Project 2

MSDA3100 : Applied Deep Learning

Session: Jan 2025 – May 2025

Professor – Ahmed Elsayed

Submitted by Group7

Nithin Rachakonda - C70313368

Sai Nithisha Marripelly – C70313009

Gautam Mehta – C70304767

On

1st May 2025

School of Professional Studies
Clark University

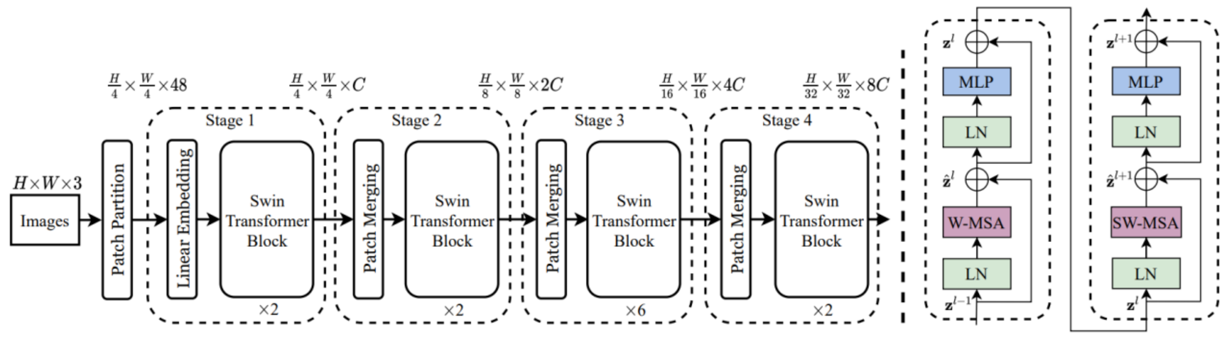
Task 1: Understanding The Swin-Transformer Paper

Vision Transformer (ViT) splits images into fixed-size patches, all processed at once. It uses global self-attention on all patches, which becomes computationally expensive for high-resolution images. While ViT captures global features effectively, it is limited in capturing local features.

These challenges are addressed by the Swin Transformer, which can capture features at different scales due to its windowed computation. Swin Transformer also splits the image into patches but processes them locally in non-overlapping windows. Self-attention is calculated within each window, allowing it to capture local features. In the next step, the shifted window mechanism enables the model to capture global features by allowing cross-window connections. Swin's architecture is hierarchical, working similarly to a Convolutional Neural Network by using patch merging at different stages. Importantly, Swin also reduces computational complexity from quadratic to linear, making it more efficient for large-scale vision tasks

Swin Transformer

The Swin Transformer takes an image as input. In the next step, the image is divided into non-overlapping patches. These patches are then flattened and linearly embedded into vectors. This is followed by a stage-wise hierarchical process consisting of four stages. Each stage includes Window-based Multi-Head Self-Attention (W-MSA) and Shifted Window Multi-Head Self-Attention (SW-MSA), along with a patch merging layer between stages to reduce resolution and increase feature dimension.



In each stage, the input patch passes through W-MSA, SW-MSA, Layer Normalization, an MLP (feed-forward network), and a patch merging layer, where the patch size is halved, and the number of channels is doubled. Stage 1 is followed by three more stages with a similar flow. These four stages are followed by an output layer where pooling is performed for the respective vision tasks.

In each Window-based Multi-Head Self-Attention block, we have a non-overlapping window partitioner, multi-head self-attention, Layer Normalization, and an MLP with a residual connection. The input is divided into fixed-size windows, and self-attention is calculated independently within each window to capture local patterns, followed by normalization, feed-forward layers, and residuals.

In Shifted Window MSA, we have a window shifter (which shifts by half the window size), multi-head self-attention, a masking mechanism to avoid information leakage, Layer Normalization, and an MLP with a residual connection. Windows are shifted to enable cross-window connections, and self-attention is computed on these new overlapping windows. Masking ensures correctness, and the output is passed through MLP and residual connections.

