Hyperparameter Tuning in Adam Optimizer for MobileNetV2

Sadaf Tanvir[1]
By : Syed Hassan Dildar (8984)
[1] Abasyn University Islamabad, Pakistan

Abstraction

Our research is all about finding the best way to adjust the inner settings of a computer model so it can do a better job of spotting tomato leaf diseases. Farming plays a huge role in Pakistan's economy, and sick crops are a major challenge for farmers. To tackle this problem, we tested several different ways of setting up the Adam optimizer, which is a tool that helps train the MobileNetV2 model to be both more accurate and faster. We tried five different setups: one standard setup, and four modified versions where we changed small things, like increasing the momentum, lowering another setting, raising a very small number called epsilon, and making the training smoother overall. All of our experiments used a collection of tomato leaf images showing both healthy and diseased plants. We carefully recorded how well the model learned from the training pictures, how accurately it could identify diseases in images it had never seen before, and how long it took to complete the training process. What we found was that the standard setup performed the best overall. It reached an accuracy of 88.76% on new, unseen images, and training took just over 13 minutes. When we increased the momentum, the training did speed up a little—finishing in just under 13 minutes—but the accuracy dipped slightly to 87.83%. Other changes also made clear trade-offs: sometimes the model trained slower but was more reliable, or it trained faster but wasn't as accurate. These results really highlight why paying attention to these small model settings is so important. If we want to build AI tools that farmers can actually trust in the field, we have to get these little details right. By carefully tuning the model, we can create systems that catch plant diseases earlier, help farmers respond more quickly, reduce the amount of crops lost each season, and support farming methods that are smarter, more productive, and better for the environment in the long run. In the end, getting these technical settings just right can make a real difference in everyday farming. It's not just about building a smart model—it's about building a helpful tool that works in the real world, where it matters most.

Keywords: Deep Learning, Hyperparameter Tuning, Adam Optimizer, MobileNetV2, Tomato Leaf Disease Detection, Precision Agriculture, Optimization, Sustainable Farming.

1. Introduction

Plants are essential components of our ecosystem, vulnerable to a range of diseases much like any other living organisms. Unmonitored and uncontrolled, these diseases can cause substantial crop yield losses, adversely affecting farmers' livelihoods and disrupting the stability of agricultural communities. It is against this backdrop that our project, the IoT-Driven Climate Monitoring and Plant Leaf Disease Classification System, emerges as an innovative and cost-effective tool designed to assist farmers. This system provides crucial disease classification and real-time climate data for plants, addressing a significant need within Pakistani agriculture. The current infrastructure and systems have proven inadequate in meeting the specific requirements of farmers for a comprehensive monitoring solution. Consequently, there is an urgent need for a tailored system that addresses the unique challenges faced by Pakistan's agricultural community. Our proposed system integrates IoT technology, classification models, and real-time sensors to bridge this critical gap. This project is not solely focused on safeguarding individual plants; it aims to protect the entire agricultural ecosystem upon which countless livelihoods depend. By leveraging technology at the intersection of agriculture, our initiative strives to ensure food security and stability for Pakistani farmers. This project is driven by a profound sense of responsibility toward the agricultural community and a commitment to harnessing technology for societal benefit. Hence, our project represents not merely a technological innovation but a foundational step towards the future of sustainable agriculture in Pakistan.

2. Mathematical Background

The Adam optimizer is a first-order gradient-based optimization algorithm that computes adaptive learning rates for each parameter. It uses both the first moment (mean) and the second moment (uncentered variance) of the gradients to adjust the learning rates. The update equations are as follows:

1. **First moment update:**
   $$ m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t $$

2. **Second moment update:**
   $$ v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 $$

3. **Bias-corrected estimates:**
   $$ \hat{m}_t = \frac{m_t}{1 - \beta_1^t} $$
   $$ \hat{v}_t = \frac{v_t}{1 - \beta_2^t} $$

4. **Parameter update rule:**
   $$ \theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} $$

Where:

- (g_t) is the gradient at time step (t).
- (\alpha) is the learning rate.
- (\beta_1) and (\beta_2) are the first and second moment decay rates.
- (\epsilon) is a small constant added to prevent division by zero.

3. Related Work

Deep learning models have emerged as a prominent approach for leaf disease diagnoses, evolving from earlier approaches that relied on classical classifiers and manually designed features. More recently, convolutional neural networks (CNNs) have become a preferred choice for feature extraction and classification tasks. Previous studies related to leaf disease classification focus on either improving the architectures of CNNs, centering around refining the feature extraction strategies or data augmentation techniques, less importance has been given to the systematic evaluation of optimization algorithms and their hyperparameters. Multiple studies have reported the use of state of the art deep and complex models such as EfficientNet family, MobileNet and MobileNetV2 for agricultural image classification. One such study by (Saleem et al., 2022) showed that particularly EfficientNet-B0, sets a new standard for model accuracy and efficiency in leaf disease classification. We see that EfficientNet-B0 is being widely adopted in applications in agricultural image classification (Farman et al., 2022), plant disease identification by (Fathimathul Rajeena et al., 2023) and (Alqahtani et al., 2023). Similarly, recent studies such as (Sarfarazi et al., 2024) and (Nurhasanah et al., 2023) use MobileNet for developing a deep learning system based on mobile for precise detecting and classifying leaf diseases. In the same way, MobilenetV2 is also being employed for leaf disease classification tasks such as reported by (Jlassi et al., 2024), (Dutta et al., 2024) and (Kumar et al., 2023) We observe that despite the critical role that optimizers play in deep learning, there is limited research on their comparative effectiveness. In Some studies, such as (Wilson et al., 2018), convergence and performance of various optimizers is examined, (Sun, 2020) provides a theoretical overview. (Dogo et al., 2022) also explore model performance under the influence of optimizers. Building on these studies, we analyze the effects of widely used optimizer adam on MobileNetV2 for leaf disease detection using real-field image datasets like PlantDoc. While previous research has focused on architectural comparisons, our study investigates the impact of different optimizer hyperparameters on MobileNetV2—in the relatively underexplored domain of leaf disease classification. Our research is focused on improving model accuracy through extensive optimizer testing thereby advancing state-of-the-art agricultural disease diagnosis. We systematically evaluate how various hyperparameters impact the performance of MobileNetV2 in plant disease classification. This is linked to ultimately advancing precision agriculture and support of sustainable farming practice.

4. Research Methodology

It describes the process of developing and validating the model for detection of diseases This session is for developing and validating the model for detection of diseases in leaves. We used TensorFlow to implement a MobileNetV2-based CNN model, pre-trained on ImageNet and fine-tuned for maize, potato, and tomato disease classes. In order to get a more accurate model, we tuned different hyperparameters of Adam optimizer as well as using image augmentation techniques such as flipping and rotation augmentations which allows the model to generalize better. The dataset was split into train, validation, and test sets. The models were trained for several epochs with their performances tracked across each run with varying setups of optimizers used in them.

Proposed CNN model for possible disease classification

We aim to provide a clear understanding of the CNN architecture for our leaf classification approach, as shown in Figure 2.

