AMATH 515 Final Project: Analysis of Different Combinations of Stochastic Gradient Descent Algorithms and Stepsizing Schemes

Alexey Yermakov
Jason Isa

ABSTRACT

Our research focused on creating a simulation playground where users could easily test different combinations of optimizer and stepsizing schemes on different datasets. We tested our playground by performing a preliminary comparison of a subset of current stochastic gradient descent algorithms and step sizing schemes available in the PyTorch library. Future work will focus on performing a more extensive comparison of more optimizers and stepsizing schemes available in the PyTorch library.

KEYWORDS

Deep Learning; Stochastic Gradient Descent Algorithms; Stepsizing Schemes; Transformers; Convolutional Neural Networks

1 INTRODUCTION

Modern deep learning hinges on variants of stochastic gradient descent. With the large volume of research in deep learning, there is an abundance of stochastic gradient descent algorithms and stepsizing schemes available to train deep learning models on. Currently, PyTorch is recognized as one of the two most popular machine learning libraries, offering free and open-source software to its users. Due to its popularity, many researchers and industry leaders; like Tesla, Microsoft, NVIDIA, and Oak Ridge National Laboratory utilize the PyTorch machine learning framework. Because of this, we performed an empirical comparison of the performance of different descent algorithms and stepsizing schemes that are currently available in the PyTorch library.

Stepsizing schemes and descent algorithms are a subsection of hyper-parameters that need to be considered in machine learning before training a model. The problem is, choosing the right optimizer and stepsizing scheme for a given machine learning problem can be very difficult and there is no one-fits-all solution. The optimizer and stepsizing scheme needs to be selected carefully based on the machine learning problem and real-world constraints. In our experiments we investigated two common machine learning problems: image classification and next token prediction.

2 PRELIMINARIES

Our experiment investigates the performance of different combinations of learning algorithms and stepsizing schemes over different datasets and deep learning models. The following sections will introduce background on each of the learning algorithms, stepsizing schemes, datasets, and models used in our experiment.

2.1 Algorithms

The following section will discuss the four learning algorithms that we tested in our experiment.

2.1.1 Adagrad. [1] Adagrad is an algorithm for gradient-based optimization which adapts the learning rate component-wise to the parameters by incorporating knowledge of past observations. This adaptability is achieved by scaling the learning rate inversely proportional to the historical sum of square gradients for each parameter. This was developed to mitigate the problem of having to choose different learning rates for each hyper-parameter dimension. The Adagrad algorithm implemented in PyTorch can be viewed in Algorithm 1. Some advantages of Adagrad are (1) it eliminates the need to manually tune the learning rate, (2) the convergence is faster and more reliable than simple stochastic gradient descent when the scaling of the weights is unequal, and (3) it is not very sensitive to the size of the master step. Adagrad can arguably be one of the most popular algorithms for machine learning and it directly influenced the development of another algorithm we tested, the ADAM algorithm.

Algorithm 1 Adagrad Algorithm

Require: **Input:** γ (lr), $f(\theta)$ (objective), λ (weight decay), τ (initial accumulator value), η (lr decay)
  Initialize: $state\_sum_0 \leftarrow 0$
  for $t = 1$ to $\ldots$ **do**
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $\tilde{\gamma} \leftarrow \gamma/(1 + (t-1)\eta)$
    if $\lambda \neq 0$ **then**
      $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    end if
    $state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2$
    $\theta_t \leftarrow \theta_{t-1} - \tilde{\gamma}\frac{g_t}{\sqrt{state\_sum_t}+\epsilon}$
  end for

2.1.2 RMSProp. [2] RMSProp is a non-published optimizer which uses a similar idea as Adagrad but the scaling of the gradient is less aggressive. This algorithm uses a moving average of the squared gradients to scale the learning rate for each parameter, stabilizing the learning process and preventing oscillations in the optimization trajectory. This adaptive learning rate feature makes RMSProp particularly well-suited for non-stationary and noisy optimization problems. The RMSProp algorithm implemented in PyTorch can be viewed in Algorithm 2.

