Effect of Adaptive Noise Injection in Convolutional Neural Network Training

Logean Romain Nicolas,
Feillard Adrien Théo,
CS-439 - Mini-Project

***Abstract*—This paper explores the impact of noise injection on Convolutional Neural Networks and Resnet18 model trained for image classification on the CIFAR-10 dataset.**

Despite advances in machine learning, some challenges like poor convergence to optimal solutions and model overfitting persist, limiting the practical application and reliability of CNNs.

The injection of noise can be used to address both these issues.

This study investigates the role of the noise in helping the model to converge to optimal solution by escaping local minima or saddle points and also address the impact of the noise as a regularizer and on the generalization of the models.

Our solution is to inject an adaptive noise, that is activated and scaled using the model parameters, such as the loss or the gradient. The goal is to inject the noise only at specific epochs, when the model might be stuck on a local minimum or might be overfitting. We tried this noise injection on four different models, using different optimizers and highlighted the strong regularizer effect but limited accuracy optimization.

I. Introduction

Convolutional Neural Networks are widely used today across various application, and are particular efficient on images classification. However, they face the same problems as many Neural Networks; they are rarely able to find an optimal solution and are prone to overfitting. A lot of well-known solution exist for this such as the momentum or the regularization. But most of these methods lack the adaptability to respond to a model's real-time convergence behavior.

Our solution implements a specific conditional Noise scheduler to inject noise precisely when the model appears to be stagnating in a local minimum or showing signs of overfitting, by analyzing real-time indicators such as the loss and the gradient norm evolution. This adaptive noise injection aim to both enable the model to escape suboptimal regions of the loss landscape for improved convergence, and acting as a form of regularization to enhance generalization.

To achieves this we choose to work on a classification task using various CNNs architecture, and a Resnet18 model on the dataset CIPHAR-10 (see II-A). Our implementation was initially based on an existing GitHub repository [1], that is a library with well-known image classification models on CIPHAR-10.

II. Models and Methods

A. Dataset

The dataset used during these experiments is CIFAR-10 [2]. It is a dataset mainly used for image classification that contains 60,000 color images, each of size 32x32 pixels, distributed across 10 distinct classes. The dataset is inherently split into 50,000 training images and 10,000 test images. To prepare the data for our models, we normalize the images and apply common data augmentation techniques : RandomCrop and RandomHorizontalFlip.

B. CNN models

The adaptive noise injection was tested on four different CNN architecture. Three of these are custom CNN of varying depth, created to investigate the impact of noise injection across different levels of model complexity. The last model is a Resnet18.

1) BabyCNN: The BabyCNN is a very simple CNN custom model, containing 20,714 trainable parameters. The idea behind this model architecture is that an underfitting model might benefit from noise to escape local minima or saddle points. This model is more precisely composed of:

- **Conv Layer 1:** 3→8 filters (k=3, p=1), followed by ReLU and Max Pooling (reduces 32×32→16×16).
- **Linear Layer 1:** 2048→10 (from flattened 8×16×16).

2) TinyCNN: The TinyCNN is a more complex CNN model, containing 545,098 trainable parameters. It is design to capture more complex features than the BabyCNN while remaining relatively small. This model is composed of:

- **Conv Layer 1:** 3→32 filters (k=3, p=1), followed by ReLU and Max Pooling (reduces 32×32→16×16).
- **Conv Layer 2:** 32→64 filters (k=3, p=1), followed by ReLU and Max Pooling (reduces 16×16→8×8).
- **Linear Layer 1:** 4096→128 (from flattened 64×8×8), followed by ReLU.
- **Linear Layer 2:** 128→10.

3) SimpleCNN: The SimpleCNN is our most complex custom model, containing 22,954,506 trainable parameters. This model is very simple in its composition but contains more trainable parameters than Resnet18 (See II-B4). However, it does not benefit from any regularization methods. The idea is to use this model to observe the regularization effect of the noise injection. This model is composed of:

- **Conv Layer 1:** 3→256 filters (k=3, p=1), followed by ReLU and Max Pooling (reduces 32×32→16×16).
- **Conv Layer 2:** 256→512 filters (k=3, p=1), followed by ReLU and Max Pooling (reduces 16×16→8×8).
- **Conv Layer 3:** 512→512 filters (k=3, p=1), followed by ReLU and Max Pooling (reduces 8×8→4×4).

