MobileNetsV2 Experiments

[Introduction 3](#_Toc162933593)

[Baseline Model 3](#_Toc162933594)

[Implementation 3](#_Toc162933595)

[Experimental Setup 5](#_Toc162933596)

[Hyperparameter Tuning 6](#_Toc162933597)

[Architectural Modifications 9](#_Toc162933598)

[Architectural Modifications – part 1 9](#_Toc162933599)

[Architectural Modifications – part 2 16](#_Toc162933600)

[Reflections and Learning 21](#_Toc162933601)

[Challenges and Lessons: 21](#_Toc162933602)

[Insights: 21](#_Toc162933603)

[Further Improvements 21](#_Toc162933604)

[ChatGPT Links 22](#_Toc162933605)

# Introduction

In the experiments conducted for this project, we aimed to explore various configurations of the MobileNetV2 architecture by implementing and evaluating its performance on the CIFAR-10 dataset for a classification task. The primary goals were to understand how the architecture responds to different settings and to identify potential areas for improvement. Our approach began with hyperparameter tuning to establish a baseline performance, followed by a series of experiments involving multiple architectural modifications.

# Baseline Model

While the original MobileNetV2 model was trained on the ImageNet dataset, we chose to use the CIFAR-10 dataset for our experiments due to its manageable size, which allowed us to explore a broader range of changes within a feasible timeframe. We adopted the architecture as described in the original MobileNetV2 paper. However, the paper did not explicitly detail certain aspects such as hyperparameters, optimization techniques, and data transformations for CIFAR-10. Therefore, we initialized our baseline model using the architecture from the paper with a set of hyperparameters and configurations determined through our experimentation process.

## Implementation

Relevant code file: Implementation.ipynb

The code defines an implementation of the MobileNetV2 architecture and trains it on the CIFAR-10 dataset.

**1.Imports:**

The code imports essential libraries including PyTorch for building and training neural networks, NumPy for numerical computations, Matplotlib for visualization, and torchvision for dataset handling.

**2. Model Architecture:**

* ConvBNActivation: Defines a block consisting of a convolutional layer followed by batch normalization and ReLU6 activation.
* InvertedResidualBlock: Implements an inverted residual block, a key component of MobileNetV2 architecture.
* CustomMobileNetV2: Consists of a total of 21 layers, including initial convolutional layers, 17 inverted residual blocks, final convolutional layers, and the classifier.

**3. Architecture Details:**

* Initial Convolutional Layer: Starts with a convolutional layer followed by batch normalization and ReLU6 activation.
* Inverted Residual Blocks: Utilize expansion, depthwise convolution, pointwise convolution, and residual connections to reduce model parameters while maintaining performance.
* Final Layers: Applies a 1x1 convolutional layer to reduce channels to the desired output size, followed by global average pooling and a fully connected layer for classification.
* Activation Function: ReLU6 activation is used throughout the model.

**4. Initialization and Hyperparameters:**

* Optimizer: Adam
* Learning rate = 0.001
* StepLR scheduler
* Batch size = 64
* Transformation : normalization
* Epochs: 30

**5. Helper Functions:**

* imshow: Displays images from the dataset.
* imshow\_with\_labels: Displays images along with their true labels and optionally predicted labels.
* train\_model: Function for training the model on the training dataset.
* evaluate\_model: Function for evaluating the model on the validation dataset.
* plot\_metrics: Function for plotting training and evaluation metrics such as loss and accuracy.

**6. Training and Evaluation:**

In the `main` function, data loaders, optimizer, and scheduler are set up.

