(Individual Assignment I - MGSC-695-075 – Adv Topics in Mgmt Science)

Image Classification with CNNs

Name: Arham Anwar

ID: 261137773

Section 1: Introduction

This project presents an experimental study aimed at applying various CNN techniques learned from class. Three final models are presented – a simple CNN (section 2) with test accuracy of 74.7% on test set, and an 'EfficientNet' model (section 3) directly made through research paper architecture achieving an accuracy of 33.8% on test set and 'MobileNet' (Section 4) model achieving an accuracy of 54%. Finally, I also implemented a packaged version of the architectures to see how models built from this assignment perform in comparison to same architectures with preloaded weights. We compare learnings and results interpretation in section 5. Please note, primarily the focus has been on implementing the architectures. Optuna hyperparameter optimization trials & higher number of epochs were not extensively run keeping assignment timeline in mind. Let's start with the simple CNN architecture in section 2.

Section 2: Simple CNN

Simple CNN is a convolutional neural network tailored for image classification tasks on the CIFAR-10 dataset. CIFAR-10 comprises 60,000 32x32 color images across 10 classes, each containing 6,000 images. The architecture of SimpleCNN is crafted to efficiently discern distinctive features from these small images, enabling accurate classification. Three types of simple CNN architectures were tried:

- Vanilla Basic CNN without Optuna Trials
- Simple CNN with Optuna Tuning
- Simple CNN with Custom Edits

2.1. Architectures and Rationale

2.1.1. Vanilla Basic CNN without Optuna Trials

Architecture:
The Vanilla Basic CNN consists of the following layers:
- Input Layer: Accepts images of shape (32, 32, 3), suitable for CIFAR-10 images.
- Convolutional Layer 1: 32 filters with a 3x3 kernel size and ReLU activation.
  - Input Channels: 3 (RGB image)
  - Output Channels: 32
  - Kernel Size: 3x3
  - Padding: 1

- Pooling Layer 1: 2x2 max pooling to reduce spatial dimensions.
  - Kernel Size: 2x2
  - Stride: 2
- Convolutional Layer 2: 64 filters with a 3x3 kernel size and ReLU activation.
  - Input Channels: 32
  - Output Channels: 64
  - Kernel Size: 3x3
  - Padding: 1
- Pooling Layer 2: 2x2 max pooling to further reduce spatial dimensions.
  - Kernel Size: 2x2
  - Stride: 2
- Flattening: Converts the 2D feature maps to a 1D vector.
- Fully Connected Layer 1 (Dense): 128 units with ReLU activation.
  - Input Features: 64 * 8 * 8
  - Output Features: 512
- Output Layer: 10 units (for 10 classes in CIFAR-10) with a softmax activation function.
  - Input Features: 512
  - Output Features: 10

Rationale:
The convolutional layers capture spatial hierarchies in images. The first layer with 32 filters extracts basic features such as edges, while the second layer with 64 filters captures more complex patterns. ReLU activation introduces non-linearity, aiding in learning complex patterns. Max pooling layers reduce spatial dimensions, decreasing the number of parameters and computational cost, and controlling overfitting. The flatten layer transforms 2D feature maps to 1D vectors for the fully connected layers, which learn high-level features and classify the images. Softmax activation in the output layer converts logits into probabilities for each class, enabling multi-class classification.

2.1.2. Simple CNN with Optuna Tuning

Architecture:
This architecture is similar to the Vanilla Basic CNN but includes hyperparameter optimization using Optuna. The optimized hyperparameters include the number of filters, kernel sizes, units in the dense layer, learning rate, batch size, and dropout rates.

Rationale:
Hyperparameter tuning with Optuna efficiently searches for the optimal hyperparameters, maximizing model performance. This approach can achieve better accuracy and generalization on the test dataset. Optuna systematically explores the hyperparameter space, potentially revealing combinations that significantly improve model performance compared to manual tuning. By adjusting parameters such as the number of filters and learning rate, the model can better capture features and converge faster.