Differences in Implementation (Original vs Berniwal)

- **Parameters Flexibility**: In the original Swin transformer code, the models are predefined. In Berniwal's implementation although there are predefined variants, it also allows custom configurations of layers, embedding dimensions, number of heads, etc.
- **Attention masking**: The original Swin uses -100 to block attention, while Berniwal's version uses -inf for a stricter and simpler way to prevent attention across shifts.
- **Stages**: The official Swin Transformer handles W-MSA and SW-MSA alternation automatically within modular blocks, while Berniwal's version simplifies this for clarity, often requiring manual alternation and customization.

Overall, we can say that Berniwal's lightweight and customizable design makes it more efficient and easier to adapt for small datasets like CIFAR-10. Lets look at the implementation of the Berniwal's Swin-transformer along with our own customized parameters.

Note: Better explanation of the paper has been done in the presentation with animations.

Task 2: Implementation of the code

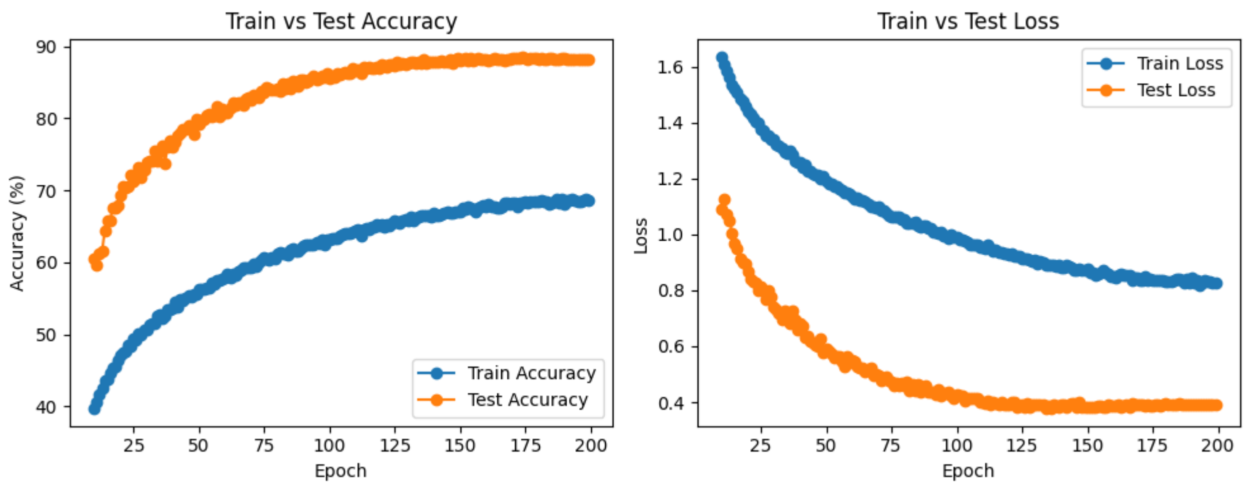
For the implementation, we have initially used the same parameters as the paper used throughout the code. Later, we changed the parameters to compare the performance.

We were provided with 3 python files: Utils.py, randomaug.py and swin_main.py. The implementation of the Swin Transformer is done in the swin_main.py file which calls the Utils.py file for the data loading process along with the augmentation of the CIFAR 10 dataset. Utils.py file calls the randomaug.py file to execute the augmentations for the dataset. The randomaug.py file defines a wide range of augmentations like shear, translations, rotations, etc. The augmentations are defined using 2 parameters i.e., number of augmentations (N) and magnitude of the augmentation (M).

1) Results of the model provided

The swin_main.py was converted to a python notebook .ipynb file and was run with the originally defined parameters. The notebook was run locally, and the remaining 2 files were stored in the same local folder. The model with the following predefined parameters:

- Downscaling factor = (2,2,2,1)
- Optimizer = Adam
- Augmentation enabled (N,M = 2,14)
- Epochs = 200

