Brain Tumor Classification Using PyTorch

1. Introduction

Deep learning has become a vital tool in medical imaging, offering new possibilities in diagnostics, particularly for tumor identification. This homework aims to classify brain MRI scans into four categories: pituitary tumor, no tumor, meningioma tumor, and glioma tumor using state-of-the-art pre-trained models from TIMM library.

The goal is to build, evaluate, and optimize multiple image classification models to achieve high accuracy and efficiency. Various training configurations, including optimizers, schedulers, and data augmentation techniques, are explored. Performance metrics such as accuracy, loss, latency, and model size are compared, with an emphasis on optimizing inference through model export.

2. Preprocessing

The dataset was split into training and testing sets, with the training set consisting of 2,850 images and the testing set consisting of 388 images. Data transformations were applied to ensure consistency and enhance model performance. Images were resized to a uniform shape of (224, 224) pixels, converted to tensors, and normalized with mean and standard deviation values appropriate for pre-trained models.

The training and testing datasets were then loaded using PyTorch's DataLoader, with batch sizes of 32. Shuffling was enabled for the training set to enhance learning. The DataLoader was used for efficient data handling, allowing for batching (to optimize GPU utilization), multi-processing (to speed up data loading), and shuffling (to prevent overfitting and improve generalization). The number of classes in the dataset is 4, corresponding to the 4 different labels. Furthermore, a random seed of 42 was set to ensure reproducibility of results across different runs.

3. Model, Train, and Test Functions

A generic create_model function was created to allow flexibility in using different pre-trained models. This modifies the final layer to match the number of classes, which is four in this case. After the model is created, it is loaded onto the appropriate device (GPU if available, otherwise CPU) to ensure efficient training and inference.

In the training loop, the model is set to training mode using model.train(). The loop iterates over the training data, performing the following steps for each batch:

- Inputs and labels are transferred to the device.
- The optimizer's gradient is reset using optimizer.zero_grad().
- The model makes predictions on the inputs, and the loss is computed using the specified loss function.

- The gradients are calculated via backpropagation (loss.backward()), and the optimizer updates the model parameters (optimizer.step()).
- Metrics such as running loss and accuracy are tracked for monitoring the model's progress.

In the testing loop, the model is set to evaluation mode using model.eval(). The loop iterates over the testing data without updating model weights:

- Inputs and labels are transferred to the device.
- Predictions are made, and the loss is computed.
- The predicted classes are compared to the true labels to calculate the accuracy.
- The testing loop is wrapped in a torch.no_grad() block to reduce memory usage and computational overhead, as no gradients are required during evaluation.

4. Effect of Learning Rates

To explore the effect of different learning rates on model performance, three learning rates (0.01, 0.001, and 0.0001) were used while training the ResNet50 model for 10 epochs. The classification loss was computed using nn.CrossEntropyLoss(), and the Adam optimizer was used along with a StepLR scheduler with a step size of 5.

Learning Rate, 0.01: Training started with relatively high loss but quickly converged, achieving a training accuracy of 99.37% by the end of 10 epochs. However, the test accuracy fluctuated significantly and only reached 73.45%. This suggests potential overfitting as the model learned the training data well but did not generalize effectively.

Learning Rate, 0.001: This learning rate provided a balance between convergence speed and stability. The model achieved a final training accuracy of 99.89% and a test accuracy of 77.32%. The test loss decreased steadily, indicating better generalization compared to the higher learning rate.

Learning Rate, 0.0001: With the smallest learning rate, the model's training process was more gradual. It achieved a training accuracy of 99.05% and a test accuracy of 72.94%. As the model learned more slowly, it showed relatively consistent test performance.

A graph was plotted (Figure 1) to visualize the convergence of training and test accuracy over the epochs. The plot shows how the model's accuracy steadily improved, highlighting the effectiveness of this learning rate in balancing learning speed and stability.