4.1. Architecture of model

Input Layer: The model begins by accepting images that have been resized to 96 by 96 pixels, with three channels representing the red, green, and blue (RGB) color components. Each image is preprocessed according to the specific requirements of the MobileNetV2 architecture to ensure compatibility and optimal feature extraction.
Convolutional Layers*:* At the core of the model are convolutional layers derived from the pre-trained MobileNetV2 architecture. These layers utilize depthwise separable convolutions to efficiently extract hierarchical features from the input images. The process enables the detection of visual patterns, ranging from simple edges to more complex textures. Notably, the base layers of MobileNetV2 remain frozen during training, thereby preserving the valuable feature representations learned from the ImageNet dataset.
Global Average Pooling: After feature extraction, a Global Average Pooling layer is applied. This layer reduces the spatial dimensions of the feature maps by calculating the average value for each map, which helps to decrease the overall dimensionality while retaining essential information.
Batch Normalization: The model incorporates a Batch Normalization layer to standardize the inputs to the subsequent dense layer. This step enhances training stability, accelerates convergence, and reduces the model's sensitivity to initial weight values.
Dropout: To address the risk of overfitting, a Dropout layer with a rate of 0.4 is introduced. During training, this layer randomly deactivates 40 percent of the neurons, promoting better generalization of the model.
Output Layer*:* The final layer consists of a number of neurons equal to the total number of plant disease categories present in the dataset. A Softmax activation function is used to generate a probability distribution across all classes, enabling the model to assign each input image to the most likely disease category.

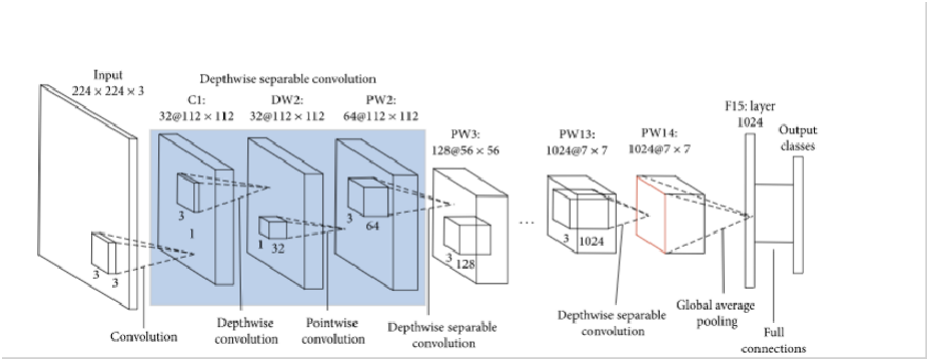

Figure 1 : MobileNet V2 Architecture

4.2. Collection of Dataset

The dataset, known as" Plant Village," was sourced from Kaggle, a reputable platform for data science and machine learning resources. Our selection of this dataset was based on its relevance to the classification of tomato leaves, aligning with the objectives of our research. Table 1 describes the datasets for tomato leaf diseases.

The dataset is made up of pictures of tomato leaves, sorted into six groups based on the condition of the leaf. These groups include Early Blight, Healthy leaves, Late Blight, Leaf Mold, Mosaic Virus, and Septoria Leaf Spot. To build and test the model, the pictures were divided into three separate sets. The training set was used to teach the model to recognize the diseases. The validation set was used to check the model's progress and adjust it during training. Finally, the testing set was used to evaluate how well the model performs on new, unseen pictures. Each set contains examples from all six categories, ensuring the model learns from and is tested on a balanced variety of leaf conditions.

Table 1: Tomato Dataset Description

Name	Total	Early Blight	Healthy	Late Blight	Leaf mold	Mosaic virus	Septoria spot
Train	5318	700	1114	1336	667	261	1240
Validate	756	100	156	191	95	37	177
Test	1519	200	318	382	190	75	354

4.3. Dataset Description

This study used a dataset containing tomato leaf images for disease classification. The dataset was systematically organized into three subsets: a training set, a validation set, and a test set, each stored in separate directories. The images were preprocessed by resizing them to 96x96 pixels to ensure uniform input dimensions for the model. To enhance the model's robustness and prevent overfitting, data augmentation techniques were applied, including random horizontal flips, rotations, zooms, and translations. These transformations created additional training variations from the original tomato leaf images. The dataset was loaded using TensorFlow's image dataset utilities, which automatically handled class labeling based on the folder structure. Transfer learning was implemented using the pre-trained MobileNetV2 architecture, where the base convolutional layers were frozen to preserve their learned features from ImageNet, and only the custom classification layers were trained on the tomato leaf data. The specific breakdown of the dataset is detailed in Table 2: Dataset Description.

Table 2:Dataset Description

Total Images	7593
Train	5318
Validation	756
Testing	1519

Before feeding the data into the deep learning model, we applied preprocessing steps to standardize and prepare the images for analysis. These preprocessing steps included resizing, MobileNetV2-specific pixel scaling, and data augmentation with random horizontal flips, rotations, zooms, and translations. To leverage pre-trained features efficiently, we applied transfer learning using the MobileNetV2 architecture. This lightweight model allowed us to benefit from knowledge acquired from the ImageNet dataset while maintaining computational efficiency, significantly enhancing the model's capability to extract relevant features from plant leaf images. We outline the training process, encompassing the Adam optimizer with different hyperparameter configurations and evaluation metrics, in our experimental results.

4.4. System Specification

Table 3 refers to the detailed system specification used in the development of our system. The system development was conducted using a local machine with an Intel Core i7 processor, 8GB of RAM, and 256GB of storage, with additional computational resources provided by Google Colab for model training and experimentation. The technical implementation was built entirely within a Python environment using TensorFlow as the core deep learning framework, along with essential libraries including NumPy for numerical operations and Matplotlib for visualization. The plant disease classification model was constructed using the MobileNetV2 architecture through transfer learning, where the pre-trained ImageNet weights were utilized and the base convolutional layers were frozen during training. Custom classification layers were added, featuring Global Average Pooling, Batch Normalization, Dropout regularization, and a final Dense layer with softmax activation. The model was trained and evaluated using a systematically partitioned dataset, with experiments conducted using various configurations of the Adam optimizer to optimize performance. The entire workflow (from data loading and augmentation to model training and result analysis) was implemented within the Colab environment.

Table 3: System Specifications

Requirements	**Specification**
Hardware	Intel Core i7 processor, minimum 8GB RAM, 256GB storage.
Platform	Google Colab (cloud-based for training and experimentation).
Core Development Framework	TensorFlow
Key Python Libraries	NumPy, Matplotlib
Model Architecture	MobileNetV2 with transfer learning
Pre-trained Weights	ImageNet
Custom Layers	Global Average Pooling, Batch Normalization, Dropout, Dense (softmax)
Dataset Structure	Systematically partitioned (Train/Validation/Test)
Optimizer	Adam (with multiple hyperparameter configurations)
Workflow Environment	Google Colab (end-to-end implementation)
Primary Tasks	Data loading, augmentation, model training, evaluation, and visualization

4.5. Hyperparameter Settings

This study tested five different configurations of the Adam optimizer to evaluate their impact on model performance.
The configurations were:

- Baseline (β_1=0.9, β_2=0.999, ε=1e-8)
- High β_1 (β_1=0.95, β_2=0.999, ε=1e-8)
- Low β_2 (β_1=0.9, β_2=0.99, ε=1e-8)
- Large ε (β_1=0.9, β_2=0.999, ε=1e-6)
- Very High Smoothing (β_1=0.99, β_2=0.9999, ε=1e-7)

The training was conducted with a fixed learning rate of 0.001 over 10 epochs, using a batch size of 32, and applied to a tomato leaf disease classification dataset consisting of six categories. Each configuration was tested with the same MobileNetV2 model architecture, featuring frozen base layers and custom classification layers, while employing data augmentation techniques including random flips, rotations, zooms, and translations to improve generalization.