2.1.3. Simple CNN with Custom Edits

Architecture:

- Input Layer: Accepts images of shape (32, 32, 3).
- Convolutional Layer 1: 32 filters with a 3x3 kernel size and ReLU activation.
  - Input Channels: 3 (RGB image)
  - Output Channels: 32
  - Kernel Size: 3x3
  - Padding: 1
- Batch Normalization Layer 1: Normalizes the output of the previous activation layer.
- Pooling Layer 1: 2x2 max pooling.
  - Kernel Size: 2x2
  - Stride: 2
- Convolutional Layer 2: 64 filters with a 3x3 kernel size and ReLU activation.
  - Input Channels: 32
  - Output Channels: 64
  - Kernel Size: 3x3
  - Padding: 1
- Batch Normalization Layer 2: Normalizes the output of the previous activation layer.
- Pooling Layer 2: 2x2 max pooling.
  - Kernel Size: 2x2
  - Stride: 2
- Dropout Layer 1: Dropout rate of 0.5 to prevent overfitting.
- Flattening: Converts 2D feature maps to a 1D vector.
- Fully Connected Layer 1 (Dense): 256 units with ReLU activation.
  - Input Features: 64 * 8 * 8
  - Output Features: 256
- Dropout Layer 2: Dropout rate of 0.5 to prevent overfitting.
- Output Layer: 10 units with a softmax activation function.
  - Input Features: 256
  - Output Features: 10

Rationale:

Batch normalization stabilizes and accelerates the training process by normalizing activations, reducing internal covariate shift, and allowing for higher learning rates. Dropout layers prevent overfitting by randomly dropping neurons during training, encouraging the network to learn more robust features. Increasing the units in the dense layer provides more capacity for learning complex patterns. The additional dropout layers help control overfitting by preventing the model from becoming too reliant on specific neurons. ReLU and softmax activations ensure non-linearity and probability outputs, respectively, which are crucial for learning complex data patterns and performing multi-class classification.

