ADNI Classifier

Introduction

Alzheimer's disease (AD) affects patients' memory, behavior and cognition. With no existing drug on the market that can completely cure this disease, early detection and intervention are crucial. Nowadays, various deep learning models have been developed for classifying Alzheimer's Disease Neuroimaging Initiative (ADNI) data, playing a key role in promoting AD detection. In this study, the Global Filter Neural Network (GFNet), a latest Vision Transformer (ViT), is used to build an ADNI classifier, aiming at achieving at least 0.8 accuracy on the test dataset.

A standard transformer architecture consists of two components: an encoder and a decoder, and it is commonly used for natural language processing tasks. Vision Transformer (ViT) is an example of an encoder-only model, where the transformer is adapted to computer vision. During the ViT processing, the input image is first divided into square patches, flattening into a one-dimensional sequence (Figure 1). These patches are linearly projected to the desired dimensionality (Figure 1). Position embeddings are added to the patches to retain information about their original locations within the image (Albanie, 2023). The resulting sequence is passed through the transformer encoder, and finally, the output is processed by a multilayer perceptron (MLP) to classify the image (Figure 1).

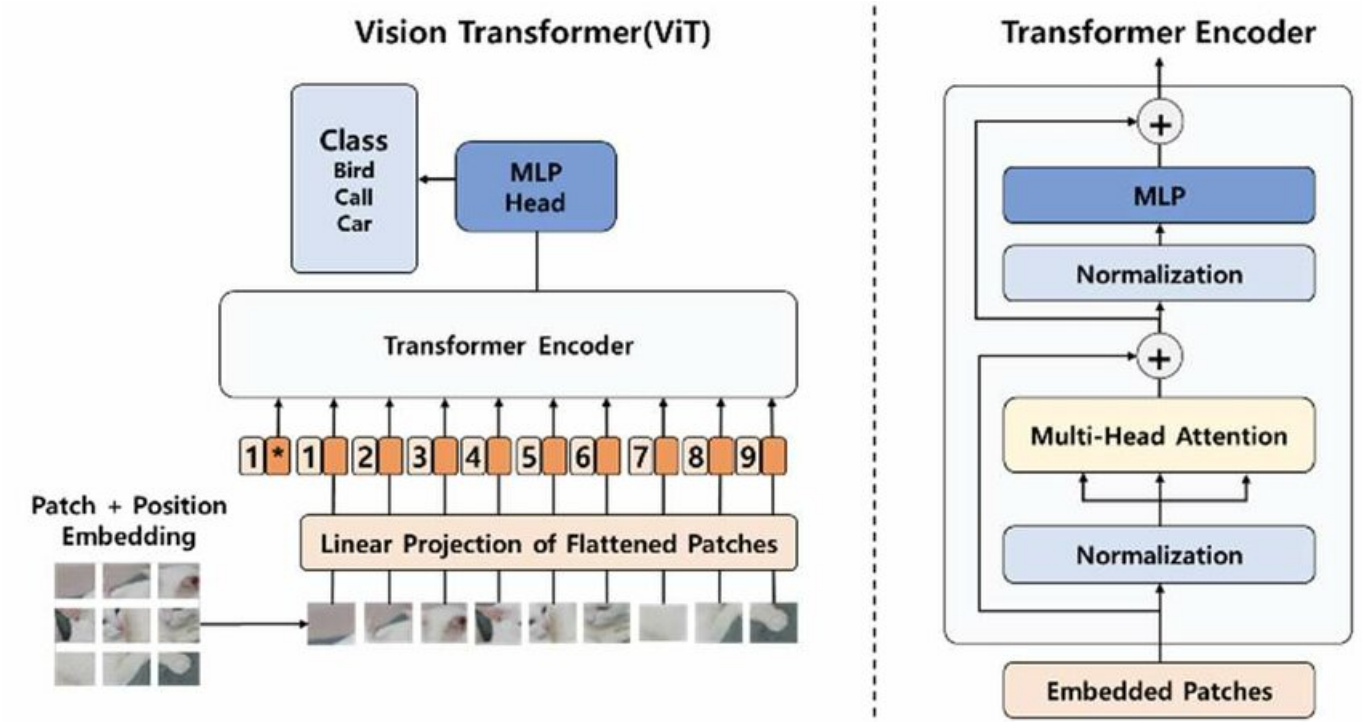


Figure 1: The Architecture of the Vision Transformer (Bang et al., 2023)