- **Linear Layer 1:** 8192→2048 (from flattened 512×4×4), followed by ReLU.
- **Linear Layer 2:** 2048→1024, followed by ReLU.
- **Linear Layer 3:** 1024→512, followed by ReLU.
- **Linear Layer 4:** 512→10.

4) ResNet-18: The ResNet-18 [3] is a widely recognized deep convolutional neural network, composed of 11,173,962 trainable. It is well known for its use of residual connections that help alleviate the vanishing gradient problem in deeper architectures.

For our experiments, we used ResNet-18 model. The model's final classification layer was replaced to match the 10 classes of the CIFAR-10 dataset (1000→10 output features).

C. Training

We trained every CNNs architecture during 300 epochs, with a weight decay of 10^{-4}. We tested various initial learning rates (η_{max}) (between 0.1 to 10^{-4}) and only reported the result of the best learning rate. We used three different optimizer, SGD without momentum, SGD with a momentum of 0.9 and Adam (with beta1 = 0.9 and beta2 = 0.999). The loss function we used is the CrossEntropyLoss.

1) Learning rate scheduler: A dynamic learning rate has been implemented through a scheduler called Cosine annealing with decaying restarts Learning rate scheduler. It consists in a combination of $torch.optim.lr\_scheduler.LinearLR$ and $torch.optim.lr\_scheduler.CosineAnnealingLR$ following those equations:

$$
\begin{aligned}
T_0 &= E_{restart} \cdot S_{epoch} \\
T_{decay} &= (E_{max} - E_{restart}) \cdot S_{epoch} \\
n &= \left\lfloor \log_{T_{mult}} \left(\frac{t}{T_0}(T_{mult} - 1) + 1 \right) \right\rfloor \\
T_i &= T_0 \cdot T_{mult}^n \\
T_{cur} &= t - T_0 \frac{T_{mult}^n - 1}{T_{mult} - 1} \\
\eta_{max,t} &= \eta_{max} - \min\left(1, \frac{t}{T_{decay}}\right) \cdot (\eta_{max} - \eta_{final\_max}) \\
\eta_t &= \eta_{min} + \frac{1}{2}(\eta_{max,t} - \eta_{min}) \left(1 + \cos\left(\frac{\pi \cdot T_{cur}}{T_i}\right)\right)
\end{aligned}
$$

η_t is the final learning rate at the current step t. This is calculated using: η_{max}, the initial maximum learning rate; η_{min} , the absolute minimum learning rate; and $\eta_{finalmax}$, the target maximum learning rate after decay at the last learning rate restart. The scheduler's state is defined by T_i, the duration of the current cycle, and T_{curr}, the position within that cycle. These are updated based on T_0, the duration of the first cycle, and T_{mult}, a cycle growth factor. The decay of the maximum learning rate occurs over T decay total steps, which is calculated from the total epochs $E_{max}$and the restart period in epochs $E_{restart}$. These conversions from epoch-based units to step-based units are performed using the variable S_{epoch}, which represents the number of steps (batches) within a single epoch. With the specified parameters in Table IV, taking an intiallearning rate with a value of 10^{-3} at epoch 0 it will decrease following a cosinus schedule over 50 reaching at minimum at 10^{-6}. Then at the restart the learning take a value lower than 10^{-3} such that at the last restart (epoch 250), the learning rate value is 10^{-5} and decay with the same cosinus pattern until 10^{-6}. This way, the trade off between exploration and precision in choosing an absolute learning rate value is avoided.