Table 4: Hyperparameter Settings

Hyperparameter	Value
Model Architecture	MobileNetV2
Optimizer	Adam
Learning Rate	0.001
β_1 (First Moment Decay)	Tested values: 0.9, 0.95, 0.99
β_2 (Second Moment Decay)	Tested values: 0.999, 0.99, 0.9999
ε (Epsilon)	Tested values: 1×10^{-8}, 1×10^{-6}, 1×10^{-7}
Loss Function	Categorical Cross-Entropy
Batch Size	32
Number of Epochs	10
Input Image Size	96 × 96
Data Augmentation	RandomFlip, RandomRotation, RandomZoom, RandomTrans-lation

5. Results and Discussion

In our tests, we tried five different versions of the Adam optimizer to see how small changes in its settings affect how well and how fast a model learns. The standard setup, called the baseline, gave strong results with 88.76% accuracy on unseen data and took about 807 seconds to train. When we increased the momentum (β_1), training became slightly faster (780 second) but accuracy dropped a bit to 87.83%, showing that more speed might come at the cost of performance. When we lowered the second momentum term (β_2), accuracy stayed high at 88.76%, but training slowed down to 875 seconds, it means this setting is accurate but not efficient. Raising a small stability number (ε) reduced accuracy to 87.43% without saving much time, so it's not very helpful here. Finally, using very smooth settings gave the best training accuracy (87.46%) but didn’t do as well on new data (87.83%), and it was slow.

Overall, the baseline Adam settings work best for balancing good accuracy and reasonable training time. Changing the optimizer can speed things up or improve training accuracy, but often at the expense of validation performance or efficiency. This tells us that unless you have a specific need for speed or stability, sticking with the default Adam values is usually the safest choice for reliable results.

Table 5: Results Analysis

Experiment	Configuration	Best Val Accuracy	Best Train Accuracy	Training Time (seconds)
1	Baseline (β_1=0.9, β_2=0.999, ε=1e-8)	0.8876	0.8633	807.39s
2	High β1(β_1 = 0.95, β_2 = 0.999, ε = 1e−8)	0.8783	0.8669	780.05s
3	Low β_2 (β_1 = 0.9, β_2 = 0.99, ε = 1e−8)	0.8876	0.8661	875.36s
4	Large ε (β_1 = 0.9, β_2 = 0.999, 1e-6)	0.8743	0.8671	796.13s
5	Aggressive Smoothing (β_1=0.99, β_2=0.9999, ε=1e-7)	0.8783	0.8746	808.05s

Figure 2 : Val Accuracy & Training Accuracy Comparison

Table 6: Baseline experiment

Epoch	Train Accuracy	Train Loss	Validation Accuracy	Validation Loss	Training Time per Epoch
1	0.5128	1.4766	0.8241	0.5490	101s
2	0.7796	0.6507	0.8571	0.4573	77s
3	0.8074	0.5397	0.8598	0.4629	81s
4	0.8347	0.5023	0.8571	0.4200	80s
5	0.8475	0.4241	0.8690	0.3961	79s
6	0.8491	0.4141	0.8823	0.3613	77s
7	0.8495	0.4167	0.8757	0.3710	76s
8	0.8632	0.4037	0.8876	0.3556	83s
9	0.8633	0.4072	0.8743	0.3677	78s
10	0.8540	0.4046	0.8717	0.3769	76s

