Project Final Report

Course: Advanced Special Topics (Optimization).

By

Harshith Kasthuri

Chandan Manabolu Narayana

Sricharan Muttineni

Motivation

Deep learning has revolutionized machine learning across various domains, but its success heavily relies on optimization algorithms that ensure efficient training and generalization. Traditionally, optimization algorithms like SGD, AdamW, and RMSprop have been manually designed, relying on fixed heuristics and assumptions about gradient-based learning. While effective in certain scenarios, these methods often struggle to adapt to diverse tasks, architectures, and datasets. This limitation highlights the need for a novel approach to discover optimization algorithms that can generalize better and perform more efficiently.

The vast search space of possible optimization algorithms presents an opportunity for innovation but also poses a significant challenge to manual exploration. Symbolic discovery methods leverage automated search techniques, such as AutoML, to systematically explore this space. By automating the design process, symbolic discovery can uncover algorithms that outperform manually designed counterparts while minimizing human bias and effort.

A key advantage of symbolic discovery lies in its use of interpretable mathematical representations. Unlike many black-box optimization methods, symbolic discovery provides insights into the underlying principles that drive an algorithm's success. This transparency not only aids in understanding but also facilitates further refinement and adaptability across new challenges.

The Lion Optimizer (Evolved Sign Momentum) showcases how symbolic discovery can revolutionize the design of optimization algorithms for machine learning. It was created to address key challenges in the traditional optimizers, such as the slow convergence of SGD and the generalization issues of AdamW.

Our objective in this project has been to evaluate and compare the performance of RMSProp, AdamW, and the recently discovered Lion Optimizer for vision-based deep learning models. Optimization algorithms are critical in determining training efficiency, convergence speed, and model accuracy, especially for complex architectures like Vision Transformers and EfficientNet. By comparing these optimizers, we aim to understand their strengths and weaknesses, assess the impact of automated discovery methods like Lion, and identify the most effective strategies for training vision-based models.

Background

1. RMSProp

RMSProp is an optimization algorithm designed to handle the challenges of non-stationary objectives often encountered in deep learning problems. By maintaining a moving average of the squared gradients, RMSProp dynamically adjusts the learning rates for each parameter. This feature makes it particularly effective for RNNs and other deep models, as it helps stabilize training by addressing issues like exploding or vanishing gradients. However, RMSProp has its limitations, especially when applied to large-scale architectures, such as those used in vision-based tasks. It may exhibit instability or slower convergence in these scenarios, making it less ideal for modern deep-learning models that require highly efficient and scalable optimizers.

2. AdamW

AdamW builds upon the widely used Adam optimizer by introducing a decoupled weight decay mechanism. This decoupling prevents the weight decay term from interfering with the gradient updates, leading to improved optimization and better generalization performance. AdamW's adaptive learning rating enables it to converge faster than traditional optimizers, making it a popular choice across various deep learning tasks and architectures.

Algorithm 1 AdamW Optimizer

```
given $\beta_1, \beta_2, \epsilon, \lambda, \eta, f$
initialize $\theta_0, m_0 \leftarrow 0, v_0 \leftarrow 0, t \leftarrow 0$
while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
    update EMA of $g_t$ and $g_t^2$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    bias correction
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
    update model parameters
    $\theta_t \leftarrow \theta_{t-1} - \eta_t(\hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda\theta_{t-1})$
end while
return $\theta_t$
```

3. Program Search Space and Optimization Discovery

A program search space is a structured set of mathematical operations and rules that can be combined to define optimization algorithms. In this space:

1. Flexibility: The search space is designed to enable the discovery of diverse and novel optimization algorithms.
2. Simplicity: Programs are easily integrated into machine learning workflows, typically resembling Python code with libraries like NumPy or JAX.
3. Algorithmic Design: The focus remains on high-level algorithmic design rather than low-level implementation details.

Using symbolic discovery techniques, optimization algorithms are represented as functions operating over n-dimensional arrays, allowing the search process to explore and evaluate many potential solutions. For example, AdamW's update rule can be expressed as a simple program in this framework, enabling its use as a starting point for further exploration.

4. Efficient Search Techniques for Optimization Discovery

The infinite and sparse nature of program search spaces poses significant challenges, as high-performing algorithms are rare and difficult to find. To address this, efficient search techniques like regularized evolutionary search and abstract execution are employed:

- **Regularized Evolution**: This method improves algorithms through iterative cycles of mutation and selection. Starting with AdamW as a baseline, the search process identifies promising candidates, gradually refining them through warm-starting, restarts, and hyperparameter tuning.
- **Abstract Execution**: To reduce redundancies and invalid programs, this technique prunes the search space by detecting syntax errors, eliminating redundant statements, and caching duplicate programs. This significantly reduces the computational cost of the search process.

5. Generalization and Program Selection

One of the greatest challenges in discovering optimization algorithms is ensuring their ability to generalize across diverse tasks and architectures. The search process often relies on proxy tasks, which are smaller and less computationally expensive, to evaluate candidate algorithms. However, a generalization gap arises when algorithms perform well on proxy tasks but fail on larger, more complex target tasks.

To address this, a funnel selection process is used in this paper, where algorithms are progressively tested on increasingly larger meta-validation tasks. This ensures that only the most robust algorithms, capable of generalizing to diverse tasks and architectures, are selected. Simplification techniques further refine the discovered programs by removing redundant or minimally impactful statements, resulting in algorithms that are not only effective but also interpretable and easy to implement.

6. Lion Optimizer

The Lion optimizer (Evolved Sign Momentum) is a recently discovered optimisation algorithm generated through symbolic discovery using a program search space []. Unlike traditional gradient-based optimisers, Lion employs a sign-based momentum update, which significantly reduces computational complexity while maintaining competitive accuracy. This unique update rule makes Lion lightweight and efficient, aligning well with the computational demands of modern deep-learning models.

Algorithm 2 Lion Optimizer (ours)

```
given $\beta_1$, $\beta_2$, $\lambda$, $\eta$, $f$
initialize $\theta_0$, $m_0 \leftarrow 0$
while $\theta_t$ not converged do
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
    update model parameters
    $c_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
    $\theta_t \leftarrow \theta_{t-1} - \eta_t(\text{sign}(c_t) + \lambda\theta_{t-1})$
    update EMA of $g_t$
    $m_t \leftarrow \beta_2 m_{t-1} + (1-\beta_2)g_t$
end while
return $\theta_t$
```

Program 2: An example training loop, where the optimization algorithm that we are searching for is encoded within the train function. The main inputs are the weight (w), gradient (g) and learning rate schedule (lr). The main output is the update to the weight. v1 and v2 are two additional variables for collecting historical information.

```
w = weight_initialize()
v1 = zero_initialize()
v2 = zero_initialize()
for i in range(num_train_steps):
  lr = learning_rate_schedule(i)
  g = compute_gradient(w, get_batch(i))
  update, v1, v2 = train(w, g, v1, v2, lr)
  w = w - update
```