2.1.3 ADAM. [5] The ADAM algorithm uses estimations of the first and second moments of the gradient to adapt the learning rate for each weight of the neural network. It dynamically adapts learning rates for each parameter individually, allowing faster convergence and improved performance on a diverse range of tasks. ADAM addresses the disadvantages of generalized performance

Algorithm 2 RMSprop Algorithm

Require: **Input:** α (alpha), γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), μ (momentum), $center$
Initialize: $v_0 \leftarrow 0$ (square average), $b_0 \leftarrow 0$ (buffer), $g_0^{ave} \leftarrow 0$
for $t = 1$ to $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  end if
  $v_t \leftarrow \alpha v_{t-1} + (1-\alpha)g_t^2$
  $\tilde{v_t} \leftarrow v_t$
  if $centered$ **then**
    $g_t^{ave} \leftarrow g_{t-1}^{ave}\alpha + (1-\alpha)g_t$
    $\tilde{v_t} \leftarrow \tilde{v_t} - (g_t^{ave})^2$
  end if
  if $\mu > 0$ **then**
    $b_t \leftarrow \mu b_{t-1} + g_t/(\sqrt{\tilde{v_t}} + \epsilon)$
    $\theta_t \leftarrow \theta_{t-1} - \gamma b_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t/(\sqrt{\tilde{v_t}} + \epsilon)$
  end if
end for

being worse than that of the simple stochastic gradient descent method, offering an improved generalizing performance. ADAM takes advantage of the speed from momentum and the ability to adapt gradients in different directions from the RMSProp algorithm. The ADAM algorithm implemented in PyTorch can be viewed in Algorithm 3.

2.1.4 ADAMW. [9] The ADAMW aglorithm modifies the implementation of wight decay in ADAM, by decoupling weight decay from the gradient update. This was meant to combat ADAM's known convergence problems. The modification helps prevent overfitting and improves the generalization performance of the model by penalizing large weights. The ADAMW algorithm implemented in PyTorch can be viewed in Algorithm 4

2.2 Stepsizing Schemes

Learning rate decay is a technique used in machine learning where, throughout the training phase, the learning rate is gradually lowered. This technique is used to tackle problems arising from the use of a fixed learning rate, like oscillations and sluggish convergence, helping to improve stability and accuracy of the model. Stepsizing schemes can be defined in a number of ways. The following sections will briefly define the two different stepsizing schemes that we tested in our experiment.

2.2.1 StepLR. StepLR decays the stepsize of each parameter group by gamma every *n* epochs. Mathematically this stepsizing scheduler takes the form,

$$lr = lr_{initial} * \left(1 - \frac{decay\_rate}{100}\right)^{epoch}.$$

Algorithm 3 ADAM Algorithm

Require: **Input:** γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), amsgrad, maximize
Initialize: $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\hat{v_0}^{max} \leftarrow 0$
for $t = 1$ to $\ldots$ **do**
  if maximize **then**
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  end if
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  end if
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
  $\hat{m_t} \leftarrow m_t/(1-\beta_1^t)$
  $\hat{v_t} \leftarrow v_t/(1-\beta_2^t)$
  if amsgrad **then**
    $\hat{v_t}^{max} \leftarrow max(\hat{v_t}^{max}, \hat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\hat{m_t}/(\sqrt{\hat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\hat{m_t}/(\sqrt{\hat{v_t}} + \epsilon)$
  end if
end for

Algorithm 4 ADAMW Algorithm

Require: **Input:** γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective), ϵ (epsilon), λ (weight decay), amsgrad, maximize
Initialize: $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\hat{v_0}^{max} \leftarrow 0$
for $t = 1$ to $\ldots$ **do**
  if maximize **then**
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  end if
  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
  $\hat{m_t} \leftarrow m_t/(1-\beta_1^t)$
  $\hat{v_t} \leftarrow v_t/(1-\beta_2^t)$
  if amsgrad **then**
    $\hat{v_t}^{max} \leftarrow max(\hat{v_t}^{max}, \hat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\hat{m_t}/(\sqrt{\hat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\hat{m_t}/(\sqrt{\hat{v_t}} + \epsilon)$
  end if
end for

2.2.2 Cosine Annealing. [8] Cosine annealing is a dynamic stepsizing scheme that adjusts the learning rate following a cosine curve.

Mathematically this stepsizing scheduler takes the form,

$$lr_t = lr^i_{min} + \frac{1}{2}\left(lr^i_{max} - lr^i_{min}\right)\left(1 + cos\left(\frac{T_{cur}}{T_i}\pi\right)\right),$$

where lr^i_{min} and lr^i_{max} are ranges for the step-size and T_{cur} account for how many epochs have been performed since the last restart. The resetting acts to re-initiate the learning learning process. This method is particularly effective in avoiding local minima and in fine tuning the model during the later stages of training.