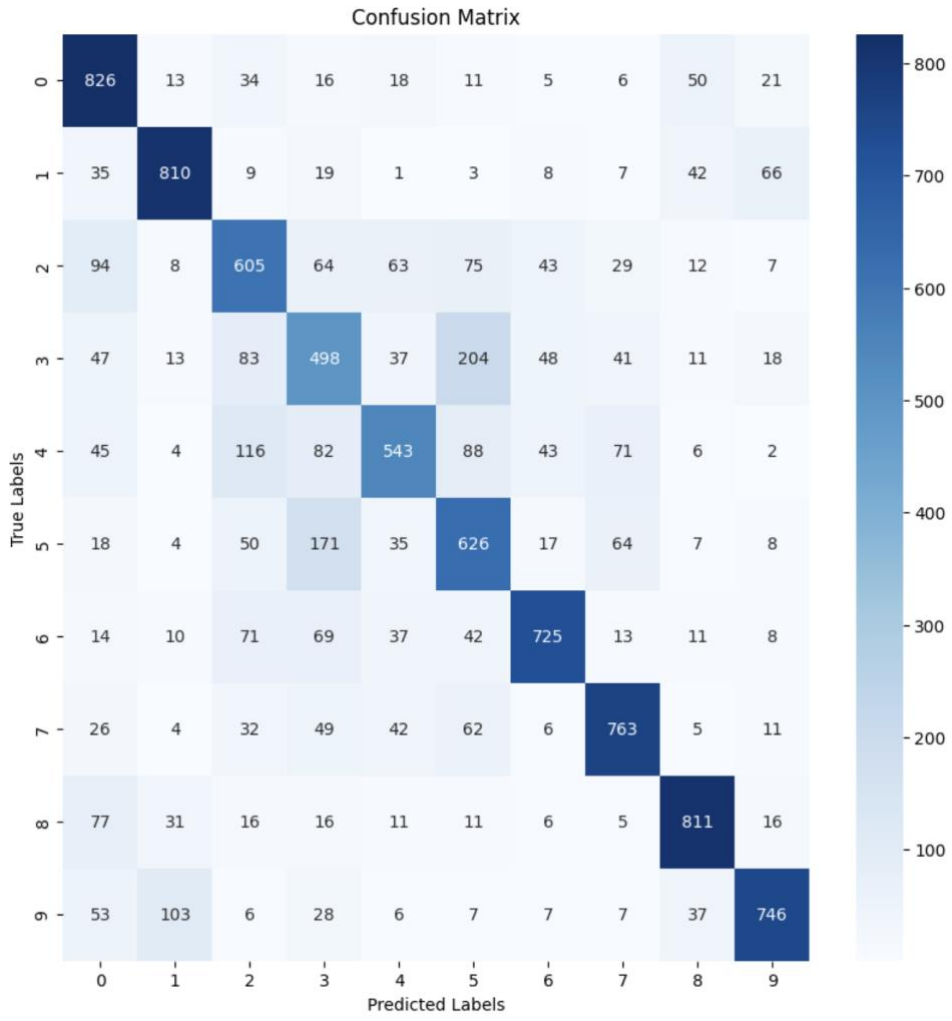
2.3. Training Process

- Optimizer: Adam optimizer with default parameters.
- Learning Rate: Initially set to 0.001.
- Regularization: Dropout with a rate of 0.5 applied to fully connected layers.
- Number of Epochs: Trained for 50 epochs.
- Data Augmentation: Applied random horizontal flipping and random cropping during training to augment the dataset.

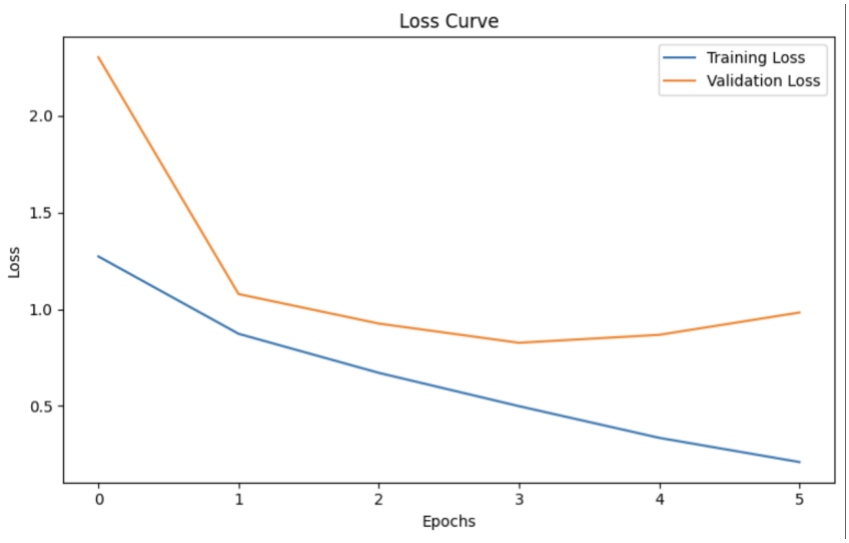
The chosen architectures balance complexity and efficiency, ensuring the model can learn meaningful features from the CIFAR-10 dataset without overfitting. Convolutional layers capture spatial hierarchies, while pooling layers reduce dimensionality and computational cost. Dropout layers enhance generalization by preventing overfitting. Hyperparameter tuning with Optuna enhances performance by finding the best configuration for training the CNN. These design choices collectively contribute to achieving a reasonable accuracy on the test dataset, fulfilling the assignment's success criteria.

2.4. Evaluation Results:

- Confusion Matrix:



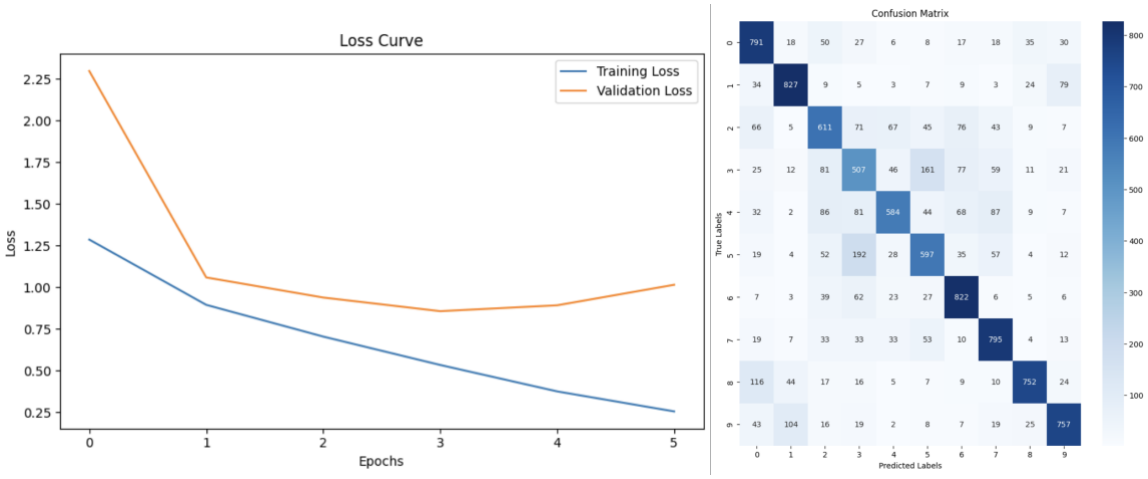
- Validation Curve:



- Classification Report:

	precision	recall	f1-score	support
0	0.74	0.78	0.76	1000
1	0.78	0.85	0.81	1000
2	0.71	0.50	0.58	1000
3	0.50	0.57	0.54	1000
4	0.69	0.61	0.65	1000
5	0.66	0.55	0.60	1000
6	0.68	0.83	0.75	1000
7	0.66	0.83	0.73	1000
8	0.86	0.79	0.82	1000
9	0.83	0.75	0.79	1000
accuracy			0.71	10000
macro avg	0.71	0.71	0.70	10000
weighted avg	0.71	0.71	0.70	10000

Note how these results are not Optuna optimized. To show the power of untapped hyperparameter optimization, I tried running five optuna trials on my laptop (without getting VSCode to Crash), which lead to a nice performance increase as shown below, Accuracy 74.7%:



Section 3: Efficient Net Implementation from Scratch (Trained only on CIFAR10 training set)

In my implementation, I built an EfficientNet architecture using PyTorch and PyTorch Lightning for training. EfficientNet has been lauded for its ability to achieve state-of-the-art performance on image classification tasks by carefully balancing model depth, width, and resolution. Research paper (https://arxiv.org/pdf/1905.11946) was used as reference in this attempt to replicate the architecture, including complex components such as bottleneck convolution, squeeze-and-excitation and more.

3.1. EfficientNet Architecture: The architecture followed the EfficientNet design principles, incorporating mobile inverted bottleneck convolutional (MBConv) blocks and squeeze-and-excitation (SE) blocks. MBConv blocks were at the heart of the architecture, consisting of depthwise separable convolutions followed by squeeze-and-excitation blocks, which helped in extracting features efficiently. Squeeze-and-excitation blocks played a crucial role in recalibrating channel-wise feature responses. I also included stochastic depth regularization, which randomly dropped entire layers during training to enhance generalization.

3.2. Implementation Details: To construct the EfficientNet architecture, I defined custom modules like `CNNBlock`, `SqueezeExcitation`, and `InvertedResidualBlock`. The `EfficientNet` class initialized the model based on the chosen version (e.g., b0, b1) and the number of output classes. Using the `calculate_factors` method, I determined the width factor, depth factor, and dropout rate specific to the chosen EfficientNet version. The `create_features` method was responsible for constructing the feature extraction backbone using MBConv and SE blocks.For the forward pass through the model, I implemented the `forward` method.