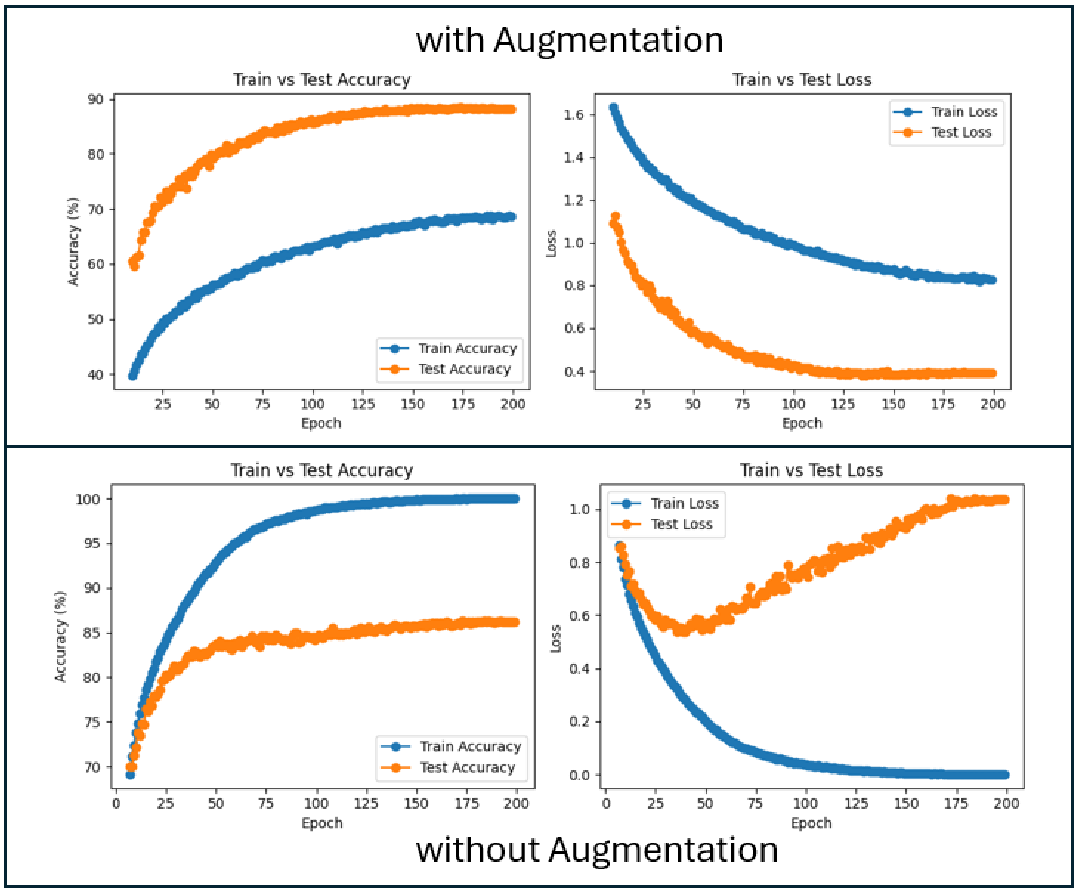


There is a huge gap between the train and test accuracies. This can be explained with the robust augmentations (Magnitude of the augmentations was set to 14) that were being given to train the model. Overall, our base model shows better generalization, as indicated by higher test accuracy. The test loss and validation loss also complement the generalizability of the model.

2) Comparison of models with and without augmentation

Here, a new model has been defined without augmentation and has been compared to our base model. The parameters defined for this model are:

- Downscaling factor = (2,2,2,1)
- Optimizer = Adam
- Augmentation disabled
- Epochs = 200