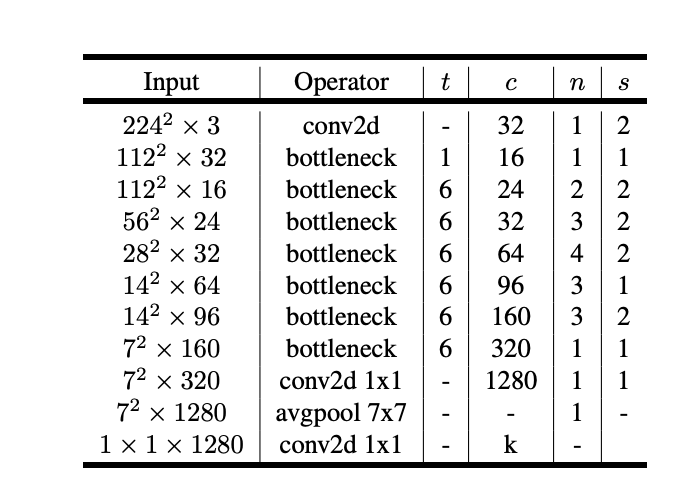
The model is trained and evaluated over a fixed number of epochs using the training and evaluation functions.

Learning rate scheduling is applied using StepLR.

**7. Main Execution:**

The `main` function is called when `\_\_name\_\_ == '\_\_main\_\_'`, initiating the training and evaluation process.

Model architecture as describes in the paper:



# Experimental Setup

**Dataset:**

Our experiments used the CIFAR-10 dataset, which is a popular dataset for evaluating image classification models. CIFAR-10 consists of 60,000 32x32 color images in 10 different classes, with 6,000 images per class. The dataset is divided into 50,000 training images and 10,000 test images.

**Hyperparameters:**

To establish a baseline and conduct subsequent experiments, we tuned various hyperparameters. While the exact values varied across experiments, they typically included learning rate, number of epochs, and optimizer settings. Specific hyperparameter values for each experiment were recorded and monitored to understand their impact on model performance.

**Evaluation Metrics:**

For the classification task on CIFAR-10, we primarily focused on accuracy as our evaluation metric. Specifically, we reported top-1 accuracy, which is the standard for CIFAR-10 evaluations. Although the original MobileNetV2 paper discusses top-1 and top-5 accuracy, the latter is more relevant for datasets with a larger number of classes, like ImageNet.

**Experiment Tracking:**

To log and monitor our experiments, we employed Weights and Biases (wandb), a tool that enabled us to track the progress and results of each model iteration.

# Hyperparameter Tuning

In this section, we conduct hyperparameter tuning on the baseline model. Through a series of configurations, we explored various settings and modifications to identify their impact on the model's performance. Each configuration represents a unique combination of hyperparameters, including learning rates, epoch counts, regularization strategies, and data augmentation techniques. The goal was to optimize these parameters to enhance the model's learning efficiency and generalization capability.

Below, we detail the configurations tested, along with the corresponding training and evaluation metrics.

However, it's important to note that due to an oversight, the training set was mistakenly used as the evaluation set, affecting the reliability of the evaluation results.

Relevant code file: experiment1.ipynb

Config 1:

Learning Rate: 0.001

Epochs: 20

Results: Train Loss - 0.282493, Train Accuracy - 90.16%, Eval Loss - 0.190753, Eval Accuracy - 94.17%.

Config 2:

Extended training to 30 epochs.

Results: Train Loss - 0.201459, Train Accuracy - 93.34%, Eval Loss - 0.134862, Eval Accuracy - 96.29%.

Config 3:

Learning Rate: 0.0005

Learning rate scheduler adjusts every 15 epochs.

Results: Train Loss - 0.119190, Train Accuracy - 95.89%, Eval Loss - 0.035135, Eval Accuracy - 99.33%.

Config 4:

Learning Rate: 0.001 with weight decay of 1e-4.

Results: Train Loss - 0.203835, Train Accuracy - 93.32%, Eval Loss - 0.135233, Eval Accuracy - 96.27%.

Config 5:

Enhanced data augmentation: Random horizontal flip, 10-degree rotation, and random cropping.

Results: Train Loss - 0.777692, Train Accuracy - 72.59%, Eval Loss - 0.737612, Eval Accuracy - 73.98%.

Config 6:

Dropout rate of 0.2.