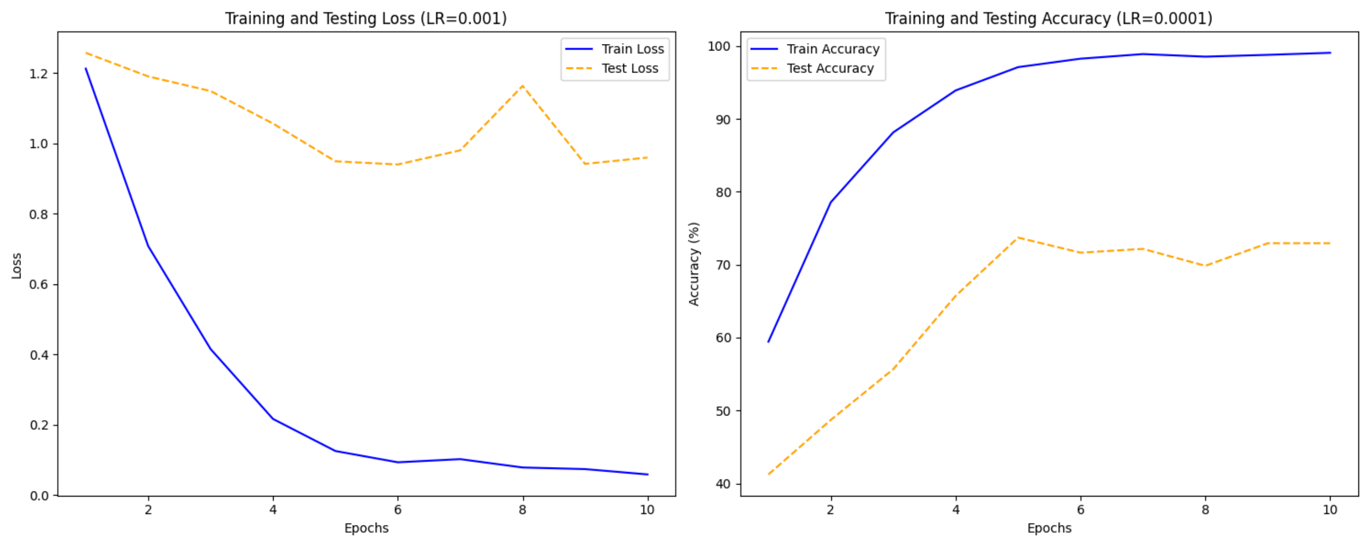


Figure 1: Training and Test accuracy and loss

5. Data Augmentation and Model Parameters

The results from the previous step indicated a high chance of overfitting, as the training accuracy was over 98% while the testing accuracy ranged between 73-77%. To address this, additional data augmentation techniques were applied, including randomly flipping images horizontally and vertically, rotating images by up to 15 degrees, adjusting brightness, contrast, saturation and converting images to grayscale with a probability of 10%.

A new model function (create_model_dropout) was also created, which included a dropout layer to help reduce overfitting by randomly deactivating neurons during training. Additionally, weight decay was added to the optimizer to regularize the model and prevent it from fitting too closely to the training data. The model showed improved generalization, with the test accuracy increasing to 79.12%.

6. Effect of Different Augmentation Techniques

To further reduce overfitting, new augmentation techniques were applied:

- RandomResizedCropAndInterpolation: This technique involves cropping images to random sizes and aspect ratios, followed by resizing them to (224, 224). It also used random interpolation methods, adding variability to the training data.
- Auto Augment: A more advanced augmentation policy was used to apply a sequence of transformations with randomized parameters, which helped increase the diversity of the training set.

A simpler model, ResNet18, was used and trained for 20 epochs to observe the impact of these new augmentation techniques over a longer period. The results showed that ResizedCropAndInterpolation was not able to improve the test accuracy, however Auto Augment was able to take it to as high as 83.25%.

The stability of the results also suggests the potential of adding early stopping as an additional strategy to prevent overfitting.