2.3 Datasets

Our experiments test the performance of different combinations of learning algorithms and stepsizing schemes on two different standard machine learning datasets, the CIFAR10 image dataset and the Tiny-Shakespeare text dataset. The following section will discuss the datasets we used.

2.3.1 CIFAR10 Image Dataset. The CIFAR-10 dataset [6] consists of 60,000 32x32 color images across 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. The classes in this dataset are "airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", and "truck". Theses classes are completely mutually exclusive, meaning that there is no overlap between classes. The CIFAR-10 dataset is commonly used to train machine learning and computer vision algorithms for the purpose of image classification. A subset of the dataset can be seen in Figure 1.

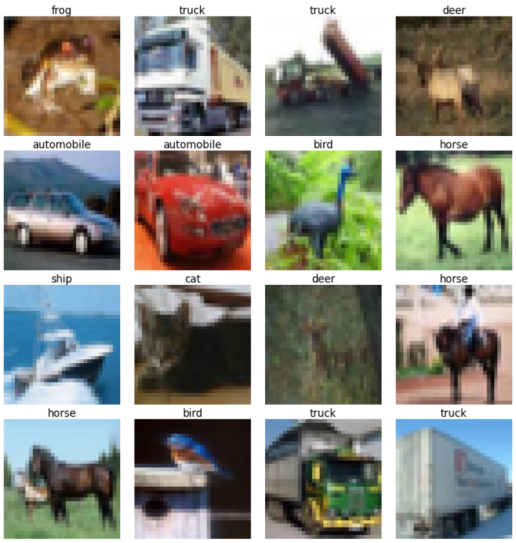


Figure 1: Subset of the CIFAR10 dataset. There are 16 labeled images shown here.

2.3.2 Tiny-Shakespeare Text Dataset. The Tiny-Shakespeare text dataset [3] is a text file consisting of a subset of the works of Shakespeare. There are 40,000 lines of text, containing a total of 202,651 words (or 1,115,394 characters).

2.4 Models

We study the effect of different descent algorithms and stepsizing schemes on two different models. This allows us to observe if there might be any benefit to choosing different algorithms and stepsizing schemes depending on the problem being solved, since it might not necessarily be the case that a "best" combination of algorithm and stepsizing scheme exists.

2.4.1 Convolutional Neural Network. We use a Convolutional Neural Network as the classifier for the CIFAR10 dataset [7]. In particular, the model is composed of:

- Convolutional layers
  - Conv2D
    (in_channels: 3, out_channels: 6, kernel_size: 3)
  - ReLU
  - MaxPool2d
    (kernel_size: 2, stride: 2)
  - Conv2D
    (in_channels: 3, out_channels: 6, kernel_size: 3)
  - ReLU
  - MaxPool2d
    (kernel_size: 2, stride: 2)
- Flatten
- Densely connected layers
  - Linear
    (in_size: 768, out_size: 384)
  - ReLU
  - Linear
    (in_size: 384, out_size: 192)
  - ReLU
  - Linear
    (in_size: 192, out_size: 10)

2.4.2 GPT. In contrast to the task of image classification, we found it interesting to study how different descent algorithms and stepsizing schemes would perform on the text-generation task. As such, we use a publicly available open-source implementation of GPT [10] called "minGPT" [4] which is based on the transformer decoder architecture [11]. The model can be simplified to the diagram in Figure 2. This model varies significantly from the previously introduced CNN since it takes text as input instead of images, is trained to predict text instead of classifying images, and is based on transformer decoders instead of convolutional layers.

For this project, we study only the loss of the model per iteration on the text prediction task, we do not study the performance of the trained model on any downstream tasks (like text classification).

3 METHODS

To obtain quantitative results, we trained the two previously described models on their respective datasets using different combinations of descent algorithms and stepsizing schemes. To hold other variables constant, we perform no hyperparameter optimization for both models. As such, the following hyperparameters were fixed:

```
# CIFAR10 default configuration values
CIFAR10_batch_size: 4096
CIFAR10_shuffle: False
CIFAR10_learning_rate: 0.001
CIFAR10_epochs: 20
```

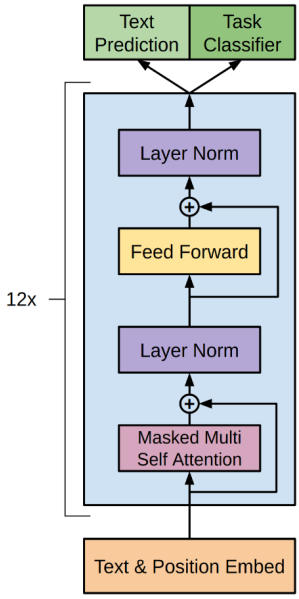


Figure 2: A simplified view of the GPT architecture using a transformer decoder.

```
# mingpt default configuration values
mingpt_batch_size: 64
mingpt_block_size: 128
mingpt_shuffle: False
mingpt_learning_rate: 0.001
mingpt_iters: 2000
```

Furthermore, each stepsizing scheme and descent algorithm used the default PyTorch hyperparameters where applicable.