2) Noise scheduler: Each models were following the same parametrized process of noise injection. The models were set to evolve without noise injection until a specified number of epochs ($MinEpochs_{beforeflag}$). This way it would allow the model to learn classification and arrive to a local optimum. Once this minimum of epochs without noise injection is passed an adaptive noise triggering flag process becomes active. The flag trigger process works by detecting gradient and weights stagnation evolution. Across an epoch window size ($Windowsize_{flag}$) specified in input parameters, 4 metrics are evaluated, the average gradient norm rate of change normalized on the window size , the average weights update norm normalized on the window size, the validation loss evolution on the window size and the train loss evolution on the window size. If either the average gradient norm rate of change or average weight norm rate of change reach a threshold (either ($Gradient_{plateau}$) or ($LowWeightupdate$)) then the flag activated will trigger noise injection on the gradient. If either the Validation Loss rate of change reach a threshold ($ValPlateau\Delta$) or the number of increasing Train Loss and Validation loss across a number of epochs reach ($NEpochsOverfit$) , the flag activated will trigger noise injection on the weights. Those two different flags category are reasoned with the idea that the noise injection on the gradient would force the model to explore new possibilities and reach a new local optimum, while the noise injection on the weights would help as a regularizer that would prevent overfitting.

Injected noise consists of generating a random noise tensor that matches the shape of the input tensor and follows either a uniform or gaussian distribution. For the Gaussian distribution, a tensor Z of the target shape, where each element is sampled from the standard normal distribution is initialized:

$$Z \sim \mathcal{N}(0, I), N = \sigma \cdot Z$$

For the uniform distribution, a tensor N is initialized by sampling each element from $[-a, a]$:

$$N \sim \mathcal{U}(-a, a), a = \sqrt{3} \cdot \sigma$$

In this setup the constant $\sqrt{3}$ is needed to ensure the uniform distribution has the same standard deviation as the Gaussian distribution and allow comparison between the two distributions. Each distribution are scaled by σ, which represent the magnitude of the noise calculated based on the epochs performance of the model and is calculated this way:

$$k = \frac{2}{\delta_{plateau}}$$
$$F_{stuck} = 1 - Sigmoid_{val}(k \cdot Improvement_{\%} - \delta_{plateau})$$
$$\sigma_{raw} = \sigma_{min} + F_{stuck} \cdot (\sigma_{max} - \sigma_{min})$$
$$D_{schedule} = 0.5 \cdot (1 + \cos(\pi \cdot \frac{Epoch_{current}}{Epoch_{max}}))$$
$$\sigma = \sigma_{final} = \sigma_{raw} \cdot D_{schedule}$$

$\delta_{plateau}$ represent the threshold on the evolution of gradient or validation loss that triggers the noise injection flag. F_{stuck} is metric representing how stuck the evolution what a value between 0 and 1. $Improvement_{\%}$ represent the improvement percentage of the gradient or validation loss over the epoch window size. The higher the gradient or validation loss goes, lower is the $Improvement_{\%}$ and the closer to 1 the F_{stuck} is. The raw noise magnitude σ_{raw} gives the noise magnitude to input solely based on the model performance with a value $[\sigma_{min}, \sigma_{max}]$. It gives a noise magnitude to inject with respect on how stuck the gradient or validation evolution is. Finally, the final noise magnitude is dampens based on the advancement of the model by $D_{schedule}$. The closest we are to the end of the model training, the less noise is injecting, allowing stabilization towards the end.

A cooldown (CD) on the noise injection timing has also been implemented to avoid injecting noise too frequently and allowing the model to learn from the injected noise. Cooldown was calculated based on the gradient stuckness factor, as it has been the most used way to inject noise and normalized on a threshold interval $[CD_{min}, CD_{max}]$ following this equation $CD = CD_{min} + F_{stuck} \cdot (CD_{max} - CD_{min})$. The use of a cooldown is also conditioned by the number of subsequent epochs where noise is injected ($ConsecutiveTrigger$). Thresholds used are described in the table V.

III. Results

TABLE I
MODEL RESULTS FOR SGD

Model	Noise Type	Accuracy		Loss	
		Train	Val	Train	Val
Resnet18	**No noise**	0.9989	0.8812	0.0051	0.6702
	Gaussian	0.8752	0.8172	0.3511	0.5475
	Uniform	0.8072	0.7754	0.5495	0.6354
SimpleCNN	**No noise**	0.8102	0.7908	0.5518	0.6078
	Gaussian	0.8087	0.7890	0.5543	0.6105
	Uniform	0.8108	0.7892	0.5508	0.6100
TinyCNN	**No noise**	0.8971	0.8318	0.2963	0.5491
	Gaussian	0.7497	0.7564	0.7132	0.7205
	Uniform	0.7489	0.7524	0.7138	0.7322
BabyCNN	**No noise**	0.5451	0.5770	1.3067	1.2235
	Gaussian	0.5441	0.5748	1.3089	1.2262
	Uniform	0.5435	0.5748	1.3091	1.2271