In contrast, the overall architecture of GFNet is similar to the ViT but with some minor changes. The multi-head attention layer in Figure 1 is replaced by a global filter layer as shown in Figure 2. The global filter layer has three key operations: 2D Fourier transform (2D FFT), an element-wise multiplication between frequency domain features and learnable global filters, and a 2D inverse Fourier transform (2D IFFT). When applying GFNet to ADNI data, the 2D FFT breaks down the spatial features (extracted from a brain image) into their frequency components, corresponded to different scales of brain structures (NTi Audio, n.d.). Hence, this transformation enables the model to analyze the global spatial patterns across the brain, which is essential for

distinguishing between Alzheimer's Disease and Normal Control brains. In the frequency domain, the network applies learnable global filters, which are designed to enhance relevant features and suppress irrelevant or noisy components (Rao et al., 2021). Since global filters operate on a broad scale, they also capture the long-range dependencies, such as the relationship between different brain regions (Rao et al., 2021), which are crucial for classification tasks. At the end, the 2D IFFT maps the modified features back to the spatial domain, enabling further processing through Feedforward Network (FFN). The next section explains the GFNet architecture and the implementation of the model in more detail.

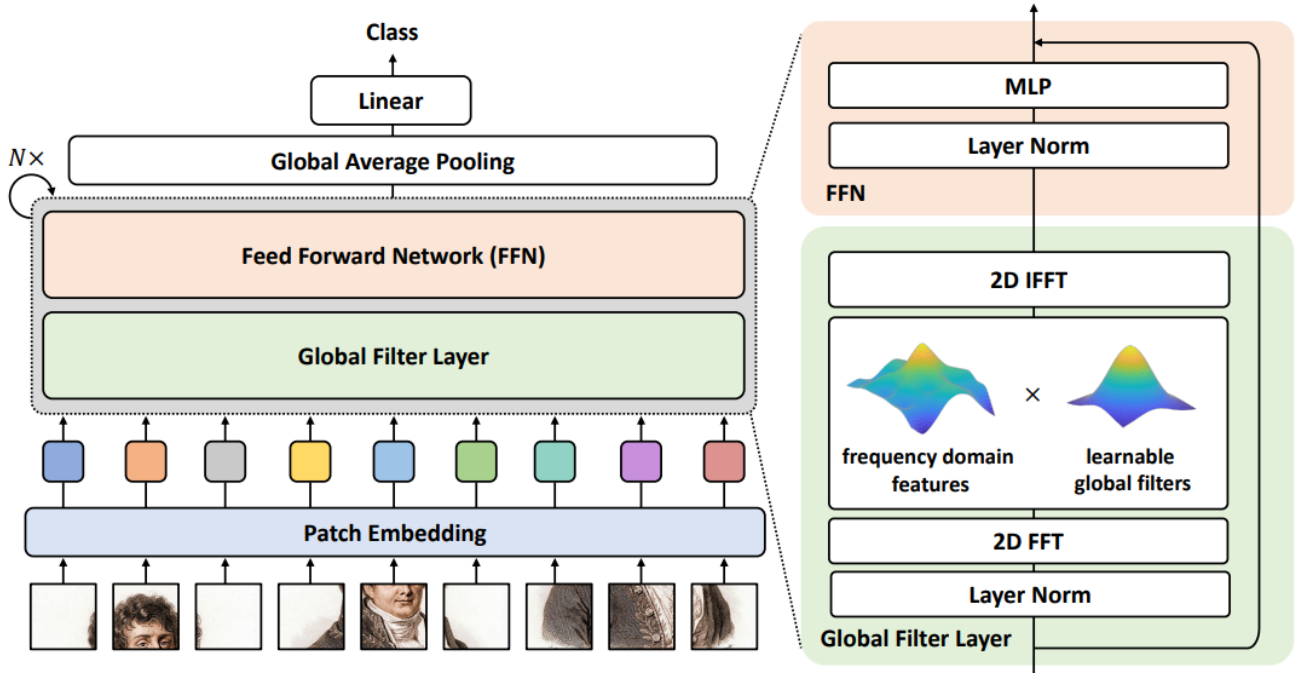


Figure 2: The Overall Architecture of GFNet (Zhao et al., 2021)