To train the CNN, we train the classifier on batches of 4,096 training images for 20 epochs. For each epoch, we iterate over all training images. After each epoch, we calculate the loss and accuracy of the model on both the training and testing datasets.

To train the GPT model, we take batches of 64 sequences of 128 characters from the text dataset. The structure of the "minGPT" code is such that there are no epochs. The training consists of perpetually obtaining the next batch from the dataset until a maximum number of iterations is achieved, looping back to the start of the dataset when necessary. We set the maximum number of iterations to 2,000 for training "minGPT".

The entire codebase used to run the project is publicly available on GitHub [13].

4 RESULTS

4.1 CNN/CIFAR10 Results

The accuracy per epoch of the CNN model for both the training and testing sets are in Figure 3. The loss per epoch of the CNN model for both the training and testing sets are in Figure 4. From these figures, it is immediately clear that the best performing combinations of step sizing scheme and learning rate are combinations of ADAM/ADAMW with StepLR/Cosine-Annealing. The best-performing combination in regard to accuracy on the test set was ADAM with StepLR, as seen in Table 1.

	ADAM	ADAMW	RMSProp	Adagrad
StepLR	**48.81%**	48.77%	44.09%	38.96%
Cosine Annealing LR	48.75%	48.72%	41.39%	38.51%

Table 1: Algorithm and stepsizing scheme test set accuracy for the CNN trained on CIFAR10.

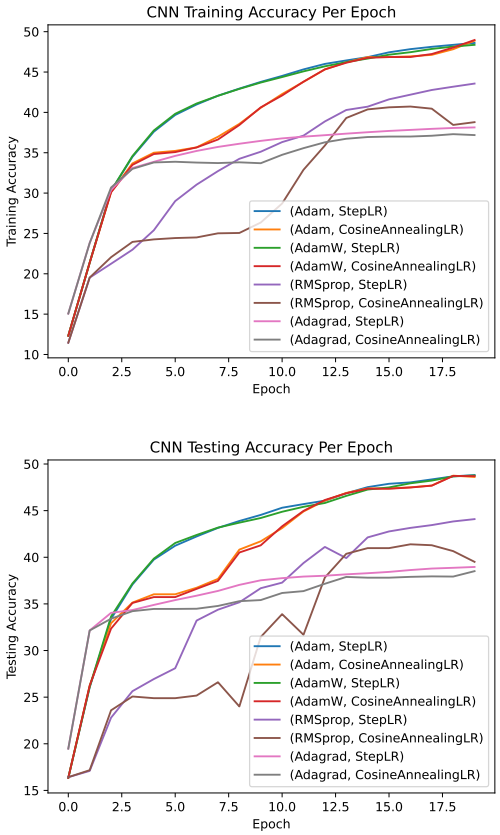
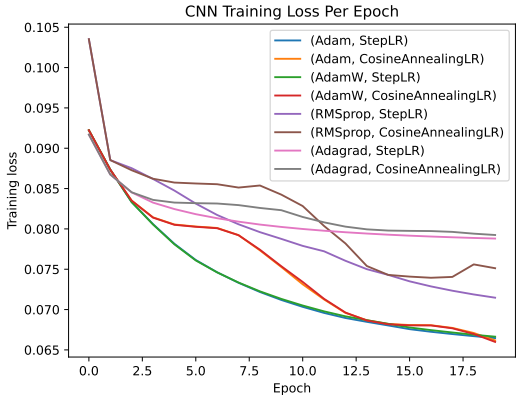


Figure 3: CNN train and test set accuracies on the CIFAR10 dataset.

