"Test Metrics for {model_name}:", metrics)
```



Here is a diagram comparing the four models based on their test accuracy and F1 scores. From the diagram the "Alzheimer Model" significantly outperforms the others in accuracy and F1 score and was selected as the final model.

Hypertuning parameters

The hyperparameters that were used to be tuned are the learning rate, gradient_accumulation_steps and weight_decay. Number of epochs will remain as 1 for quick training and batch size will remain 4 because it crashes due to limited RAM in bigger numbers.

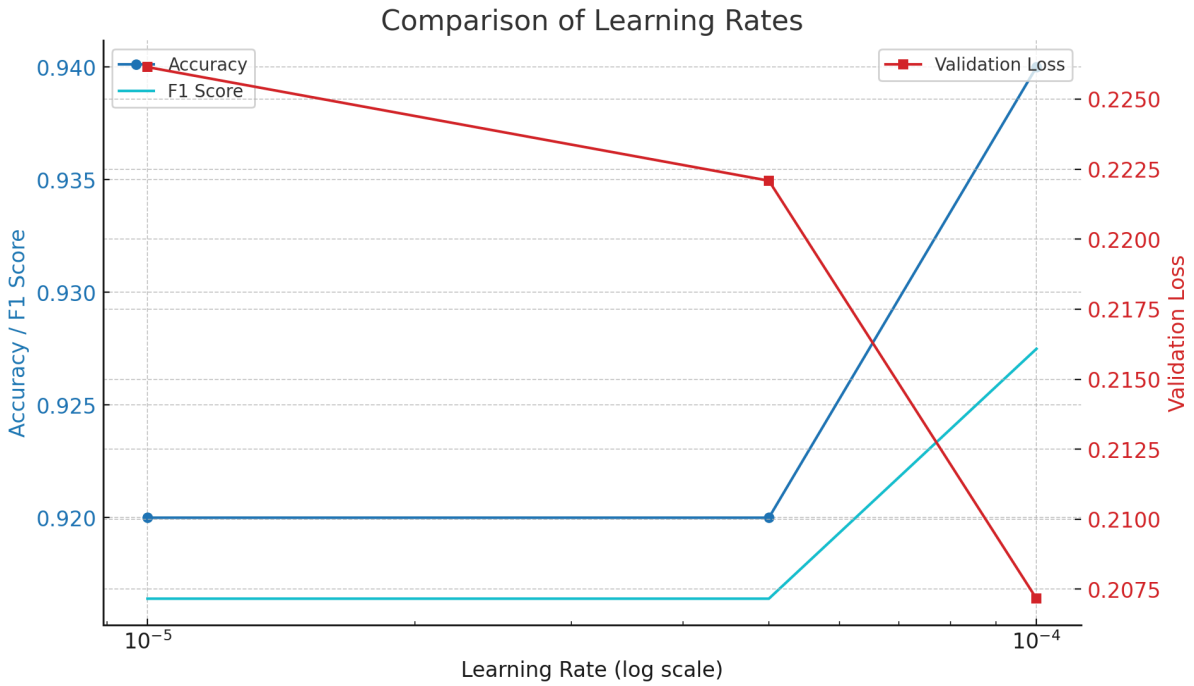
Learning rate

Three different learning rates were used:

- **1e-5**: For slower, stable fine-tuning, especially on noisy or complex datasets.
- **5e-5**: A practical middle ground for pretrained models.
- **1e-4**: Faster learning for potentially simpler datasets.