Program 3: Initial program (AdamW). The bias correction and ϵ are omitted for simplicity.

```
def train(w, g, m, v, lr):
  g2 = square(g)
  m = interp(g, m, 0.9)
  v = interp(g2, v, 0.999)
  sqrt_v = sqrt(v)
  update = m / sqrt_v
  wd = w * 0.01
  update = update + wd
  lr = lr * 0.001
  update = update * lr
  return update, m, v
```

Program 4: Discovered program after search, selection and removing redundancies in the raw Program 8. Some variables are renamed for clarity.

```
def train(w, g, m, v, lr):
  g = clip(g, lr)
  g = arcsin(g)
  m = interp(g, v, 0.899)
  m2 = m * m
  v = interp(g, m, 1.109)
  abs_m = sqrt(m2)
  update = m / abs_m
  wd = w * 0.4602
  update = update + wd
  lr = lr * 0.0002
  m = cosh(update)
  update = update * lr
  return update, m, v
```

Main Idea and Implementation Details

The primary focus of this project was to evaluate and compare the performance of three optimization algorithms—**RMSProp**, **AdamW**, and **Lion**—on fine-tuning tasks for pre-trained deep learning models. These optimizers were tested on three state-of-the-art architectures:

1. **EfficientNetV2**:
   A convolutional neural network known for its highly efficient scaling and use of **Fused-MBConv layers** for faster inference and training. In this study, EfficientNetV2 was pre-trained on the **ImageNet-21K** dataset and fine-tuned on the **CIFAR-100** dataset [2].
2. **VisionTransformer(ViT-B/32)**:
   A transformer-based architecture that achieves exceptional performance on image classification tasks by leveraging **patch-based self-attention mechanisms**. ViT-B/32 was pre-trained on **ImageNet-21K** and fine-tuned on the **CIFAR-10** dataset [3].
3. **Mixer-B/16**:
   An **MLP-Mixer** architecture that uses token-mixing and channel-mixing MLP layers to capture dependencies across patches and channels. Mixer-B/16 was also pre-trained on **ImageNet-21K** and fine-tuned on the **CIFAR-10** dataset [4].

This setup enabled a comprehensive evaluation of the optimizers across distinct architectural paradigms—convolutional (EfficientNetV2), transformer-based (ViT-B/32), and MLP-based (Mixer-B/16). By fine-tuning these pre-trained models on CIFAR datasets, we primarily assessed the accuracy and performance of the Lion optimizer compared to RMSProp and AdamW, analyzing its adaptability, generalization, and efficiency in various contexts.

1) Fine Tuning EfficientNetV2 for CIFAR-100

EfficientNetV2 is a state-of-the-art image classification model designed to optimize both training speed and accuracy. It introduces two key block types: MBConv, which uses depthwise separable convolutions and squeeze-and-excitation (SE) blocks to reduce computation, and Fused-MBConv, which replaces depthwise separable convolutions with standard convolutions for faster processing in early layers. This combination allows EfficientNetV2 to balance computational efficiency and performance, making it highly suitable for large-scale vision tasks. By selectively applying Fused-MBConv in shallow layers and MBConv in deeper layers, the model achieves an optimal trade-off between speed and accuracy.

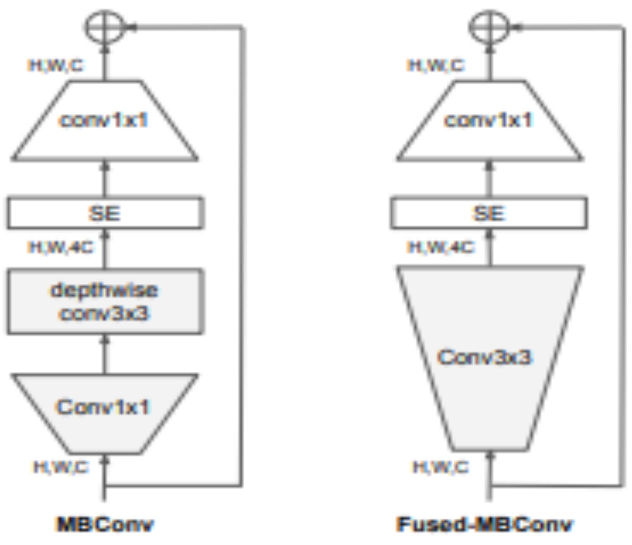


Figure 2. Structure of MBConv and Fused-MBConv.

We followed the same procedure for finetuning outlined in the GitHub repository associated with this paper. Specifically, we ran their code with modifications to use different optimizers [5].

Fine-tuning involves initializing the model with pre-trained weights and adapting it to a new dataset, in our case, using different optimizers. The code snippet below illustrates how this is achieved.

```
if config.train.ft_init_ckpt:  # load pretrained ckpt for finetuning.
  model(tf.keras.Input([None, None, 3]))
  ckpt = config.train.ft_init_ckpt
  utils.restore_tf2_ckpt(model, ckpt, exclude_layers=('_fc', 'optimizer'))
```

Loading Pre-Trained Checkpoints:

- The condition if config.train.ft_init_ckpt checks whether a pre-trained checkpoint is specified for fine-tuning.
- The variable ckpt holds the path to the pre-trained checkpoint, provided in the configuration.

Model Initialization:

- The model(tf.keras.Input([None, None, 3])) line initializes the model, specifying the input shape as (None, None, 3) for images (height, width, and 3 channels for RGB).

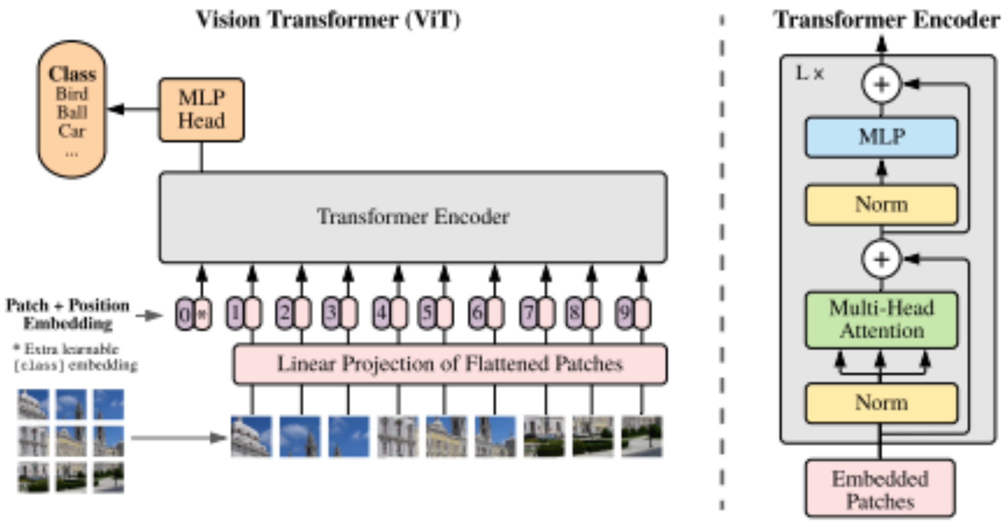
Restoring Weights:

- The function utils.restore_tf2_ckpt(model, ckpt, exclude_layers=['_fc', 'optimizer']) is used to restore weights from the checkpoint into the model.

