Impact of Learning Rate, Optimizer, Learning Rate Scheduler and Batch Size on OOD generalization in Deep learning

Devikalyan Das, Pranay Raj Kamuni, Saurabh Kumar Pandey
Visual Computing, Saarland Univeristy

***Abstract*—Deep learning is a sub-branch of AI. Due to it's representational learning capabilities in real world problems, it's popularity and research bucket is increasing day by day, with endless amount of new models being introduced every day, performing better than the previous one. The core assumption of independent and identically distributed makes the learning problem simple for these models but introduces generalization problems in real world applications. In this project we investigated the effect of different optimizers, hyper-parameters, batch-sizes, etc on the OOD generalization performance of a deep neural network.**

I. Introduction

In recent times AI has taken over the world by storm. Today web searches, shopping, social media, automobiles, cell phones all are powered by AI in one form or the other. Knowingly or unknowingly we interact with these systems every day, thus making them an integral part of our daily life. In the last few decades the research and applications in the area of AI has impacted our lives more than ever, with each day a new system being introduced to solve a task once thought unsolvable. The heart of all these advances in AI is deep learning. Traditional machine learning techniques were very limited in the sense that they required a lot of domain knowledge to design tools to extract features or get a feature representation to use it with a learning system in-order to accomplish the task at hand. These methods were neither scalable nor widely adopted, if we leave few of the exceptions. The introduction of representation learning methods that can learn patterns directly from the data changed the field for ever.

As more and more industries are adopting deep learning solutions every day, its very important to validate the generalization capabilities of these models(specially in mission critical systems). The underlying assumption of all deep learning models is the independent and identically distributed(i.i.d) data. In simple terms the model assumes the training and test data comes from the same distribution, which is not true in practical scenarios. For instance a good image classifier that can separate cows from camels with high accuracy suddenly fails when an image of a cow on a beach is provided. It turns out that instead of learning the features of cows or camels, in this case the model learns the background to differentiate between the two subjects at test(i.e cows are generally in images with green background which represents farms). There are few approaches suggested to overcome the OOD generalization issues[6],[2] i.e Invariant risk minimization(IRM). The idea of IRM is simply to replace good old empirical risk minimization(ERM) with a different risk which forces the model to learn features that are invariant under different data distribution. All though the idea seems very promising, a lot of criticism and theoretical evidence were presented to disprove the claims of IRM [3],[7] i.e IRM at best will be as good as ERM but not better. Instead of searching for a new direction, we decided to evaluate OOD generalization with tools at hand i.e different optimizers, batch sizes etc, in-order to understand the effect these fundamental choice of a deep neural network model have on OOD generalization.

II. Methods

A. *Optimizers Used*

The gradient descent algorithm is the most useful workhorse for minimizing loss functions in practice. It is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. As mentioned in the title, for our analysis we experimented with 4 differnet optimizers.

1) Mini-batch SGD[8]: Mini-batch Stochastic gradient descent (SGD) computes the gradient of the cost function w.r.t. to the parameters θ for a mini-batch of training dataset.
$$\mathbf{x}_{t+1} := \mathbf{x}_t - \gamma_t \nabla f_i\left(\mathbf{x}_t\right)$$
2) RMSProp[1]: Root Mean Squared Propagation, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter. The use of a decaying moving average allows the algorithm to forget early gradients and focus on the most recently observed partial gradients seen during the progress of the search, overcoming the limitation of AdaGrad.

$$E\left[g^2\right]_t = \beta E\left[g^2\right]_{t-1} + (1-\beta)\left(\frac{\delta C}{\delta w}\right)^2$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{E\left[g^2\right]}} \frac{\delta C}{\delta w}$$

3) Adam[4]: Adaptive Moment Estimation is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1). They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

4) AdamW[5]: An extension of the Adam optimizer with an addition of weight decay to its optimization to move towards convergence at a faster pace, the details can be looked on the blog by fast.ai team.

B. Dataset

We are using the NICO dataset, which is from an image recognition competition which contains data labeled in 60 categories, we use the 15 most popular categories for our project, which are present in different domains/context like autumn, grass, rock, outdoor etc. to perform our experiments. We use all the data from n-1 contexts, shuffle them to use it for training, then take the nth context related data (different from the n-1 context related data used for training) to use it for validation to check speed and performance at which it generalizes when the distribution is out of context.