Description of the Algorithm

The GFNet model is implemented across four Python files: `dataset.py`, `modules.py`, `train.py` and `predict.py` The purpose of each file and the explanation of the code is described below:

1. dataset.py

The original ADNI brain data was preprocessed into training and testing datasets and has two classes: Alzheimer's disease and Normal Cognitive (NC). The input images are 2D brain slices in grayscale (converted into 'RGB' during loading as required the input channels for GFNet are 3) and have a size of 256 pixels in width and 240 pixels in height. The first 10 input images from the training dataset are shown below:

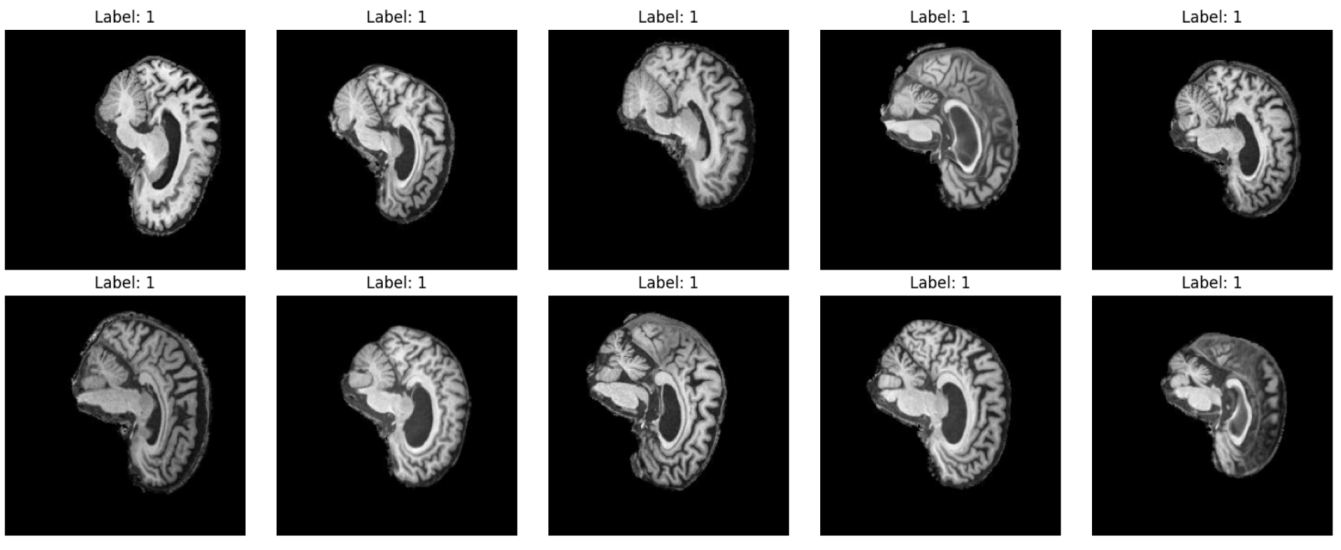


Figure 3: First 10 Images from the Training AD Dataset with Labels.

Since the images are compressed, the function `extract_zip(zip_path, extract_to)` is created to extract all the images within the zip file to the folder assigned to the 'extract_to'. The custom dataset class `ADNIDataset` is used to load images from both 'AD' and 'NC' classes from respective folder and assign 1 and 0 respectively. In order to normalize the image data, the mean and standard deviation of all pixels were calculated using `get_mean_std(loader)` (Saturn Cloud, 2023). However, it was found that mean and std values for both datasets are extremely small, indicating that the image has been normalized during preprocessing (very little variation in pixels values). The most notable aspect of the last function `get_data_loaders` is the use of data augmentations on the training dataset, generating new images to further improve model's performance. Moreover, 20% of the training dataset is used to validate the performance of the model during training.

2. modules.py

The source code for the GFNet model was developed by Yongming Rao, Wenliang Zhao, Zheng Zhu, Jiwen Lu and Jie Zhou in 2021 (Rao et al., 2021). The code was directly used to implement the ADNI image classification task. As mentioned in Introduction, the model's architecture consists of three major parts: the patch embedding, the global filter layer and MLP.