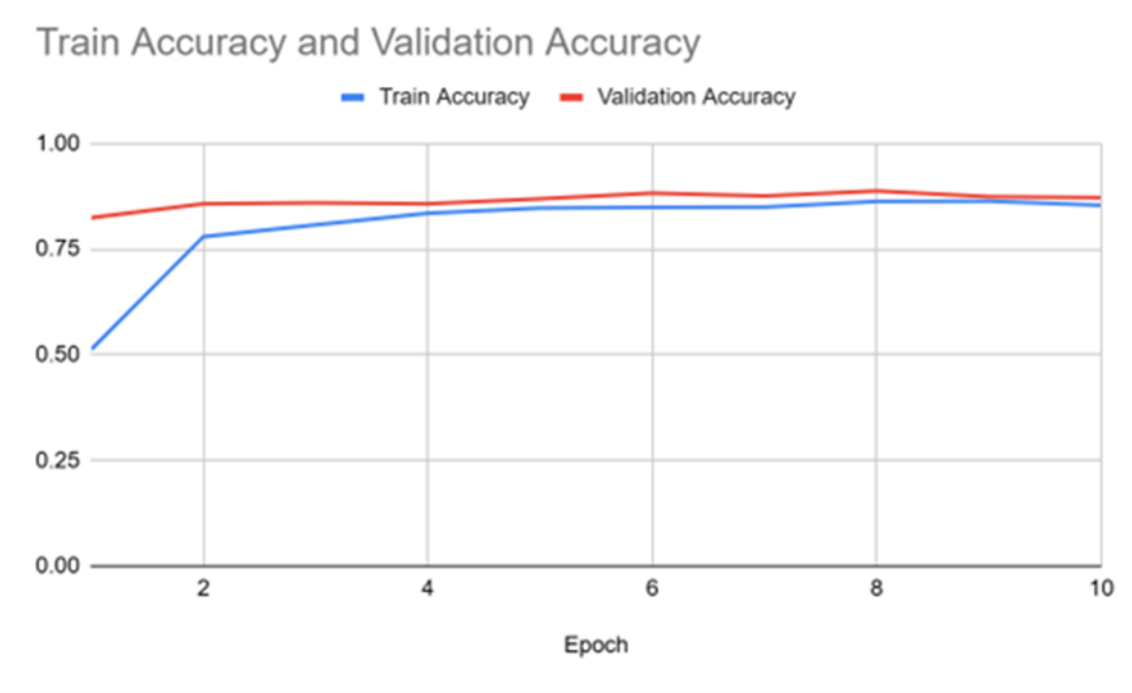

Figure 3 : Baseline Accuracy Comparison

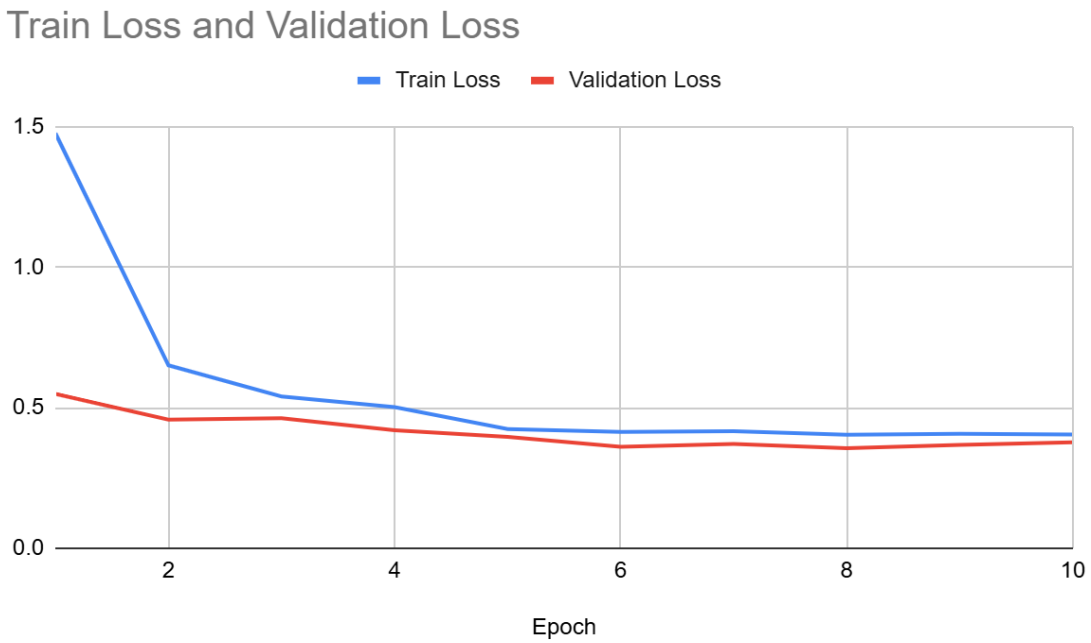

Figure 4: Baseline Loss Comparison

Table 7: High Beta 1 Experiment

Epoch	Train Accuracy	Train Loss	Validation Accuracy	Validation Loss	Training Time per Epoch
1	0.5209	1.4061	0.8095	0.5361	85s 476ms
2	0.7966	0.6224	0.8439	0.4613	78s 462ms
3	0.8216	0.5022	0.8519	0.4452	77s 457ms
4	0.8294	0.5016	0.8598	0.4104	81s 478ms
5	0.8446	0.4566	0.8585	0.4174	77s 459ms
6	0.8495	0.4397	0.8717	0.3936	77s 456ms
7	0.8493	0.4229	0.8783	0.3701	76s 456ms
8	0.8476	0.4032	0.8783	0.3683	78s 464ms
9	0.8628	0.3780	0.8585	0.4042	76s 452ms
10	0.8666	0.3889	0.8743	0.3888	77s 459ms

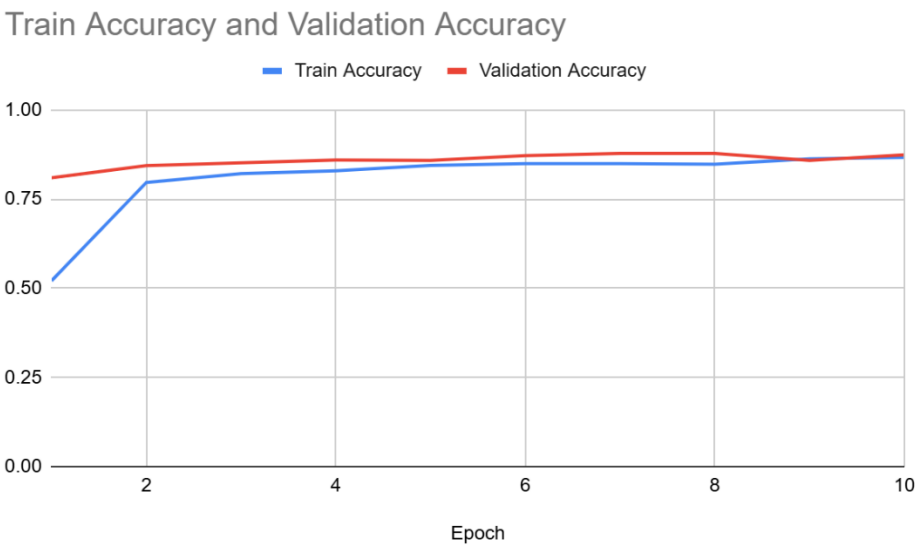

Figure 5: **High Beta 1** ***Accuracy Comparison***

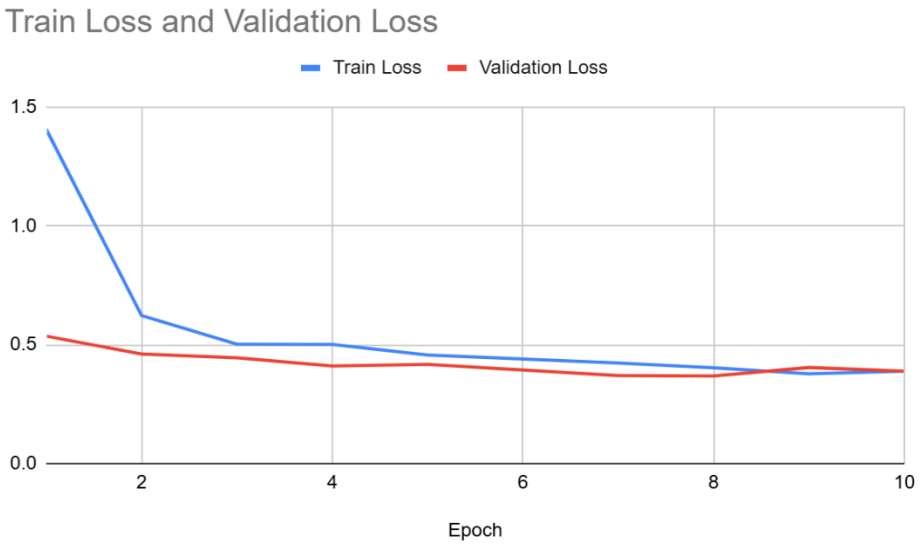

Figure 6: **High Beta 1** ***Loss Comparison***

Table 8: Low Beta 2 Experiment

Epoch	Training Accuracy	Training Loss	Val Accuracy	Val Loss	Training Time per Epoch
1	0.4953	1.5015	0.8175	0.5237	91s 494ms
2	0.7916	0.6185	0.8519	0.4297	89s 477ms
3	0.8257	0.5098	0.8466	0.4379	78s 466ms
4	0.8328	0.4814	0.869	0.3957	79s 469ms
5	0.8478	0.4404	0.8677	0.3567	82s 489ms
6	0.8498	0.4282	0.8651	0.3867	138s 464ms
7	0.8468	0.4327	0.8876	0.3464	83s 470ms
8	0.8597	0.394	0.873	0.3686	82s 488ms
9	0.8623	0.3883	0.873	0.364	81s 481ms
10	0.8582	0.4205	0.881	0.377	89s 477ms