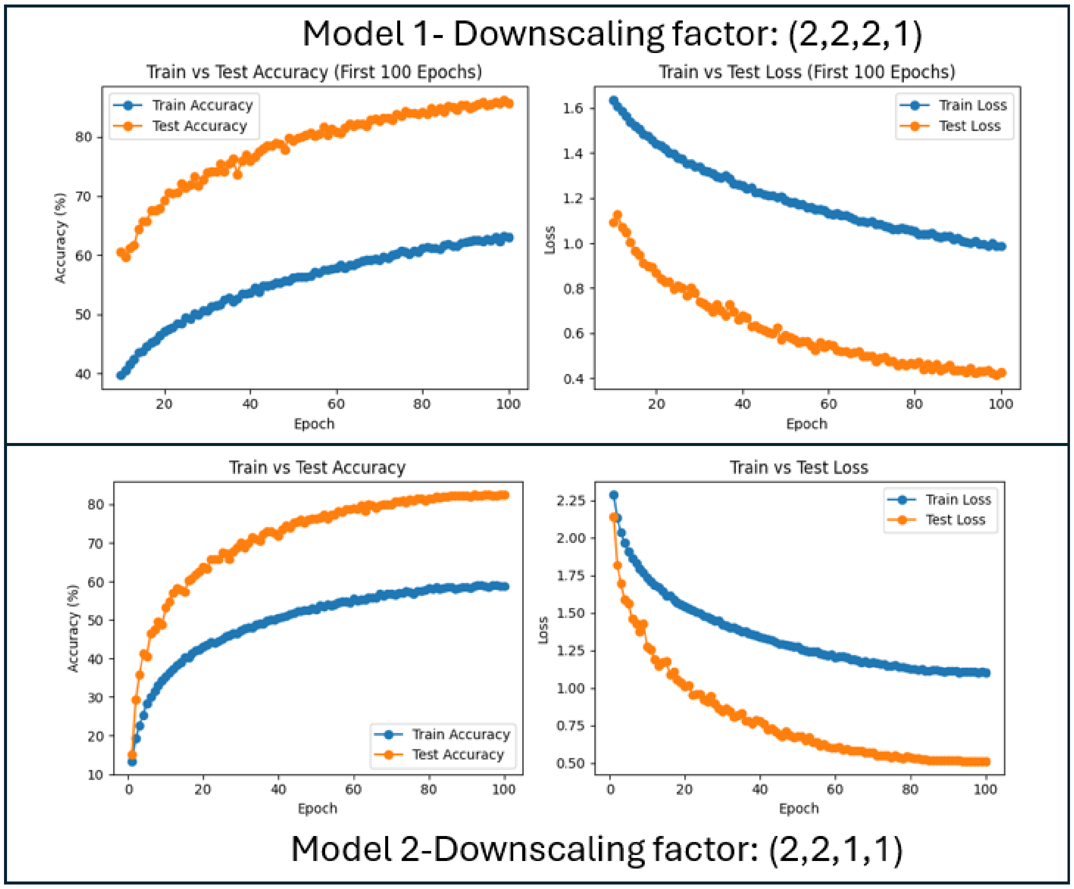


With augmentation, the model generalizes better and achieves higher test accuracy. Without augmentation, the model overfits. Augmentation also helps in reducing the gap between train and test performance as visualized. We can conclude that, without augmentation, the model memorizes training data but performs poorly on unseen data.

3) Comparison of models with different downscaling factors

Here, a new model has been defined with different downscaling factors (2,2,1,1) and has been compared to our base model. The parameters defined for this model are:

- Downscaling factor = (2,2,1,1)
- Optimizer = Adam
- Augmentation enabled (N,M) = (2,14)
- Epochs = 100