In `PatchEmbed` class, the utility function `to_2tuple` ensures that both single `img_size` and `patch_size` inputs are converted into tuples of two values. For example, a single input 224 will be transformed to `(224, 224)`, indicating 224 x 224 pixels.

```
self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size,
stride=patch_size)
```

```
def forward(self, x):
        B, C, H, W = x.shape
        # FIXME look at relaxing size constraints
        assert H == self.img_size[0] and W == self.img_size[1], \
            f"Input image size ({H}*{W}) doesn't match model ({self.img_size[0]}*
{self.img_size[1]})."
```

```
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```

During the forward pass, the assertion makes sure the input image size matches the expected size. The input image is first projected into patch embeddings using the convolution operation `self.proj`, where each patch is transformed into an embedding vector of a specific dimension: `embed_dim`. The output of `self.proj` has the shape `(B, embed_dim, H/patch_size, w/patch_size)`. This is followed by `flatten(2)` which flattens the spatial dimensions (height and width) of the output into a single dimension `num_patches = (H/patch_size)*( w/patch_size)`. The resulting output, in the form `(B, embed_dim, num_patches)` is then switched into `(B, num_patches, embed_dim)`using `transpose(1,2)`.

```
self.complex_weight = nn.Parameter(torch.randn(h, w, dim, 2, dtype=torch.float32)
* 0.02)
```

```
def forward(self, x, spatial_size=None):
        B, N, C = x.shape
        if spatial_size is None:
            a = b = int(math.sqrt(N))
        else:
            a, b = spatial_size

        x = x.view(B, a, b, C)

        x = x.to(torch.float32)

        x = torch.fft.rfft2(x, dim=(1, 2), norm='ortho')
        weight = torch.view_as_complex(self.complex_weight)
        x = x * weight
        x = torch.fft.irfft2(x, s=(a, b), dim=(1, 2), norm='ortho')

        x = x.reshape(B, N, C)

        return x
```

In `GlobalFilter`, `B`, `N` and `C` correspond to the output from `PatchEmbed`. The `self.complex_weight` defines the learnable parameter with shape `(h, w, dim, 2)`. However, the flat sequence produced by the patch embedding need to be reshaped into a 2D spatial format for FFT operation. As a result, N is sqaure-rooted into `a` and `b` if `spatial_size' is not specified. By multiplying the input by the learned complex weight, the model can adjust specific frequency components within the input.

```
class Mlp(nn.Module):
    def __init__(self, in_features, hidden_features=None, out_features=None,
act_layer=nn.GELU, drop=0.):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
```

```
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x
```

In MLP, the fully connected layers `self.fc1` and `self.fc2` map the input features to a hidden layer of size `hidden_features`, and then the `out_features`. The `nn.layer` applies a linear transformation in the form of `y = xW+b` with weight matrix w and bias b. The activation function `GELU` by default applies a non-linear transformation to the output of the fc1, which helps the model to learn complex patterns. Moreover, the `self.drop` is a regularization technique that randomly sets a fraction of elements to zero during training to prevent overfitting.

Furthermore, the `Block` class represents a building block used in GFNet.

```
class Block(nn.Module):

    def __init__(self, dim, mlp_ratio=4., drop=0., drop_path=0.,
act_layer=nn.GELU, norm_layer=nn.LayerNorm, h=14, w=8):
        super().__init__()
        self.norm1 = norm_layer(dim)
        self.filter = GlobalFilter(dim, h=h, w=w)
        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim,
act_layer=act_layer, drop=drop)

    def forward(self, x):
        x = x + self.drop_path(self.mlp(self.norm2(self.filter(self.norm1(x)))))
        return x
```

The layer normalization is used both before and after the filter, as shown in Figure 2. Additionally, an advanced technique known as stochastic depth (drop path) is applied to Mlp output, allowing some blocks to be skipped during training to reduce overfitting. If `drop_path = 0`, no operation will perform on the input due to `nn.Identity()`.

Essentially, the GFNet is built using the four classes mentioned above. Another point worth noting in `GFNet` is that the position embeddings are added to the patch embeddings to encode spatial information in `GFNet` class, and dropout is applied for regularization.