Results: Train Loss - 0.264754, Train Accuracy - 90.78%, Eval Loss - 0.193498, Eval Accuracy - 94.01%.

Config 7:

Combines Config 5's data augmentation with Config 6's dropout rate.

Results: Train Loss - 0.817149, Train Accuracy - 70.92%, Eval Loss - 0.775910, Eval Accuracy - 72.56%.

Config 8:

Learning Rate: 0.0005 with weight decay and "ReduceLROnPlateau" scheduler.

Results: Train Loss - 0.555098, Train Accuracy - 80.22%, Eval Loss - 0.451609, Eval Accuracy - 84.65%.

Config 9:

Learning Rate: 0.0005 with higher dropout rate (0.3).

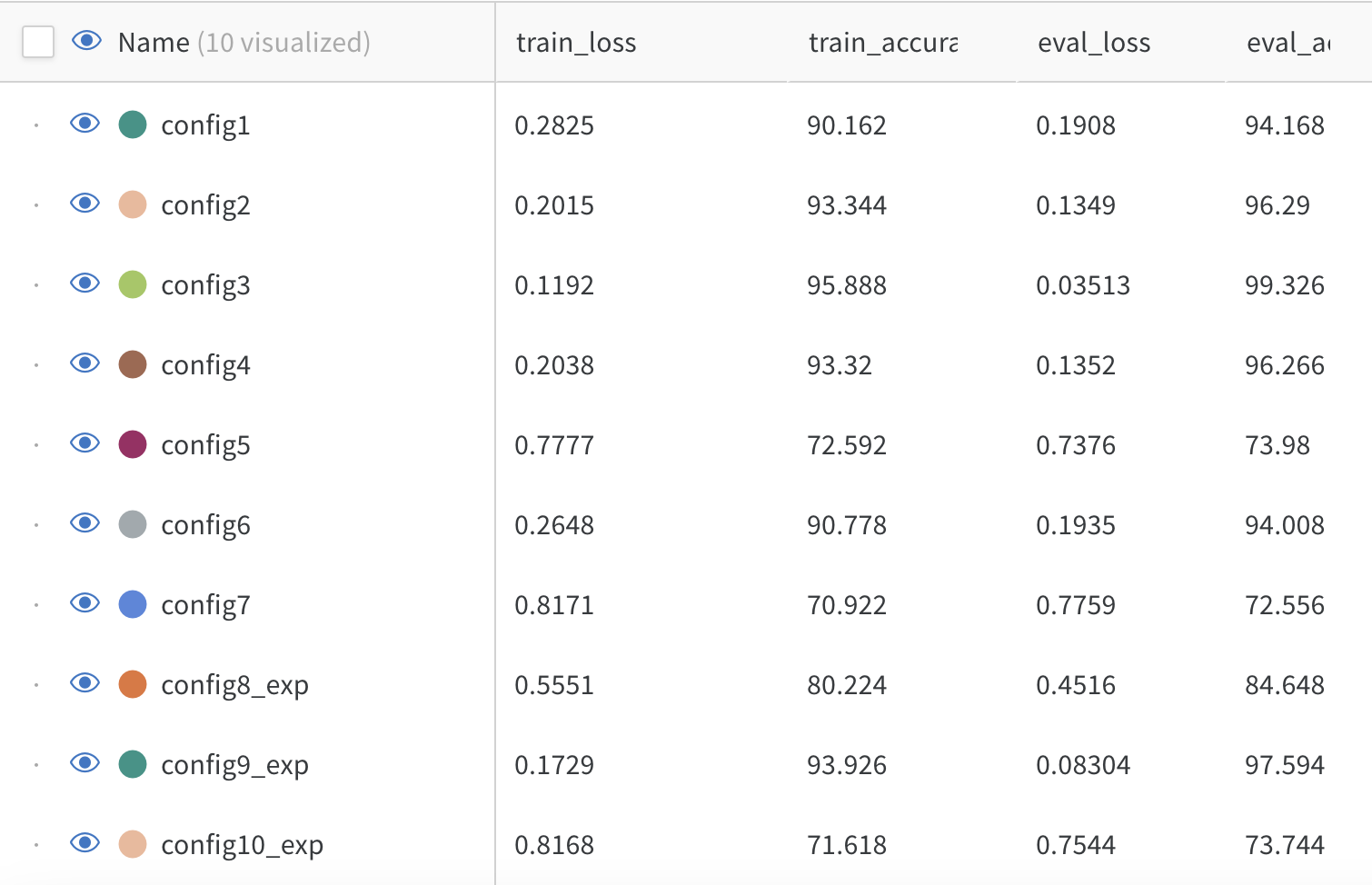
Results: Train Loss - 0.172900, Train Accuracy - 93.93%, Eval Loss - 0.083035, Eval Accuracy - 97.59%.