By excluding task-specific layers like _fc and the optimizer, the fine-tuning process focuses on adapting the model to the target dataset CIFAR-100, using the specified optimizer (RMSProp, AdamW, or Lion).

2) Fine-tuning ViT-B/32 for CIFAR-10

The Vision Transformer (ViT-B/32) is a transformer-based architecture designed for image recognition tasks, leveraging the power of self-attention mechanisms traditionally used in natural language processing. Unlike convolutional neural networks, ViT processes images as sequences of patches, treating each patch as a token. Specifically, ViT-B/32 divides an image into fixed-size patches of 32x32 pixels, which are then flattened and projected into a latent space. These patch embeddings, along with positional encodings, are fed into a standard transformer encoder.



We followed the same procedure for fine-tuning outlined in the GitHub repository associated with this paper. Specifically, we ran their code with modifications to use different optimizers [6].

Procedure Followed to Fine-Tune Vision Transformer(ViT):

i) Pre-trained Weights Loaded:

- Pre-trained weights (params) from the checkpoint were loaded as a strong initialization for fine-tuning.

ii) Training Steps and Learning Rate Schedule:

- Total training steps were set using config.total_steps. A dynamic learning rate schedule (lr_fn) with a base learning rate, cosine decay, and warmup steps was created for stable training.

iii) Optimizers Implemented:

- RMSProp, AdamW, and Lion optimizers were applied with gradient clipping to compare their adaptability, efficiency, and generalization capabilities.

iv) Update Function:

- An update function (update_fn_repl) was used to compute the training loss, calculate gradients, and update model parameters.

v) Checkpoints and Replication:

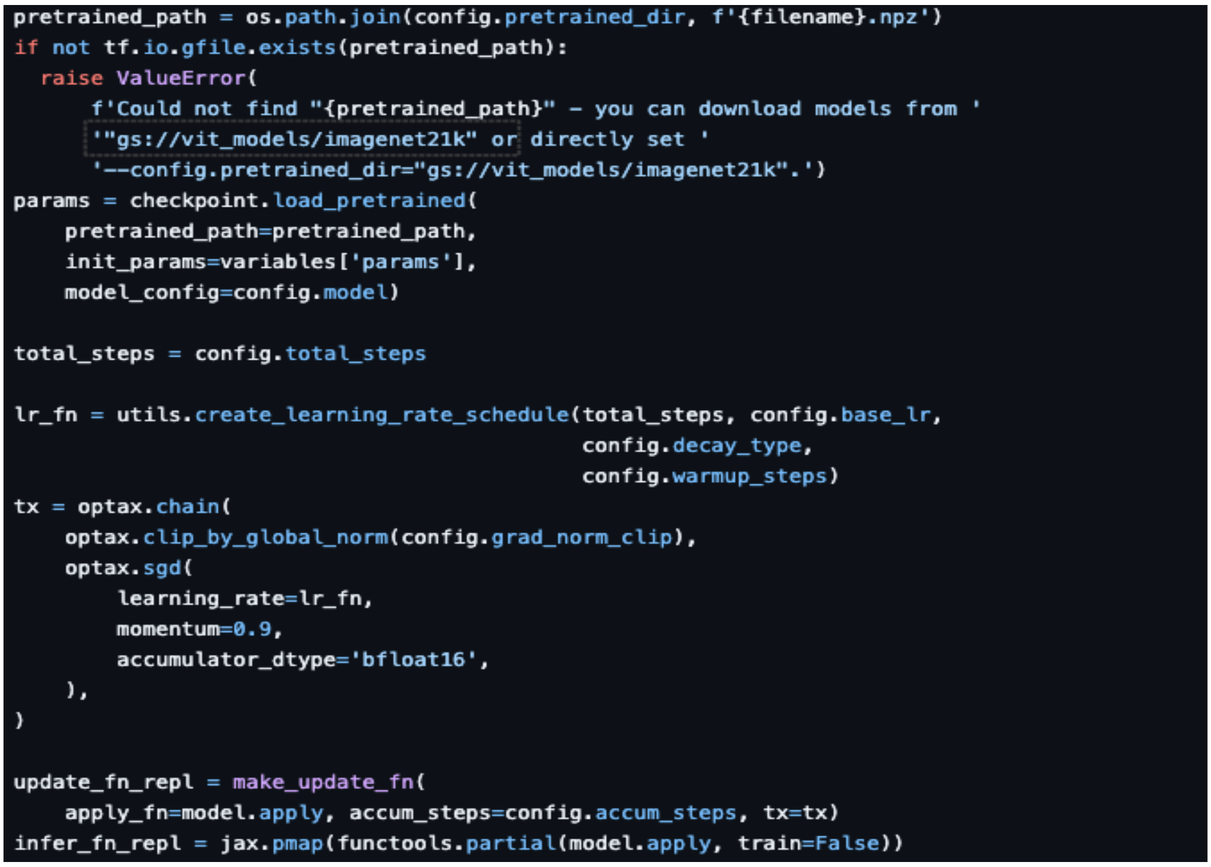
- Checkpoints were restored for parameters and optimizer states, and these were replicated across devices for distributed training.

vi) Training Loop and Evaluation:

- Batches from the training dataset were processed to fine-tune the model. Periodic evaluations on the test dataset were performed to track accuracy and performance.

vii) Checkpoints Saved:

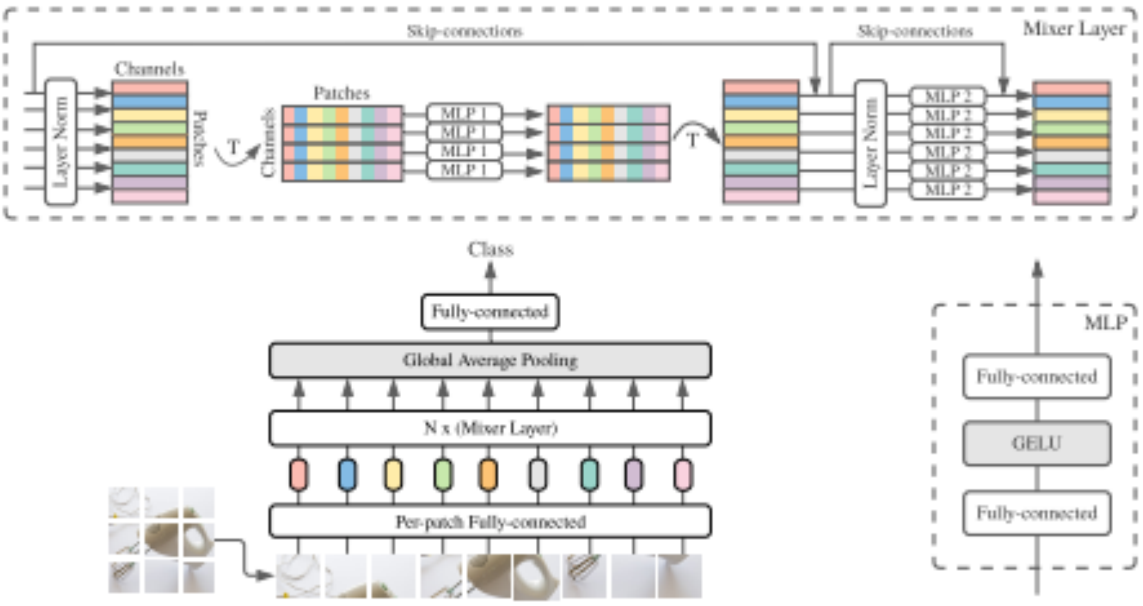
- Model parameters and optimizer states were saved periodically to resume training if needed.



```
pretrained_path = os.path.join(config.pretrained_dir, f'{filename}.npz')
if not tf.io.gfile.exists(pretrained_path):
  raise ValueError(
      f'Could not find "{pretrained_path}" - you can download models from '
      '"gs://vit_models/imagenet21k" or directly set '
      '--config.pretrained_dir="gs://vit_models/imagenet21k".')
params = checkpoint.load_pretrained(
    pretrained_path=pretrained_path,
    init_params=variables['params'],
    model_config=config.model)

total_steps = config.total_steps

lr_fn = utils.create_learning_rate_schedule(total_steps, config.base_lr,
                                            config.decay_type,
                                            config.warmup_steps)
tx = optax.chain(
    optax.clip_by_global_norm(config.grad_norm_clip),
    optax.sgd(
        learning_rate=lr_fn,
        momentum=0.9,
        accumulator_dtype='bfloat16',
    ),
)

update_fn_repl = make_update_fn(
    apply_fn=model.apply, accum_steps=config.accum_steps, tx=tx)
infer_fn_repl = jax.pmap(functools.partial(model.apply, train=False))
```

3) Fine-tuning Mixer-B/16 for CIFAR-10

Mixer-B/16 is an MLP-Mixer architecture that replaces traditional convolutional or self-attention layers with fully connected layers, making it a unique approach to image recognition. The model begins by dividing input images into non-overlapping patches of size 16x16, which are linearly embedded and processed through a series of "Mixer Layers." Each Mixer Layer has two components: a token-mixing MLP, which captures spatial dependencies across patches, and a channel-mixing MLP, which captures inter-channel relationships. Skip connections are added to maintain stable gradients during training. After processing through multiple Mixer Layers, the final representation is aggregated using global average pooling, and the output is passed to a fully connected classification head.



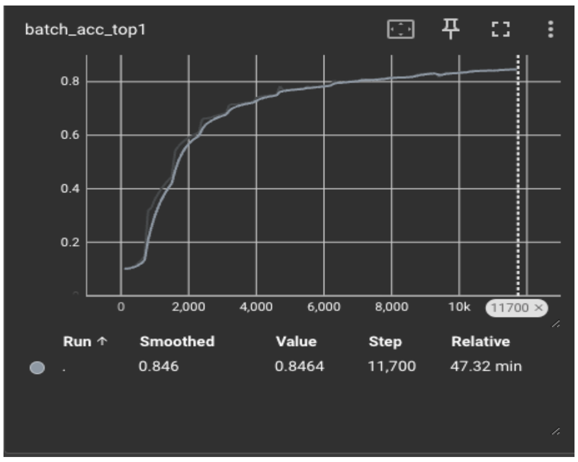
We followed the same code and procedure used for fine-tuning ViT-B/32 to fine-tune Mixer-B/16 on the CIFAR-10 dataset.

Results

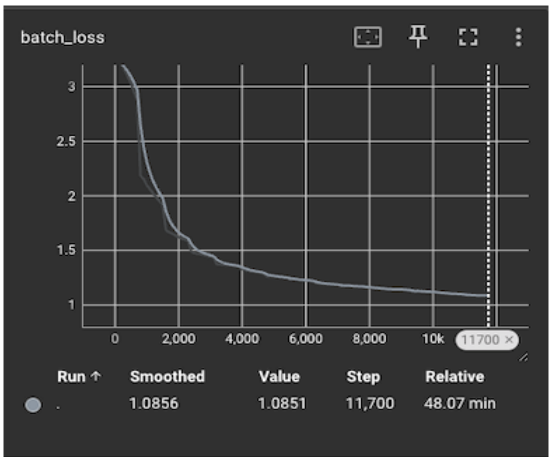
These are the few key metrics in model training and evaluation

- Batch Accuracy(Top-1): The percentage of samples in a batch where the model's top prediction matches the true label.
- Epoch Accuracy(Top-1): The average Top-1 accuracy across all batches in an epoch represents the model's overall performance.
- Epoch Accuracy (Top-5): The average Top-5 accuracy across all batches in an epoch, measuring how often the true label is within the top 5 predictions.
- Evaluation Accuracy (Top-1): The Top-1 accuracy is calculated on the validation or test dataset, reflecting how well the model generalizes.

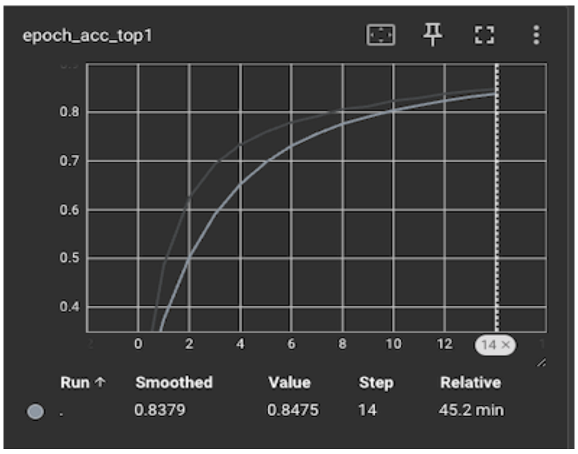
EfficientNetV2 with RMSProp.