C. Model

For our case we used the very popular CNN model, the ResNet 34 model from the list of pytorch models(not a model with pretrained weights, as it may give wrong results when we perform our experiments) and perform our image recognition task.

III. Experiments

For this project, our goal is to achieve better and faster OOD generalization by investigating the influence of choosing different batch sizes and learning rates along with their schedulers. We have used various established optimization algorithms such as ADAM, SGD, RMSPROP and ADAMW for comparing the optimization performance and speed for our task.

For each experimental condition (choice of optimizer, batch size, learning rate with and without schedulers), the model was trained on 5 contexts with 15 classes in each context and evaluated on a validation set belonging to a separate context. Batch sizes of 64 and 128 were used in the experiments. We have kept the number of epochs fixed at 50 for every experiment to find out rate of convergence of different optimizers. We selected learning rate of 0.01 and 0.0001 to understand the impact of various learning rates. Also, we used MultiStepLR as learning rate scheduler and chose to reduce the learning rate by a factor of 0.1 after every 20 epochs as training progresses which will ensure stability. For this classification task, Cross-Entropy loss function was used.

A. Figures and Tables

The loss and accuracy have been computed during training and validation have been plotted. The comparison of training loss for batch size of 64 with learning rate of 0.0001 has been provided in Figure 4.

IV. Results

In the figure1 4 we present the results from the experiment related to usage of 4 different optimizers, with a constant lr of 1e-4, batch size = 64, and with 2 settings of LR schedulers(please refer the legend for more details), and below are the most important takeaways(the plots are smoothed using exponentially weighted function, alpha = 0.6):

- The first thing we notice is that the SGD optimizer does not show an increase in train or validation accuracy and no proper decrease in the loss values, this could mostly imply that the learning rate used here is not proper for convergence given the graph of the validation loss, it looks like the LR was too low, hence even the case with LR scheduler shows no improvement.
- The second most important thing we see is the validation loss/accuracy is very erratic and doesn't move smoothly in all the cases, it is because of the nature of the problem, as the train and valid datasets are from different distribution, and its effects are clearly visible here.
- We see that the optimizer with LR schedulers perform relatively better when compared to their counterparts without any LR scheduler. This shows that the model converges faster when we first reach near a good minima and slowly try to settle on to the global minima.
- In case of Adam and RMSProp optimizer without a scheduler, we see that after a certain point the loss value gets constant and the accuracy keeps reducing, this could