4.2 minGPT/Tiny-Shakespeare Results

The accuracy per iteration for the GPT model is in Figure 5. Once again, we see that there is an apparent tie between different combinations: ADAMW with either StepLR or CosineAnnealingLR. However, ADAMW with StepLR achieves the lowest minimum



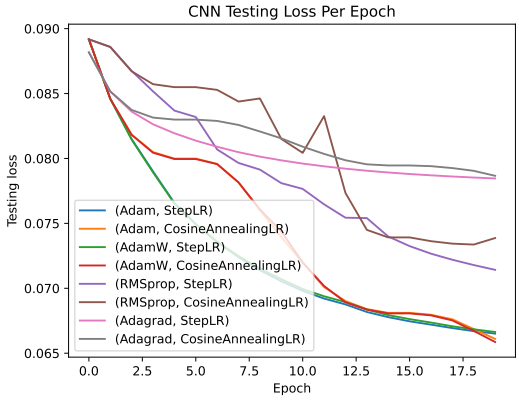


Figure 4: CNN train and test set losses on the CIFAR10 dataset.

loss across all combinations, as seen in Table 2. Zooming into a portion of the figure (as in Figure 6), we see that the remaining combinations of descent algorithm and stepsizing schemes perform relatively similarly, due to the significant overlap in their losses.

	ADAM	ADAMW	RMSProp	Adagrad
StepLR	2.503	**1.546**	2.507	2.577
Cosine Annealing LR	2.500	1.578	2.504	2.589

Table 2: Algorithm and stepsizing scheme test set minimum loss for minGPT trained on tiny-shakespeare.

5 DISCUSSION

From our results, we see that ADAMW with StepLR acheives the best performance when applied to both datasets (although it is 0.04% behind the top-performing ADAM with StepLR on the CNN classification accuracy). The best performing descent algorithm for both experiments being ADAMW makes sense intuitively: it utilizes the best of ADAM, RMSProp, and Adagrad in a single algorithm [12]. We also see that the performance of the descent algorithm

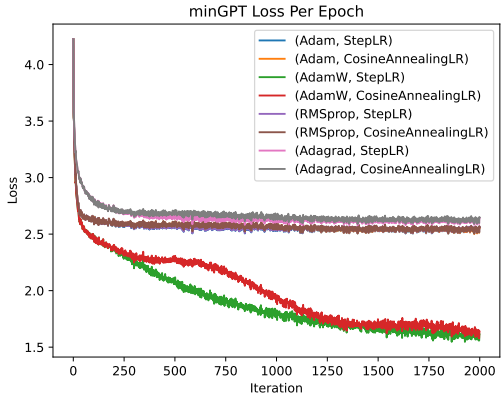


Figure 5: minGPT test loss per iteration on the tiny-shakespeare dataset.

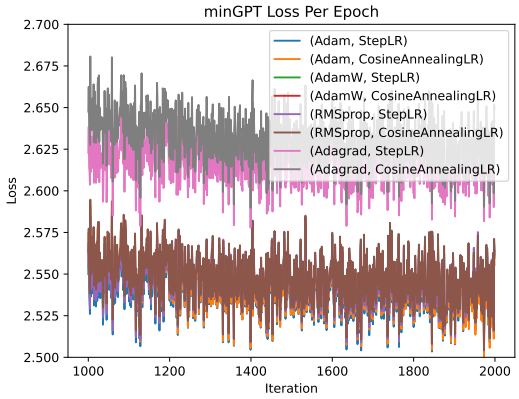


Figure 6: minGPT test loss per iteration on the tiny-shakespeare dataset. This figure is zoomed into a region of Figure 5.

improves with how recently developed the descent algorithm is: Adagrad performing the worst, followed by RMSProp, followed by ADAM, followed by ADAMW. This also makes sense from a logical point of view as newer algorithms should ideally perform better.

It's important to note that for the minGPT model, the difference between using ADAMW and ADAM is large, resulting in significantly different minimum training loss when compared with the CNN, where the choice of the two descent algorithms has a much smaller final effect. It's not immediately clear why this is the case, but it is worth pointing out.

Furthermore, from Table 1 and Table 2 we see that the descent algorithm has a much stronger influence on model performance than the step-sizing scheme. Fixing the descent algorithm and observing which step-sizing schemes perform better shows us that StepLR performs better than Cosine Annealing always for the CNN and sometimes for minGPT. From this we can conclude that it's unclear whether the step-sizing scheme has an outsized effect on

the results for minGPT, and that StepLR performs marginally better than Cosine Annealing for the CNN.