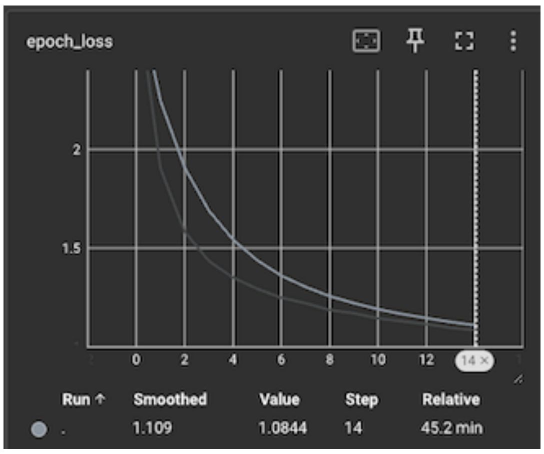
Run ↑	Smoothed	Value	Step	Relative
● .	0.846	0.8464	11,700	47.32 min



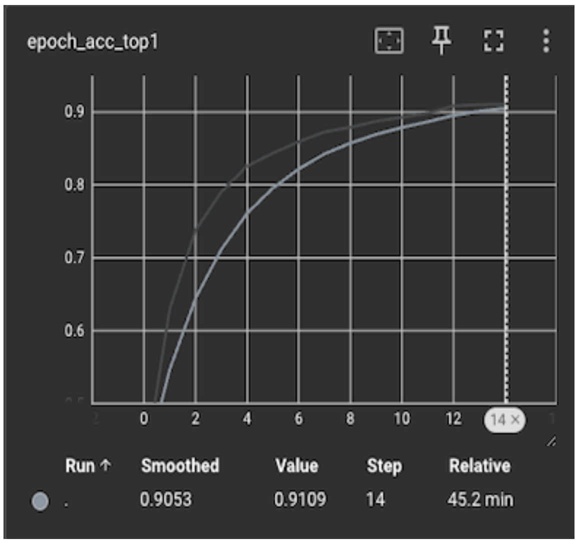
Run ↑	Smoothed	Value	Step	Relative
● .	1.0856	1.0851	11,700	48.07 min



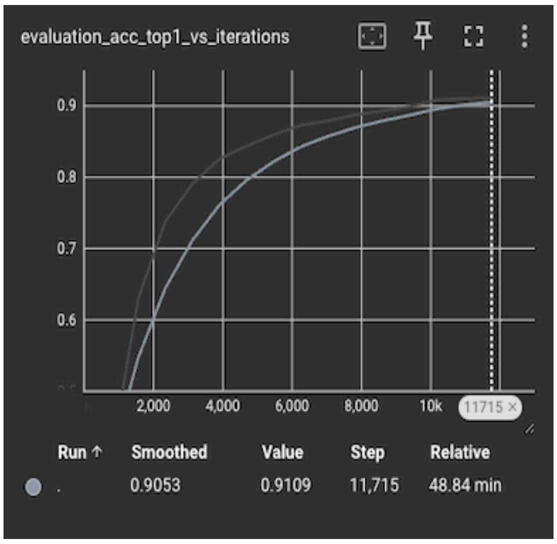
Run ↑	Smoothed	Value	Step	Relative
● .	0.8379	0.8475	14	45.2 min



Run ↑	Smoothed	Value	Step	Relative
● .	1.109	1.0844	14	45.2 min

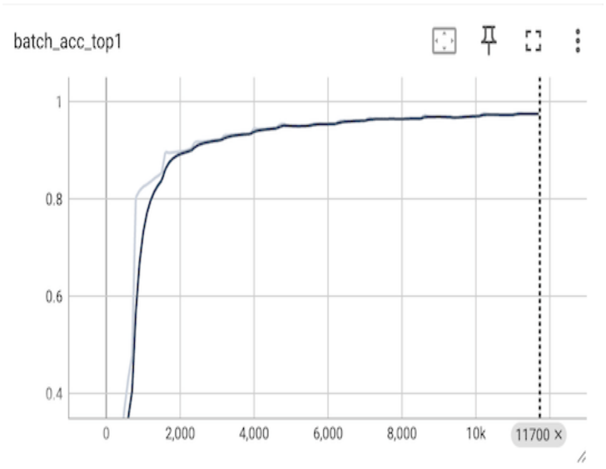


Run ↑	Smoothed	Value	Step	Relative
● .	0.9053	0.9109	14	45.2 min

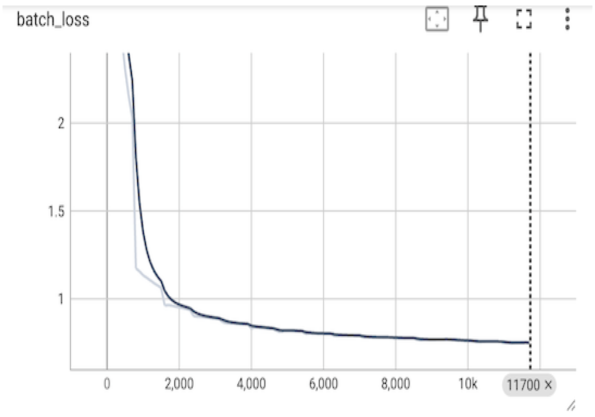


Run ↑	Smoothed	Value	Step	Relative
● .	0.9053	0.9109	11,715	48.84 min

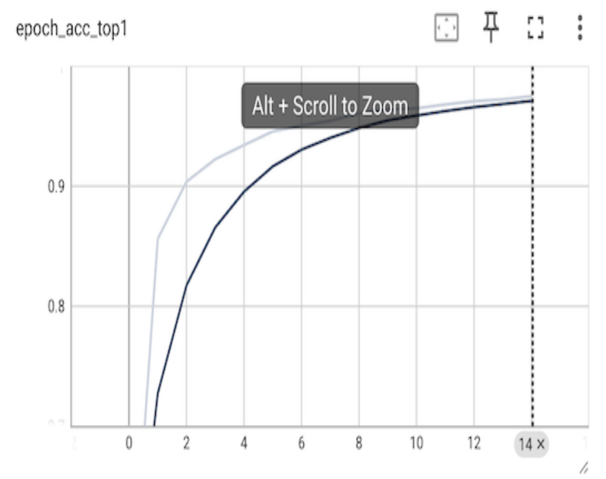
EfficientNetV2 with AdamW



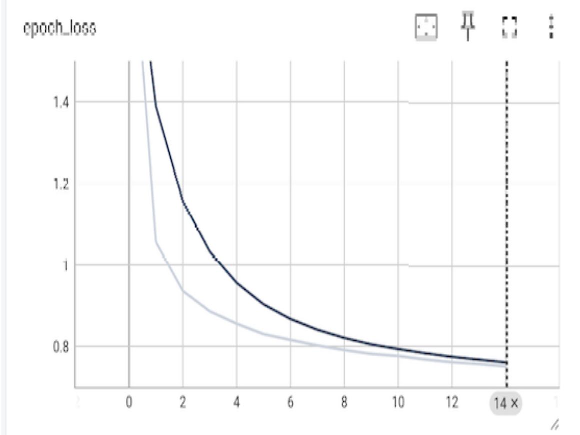
Run ↑	Smoothed	Value	Step	Relative
● .	0.9746	0.9748	11,700	49.8 min