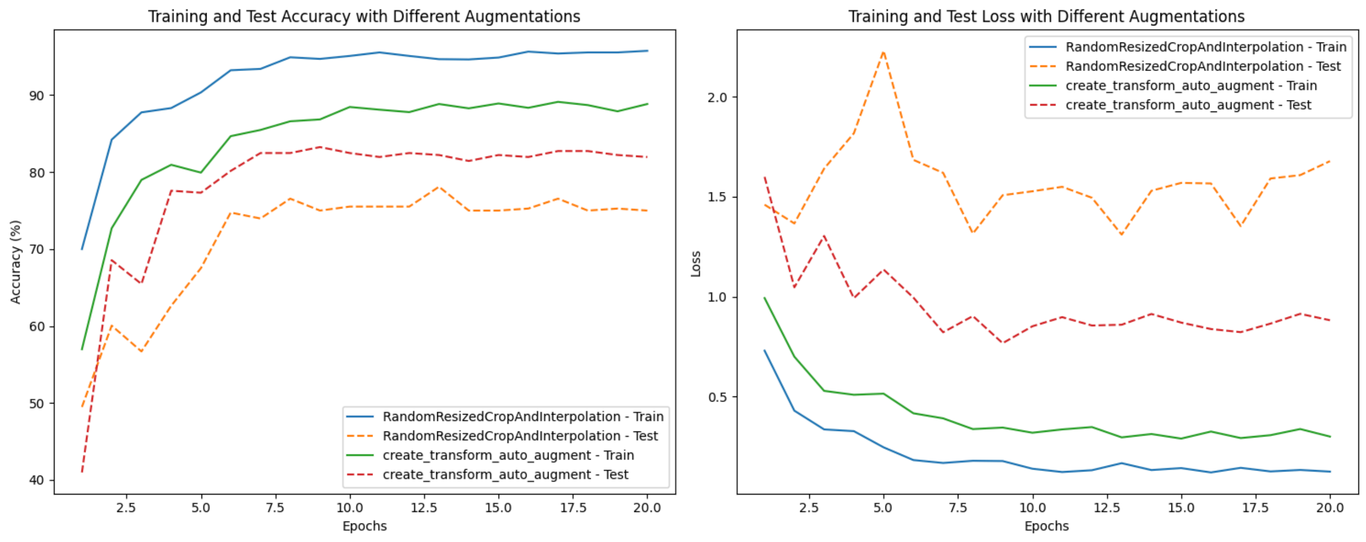


Figure 2: Accuracy and loss with different data augmentation techniques

7. Effects of Different Optimizers and Global Pooling

To further analyze model performance, different optimizers were explored, i.e switching from Adam to Stochastic Gradient Descent (SGD) and AdamW. The pooling strategy was changed from Average pooling to Max pooling and the model was changed to DenseNet121 to evaluate their effects.

SGD Optimizer with Max Pooling showed steady improvement, reaching a final test accuracy of 74.74% only which is a reduction from previous test accuracies. The AdamW optimizer resulted in better performance, with the test accuracy reaching almost 83%. The use of weight decay in AdamW likely contributed to better generalization compared to SGD, however the overall result also depends on the new model and pooling strategies used.

8. Effects of Different Schedulers

Since the AdamW optimizer performed well, it was retained for further experiments with the DenseNet121 model, and average pooling was used. Two new schedulers, CosineAnnealingLR and ExponentialLR, were explored as alternatives to the StepLR scheduler.

- CosineAnnealingLR Scheduler: The model trained with CosineAnnealingLR demonstrated smooth convergence and achieved a final test accuracy of 82.73%. This scheduler helps the model converge by gradually reducing the learning rate, allowing for more stable training and better generalization.
- ExponentialLR Scheduler: The model trained with ExponentialLR reached a final test accuracy of 79.90%. While it also provided good results, the convergence was slightly less.