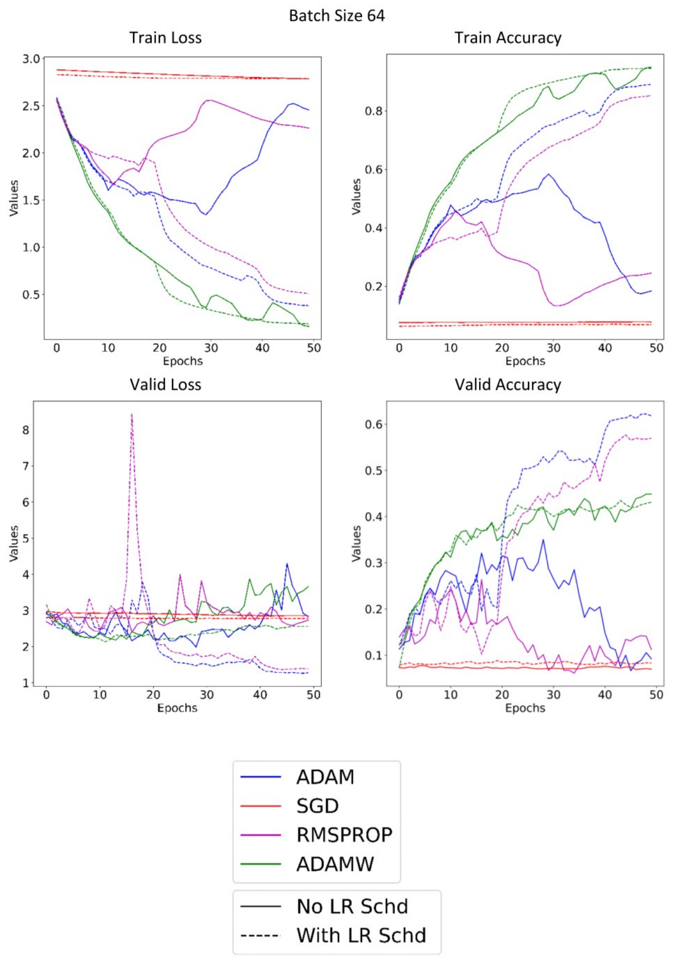

Fig. 1: Train Loss for various optimizer with batch size of 64.

mean that due to high LR value, the optimizer made the model reach a local minima and started settling there, whereas having an LR scheduler has made things easier for the optimizer for faster convergence and generalization.

- In case of AdamW optimizer the training accuracy is pretty high when compared to the Adam and RMSProp optimizers with an scheduler but the validation accuracy shows that AdamW optimizer has relatively lower accuracy when compared to the other two, this could mean that the AdamW optimizer is over-fitting the model, hence bad results on validation set.
- We see that the batch size is not really a huge factor for faster convergence and generalization of models when we use OOD datasets(refer apendix for images)
- From 5,4 for lr 1e-2, we see that the model has converged too quickly to a sub optimal solution, no matter which optimizer is used. As we see in the plots, the training loss has been sub optimally settled at a very high value, and even the use of LR scheduler has not shown any good progress. Also the erratic changes in the validation accuracy confirms that the model has not been trained properly for the task.

V. Conclusion

From the results we can conclude that although we see a higher training accuracy in the case of AdamW optimizer(with or without LR scheduler), we observe that the validation accuracy is lower and the validation loss is not decreasing, suggesting that the model is over-fitting on training data, but when it comes to Adam and RMSProp optimizer with an LR scheduler, we see that although the training accuracy is not as high as the AdamW optimizer, the validation accuracy is relatively higher and the validation loss values are decreasing gradually. Hence we can say that the Adam + LR scheduler or RMSProp + LR schedulers are a good option for optimizing the models with an OOD dataset.
As a future work, we can extend or study on different learning rate values, various other optimizer, with different LR scheduling techniques, and other techniques like label smoothing etc.

Acknowledgements

We are grateful for kaggle to provide us the platform to train our models, and NICO Challenge 2022[9] for the wonderful challenge and providing us the dataset.

References

[1] Rmsprop. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[2] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization, 2019.

[3] Ruocheng Guo, Pengchuan Zhang, Hao Liu, and Emre Kiciman. Out-of-distribution prediction with invariant risk minimization: The limitation and an effective fix, 2021.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[5] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[6] Jonas Peters, Peter Bühlmann, and Nicolai Meinshausen. Causal inference using invariant prediction: identification and confidence intervals. 2015.

[7] Elan Rosenfeld, Pradeep Ravikumar, and Andrej Risteski. The risks of invariant risk minimization, 2020.

[8] Ohad Shamir and Tong Zhang. Stochastic gradient descent for non-smooth optimization: Convergence results and optimal averaging schemes. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 71–79, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[9] Renzhe Xu Han Yu Zheyan Shen Peng Cui Xingxuan Zhang, Yue He. Nico++: Towards better benchmarking for domain generalization, 2022.

Appendix

Please find the images below for various experiments of our project with different setups:

1) Learning rate = 1e-2, 1e-4
2) Batch Size = 64, 128
3) With LR scheduler, without LR scheduler
4) Optimizer = Adam, SGD, AdamW, RMSProp

For more details regarding the experiments and the implementation, please clone the GitHub repo.https://github.com/DevikalyanDas/Optimization-In-ML