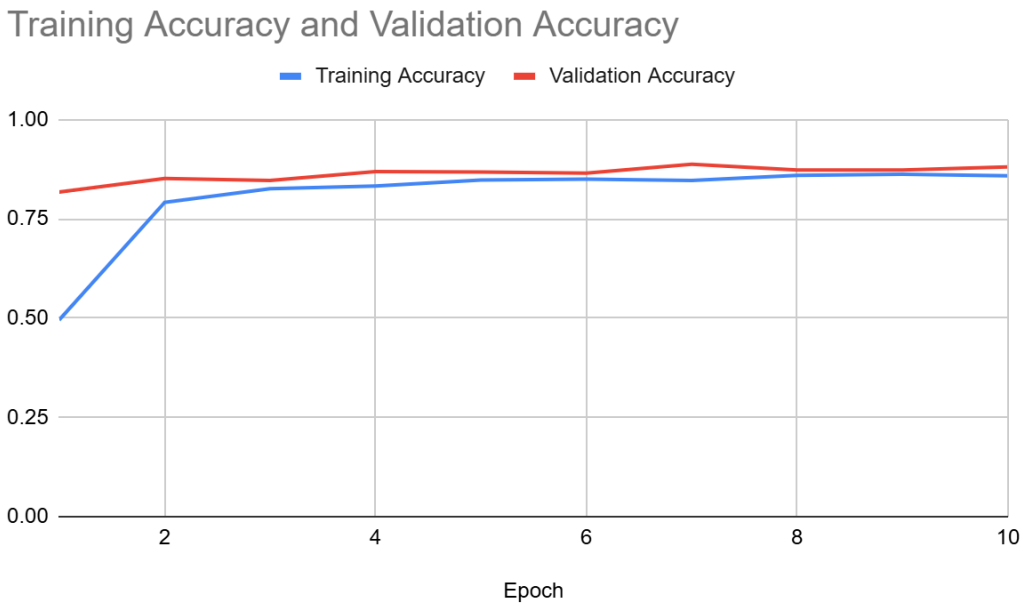

Figure 7: **Low Beta 2** ***Accuracy Comparison***

Figure 8: **Low Beta 2** ***Loss Comparison***

Table 9: Large Epsilon Experiment

Epoch	Training Accuracy	Training Loss	Val Accuracy	Val Loss	Training Time per Epoch
1	0.4994	1.5179	0.8082	0.5678	89s 494ms
2	0.7830	0.5975	0.8280	0.4861	77s 460ms
3	0.8218	0.5299	0.8492	0.4548	78s 468ms
4	0.8388	0.4892	0.8743	0.3804	77s 459ms
5	0.8474	0.4449	0.8743	0.3961	77s 457ms
6	0.8454	0.4302	0.8730	0.3999	81s 483ms
7	0.8550	0.4056	0.8664	0.4018	79s 468ms
8	0.8661	0.3982	0.8545	0.4185	79s 474ms
9	0.8546	0.4014	0.8651	0.3779	78s 468ms
10	0.8505	0.4248	0.8704	0.3818	81s 481ms

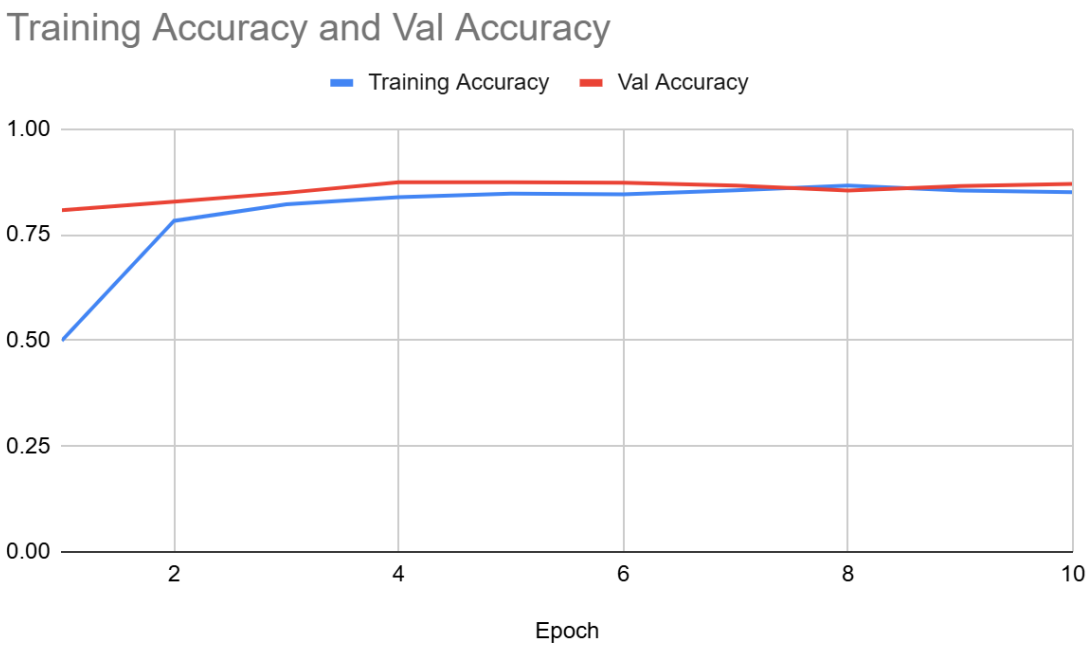

Figure 9: Large Epsilon Accuracy Comparison

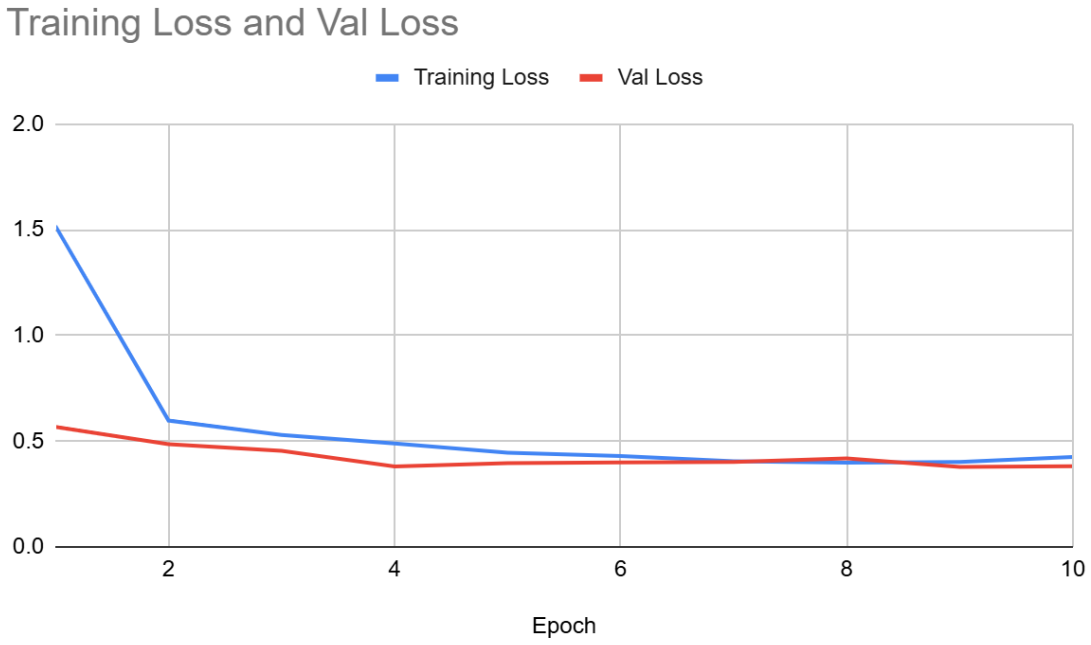

Figure 10: Large Epsilon Loss Comparison

Table 9: Very High Smoothing Experiment

Epoch	Training Accuracy	Training Loss	Val Accuracy	Val Loss	Training Time per Epoch
1	0.5221	1.4792	0.8029	0.5429	89s 496ms
2	0.7936	0.6398	0.8426	0.4567	80s 479ms
3	0.8423	0.5050	0.8466	0.4069	80s 476ms
4	0.8288	0.4970	0.8585	0.3867	84s 500ms
5	0.8532	0.4440	0.8743	0.3369	78s 464ms
6	0.8593	0.4033	0.8677	0.3424	78s 464ms
7	0.8632	0.4116	0.8783	0.3380	77s 462ms
8	0.8707	0.3863	0.8690	0.3422	77s 461ms
9	0.8518	0.4203	0.8598	0.3568	82s 464ms
10	0.8584	0.4034	0.8545	0.3768	78s 463ms