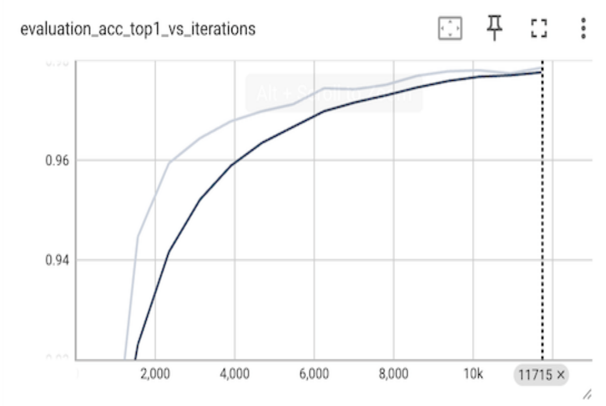
Run ↑	Smoothed ↑	Value	Step	Relative
● .	0.751	0.7507	11,700	49.8 min



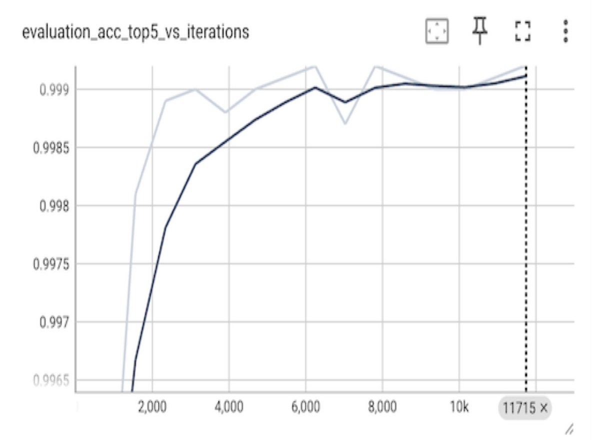
Run ↑	Smoothed	Value	Step	Relative
● .	0.971	0.9749	14	47.18 min



Run ↑	Smoothed	Value	Step	Relative
● .	0.7605	0.7507	14	47.18 min

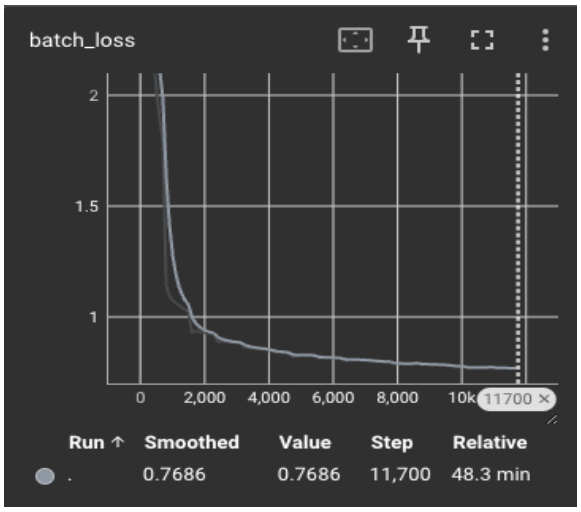


Run ↑	Smoothed	Value	Step	Relative
● .	0.9776	0.9785	11,715	50.84 min

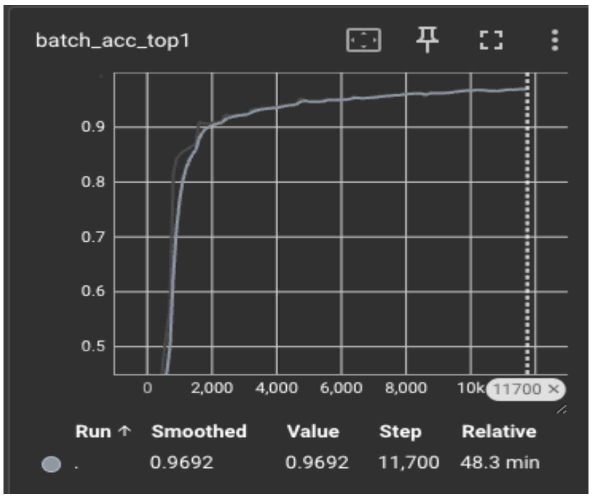


Run ↑	Smoothed	Value	Step	Relative
● .	0.9991	0.9992	11,715	50.84 min

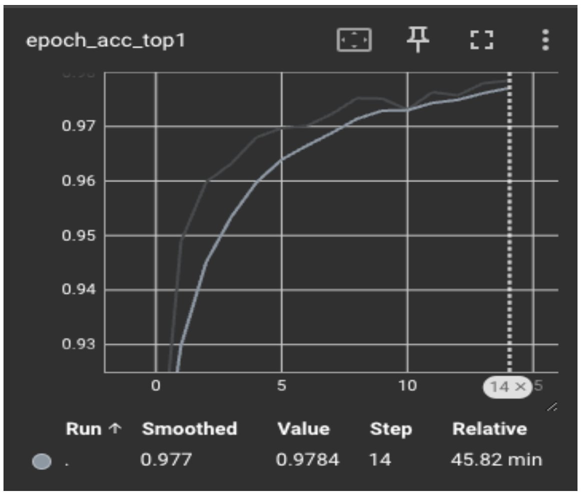
EfficientnetV2 with Lion



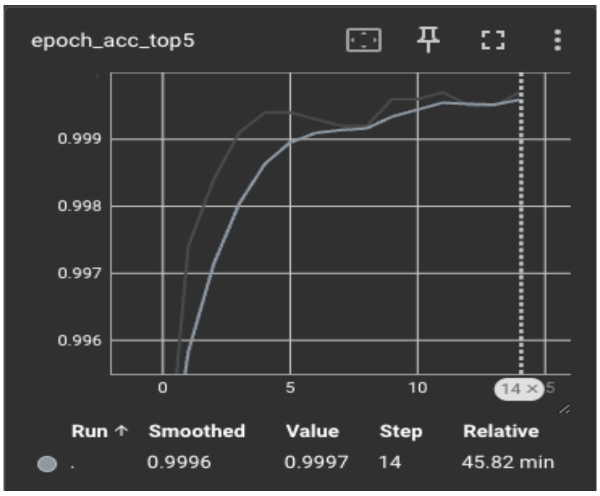
Run ↑	Smoothed	Value	Step	Relative
● .	0.7686	0.7686	11,700	48.3 min