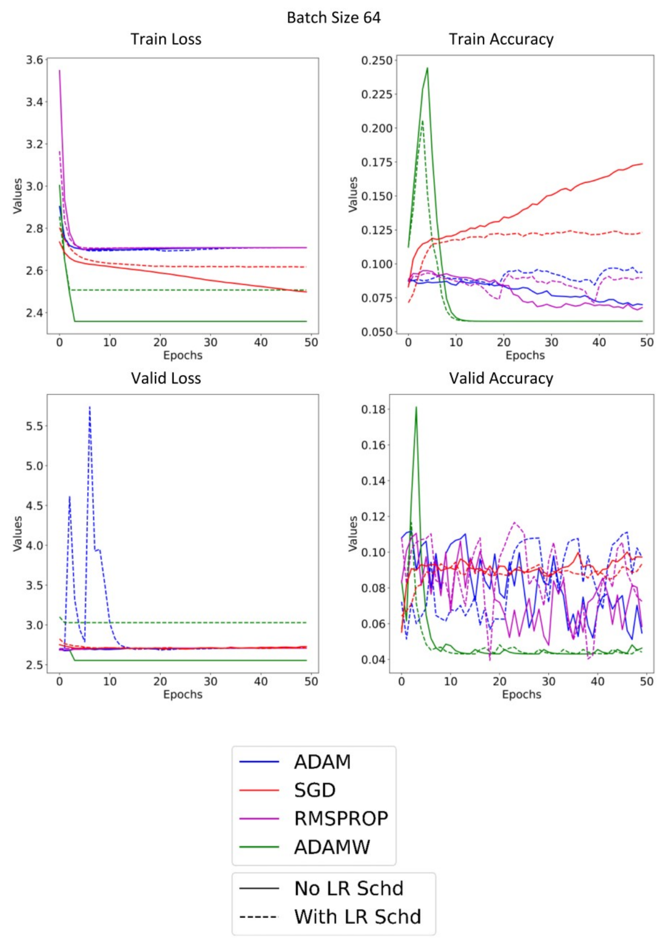

Fig. 2: Train Loss for various optimizer with batch size of 64 and LR = 1e-2.

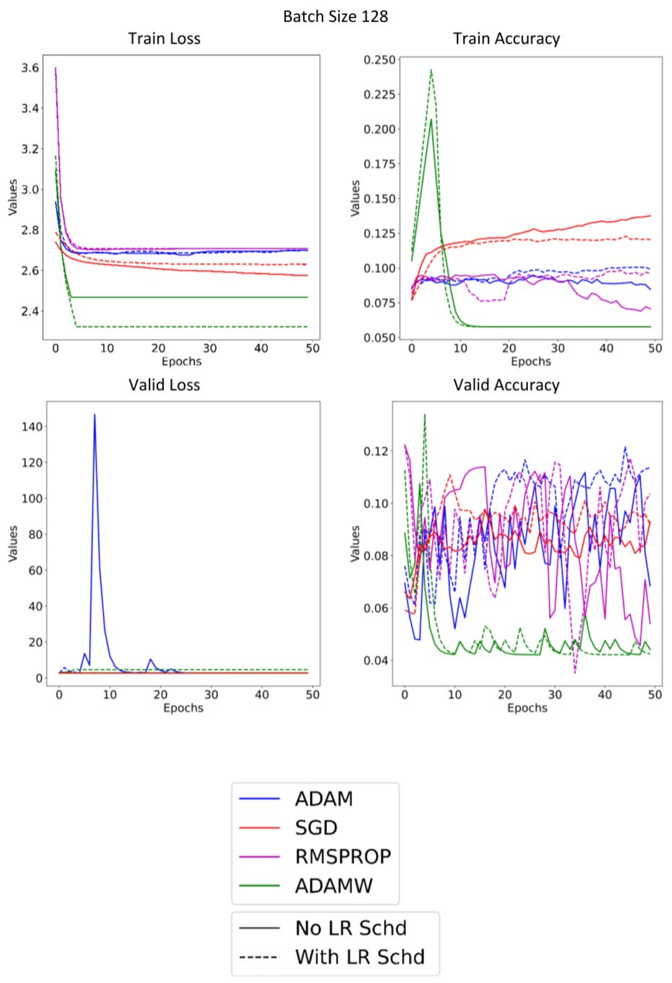

Fig. 3: Train Loss for various optimizer with batch size of 128 and LR = 1e-2.