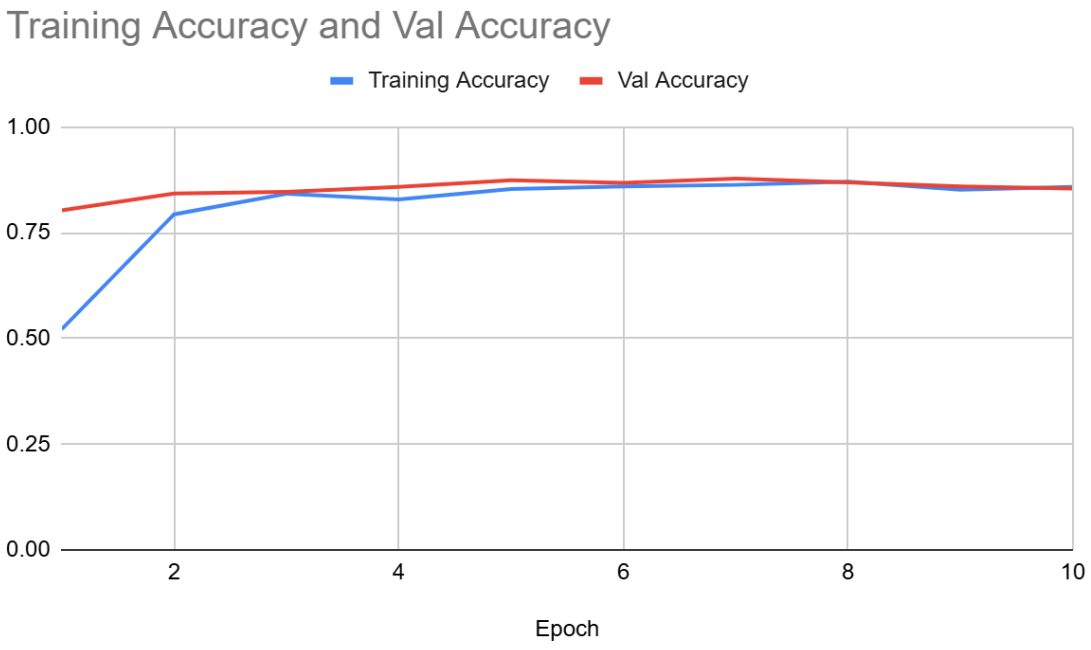

Figure 11: Very High Smoothing Accuracy Comparison

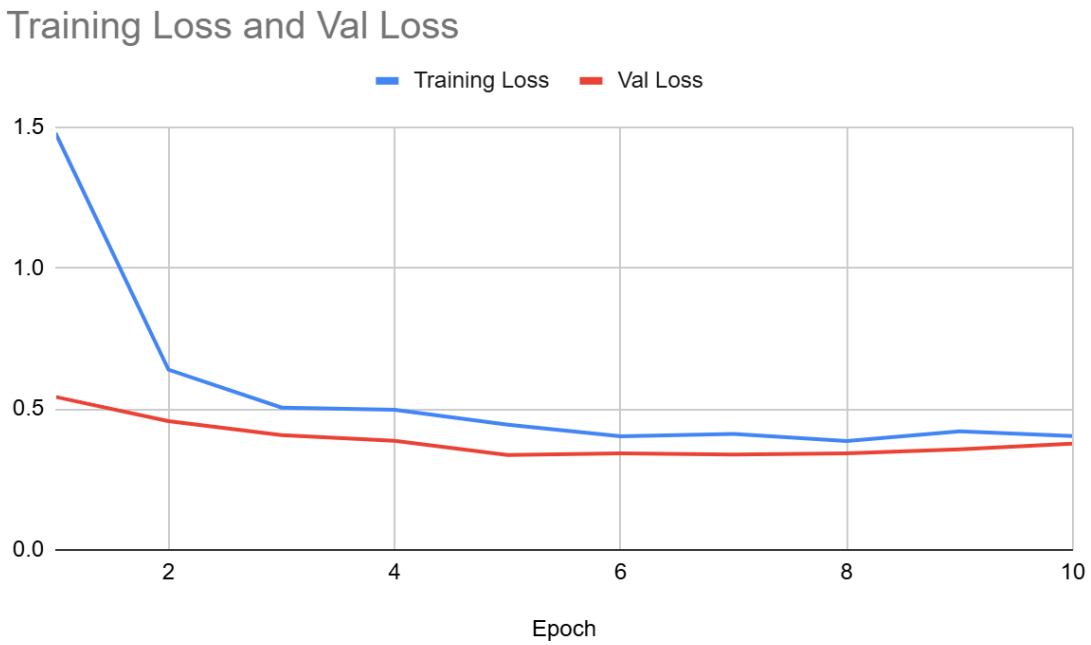

Figure 12: Very High Smoothing Loss Comparison

Outputs

```
Running experiment: baseline
Epoch 1/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 101s 568ms/step - accuracy: 0.5128 - loss: 1.4766 - val_accuracy: 0.8241 - val_loss: 0.5490
Epoch 2/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 77s 453ms/step - accuracy: 0.7796 - loss: 0.6507 - val_accuracy: 0.8571 - val_loss: 0.4573
Epoch 3/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 81s 480ms/step - accuracy: 0.8074 - loss: 0.5397 - val_accuracy: 0.8598 - val_loss: 0.4629
Epoch 4/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 80s 471ms/step - accuracy: 0.8347 - loss: 0.5023 - val_accuracy: 0.8571 - val_loss: 0.4200
Epoch 5/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 79s 468ms/step - accuracy: 0.8475 - loss: 0.4241 - val_accuracy: 0.8690 - val_loss: 0.3961
Epoch 6/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 77s 457ms/step - accuracy: 0.8491 - loss: 0.4141 - val_accuracy: 0.8823 - val_loss: 0.3613
Epoch 7/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 76s 451ms/step - accuracy: 0.8495 - loss: 0.4167 - val_accuracy: 0.8757 - val_loss: 0.3710
Epoch 8/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 83s 460ms/step - accuracy: 0.8632 - loss: 0.4037 - val_accuracy: 0.8876 - val_loss: 0.3556
Epoch 9/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 78s 466ms/step - accuracy: 0.8633 - loss: 0.4072 - val_accuracy: 0.8743 - val_loss: 0.3677
Epoch 10/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 76s 455ms/step - accuracy: 0.8540 - loss: 0.4046 - val_accuracy: 0.8717 - val_loss: 0.3769
Finished: baseline (Test Acc: 0.8637)
```