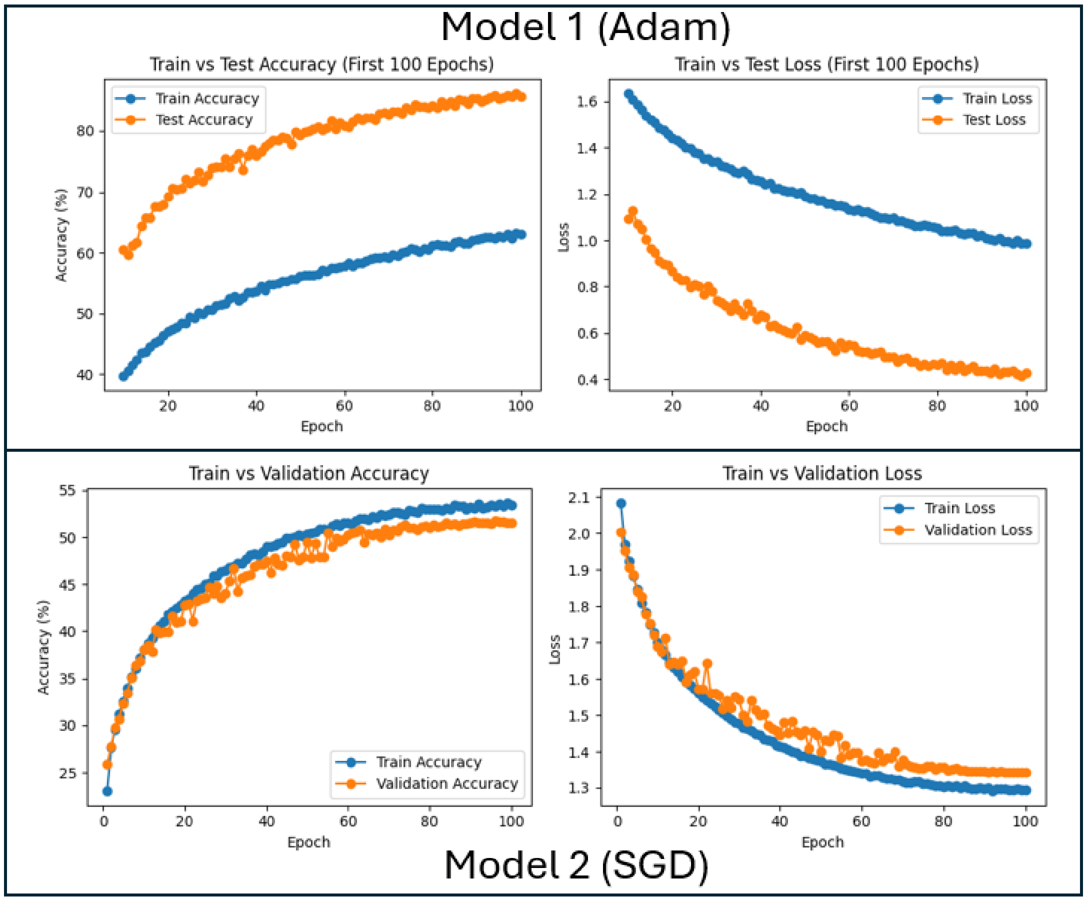


Let's call our base model as model 1 and the model with downscaling factors (2,2,1,1) as model 2. Model 1 achieves higher test accuracy, indicating better learning capacity. Model 2 converges faster but reaches lower accuracy, suggesting limited learning. More aggressive downscaling in model 1 allows capturing global features effectively. Overall, it has been observed that progressive downscaling improves model performance by efficient hierarchical feature learning.

4) Comparison of models with different Optimizers

Here, a new model has been defined with different optimizer SGD and has been compared to our base model. The parameters defined for this model are:

- Downscaling factor = (2,2,2,1)
- Optimizer = SGD
- Augmentation disabled
- Epochs = 100



Let's call our base model as model 1 and the model with SGD optimizer as model 2. Model 1 achieves significantly higher test accuracy than model 2, indicating better generalization. Model 2 has slower learning and lower accuracy, suggesting possible underfitting. Loss curves in model 2 converge earlier, while Model 1 continues to improve steadily. We can conclude that in the Adam optimizer helps the model learn better and faster than SGD in this case.

4) Comparison of all the models after 100 epochs

Below is a table that summarizes all the results from all the models trained.

S.No	Optimizer	Downscaling factor	Augmentation	Train_Acc	Test_Acc	Train_Loss	Test_Loss
1	Adam	(2,2,2,1)	Yes	63.09	85.64	0.99	0.42
2	Adam	(2,2,2,1)	No	98.64	84.5	0.03	0.76
3	Adam	(2,2,1,1)	Yes	58.7	82.37	1.1	0.5
4	SGD	(2,2,2,1)	No	53.37	51.15	1.29	1.34

- **Model 1** showed the highest test accuracy of 85.64% among the four models. This indicates good generalization to unseen data, even though the training accuracy was lower. This could be due to the use of data augmentation.
- **Model 2** achieved the highest training accuracy of 98.64% but a lower test accuracy of 84.5% and a higher test loss of 0.76. This shows signs of overfitting, possibly due to the absence of data augmentation.
- **Model 3** had a lower overall performance, with 58.7% training accuracy and 82.37% test accuracy. The model appears to be underfitting, indicating limited learnability with the current configuration.
- **Model 4** showed the poorest performance, with 53.37% training accuracy and 51.15% test accuracy. This could be attributed to the use of the SGD optimizer and the lack of data augmentation.

Conclusion: Robust data augmentation using randomaug improved the model's generalization ability, as observed in the results. The Adam optimizer outperforms SGD when used with Berniwal's Swin Transformer implementation on the CIFAR-10 dataset.

Appendix (Test Samples)

Test samples were randomly selected for the base model, and the predicted classes are shown below.