```
self.pos_embed = nn.Parameter(torch.zeros(1, num_patches, embed_dim))
self.pos_drop = nn.Dropout(p=drop_rate)
```

```
def forward_features(self, x):
        B = x.shape[0]
        x = self.patch_embed(x)
        x = x + self.pos_embed
        x = self.pos_drop(x)

        for blk in self.blocks:
            x = blk(x)

        x = self.norm(x).mean(1)
        return x
```

3. train.py

There are three functions contained in `train.py`. The first is the `plot_metrics` whcih generates the graphs for training and validation losses, as well as accuracy over the epochs, and saves the graph to `save_path` if provided. The other two functions, `train` and `validate`, are responsible for training and validating the model, calculating the average loss, accuracy, and confusion matrix. To maintain repeatability, set seeds during training. Since hyperparameters in training GFNet are significant on its performance, an Optuna study is employed to identify the optimal values for the learning rate, weight decay rate and drop path rate within the provided range. A total of 10 trials are conducted, with 30 epochs run for each trial.The best model (model with the highest val accuracy) from each trial is also saved.

```
learning_rate = trial.suggest_loguniform('lr', 1e-5, 1e-3)
weight_decay = trial.suggest_loguniform('weight_decay', 1e-5, 1e-3)
drop_path_rate = trial.suggest_uniform('drop_path_rate', 0.0, 0.5)
```

The model, criterion and optimizer are initilized as below:

```
model = GFNet(
                img_size=512,
                patch_size=16, embed_dim=512, depth=19, mlp_ratio=4,
drop_path_rate=drop_path_rate,
                norm_layer=partial(nn.LayerNorm, eps=1e-6)
        )

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=weight_decay)
model.to(device)
```

Once the optimization is completed, the best hyperparameters obtained will then be used to train the model for 70 epochs, with the best model saved to `model.pth`, and the losses and accuracy visualized.

4. predict.py

The best model identified during training is used to evaluate the performance on the test dataset, and the resulting confusion matrix is displayed.

```
model.load_state_dict(torch.load('model.pth'))
test_loss, test_accuracy, cm = validate(model, test_loader, criterion, device)
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")

print(f"Confusion Matrix:\n{cm}")
```

Result Analysis

According to the algorithm, the optimal hyperparameters are shown as below:

```
Best hyperparameters:  {'lr': 0.00016023270571709292, 'weight_decay': 1.8234018632998165e-05, 'drop_path_rate': 0.08789879211387502}
Best validation accuracy:  0.9577137546468402
```

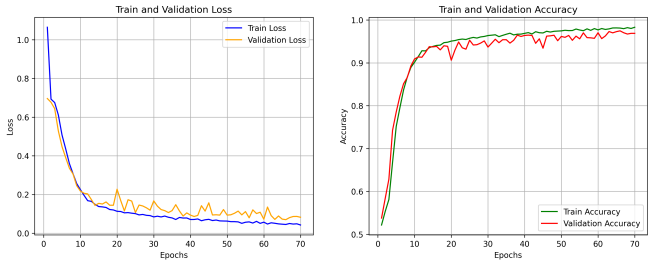
Figure 4: The Best Hyperparameters (with the Highest Val Accuracy) out of 10 Trials

However, after reviewing the test accuracy across 10 trials, I found the set of hyperparameters of the 8th trial interesting.

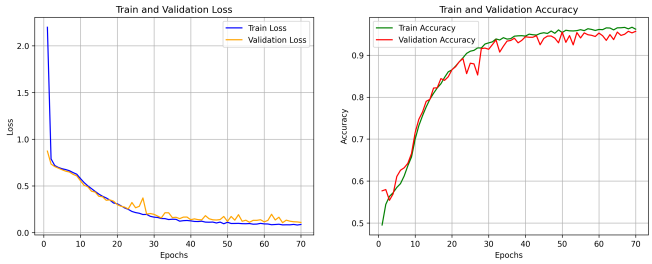
```
--- Trial 8 ---
Hyperparameters for trial 8: lr=1.2655138149073937e-05, weight_decay=9.472938882625012e-05, drop_path_rate=0.17728764992362356
using linear droppath with expect rate 0.08864382496181178
```

Figure 5: Hyperparameters with High Test Accuracy during training across 30 epochs

Although this set did not achieve a relatively high validation accuracy across 30 epochs, it reached a test accuracy of 0.67 multiple times during training, while the rest trials only reached a maximum of 0.66. As a result, both sets of hyperparameters were used to retrain the model over 70 epochs to evaluate their performance. The plots for losses and accuracy of both models are shown as below:



Graph 1: Losses and Accuracy of the Best Model (optimal hyperparameters) across 70 Epochs



Graph 2: Losses and Accuracy of the Best Model (8th Trial's Hyperparameters) across 70 Epochs

Graph 1 presents two subplots illustrating the performance of the GFNet model trained with the optimal hyperparameters resulting from 10 trials. During the early epochs, both the training and validation losses

decrease dramatically, indicating that the model is learning efficiently and reducing error. Although the validation loss starts to fluctuate after the 30th epoch, it continues to decrease alongside the training loss. Despite this, the loss curves for training and validation gradually shows a tendency to disperse, suggesting the probability of overfitting. A similar trend is observed in the accuracy plot: while both accuracies increase as losses decrease, a small gap exists between them. In contrast, the loss and accuracy plots in Graph 2 (for the model trained with hyperparameters giving relatively high test accuracy) suggest that the model is not overfitting. The curves for both loss and accuracy are closely aligned, revealing that the model is generalizing well to unseen validation. Additionally, the model trained with the optimal hyperparameters achieves a validation accuracy of approximately 0.97, whereas the model trained with the hyperparameters from the 8th trial reaches a slightly lower accuracy of around 0.95. However, when testing the best models from both sets of hyperparameters, the model with 8th trial's hyperparameters outperformed, achieving a higher test accuracy of 0.6799 on the test set (Figure 7) compared to 0.6607 (Figure 6). The higher train and validation accuracy but lower test accuracy is likely due to the overfitting, where the trained model struggles to make correct predictions on the new test data (IBM, n.d.). The optimal hyperparameters have a `drop_path_rate` of 0.088, whereas trial 8 has a rate of 0.177. The lower drop path rate may be a key factor contributing to the overfitting, as fewer layers are dropped during training, allowing the model to memorize the training data instead of learning general patterns.

Figure 6: The Test Accuracy and the Confusion Matrix of the Best Model (Optimal Hyperparameters)

Figure 7: The Test Accuracy and the Confusion Matrix of the Best Model (8th Trial's Hyperparameters)

As a result, the model trained using trial 8's hyperparameters has a better performance.