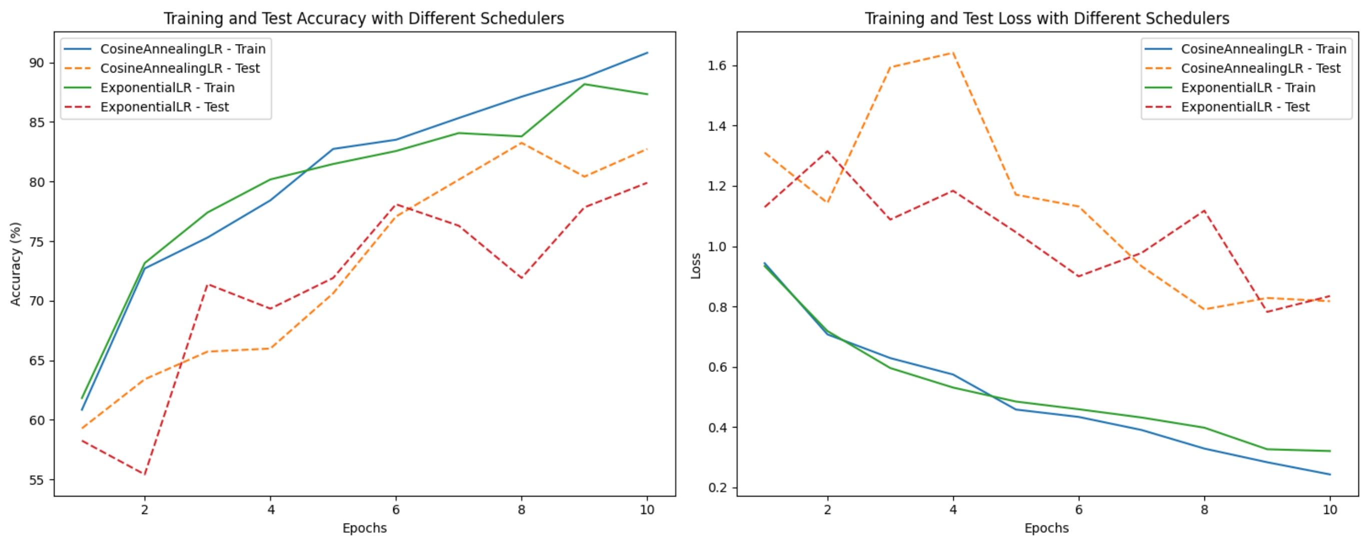


Figure 3: Accuracy and loss with different schedulers

9. Model Inference

For model inference, a saved model was loaded, exported to TorchScript, and evaluated for accuracy and latency. The accuracy was calculated using the evaluate_model function, which computes the average loss and accuracy over the test dataset. The latency was measured using the measure_latency function, which calculates the average latency of the model over multiple iterations, providing insight into the model's inference speed.

A model mapping was used to map model filenames to their respective creation functions. Although this mapping may seem irrelevant for my case as the 3 models I tested with are created from the same function, it is crucial when different models are created using distinct functions with varying parameters. If the initial ResNet50 (without weight decay) model were to be tested then the mapping would be useful as it would map to a different creation function which did not have dropout and other parameters initialized.

10. Model Inference Comparisons

The inference results were tested for 3 models out of all the models saved from above experiments.

Model 1: DenseNet121 with AdamW optimizer and CosineAnnealingLR scheduler

- Test Loss: 0.8175
- Test Accuracy: 82.73%
- Latency: 18.72 ms
- Model Size: 30.77 MB

Model 2: ResNet18 with Adam optimizer, StepLR scheduler and auto augment transformation

- Test Loss: 0.8819
- Test Accuracy: 81.96%
- Latency: 2.66 ms
- Model Size: 45.92 MB

Model 3: Resnet50 with Adam optimizer, basic data transform and weight decay of 1e-4

- Test Loss: 1.0107
- Test Accuracy: 79.12%
- Latency: 6.85 ms
- Model Size: 98.65 MB