```
random_hyperparameter = [
    {"learning_rate": 1e-4},
    {"learning_rate": 5e-5},
    {"learning_rate": 1e-5},
]
```

Results:

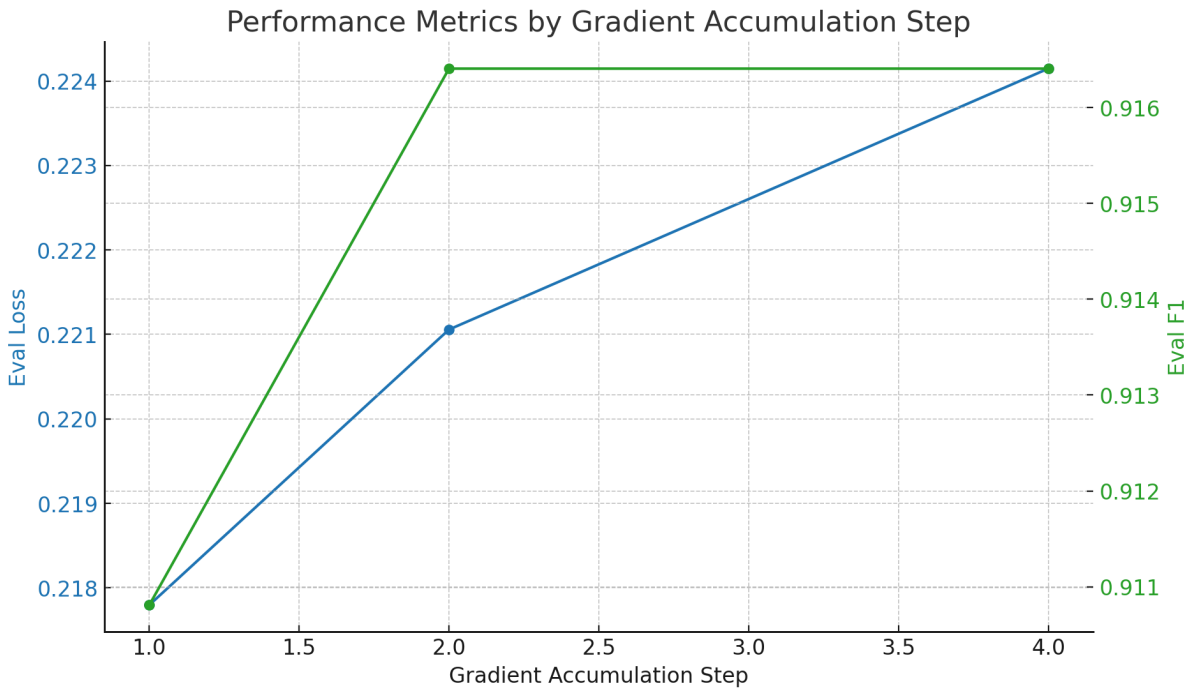


The diagram compares the performance of different learning rates (1e-04, 5e-05, and 1e-05) based on validation loss, accuracy, and F1 score:

- Best Learning Rate: 1e-04, as it provides the highest accuracy (0.94) and F1 score (0.93) while achieving the lowest validation loss (0.207).

Gradient accumulation steps

Three different gradient_accumulation_steps were used: 1, 2 ,4

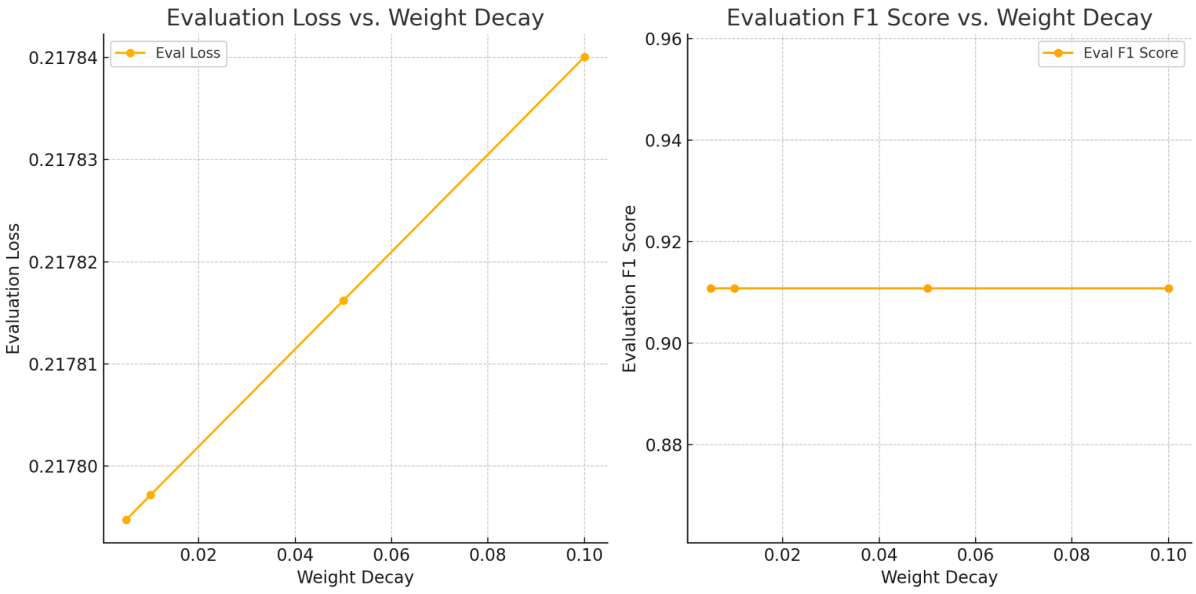


The best gradient accumulation step is **1**, as it results in the lowest evaluation loss (0.2178).

Weight decay

Four different weight decays were used:

```
random_hyperparameter = [
    {"weight_decay": 0.005},
    {"weight_decay": 0.01},
    {"weight_decay": 0.05},
    {"weight_decay": 0.1},
]
```



While all weight decay values yield similar results in terms of accuracy and F1 score, 0.005 achieves the lowest evaluation loss. Hence, it can be considered the optimal weight decay value for this model.

Final Model Training

Dataset Initialization

```
processor = AutoImageProcessor.from_pretrained("Falah/Alzheimer_classification_model")
```

Processor: Loads the Hugging Face image processor for preprocessing images.

```
train_dataset = AlzheimerDataset(data_train, processor)
test_dataset = AlzheimerDataset(data_test, processor)
```

Datasets: Creates PyTorch datasets for training (train_dataset) and testing (test_dataset) using the custom AlzheimerDataset class.

Model Initialization

```
model = AutoModelForImageClassification.from_pretrained(
    "Falah/Alzheimer_classification_model", num_labels=4
)
```

Loads a pre-trained Hugging Face image classification model with four output labels (classes).

num_labels=4: Specifies that the task involves four dementia levels:

- 0: Mild_Demented
- 1: Moderate_Demented
- 2: Non_Demented
- 3: Very_Mild_Demented

Model Selection: A pre-trained model from HuggingFace's AutoModelForImageClassification library was chosen. The specific model used was Falah/Alzheimer_classification_model, an image classification model trained on images of people with dementia. This model is a fine-tuned version of google/vit-base-patch16-224-in21k on the imagefolder dataset. It achieves the following results on the evaluation set:

- Loss: 0.4065
- Accuracy: 0.8375

The following hyperparameters were used during training:

- learning_rate: 5e-05
- train_batch_size: 16
- eval_batch_size: 16
- seed: 42
- gradient_accumulation_steps: 4
- total_train_batch_size: 64
- optimizer: Adam with betas=(0.9,0.999) and epsilon=1e-08
- lr_scheduler_type: linear
- lr_scheduler_warmup_ratio: 0.1
- num_epochs: 10

Training Configuration

```
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=1e-4,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=5,
    weight_decay=0.005,
    logging_dir="./logs",
    logging_steps=50,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    fp16=True,
    gradient_accumulation_steps=1,
)
```

The TrainingArguments class is used to configure various aspects of the training and evaluation process.

evaluation_strategy="epoch":

- Configures the evaluation to run at the end of each training epoch.

- Provides periodic feedback on the model's performance, ensuring that improvements and potential issues are monitored during training.

save_strategy="epoch":

- Saves the model checkpoint at the end of each epoch.
- Prevents data loss and allows resumption of training from the last checkpoint in case of interruptions.

learning_rate=1e-4:

- Sets the learning rate which controls the step size in gradient updates.

per_device_train_batch_size=4:

- Defines the number of samples processed in a single forward and backward pass per GPU/CPU during training.
- *Training with a bigger batch size than 4 would crash the session because it used all available RAM.*

per_device_eval_batch_size=4:

- Sets the batch size for evaluation, ensuring consistency with the training batch size.
- Allows the model to process the same number of images per step during evaluation.
- *Training with a bigger batch size than 4 would crash the session because it used all available RAM.*

num_train_epochs=5:

- Specifies that the training process will run for five full passes over the dataset.

weight_decay=0.01:

- Adds a regularization term to the loss function to penalize large weights, helping to prevent overfitting.

load_best_model_at_end=True:

- Ensures that the best-performing model (based on the evaluation metric) is automatically loaded after training.

metric_for_best_model="accuracy":

- Specifies accuracy as the primary metric for determining the best model.

Training and Evaluation