Another interesting observation is in Figure 3. We can clearly see from the lines for (Adam, CosineAnnealingLR) and (AdamW, CosineAnnealingLR) that the learning rate has a direct effect on the accuracy of the model. This is seen from the "wiggle" of the accuracy curves as the learning rate is changing. Compare those two lines with the learning rates in Figure 7 and we see that a higher learning rate results in a larger rate of increase in accuracy. A similar observation can be made in Figure 5 for the (AdamW, CosineAnnealingLR) curve due to the presence of a "wiggle". However, in both instances, there does not seem to be a direct benefit to using Cosine Annealing since its loss catches up to, but does not exceed, the losses from StepLR. We can conclude from this observation that for the CNN model, a higher learning rate yields a faster rate of increase in testing accuracy.

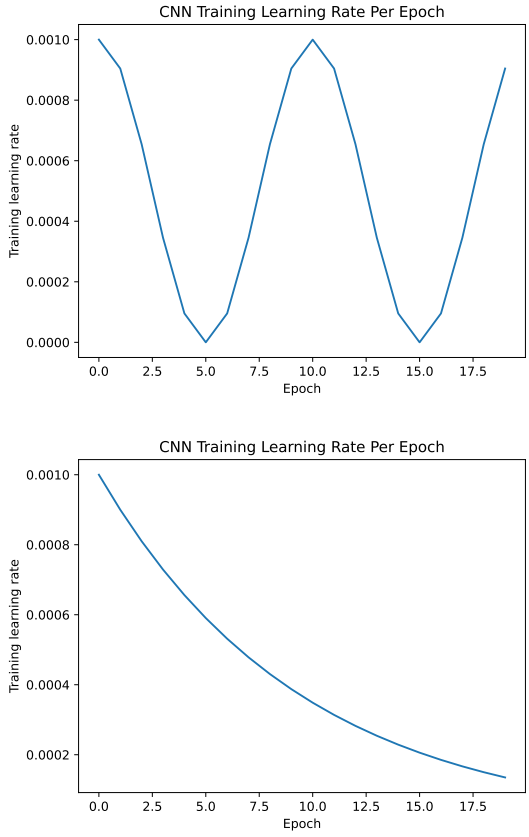


Figure 7: CNN learning rates per epoch for ADAM. First figure is Cosine Annealing and the second figure is StepLR.

6 CONCLUSION

Deep learning is a powerful tool which can be applied in a wide variety of situations. Its popularity has recently exploded due to recent advancements in hardware, allowing large datasets to be trained on large models which can be used in general settings. One particular example of this is OpenAI's ChatGPT, which uses a GPT-based model to provide a powerful assistant to anyone with an internet connection. However, as seen in this project, simply having a model and a dataset is not enough to get good results. The choice of batched stochastic gradient descent algorithm and stepsizing schemes can have drastic effects on model performance and should be considered when training any deep learning model.

From this project, we learned that ADAMW is a great candidate for descent algorithm because it is the most recent one, allowing it to be the result of years of research and applications. However, our analysis of stepsizing schemes is not as enlightening: this choice didn't seem to have an outsized effect on final performance. Although we did see that the stepsizing scheme does affect model performance, it was not as significant as the choice of descent algorithm. It may be the case that various step-sizing schemes were developed in the field of machine learning to achieve slightly better performance than existing models, achieving new state-of-the-arts. However, from this project we believe that in a practical setting, you can obtain "good-enough" performance with virtually any reasonable stepsizing scheme.

As was mentioned earlier in the paper, the only variables we studied were the choices of descent algorithms and stepsizing schemes. We fixed all other hyperparameters. Future work would benefit from studying the individual hyperparameters of each stepsizing scheme, gradient descent algorithm, and deep learning model. Even then, more models, algorithms, and schemes are constantly being developed, resulting in a combinatorial explosion of combinations that can be studied.

As the field of machine learning matures, novel model architectures are going to be developed. Each of these architectures is powered by the incredibly powerful method of gradient descent, allowing the model to train on whatever dataset researchers and engineers decide to throw at it. As such, the use of different combinations of gradient descent and learning rates are here to stay and they have a direct role on the downstream performance of the model.

REFERENCES

[1] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* 12, 61 (2011), 2121–2159. http://jmlr.org/papers/v12/duchi11a.html

[2] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

[3] Andre Karpathy. 2015. https://github.com/karpathy/char-rnn.

[4] Andre Karpathy. 2023. https://github.com/karpathy/minGPT.

[5] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[6] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[8] Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

[9] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG]

[10] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[12] Philipp Wirth. 2024. https://www.lightly.ai/post/which-optimizer-should-i-use-for-my-machine-learning-project.

[13] Alexey Yermakov. 2024. https://github.com/yyexela/AMATH515_Project.

7 LINK TO GITHUB CODE

https://github.com/yyexela/AMATH515_Project