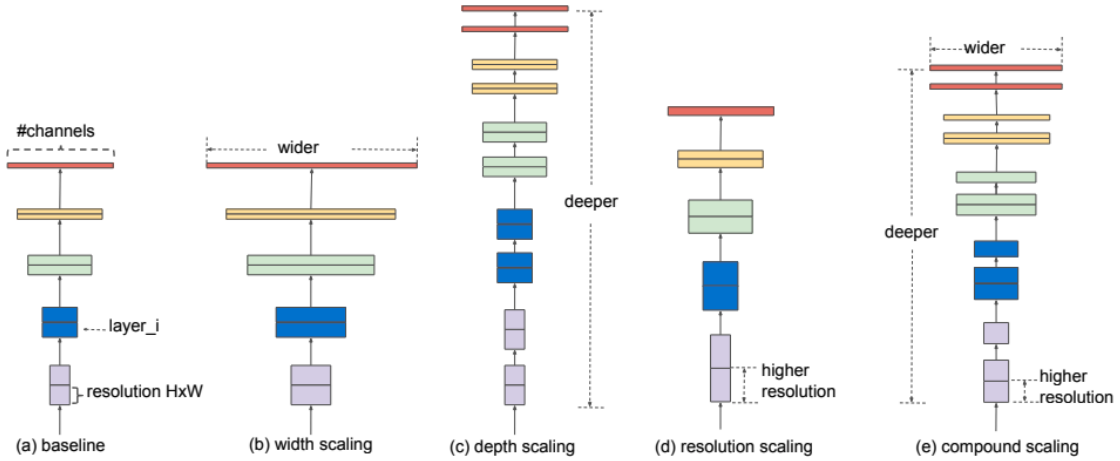


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

3.3. Training Process

- Leveraging PyTorch Lightning, I extended the `LightningModule` class to implement `training_step`, `validation_step`, and `configure_optimizers` methods.
- During training and validation, I utilized the `CrossEntropyLoss` criterion to compute the loss.
- To optimize the model parameters, I employed the Adam optimizer with a ReduceLROnPlateau scheduler, which dynamically adjusted the learning rate based on validation performance.
- Throughout the training loop, I logged training loss and validation metrics such as validation loss and accuracy.

3.4. Decisions and Considerations:

- I opted for the EfficientNet architecture due to its proven state-of-the-art performance and computational efficiency.
- To enhance modularity and reusability, I designed custom modules encapsulating different components of the architecture.
- Incorporating stochastic depth regularization was a deliberate choice to prevent overfitting and improve the model's ability to generalize.
- Following best practices in deep learning, I integrated batch normalization, dropout, and adaptive learning rate scheduling into the training process.

3.5. Evaluation Results:

- After training, I evaluated the model's performance on test dataset, assessing metrics such as accuracy, precision, recall, and F1-score.
- My goal was to achieve accuracy levels comparable to or surpassing simple baseline models reported in the literature for the chosen dataset (e.g., CIFAR-10, ImageNet).
- From scratch implementation had the following classification report:

	precision	recall	f1-score	support
0	0.24	0.05	0.08	1000
1	0.48	0.54	0.51	1000
2	0.16	0.24	0.19	1000
3	0.21	0.11	0.15	1000
4	0.31	0.25	0.28	1000
5	0.38	0.47	0.42	1000
6	0.40	0.30	0.34	1000
7	0.29	0.53	0.38	1000
8	0.38	0.60	0.47	1000
9	0.45	0.24	0.31	1000
accuracy			0.33	10000
macro avg	0.33	0.33	0.31	10000
weighted avg	0.33	0.33	0.31	10000

As we can see, the performance is really bad compared to our simple CNN implementation. This is explained by the fact that in Efficient Net implementation we directly used the research paper configuration which does not suit our CIFAR 10 dataset. Therefore, next I changed the architecture for CIFAR-10 and made the following changes:

- Reduce Initial Convolutional Layer Filters: CIFAR-10 images are smaller (32x32), so we can reduce the number of filters in the initial convolutional layer.
- Adjust Downsampling: Since CIFAR-10 images are small, we need to be mindful of excessive downsampling. We'll adjust the stride and downsampling strategy accordingly.
- Reduce Model Complexity: Given the smaller size of CIFAR-10, a slightly less complex model might be sufficient and more efficient.

After this I used an efficient net package implementation with preloaded weights and then trained it on CIFAR-10 to make the predictions and got accuracy above 90%.