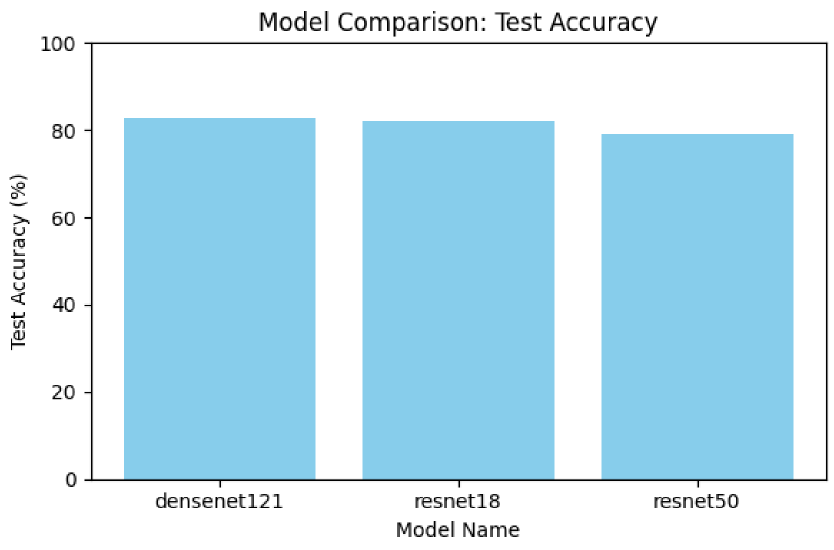


Figure 4: Model comparison for test accuracy

All three tested models have similar accuracy on the testing set. DenseNet121 performs the best as it stabilizes training which is helped by the use of AdamW with CosineAnnealingLR. ResNet18 with Auto Augment performed closely but slightly lower, showing that augmentation techniques can also significantly improve performance. ResNet50, despite being deeper and theoretically more capable, may have been overfitting due to its complexity and the use of basic data augmentation techniques only, which led to lower test accuracy.

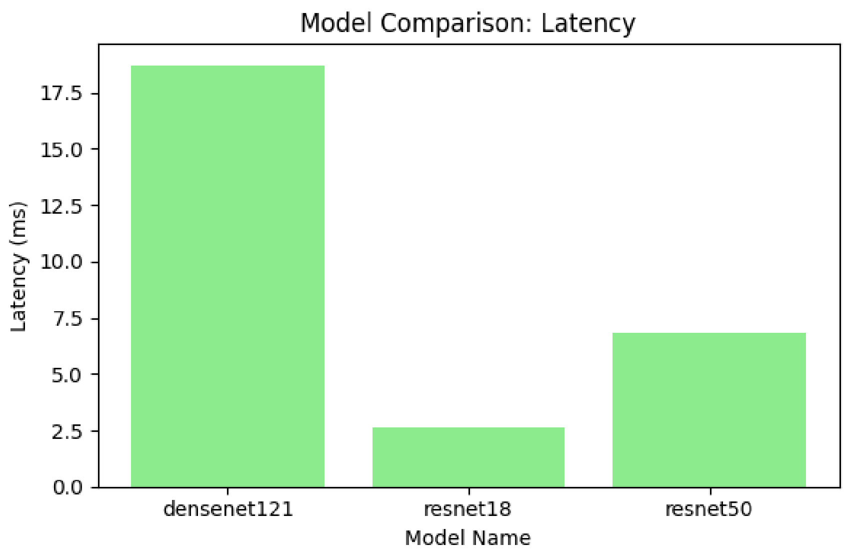


Figure 5: Model comparison for latency in ms

ResNet18's low latency is attributable to its relatively smaller model depth compared to DenseNet121 and ResNet50. This makes it faster for inference, which is advantageous for scenarios where prediction speed is critical. DenseNet121, although having higher accuracy, has increased latency, possibly due to the complex connections between layers. ResNet50, being deep and complex, has moderate latency, which still makes it usable but less ideal for time-sensitive tasks.