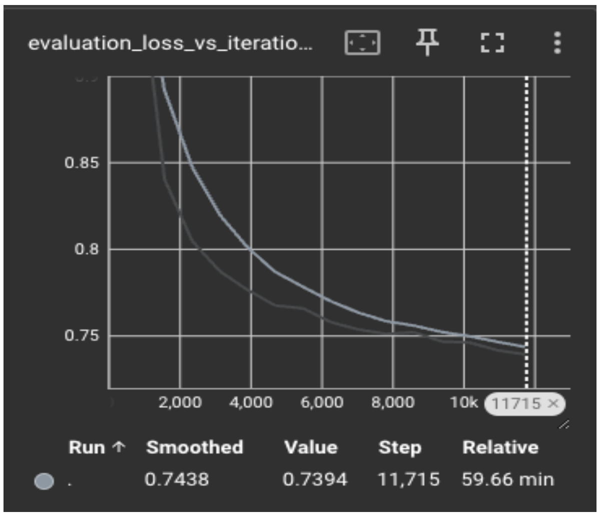
Run ↑	Smoothed	Value	Step	Relative
● .	0.9692	0.9692	11,700	48.3 min



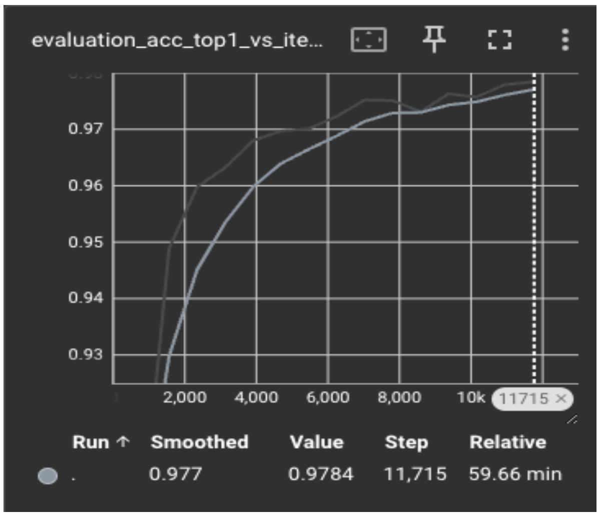
Run ↑	Smoothed	Value	Step	Relative
● .	0.977	0.9784	14	45.82 min



Run ↑	Smoothed	Value	Step	Relative
● .	0.9996	0.9997	14	45.82 min



Run ↑	Smoothed	Value	Step	Relative
● .	0.7438	0.7394	11,715	59.66 min

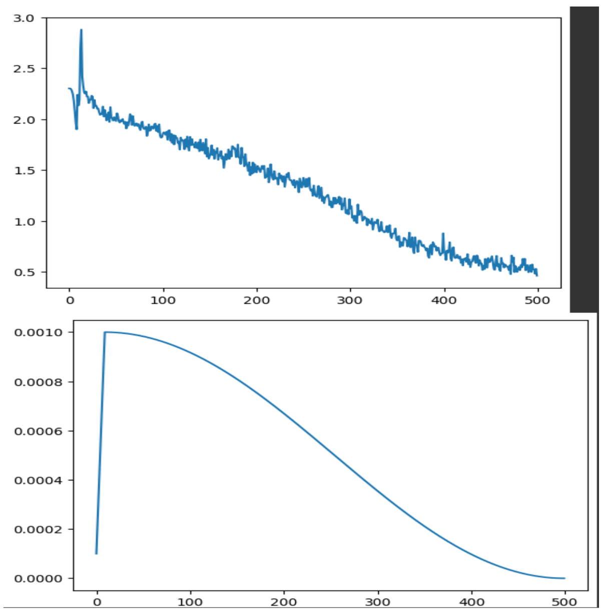


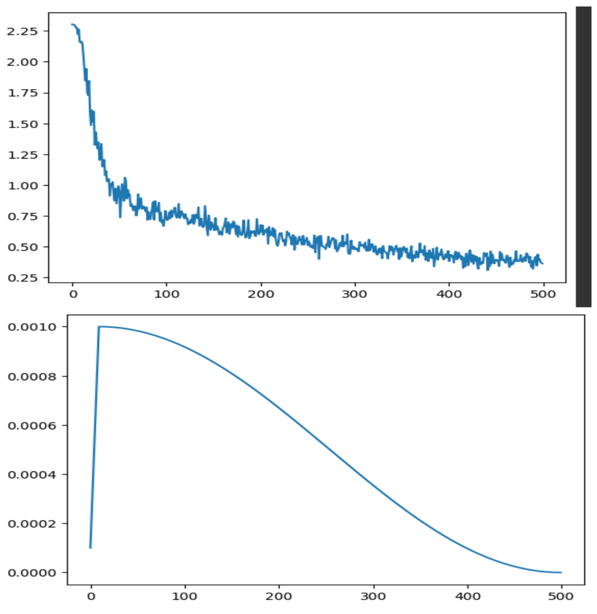
Run ↑	Smoothed	Value	Step	Relative
● .	0.977	0.9784	11,715	59.66 min

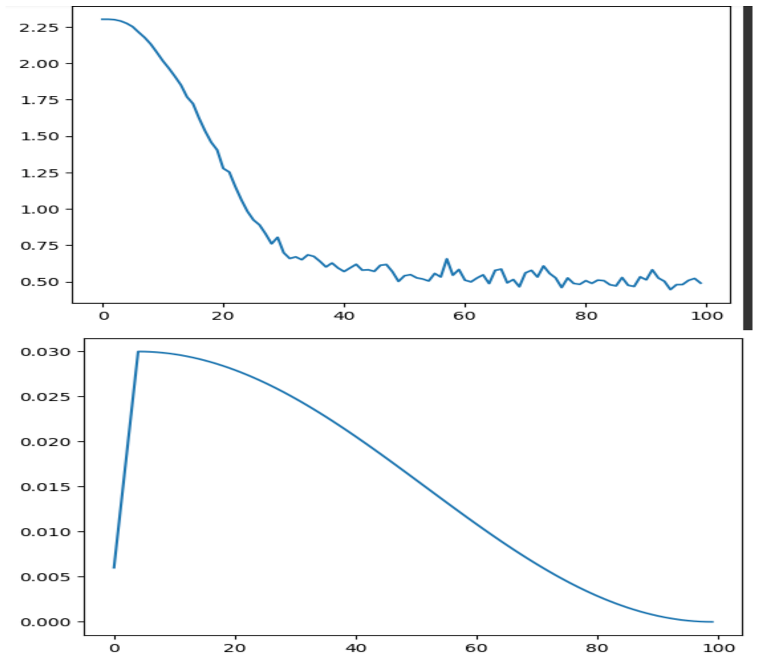
- After testing 3 optimizers, the Lion optimizer performed well with the Epoch_top_1 accuracy of 97.7%.
- AdamW with 97.1%(Epoch_top1_acc).
- RMSprop with 90.53%(Epoch_top1_acc).

ViT-B/32 Fine Tuning using RMSProp, AdamW and Lion Optimizers

The Below graphs indicate loss and learning rates over training steps.