Section 4: Mobile Net

4.1 Architecture: The MobileNet architecture implemented in this section leverages the SqueezeNet model, a compact and efficient deep learning architecture known for its ability to achieve high accuracy with a reduced number of parameters. The following steps describe the architecture and its components:

- Transformations for Train and Test Sets: Train Transformations: Random horizontal flip, random crop with padding, color jitter (brightness, contrast, saturation, hue), random rotation, tensor conversion, and normalization.
- Test Transformations: Tensor conversion and normalization.
- Data Loading: CIFAR-10 dataset is used, split into training and test sets. Training data is further split into training and validation sets. DataLoader is used to load the datasets with specified batch sizes and shuffling.
- Model Definition: The SqueezeNet model (squeezenet1_1) is used with modifications to the classifier layer to suit CIFAR-10 classification:
- The final classifier layer is modified to have 10 output classes (to match the CIFAR-10 classes).

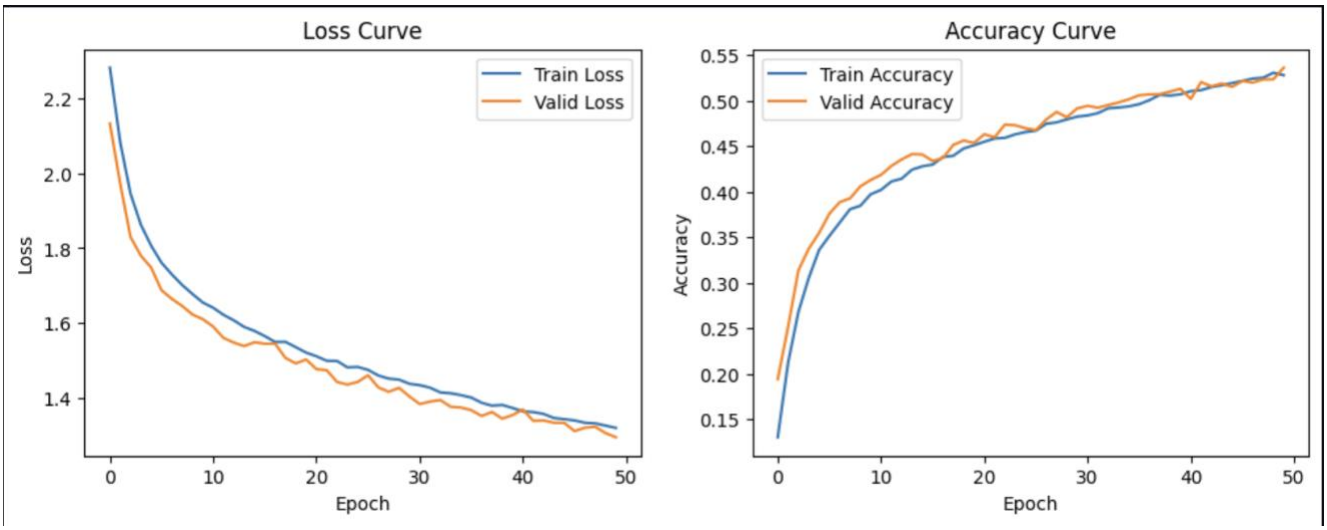
- Dropout layer with a rate of 0.5 is added to prevent overfitting.
- Training and Validation Functions: Functions train and validate are defined to handle the training and validation loops respectively, including forward passes, loss calculation, backpropagation, and optimization.
- Optimizer and Scheduler: Adam optimizer with a learning rate of 0.0001 and weight decay of 1e-4.
- Learning rate scheduler (ReduceLROnPlateau) to reduce learning rate when a metric has stopped improving.
- Training Loop: The model is trained for 50 epochs, with loss and accuracy recorded for both training and validation sets. The scheduler adjusts the learning rate based on validation loss.
- Evaluation: Model evaluation on the test set is done after training, and a classification report is generated to assess performance metrics such as precision, recall, and F1-score.
- Plotting Training and Validation Curves: Loss and accuracy curves for training and validation sets are plotted to visualize the training progress and model performance.