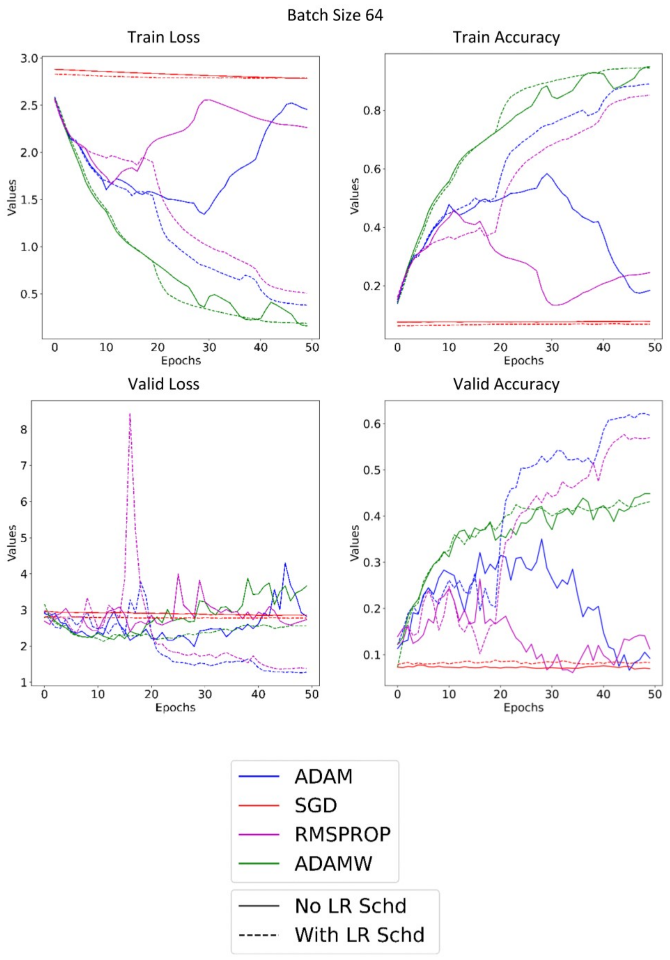

Fig. 4: Train Loss for various optimizer with batch size of 64 and LR = 1e-4.

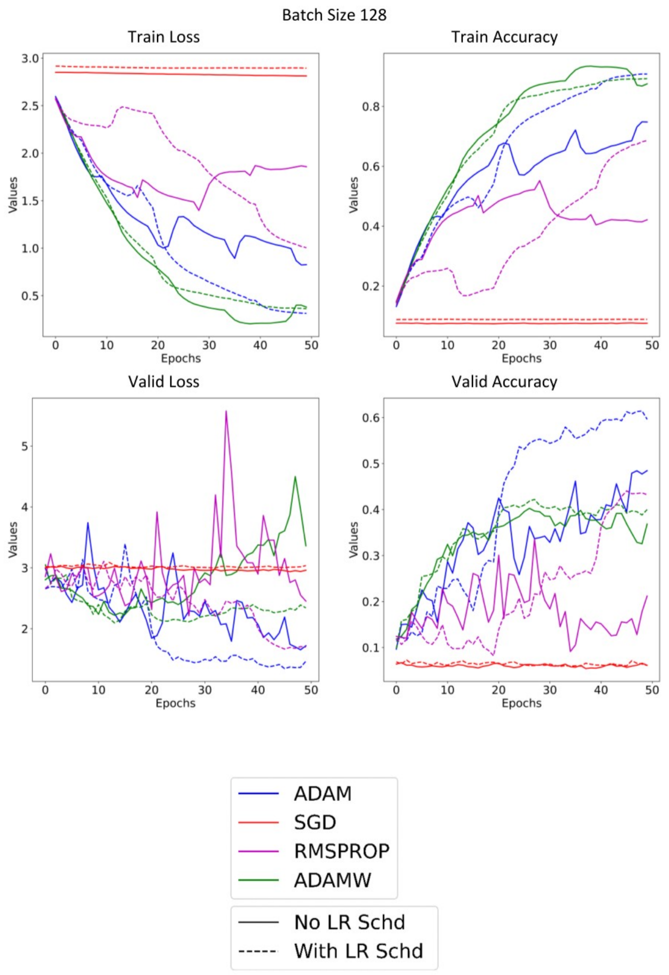

Fig. 5: Train Loss for various optimizer with batch size of 128 and LR = 1e-4.