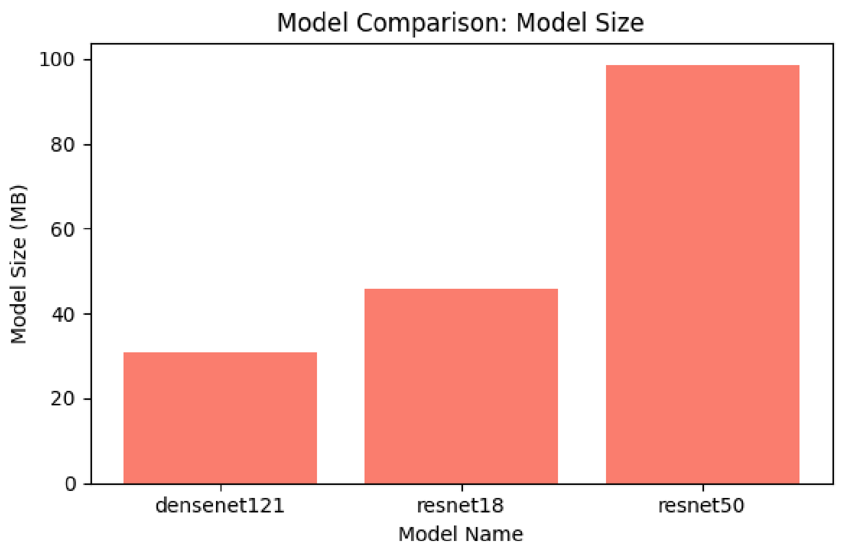


Figure 6: Model comparison for Model size in MB

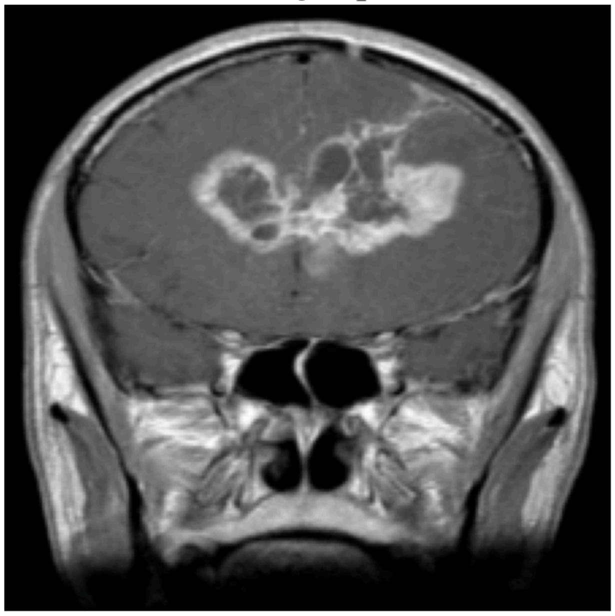
DenseNet121 had the smallest model size due to its efficient use of feature maps, making it suitable for environments with memory constraints. ResNet50's larger model size can be attributed to its deep and numerous convolutional layers. Though ResNet50 is generally known for its feature extraction capabilities, its large model size may make it impractical in edge or mobile scenarios compared to the more compact DenseNet121 or ResNet18.

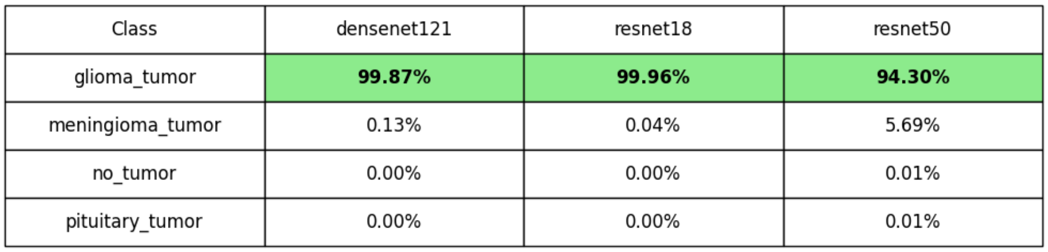
11. Testing with Random Image

To further evaluate model performance, a random image from the test dataset was selected and used for inference across the three models specified in the previous section. The aim is to analyze how well each model performed in terms of classification probabilities and predictions for a specific instance.

The results show that each of the models were able to classify the image by assigning the highest probability to the true label ‘glioma tumor’.

True Label: glioma_tumor





Class	densenet121	resnet18	resnet50
glioma_tumor	**99.87%**	**99.96%**	**94.30%**
meningioma_tumor	0.13%	0.04%	5.69%
no_tumor	0.00%	0.00%	0.01%
pituitary_tumor	0.00%	0.00%	0.01%

Predicted Probabilities (%) for Each Class across different Models