Config 10:

More aggressive data augmentation: Random flip, rotation, color jitter, and cropping.

Results: Train Loss - 0.816806, Train Accuracy - 71.62%, Eval Loss - 0.754390, Eval Accuracy - 73.74%.

Run time summary:



Prior to recognizing the oversight in our evaluation process, the preliminary results suggested that several configurations achieved notably high accuracies. This outcome led us to initially conclude that we had identified a set of effective hyperparameters. As a result, we made the decision to postpone further experimentation, and move on with architectural modifications.

# Architectural Modifications

After establishing a baseline with hyperparameter tuning, we proceeded to explore architectural changes to MobileNetV2.

In this section, the first few experiments used these hyperparameters:

configA = {

"num\_classes": 10,

"learning\_rate": 0.0005,

"batch\_size": 64,

"epochs": 30,

"optimizer": "Adam",

"scheduler\_step\_size": 15,

"scheduler\_gamma": 0.5,

"first\_layer\_channels": 32,

"expansion\_layer\_channels": 1280

}

## Architectural Modifications – part 1

In this section, we detail the experiments conducted on architectural modifications, exploring various changes to the original model's structure. Each experiment was designed to test a specific aspect of the architecture, ranging from layer configurations to activation functions. However, it's important to note that during this phase, the evaluation continued erroneously on the training set. Once we identified models with promising training accuracy, we re-evaluated them using the correct test set. Consequently, some models have two sets of results, with the second set reflecting the true test set performance.

Relevant code file: experiment2.ipynb

### Model A

In this model we experimented using the first 10 layers of the original architecture.

Run summary:

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 99.2 |
| eval\_loss | 0.0299 |
| train\_accuracy | 96.922 |
| train\_loss | 0.08681 |

In this experiment accuracy was surprisingly high, but we ran this model later with test set and got the following result:

Run summary:

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 67.08 |
| eval\_loss | 1.91413 |
| train\_accuracy | 97.102 |
| train\_loss | 0.08131 |

### Model B

In this model we experimented reducing expansion ratio to 2.

Expansion purpose is to expand the number of channels in the data before it goes into the depthwise convolution.

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 86.41 |
| eval\_loss | 0.40227 |
| train\_accuracy | 80.964 |
| train\_loss | 0.53509 |

### Model C

In this model, we experimented increasing expansion ratio to 10.

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 98.484 |
| eval\_loss | 0.0498 |
| train\_accuracy | 96.372 |
| train\_loss | 0.10573 |

Model performance with test set:

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 68.77 |
| eval\_loss | 1.58786 |
| train\_accuracy | 96.75 |
| train\_loss | 0.09628 |

#### Appearently reducing the expansion ratio did not yield the best resuls unlike increasing expansion.

### Model D

In this model we experimented using bigger kernel size = 5.

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 96.306 |
| eval\_loss | 0.11459 |
| train\_accuracy | 93.58 |
| train\_loss | 0.18533 |

Model performance on test set:

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 63.56 |
| eval\_loss | 1.64782 |
| train\_accuracy | 92.564 |
| train\_loss | 0.21013 |

In the following models we experimented different activation functions.