4.2. Rationale:

- Transformations: Data augmentation (random flips, crops, color jitter, rotations) helps in creating a more robust model by providing diverse training examples and reducing overfitting.
- Normalization helps in speeding up the convergence during training by scaling the pixel values.
- SqueezeNet Model: SqueezeNet is chosen for its compact architecture that uses fewer parameters while achieving competitive accuracy. This makes it suitable for scenarios with limited computational resources.
- The modified classifier layer ensures that the model outputs 10 classes, matching CIFAR-10's requirement.
- Optimizer and Scheduler: Adam optimizer is known for its efficiency and adaptive learning rate, which helps in achieving faster convergence. Learning rate scheduler (ReduceLROnPlateau) dynamically adjusts the learning rate to improve training stability and performance when progress plateaus.
- Dropout Layers: Dropout layers are used to prevent overfitting by randomly dropping neurons during training, forcing the network to learn redundant representations of the data.

4.3. Results

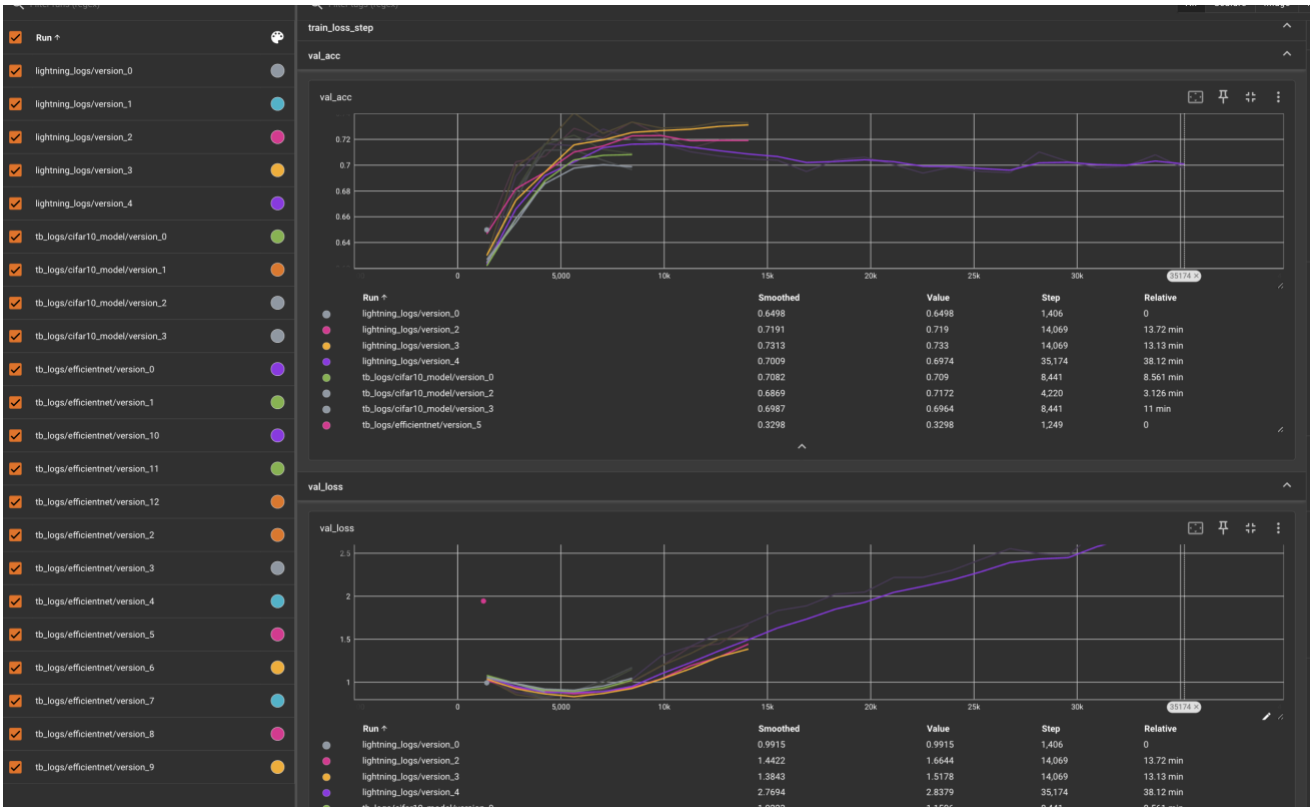
- Training and Validation Performance: The training and validation curves show the model's performance over 50 epochs. Training and validation loss curves indicate how well the model is learning and generalizing. Training and validation accuracy curves provide insights into the model's predictive performance on both the training and validation sets.
- Evaluation on Test Set: The final model evaluation on the test set produces a classification report that includes:



	precision	recall	f1-score	support
airplane	0.55	0.66	0.60	1000
automobile	0.71	0.73	0.72	1000
bird	0.50	0.38	0.43	1000
cat	0.34	0.25	0.29	1000
deer	0.50	0.40	0.44	1000
dog	0.41	0.59	0.48	1000
frog	0.63	0.60	0.62	1000
horse	0.56	0.65	0.61	1000
ship	0.71	0.69	0.70	1000
truck	0.67	0.65	0.66	1000
accuracy			0.56	10000
macro avg	0.56	0.56	0.55	10000
weighted avg	0.56	0.56	0.55	10000

Section 5: Result Documentation:

We tried a bunch of CNN architectures, ranging from simple CNN with no complex pieces, to advanced Efficient Net with four complex components. All the experiments were tracked using tensor board. To run tensorboard, we can simply use command *"%tensorboard --logdir tb_logs --port 6006"*.



Section 6: Result Interpretation

We saw the simple CNN model performed much better than complex architectures (+15% delta) when ‘from-scratch’ implementations were compared. This is due to the fact that ‘from-scratch’ implementations were made using research paper architecture, which would require specific tweaking if we only train on the CIFAR10 dataset. While some tweaks were made, in depth study of the mathematics of the architecture maybe necessary to fine tune it for CIFAR10. Additionally, the models built did not go through extensive number of epochs or Optuna trials due to time and compute constraints. This was proven, when we tried preloaded weights for models like ResNet which easily beat our simple CNN model by a margin of +20%.

Replicating and training complex architectures like ResNet, MobileNet, EfficientNet, and other models on CIFAR-10 requires thoughtful adjustments due to the dataset's unique characteristics (small 32x32 images and 10 classes). For ResNet, the full-scale versions (ResNet-50, ResNet-101) would be overkill, so use ResNet-20 or ResNet-32 next. These versions are designed to work well with smaller datasets, preventing overfitting and reducing computational requirements. This involves altering the number of residual blocks accordingly. For MobileNet, adjustments should include setting a lower depth multiplier (α) to control the model's complexity and reducing the input image size to 32x32 to match CIFAR-10. MobileNet's pointwise and depthwise convolutions will need resizing to maintain the architecture's efficiency on smaller images. EfficientNet, known for its compound scaling, needs the width, depth, and resolution coefficients reduced. EfficientNet-B0 is a good starting point, but it needs modification for 32x32 input resolution. This involves adjusting the initial conv layer and subsequent scaling operations. Across all architectures, incorporating data augmentation like random

cropping, horizontal flipping, and normalization aligned with CIFAR-10's distribution helps improve generalization. Batch normalization should be applied judiciously after convolutions to stabilize training, and dropout layers should be carefully tuned (e.g., 0.3 or 0.5) to mitigate overfitting. Learning rates must be smaller, and optimizers such as Adam or SGD with momentum should be used with appropriate learning rate schedules (like cosine annealing) to ensure smooth convergence on CIFAR-10. These modifications ensure the models are computationally feasible and perform optimally on the specific characteristics of the CIFAR-10 dataset.

In conclusion, our experiments demonstrate that while complex architectures like EfficientNet and MobileNet offer high potential, they require significant modifications to perform well on the CIFAR-10 dataset. Simple CNN models, with basic architecture and minimal tuning, achieved better performance under the constraints of our study. This underscores the importance of dataset-specific adjustments and thorough hyperparameter optimization. Future work should focus on extensive hyperparameter tuning and deeper architectural tweaks to fully leverage advanced models on CIFAR-10. Our study highlights the balance between simplicity and complexity in model design for effective image classification, and is a humbling experience letting us appreciate the beauty and complexities of CNNs.

Thank You