Figure 13: Baseline Experiment

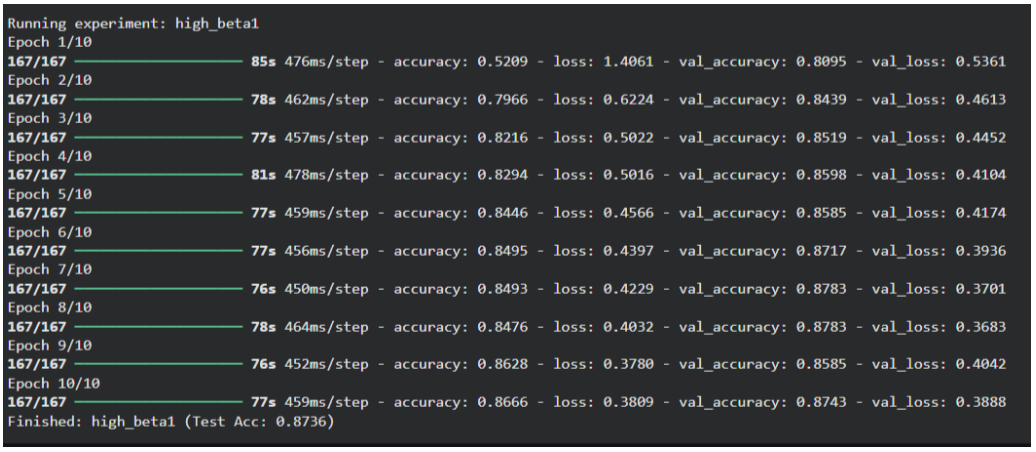

Figure 14: High Beta 1 Experiment

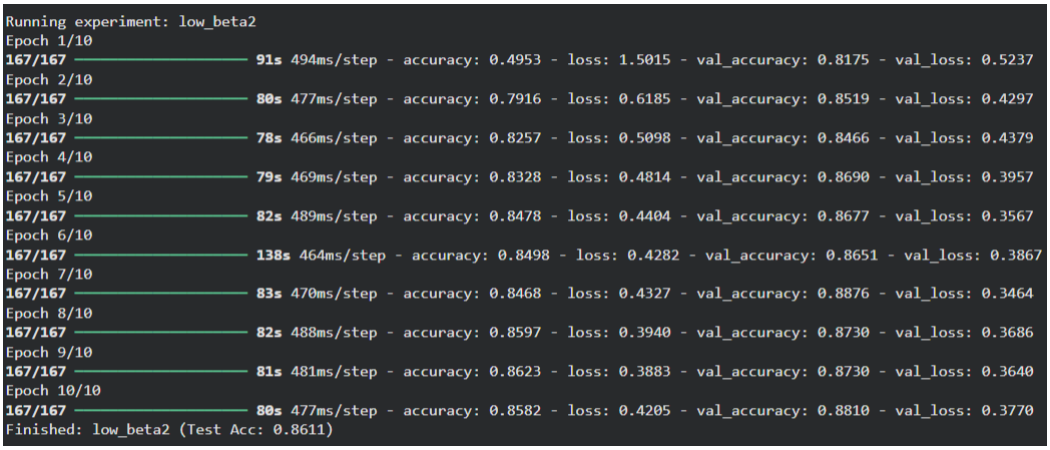

Figure 15: Low Beta 2 Experiment

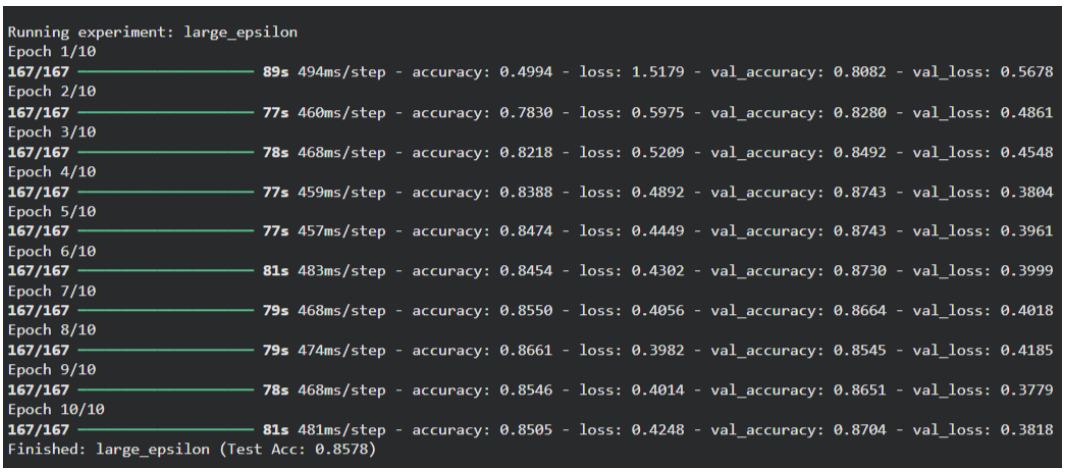

Figure 16: Large Epsilon Experiment

```
Running experiment: very_high_smoothing
Epoch 1/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 89s 496ms/step - accuracy: 0.5221 - loss: 1.4792 - val_accuracy: 0.8029 - val_loss: 0.5429
Epoch 2/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 80s 479ms/step - accuracy: 0.7936 - loss: 0.6398 - val_accuracy: 0.8426 - val_loss: 0.4567
Epoch 3/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 80s 476ms/step - accuracy: 0.8423 - loss: 0.5050 - val_accuracy: 0.8466 - val_loss: 0.4069
Epoch 4/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 84s 500ms/step - accuracy: 0.8288 - loss: 0.4970 - val_accuracy: 0.8585 - val_loss: 0.3867
Epoch 5/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 78s 464ms/step - accuracy: 0.8532 - loss: 0.4440 - val_accuracy: 0.8743 - val_loss: 0.3369
Epoch 6/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 78s 464ms/step - accuracy: 0.8593 - loss: 0.4033 - val_accuracy: 0.8677 - val_loss: 0.3424
Epoch 7/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 77s 462ms/step - accuracy: 0.8632 - loss: 0.4116 - val_accuracy: 0.8783 - val_loss: 0.3300
Epoch 8/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 77s 461ms/step - accuracy: 0.8707 - loss: 0.3863 - val_accuracy: 0.8690 - val_loss: 0.3422
Epoch 9/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 82s 464ms/step - accuracy: 0.8518 - loss: 0.4203 - val_accuracy: 0.8598 - val_loss: 0.3568
Epoch 10/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 78s 463ms/step - accuracy: 0.8584 - loss: 0.4034 - val_accuracy: 0.8545 - val_loss: 0.3768
Finished: very_high_smoothing (Test Acc: 0.8545)
```