In the original MobileNetV2 architecture, the ReLU6 activation function was selected mainly for its compatibility with low-precision computations, which is particularly beneficial for deployment on mobile devices where computational resources are limited. ReLU6 is designed to cap the activation at a maximum value of 6, which aids in the training of models in low-precision formats by constraining the range of the activation outputs.

Additionally, the use of ReLU6 is motivated by considerations related to the manifold of interest. It helps in maintaining a manageable range of activation values, contributing to more stable training and inference processes.

We decided to experiment with a set of alternative activation functions that potentially share some beneficial properties with ReLU6.

The activation functions we selected for experimentation were:

* LeakyReLU: Unlike ReLU6, which caps the activation at 6, LeakyReLU allows for a small, non-zero gradient when the unit is inactive. This can help prevent the dying ReLU problem associated with standard ReLU activation functions. However, unlike ReLU6, it does not have an explicit upper bound.
* SiLU (Sigmoid Linear Unit): Also known as the Swish function, SiLU is a smooth, non-monotonic function that combines aspects of ReLU and sigmoid functions. It doesn't explicitly cap the activation like ReLU6, but it tends to push the activations towards zero for negative inputs and has a self-gating property, which can make it more robust during training.
* Hardswish: This function is designed to be a computationally efficient approximation of the SiLU. It introduces a piecewise linear approximation, which can be more hardware-friendly, particularly for fixed-point arithmetic operations on mobile or embedded devices. Hardswish has a bounded output range, which can contribute to numerical stability similar to ReLU6. However, the range is typically between -3 and +6, slightly different from ReLU6's 0 to 6 range.
* Hardsigmoid: This is a piecewise linear approximation of the sigmoid function, designed to be more computationally efficient. Like Hardswish, it offers a bounded output, typically between 0 and 1, which helps in numerical stability and is beneficial for low-precision computation. While the range is different from ReLU6, the concept of bounding the activation to a fixed range is similar, which is crucial for training stability and efficient inference in quantized models.

### Model with LeakyRelu

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 97.572 |
| eval\_loss | 0.07938 |
| train\_accuracy | 94.78 |
| train\_loss | 0.14957 |

Model performance on test set:

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 66.72 |
| eval\_loss | 1.57126 |
| train\_accuracy | 95.128 |
| train\_loss | 0.13981 |

### Model with SiLU

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 99.014 |
| eval\_loss | 0.03273 |
| train\_accuracy | 96.988 |
| train\_loss | 0.0846 |

Model performance with test set:

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 68.39 |
| eval\_loss | 1.72028 |
| train\_accuracy | 97.05 |
| train\_loss | 0.08669 |

### Model with Hardswish

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 98.814 |
| eval\_loss | 0.04066 |
| train\_accuracy | 96.372 |
| train\_loss | 0.10244 |

Model performance with test set:

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 66.51 |
| eval\_loss | 1.76056 |
| train\_accuracy | 96.554 |
| train\_loss | 0.09939 |

### Model with Hardsigmoid

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 70.432 |
| eval\_loss | 0.81114 |
| train\_accuracy | 69.376 |
| train\_loss | 0.8543 |

### Model with Squeeze and Excitation Block (SE Block)

In this model we integrated a block called Squeeze and excitation block

The Squeeze-and-Excitation (SE) block takes into account the interdependencies between different feature channels of CNNs.

The reason behind incorporating SE blocks into our experiments was based on their proven ability to boost network performance across a variety of tasks at almost no computational cost.