```
def compute_metrics(pred):
    labels = pred.label_ids
    preds = np.argmax(pred.predictions, axis=1)
    acc = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds, average="weighted")
    return {"accuracy": acc, "f1": f1}
```

The compute_metrics function calculates evaluation metrics, accuracy and F1-score, for model performance during training and evaluation.

```
os.environ["WANDB_DISABLED"] = "true"
```

Prevents Hugging Face Trainer from automatically logging metrics and training progress to the Weights & Biases (W&B) platform. It was disabled because otherwise I would need to create an account, acquire an API key and set the API key.

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=processor,
    compute_metrics=compute_metrics
)

trainer.train()
```

The Trainer class in Hugging Face is an API for training and evaluating models.

Components:

1. model: the model used was Falah/Alzheimer_classification_model
2. training_args: the training arguments were defined above
3. Train_dataset: the train dataset we got from Hugging Face
4. eval_dataset: the test dataset we got from Hugging Face

5. tokenizer: the Hugging Face image processor for preprocessing images
6. compute_metrics: defines the metrics used to evaluate model performance on the validation dataset

With the train() method we start the training loop, where the model processes data from the train_dataset and evaluates performance on the eval_dataset using the compute_metrics function.

```
print("Evaluating the model on test data")
metrics = trainer.evaluate(test_dataset)
print("Test Metrics:", metrics)
```

The evaluate method computes the evaluation metrics, accuracy, F1-score, on the test_dataset.

Model Saving

```
trainer.save_model("./fine_tuned_model")
processor.save_pretrained("./fine_tuned_processor")
```

Save the fine-tuned model's weights and configuration and also save the processor for future use. The training time was around 7 hours, so instead of retraining the model to predict new values we can load the fine tuned model and processor to make predictions on the loaded model.

Results

The final model was evaluated on the test dataset, achieving the following metrics:

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.137700	0.280242	0.903125	0.902699
2	0.092700	0.282033	0.905469	0.905381
3	0.088600	0.284234	0.906250	0.906182
4	0.119200	0.284501	0.907813	0.907808
5	0.079300	0.285060	0.907813	0.907808

Epoch 5 stands out as the best model. Epoch 5 has the lowest training loss (0.079300), followed by epoch 3 (0.088600). Epochs 4 and 5 have the highest accuracy and F1 score (0.907813 and 0.907808, respectively). Epoch 5 has the highest validation loss (0.285060), however it is not that big of a difference from the lowest validation loss (0.280242).

Test evaluation metrics:

'eval_loss': 0.2845005393028259

'eval_accuracy': 0.9078125

'eval_f1': 0.9078079269375354

'epoch': 5.0

eval_accuracy and eval_f1 confirm strong generalization on the test dataset. eval_loss is low, supporting consistent performance across training and test datasets. These metrics suggest that the fine-tuned model is well-optimized.

Code Repository

The complete implementation, including preprocessing scripts, training loops, and evaluation code, is available in the submitted repository.

mmmichail/DeepLearning

References

[1] Falah, "Alzheimer MRI Dataset," [Online]. Available: https://huggingface.co/datasets/Falah/Alzheimer_MRI. [Accessed: Jan. 7, 2025].

[2] Falah, "Alzheimer Classification Model," Hugging Face, [Online]. Available: https://huggingface.co/Falah/Alzheimer_classification_model. [Accessed: Jan. 7, 2025].