```
model = GFNet(
                img_size=512,
                patch_size=16, embed_dim=512, depth=19, mlp_ratio=4,
drop_path_rate=0.17728764992362356,
                norm_layer=partial(nn.LayerNorm, eps=1e-6)
        )
criterion = nn.CrossEntropyLoss()
optimizer = optim.RAdam(model.parameters(), lr=1.2655138149073937e-05,
weight_decay=9.472938882625012e-05)
```

However, this model has a relatively high false positive rate (1962 FP cases compared to 1858 cases). During the situation of detecting whether a patient has Alzheimer's Disease, a higher false positive rate means incorrectly diagnosing healthy individuals as having the disease, which can cause unnecessary stress and potentially inappropriate treatments. Lastly, there are still limitations in the training process, as the final result falls short of the expected 0.8 accuracy on the test set. For instance, the Optuna study was limited to only 10 trials with 30 epochs per trial, which may have contributed to overfitting when selecting the best hyperparameters. Therefore, more trials with a higher number of epochs could help identify the true best hyperparameters. To reach the target accuracy of 0.8, more advanced techniques should be employed, such as using the pre-trained model and fine-tuning it on the given dataset.

References

1. Albanie, S. (2023). *Vision Transformer Basics.* YouTube. https://www.youtube.com/watch?v=vsqKGZT8Qn8
2. Bang, J.-H., Park, S.-W., Kim, J.-Y., Park, J., Huh, J.-H., Jung, S.-H., & Sim, C.-B. (2023). *CA-CMT: Coordinate attention for optimizing CMT Networks.* IEEE Access, 11, 76691–76702. https://doi.org/10.1109/access.2023.3297206
3. IBM. (n.d.). *Overfitting.* https://www.ibm.com/topics/overfitting
4. NTi Audio. (n.d.). *Fast Fourier Transform (FFT): Basics, strengths and limitations.* NTi Audio. https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft.
5. OpenAI. (2024). ChatGPT (Oct 2024 version) [Large language model]. https://openai.com
6. Saturn Cloud. (2023). *How to normalize image dataset using PyTorch.* Saturn Cloud. https://saturncloud.io/blog/how-to-normalize-image-dataset-using-pytorch/
7. Zhao, R., Liu, Z., Lin, J., Lin, Y., Han, S., & Hu, H. (2021). *Global Filter Networks for Image Classification.* arXiv. https://arxiv.org/abs/2107.00645