[Details about this block](https://medium.com/@tahasamavati/squeeze-and-excitation-explained-387b5981f249)

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 97.976 |
| eval\_loss | 0.07247 |
| train\_accuracy | 94.032 |
| train\_loss | 0.1705 |

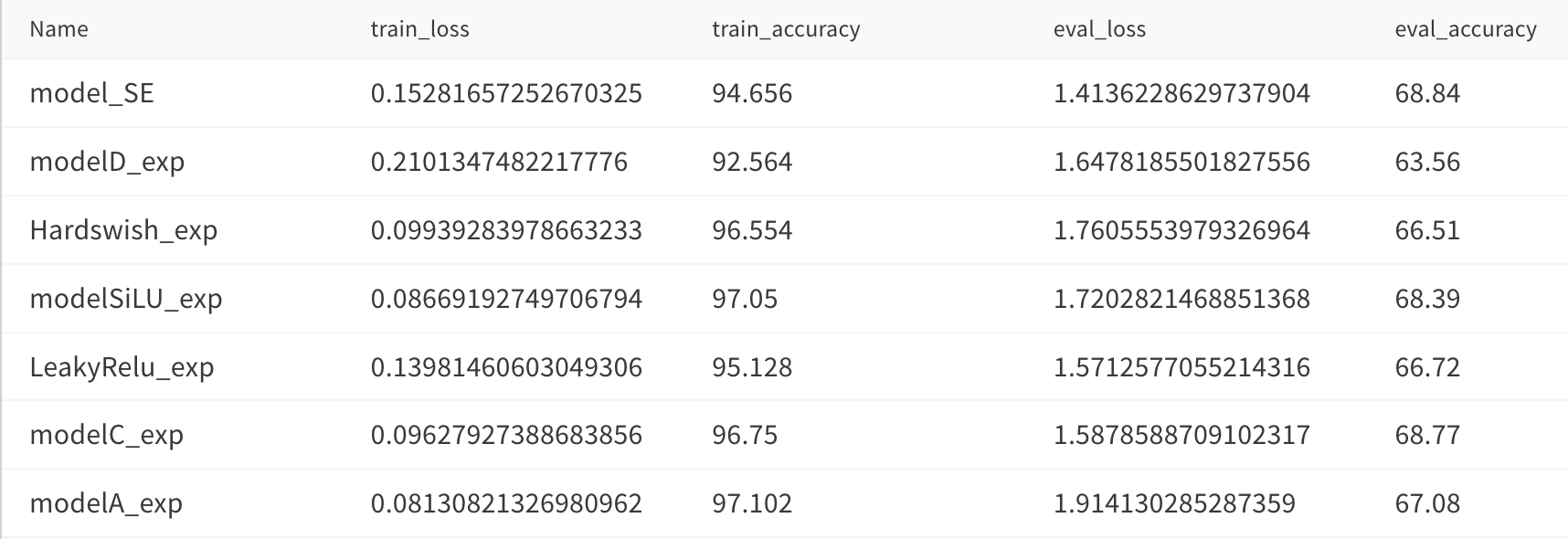
Model performance with test set:

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 68.84 |
| eval\_loss | 1.41362 |
| train\_accuracy | 94.656 |
| train\_loss | 0.15282 |

Based on the results, we can observe that all the models have high variance and are overfitting the train set, because there is a huge gap between the training and test accuracy and loss.

Here’s a table summarizing the models that were trained with test set:



After observing the results, we spotted multiple problems:

* High variance, the models did not generalize well on unseen data, this might indicate an overfitting issue.
* Most models reached their best testing accuracy before the 30th epoch. This suggests that the number of epochs might be more than what the models need.
* Test/Eval loss started to increase after around epoch 15. An initial learning rate of 0.0005, along with a learning rate scheduler that drops the learning rate by half every 15 epochs, may be the reason behind the increasing loss. The reduced learning rate might have led to the models excessively fine-tuning to the training data, and continued training beyond the model’s need (as discussed in the previous bullet) may also be a reason for it.

[Plots summarizing models' performances](https://wandb.ai/malak-y17/mobilenetv2_arch/reports/-Run-Summaries-Part-1--Vmlldzo3MzY4MDI4?accessToken=ry3bhicyhezlotb5qpczb6n0q5nle4yitqpvsest7y79zcrg8z2pw9g6wqi6lfa4)

In order to improve the performance, adjustments to the hyper parameters is needed.