Figure 17: Very Hight Smoothing Experiment

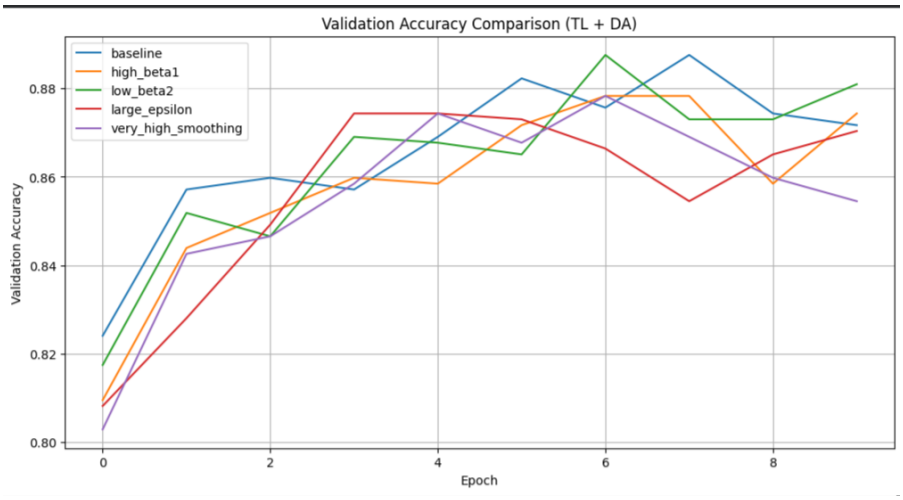

Figure 18: Validation Accuracy Comparison

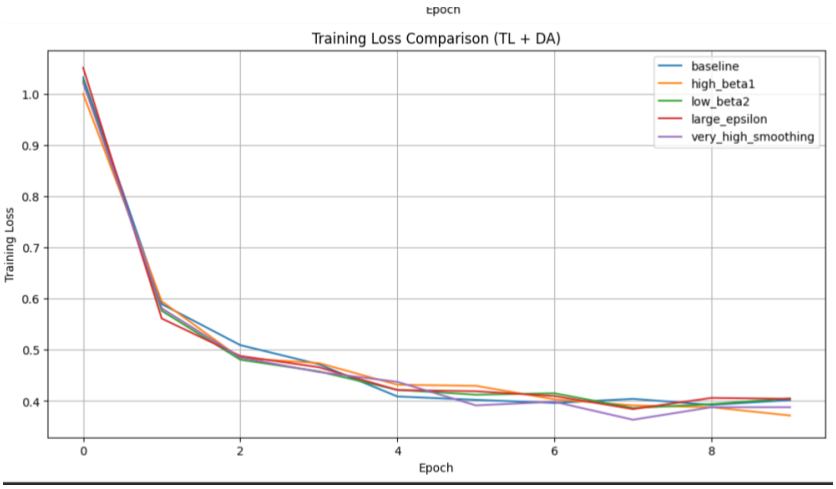

Figure 19: Training Loss Comparison

6. Conclusion and Future Work

6.1. Conclusion

After testing, the normal Adam settings work best. They got 88.76% correct on new data and trained in about 13 minutes. Some changes made training faster or improved scores during training, but they didn't work as well on new, real data. That's why the standard settings are the safest choice. Yes, small changes in the settings matter a lot. Turning up the speed setting made training faster but lowered accuracy. Turning down another setting kept accuracy high but made training take longer. It's always a balance: you give up one thing to get another — speed, accuracy, or stability. So, if you want good results on real data without waiting too long, just use the default Adam. If you care more about speed, you can change things but test carefully so you don't lose accuracy when it really matters.

6.2. Future Work

We still have a lot we could test in the future:

- **Try more numbers**: We only checked a few settings. We could test more values for the optimizer's controls to see if we can find an even better mix of speed and accuracy.
- **Test on other kinds of data**: All our tests were on one dataset. It would be cool to see if we get the same results on different data or other tasks.
- **Mix changes together**: We changed one thing at a time. Next time, we could combine changes like turning one knob up and another down to see if we can get the good parts of both.
- **Watch training more closely** : We only looked at the final score and time. Watching how the model learns step-by-step could help us understand why some settings work better than others.
- **Try different optimizers** : We only used Adam. We could also try other training methods like SGD or RMSprop to see if they do an even better job.
- **Let the computer search for us** : Instead of trying everything by hand, we could use tools that automatically look for the best settings. This might save time and find combinations we hadn't thought of.

By trying these ideas, we can make models that work better, train faster, and are more dependable in real use.

7. References

1. Y. Gai and H. Wang, “Plant Disease: A Growing Threat to Global Food Security,” Agronomy, vol. 14, no. 8, pp. 1615, 2024.

2. S. Konduru, M. Amiruzzaman, V. D. Avina, and M. R. Islam, “Plant Disease Detection and Classification Using Deep Learning Models,” ResearchGate, 2023.

3. B. Yang, M. Li, F. Li, and H. Wang, “A Novel Plant Type, Leaf Disease, and Severity Identification Framework Using CNN and Transformer with Multi-label Method,” Scientific Reports, vol. 14, no. 1, pp. 116644, 2024.

4. M.H. Saleem, J. Pot- gieter, and K. M. Arif, “Plant Disease Classification: A Comparative Evaluation of Convo- lutional Neural Networks and Deep Learning Optimizers,” Plants, vol. 9, no. 10, pp. 1319, 2020.

5. H. Farman, J. Ahmad, B. Jan, Y. Shahzad, M. Abdullah, and S. Ali, “Efficient-Net-based Robust Recognition of Peach Plant Diseases in Field Images,” Comput. Mater. Contin., vol. 71, no. 1, pp. 2073–2089, 2022.

6. P. P. Fathimathul Rajeena, S. U. Aswathy, M. A. Moustafa, and M. A. S. Ali, “Detecting Plant Disease in Corn Leaf Using EfficientNet Architecture—An Analytical Approach,” Electronics, vol. 12, no. 8, pp.1938, 2023.

7. M. M. Alqahtani, A. K. Dutta, S. Almotairi, M. Ilayaraja, A. A. Albraikan, F. N. Al-Wesabi and M. Al Duhayyim, “Sailfish Optimizer with EfficientNet Model for Apple Leaf Disease Detection,” Computers, Materials & Continua, vol. 75, no. 1, pp. 217-232, 2023.

8. S. Ruder, “An Overview of Gradient Descent Optimization Algorithms,” arXiv Preprint arXiv., 2016.

9. J. Heaton, “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learn-ing: The MIT Press, 2016, 800 pp, ISBN: 0262035618,” Genet. Program. Evolvable Mach., vol. 19, no. 1, pp. 305 – 307, 2018.

10. A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, “The Marginal Value of Adaptive Gradient Methods in Machine Learning,” Ad- vances in neural information processing systems, vol. 30, 2017.

11. R. Y. Sun, “Optimiza- tion for Deep Learning: An Overview,” Journal of the Operations Research Society of China, vol. 8, no. 2, pp. 249-294, 2020.

12. E. M. Dogo, O. J. Afolabi, and B. Twala, “On Convolutional Neural Networks with Varying Depth and Width for Image Classification,” Appl. Sci., vol. 12, no. 23, pp. 11976, 2022.

13. P. Singh, P. Singh, U. Farooq, S. S. Khurana, J. K. Verma, and M. Kumar, “CottonLeafNet: Cotton Plant Leaf Disease Detection Using Deep Neural Networks,” Multimed. Tools Appl., vol. 82, no. 24, pp. 37151–37176, 2023.

14. İ. Kunduracıoğlu and İ. Paçal, “Deep Learning-based Disease Detection in Sugarcane

Leaves: Evaluating EfficientNet Models," J. Oper. Intell., vol. 2, no. 1, pp. 321–335, 2024.

15. L. T. Duong, P. T. Nguyen, C. Di Sipio, and D. Di Ruscio, "Automated Fruit Recognition using EfficientNet and MixNet," Comput. Electron. Agric., vol. 171, pp. 105326, 2020.

16. W. Shafik, A. Tufail, C. De Silva Liyanage, and R. A. A. H. M. Apong, "Using Transfer Learning-based Plant Disease Classification and Detection for Sustainable Agriculture," BMC Plant Biology, vol. 24, no.1, pp. 136, 2024.

17. I. Khan, S. S. Sohail, D. Ø. Madsen, and B. K. Khare, "Deep Transfer Learning for Fine-Grained Maize Leaf Disease Classification," J. Agric. Food Res., vol. 16, pp. 101148, 2024.

18. S. Choubey and Divya, "Lightweight Federated Transfer Learning for Plant Leaf Disease Detection and Classification Across Multiclient Cross-Silo Datasets," in BIO Web of Conferences, vol. 82, pp. 05018, 2024.

19. S. P. Mohanty, D. P. Hughes, and M. Salathe, "Using Deep Learning for Image-based Plant Disease Detection," Front. Plant Sci., vol. 7, pp. 1419, 2016.

.