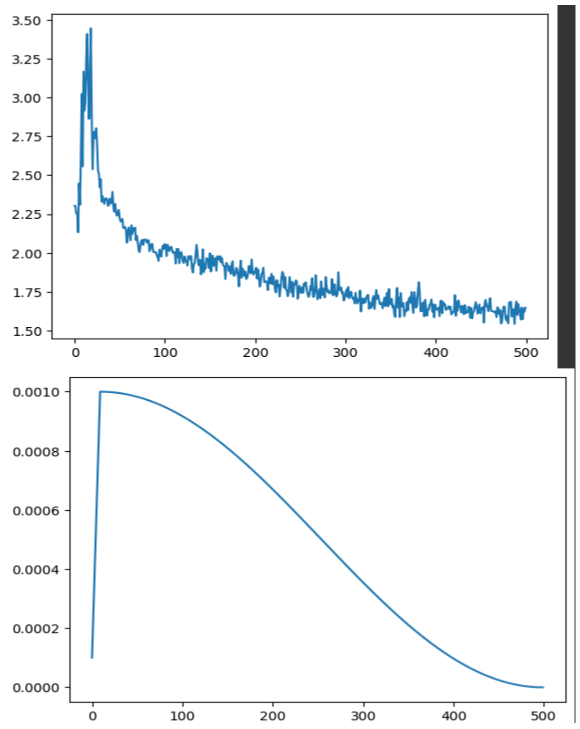

RMSProp Optimizer


AdamW optimizer

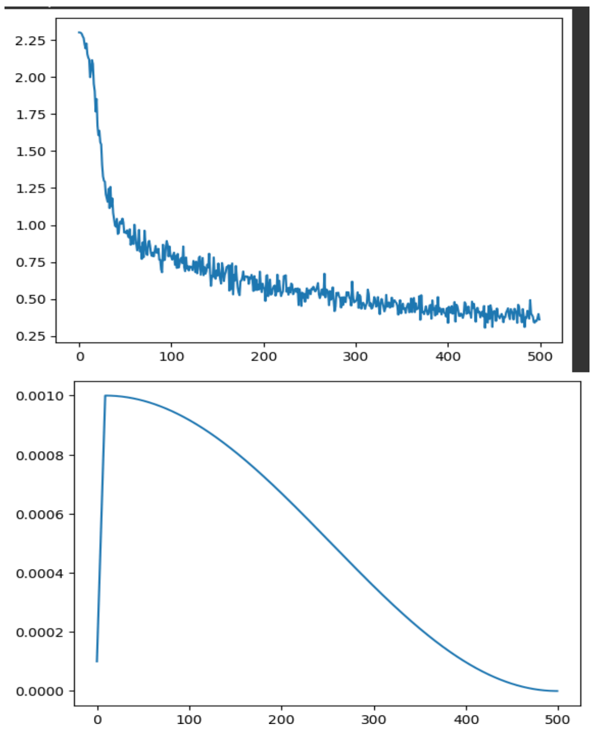

Lion Optimizer

Mixer-B/16 Fine Tuning using RMSProp, AdamW and Lion optimizers

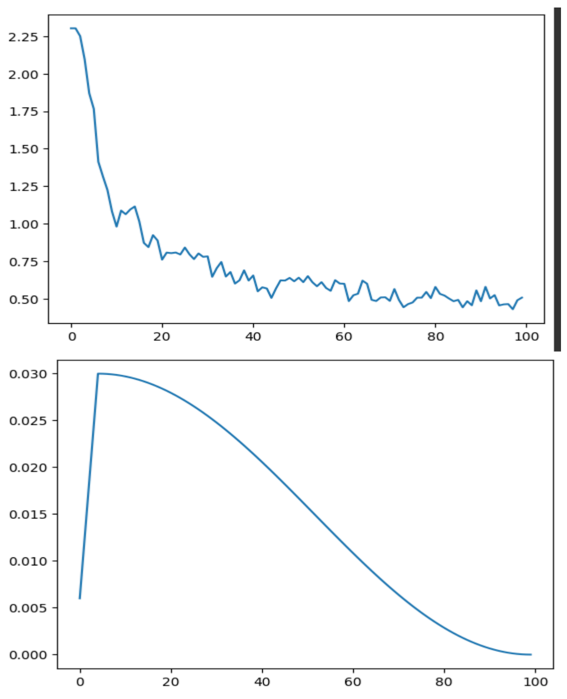
The Below graphs indicate loss and learning rates over training steps.



RMSProp Optimizer



AdamW Optimizer



Lion Optimizer

Model Finetuning	RMSProp Accuracy	AdamW Accuracy	Lion Accuracy
EfficientNetV2(CIFAR100)	90.53	97.1	**97.7**
ViT-B/32(CIFAR10)	92.66	96.31	**97.666**
Mixer-B/16(CIFAR10)	88.44	96.47	**96.7**

The results obtained align with the paper's findings that the Lion optimizer consistently achieved the highest accuracy across all image classification models, making it the most effective for fine-tuning tasks. AdamW performed closely, demonstrating robust generalization capabilities, while RMSProp showed comparatively lower accuracy. However, the performance of RMSProp and other manually tuned optimizers is highly dependent on hyperparameter tuning. This highlights the importance of selecting the right optimizer, with Lion demonstrating superior adaptability and performance across diverse models and datasets for fine-tuning.

Team Contribution

This project was a collaborative effort by Harshith Kasthuri, Chandan Narayana, and Sricharan Muttineni, with each member contributing equally to fine-tuning and evaluating the models.

Harshith Kasthuri fine-tuned the **EfficientNetV2** model on CIFAR-100, testing optimizers like RMSProp, AdamW, and Lion, and analyzing their performance on this convolutional model.

Chandan Narayana worked on fine-tuning the **ViT-B/32** model on CIFAR-10, focusing on testing optimizers and evaluating their effectiveness for transformer-based architectures.

Sricharan Muttineni fine-tuned the **Mixer-B/16** model on CIFAR-10, adapting the MLP-Mixer architecture and analyzing optimizer performance.

We altogether worked on the report collaboratively and completed the project successfully.

References

[1] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le, "Symbolic Discovery of Optimization Algorithms," *Advances in Neural Information Processing Systems (NeurIPS)*, 2023, https://arxiv.org/pdf/2302.06675.

[2] Mingxing Tan and Quoc V. Le, "EfficientNetV2: Smaller Models and Faster Training," *Proceedings of the International Conference on Machine Learning (ICML)*, 2021, https://arxiv.org/pdf/2104.00298.

[3] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021, https://arxiv.org/pdf/2010.11929.

[4] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy, "MLP-Mixer: An All-MLP Architecture for Vision," *Advances in Neural Information Processing Systems (NeurIPS)*, 2021, https://arxiv.org/pdf/2105.01601.

[5] Mingxing Tan and Quoc V. Le, "EfficientNetV2: Smaller Models and Faster Training," GitHub repository, available at: https://github.com/google/automl/tree/master/efficientnetv2.

[6] Google Research, "Vision Transformer: Transformers for Image Recognition at Scale," GitHub repository, available at: https://github.com/google-research/vision_transformer.