## Architectural Modifications – part 2

Relevant code file: experiment\_test.ipynb

To address the problems faced in the previous section, we introduce this new set of hyper parameters:

config = {

"num\_classes": 10,

"learning\_rate": 0.001,

"epochs": 20,

"optimizer": "Adam",

"weight\_decay": 1e-4,

"dropout\_rate": 0.3,

"scheduler\_step\_size": 10,

"scheduler\_gamma": 0.7,

"first\_layer\_channels": 32,

"expansion\_layer\_channels": 1280

}

We decreased learning and number of epochs, added L2 regularization (weight decay), and dropout.

Also, we added applied data augmentation, randomly cropping, and flipping the images.

transform = transforms.Compose([

transforms.RandomCrop(32, padding=4),

transforms.RandomHorizontalFlip(),

transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),

])

In this section, we generated a set of unique models based on combinations of activation functions, block types, and architectures. Specifically, we explored the following sets of parameters:

activation\_functions = [nn.ReLU6, nn.SiLU, nn.Hardswish]: This set includes three different activation functions that we tested in our models.

block\_types = ["Regular", "SE"]: This set distinguishes between using blocks with or woithout Squeeze-and-Excitation (SE) modules.

archs = [original\_architecture, arch\_expansion10]: This set includes two architectural variations, one with the original architecture and another with an expansion factor of 10.

activation\_functions = [nn.ReLU6, nn.SiLU, nn.Hardswish]

block\_types = ["Regular", "SE"]

archs = [original\_architecture, arch\_expansion10]

for activation\_function in activation\_functions:

for arch in archs:

for block\_type in block\_types:

create\_model(arch, block\_type, activation\_function, config, trainloader, testloader)

This process resulted in the creation of 12 unique models.

The follwing schedule summarizes the models performance sorted descendingly by test (eval) accuracy:



Based on the results we can observe the following:

1. **SiLU Activation Function**: The models using the SiLU activation function generally achieved the highest training and evaluation accuracies among the three activation functions tested.
2. **SE Blocks**: Models utilizing SE blocks (Squeeze-and-Excitation modules) appear to have performed better in terms of evaluation accuracy compared to the models without SE blocks.
3. **Architecture Variations**: Models with the expansion factor of 10 have generally outperformed those with the original architecture’s factor of 6 in terms of evaluation accuracy, implying that the increased complexity allows the model to learn more nuanced features of the data.
4. **ReLU6 Activation Function**: Models with the ReLU6 activation function have the lowest evaluation accuracies and the highest evaluation losses. This might suggest that ReLU6 is either not as effective for capturing the complexities of your data or it is not pairing well with the other model configurations.
5. **Overfitting**: While overfitting is not as severe as in the previous set of models (based on the smaller gaps between training and evaluation accuracy), it is still present, especially in models with the ReLU6 activation function.
6. **Underfitiing**: There might be a bias problem here, since previous models scored high training accuracies, and in this experiment the models have training accuracies ranging from 74% to 80%.
7. **Hyperparameters and Data Augmentation**: The combination of hyperparameters and data augmentation strategies applied contributed to a more regularized training process. The use of weight decay and dropout, along with the data transformations, helped in reducing overfitting problem.
8. **Learning Rate Scheduler**: The problem faced with previous models from part 1 was not noticed here, where evaluation losses started to increase after epoch 15.

[Plots summarizing performances](https://wandb.ai/malak-y17/arch_3/reports/Run-Summaries-Part-2--Vmlldzo3MzczNzQ5?accessToken=m7fqjdcly356a2mts8771aaqtsgut9hc2f7a41a8101x9s1rnd475gbvmqfn3782)

An important observation from this experiment is that the model “ReLU6\_original\_Regular” employing the original architecture's expansion factor (6) and the ReLU6 activation function, registered the lowest evaluation accuracy among all the models tested. This observation suggests that the modifications introduced have contributed positively to enhancing model performance.

To address the issue of bias, we could increase the number of epochs, optimize our hyperparameters, or increase the model's complexity by adding more layers.

### Width Experiment

In this experiment width scaling is applied on the netweork’s layers to make them wider.

(The concepts of width multiplier and depth multiplier as hyperparameters were introduced in MobileNetV1, here we experiment width only.)

Relevant code file: Modification\_width.ipynb

For this experiment number of epochs was set to 30, we did not perform any regularization, and applied data augmentation, as following:

transform = transforms.Compose([

transforms.RandomCrop(32, padding=4),

transforms.RandomHorizontalFlip(),

transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),

])

Those are the hyperparameters used :

config = {

"num\_classes": 10,

"learning\_rate": 0.001,

"epochs": 30,

"optimizer": "Adam",

"scheduler\_step\_size": 10,

"scheduler\_gamma": 0.7,

"width" : 2

}

While keeping original architecture’s configurations, such as ReLU6 as activation function and using expansion factor = 6.

**Run summary:**

|  |  |
| --- | --- |
| epoch | 30 |
| eval\_accuracy | 88.400 |
| eval\_loss | 0.383 |
| train\_accuracy | 93.948 |
| train\_loss | 0.173 |

Note: The width model experiment was conducted concurrently with the models from the previous section. Due to this parallel execution, the width model experiment was not integrated with the combinations explored in the prior section.

This model has surpassed all previous models in terms of train and test accuracies. By doubling the neurons in each layer (making the network wider), we’ve increased its capacity to represent complex functions and capture detailed data patterns. This expansion in parameters enables the network to learn a broader range of patterns, potentially enhancing its generalization capabilities. In addition to that, training the model for 30 epochs allowed the model to learn and adjust its parameters for optimal performance. Consequently, it’s fair to conclude that the implementation of width scaling played a significant role in improving the performance.

# Reflections and Learning

## Challenges and Lessons:

One of the primary challenges we encountered during our experimentation was the extensive running time required for each model training session. The breadth of experiments we aimed to conduct was vast, and, constrained by the assignment's deadline, we could not explore every idea we had in mind. To mitigate this, we utilized Google Colab's advanced GPUs and our personal GPUs.

A critical lesson learned was the importance of validation. Initially, we validated models on the training set, a misstep that skewed our perception of model performance, and stopped exploring other hyperparameters and configurations that might be good. For example, exploring other optimizers, trying different hyperparameter combinations via random search to potentially uncover more optimal configurations.

## Insights:

This project allowed us to step into the shoes of researchers, giving us a firsthand experience in developing hypotheses, conducting experiments, and refining our approach based on the outcomes. It was an excellent opportunity to apply our knowledge in a practical setting, diving into the details of how to enhance neural network architectures.

# Further Improvements

For future improvements, here are some directions and ideas that we wanted to experiment, but did not have time to:

* Combining Width Scaling with SE Block: Integrating width scaling alongside the inclusion of SE blocks could provide a dual approach to enhancing model capacity and efficiency. Further, assessing the interaction between different activation functions findings regarding model performance and efficiency.
* Exploring Compound Scaling (EfficientNet's Technique): Adopting EfficientNet's compound scaling method, which systematically scales network width, depth, and resolution, could offer a structured pathway to optimizing our network's architecture, potentially leading to more balanced and effective models.
* Extending Experiments to ImageNet: After identifying top-performing models in our current setup, a logical next step would be to evaluate these models on a more challenging and diverse dataset like ImageNet. This would provide a robust test of the models' generalizability and scalability, offering valuable insights into their applicability to broader real-world tasks.

# ChatGPT Links

<https://chat.openai.com/share/1beed686-41a9-4896-9da0-de6c9e0fb99f>

<https://chat.openai.com/share/c362ab3e-982a-4969-89ab-76a4b2fb1d4c>

<https://chat.openai.com/share/24473d22-7408-4ad4-b687-c27439be6ceb>

some chats could not be exported because they are either too long or they have images and ChatGPT does not export them.