The results for the the different optimizers used reflects several insights on the effect of noise injection on weights or gradients. For the parameters specified, the noise injection was mainly located on the gradient as displayed in Figure 1. Overall the use of noise injection has a equal or negative impact on the classification accuracy. Almost all the training accuracy and validation accuracy are lower when injecting noise and almost all training loss and validation loss are higher when injecting noise. No differences are noticeable between the different noise distribution. However the noise injection acted as a strong regularizer the models Resnet 18, SimpleCNN and TinyCNN, when using the SGD optimizer, and SGD optimizer with momentum. For example when the noise is injected in the TinyCNN using SGD, the gap between the loss on Training data and Validation data drop from 0.2528 to around 0.02. For Adam optimizer the regularizing effect is non existent for the Resnet 18 and SimpleCNN models. For Tiny CNN, we can observe a strong regularizer effect.

TABLE II
MODEL RESULTS FOR SGD WITH MOMENTUM

Model	Noise Type	Accuracy		Loss	
		Train	Val	Train	Val
Resnet18	**No noise**	0.9998	0.9130	0.0010	0.4997
	Gaussian	0.8031	0.7938	0.5571	0.6080
	Uniform	0.8025	0.7928	0.5607	0.6110
SimpleCNN	**No noise**	0.9998	0.8978	0.0011	0.5719
	Gaussian	0.8445	0.8236	0.4430	0.5233
	Uniform	0.8448	0.8148	0.4466	0.5487
TinyCNN	**No noise**	0.9006	0.8326	0.2861	0.5249
	Gaussian	0.7529	0.7598	0.7027	0.7158
	Uniform	0.7887	0.7886	0.6084	0.6178
BabyCNN	**No noise**	0.5909	0.6296	1.1908	1.1053
	Gaussian	0.5799	0.6174	1.2220	1.1262
	Uniform	0.5528	0.5900	1.2796	1.1904

TABLE III
MODEL RESULTS FOR ADAM

Model	Noise Type	Accuracy		Loss	
		Train	Val	Train	Val
Resnet18	**No noise**	0.9998	0.9034	0.0009	0.5117
	Gaussian	0.9823	0.8934	0.0548	0.4070
	Uniform	0.9813	0.8936	0.0567	0.4126
SimpleCNN	**No noise**	0.9999	0.8740	0.0011	0.8718
	Gaussian	0.9779	0.8564	0.0646	0.6843
	Uniform	0.9777	0.8558	0.0645	0.6527
TinyCNN	**No noise**	0.8480	0.8078	0.4435	0.5677
	Gaussian	0.7948	0.7858	0.5912	0.6250
	Uniform	0.7942	0.7858	0.5921	0.6300
BabyCNN	**No noise**	0.5890	0.6218	1.1941	1.1140
	Gaussian	0.5525	0.5938	1.2859	1.1879
	Uniform	0.5579	0.5906	1.2790	1.1945

An interesting behaviors of our model appear on the smallest architectures (BabyCNN and TinyCNN), the validation accuracy is higher than the training accuracy. Due to their small number of trainable parameters, they struggle with the augmented complexity induced by the data-augmentation, that's probably the reason of such a weird result.

IV. Conclusion

From the results observed, the effects of a targeted injection of noise into the weights or the gradients are mitigated. Although it act as a strong regularizer for complex enough models, the negative impact on accuracy remains too high, implying that the validation accuracy become lower than the baseline. Hence, the noise injection as a regularizer, seems suboptimal when compared with traditional regularizer.

APPENDIX

TABLE IV
INPUT PARAMETER OF THE COSINE ANNEALING WITH DECAYING RESTARTS LEARNING RATE SCHEDULER.

Parameters	Value in Experiment	Description
η_{max}	$10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$	Sets the initial maximum learning rate for the scheduler.
η_{final_max}	1^{-5}	Sets the maximum learning rate for the scheduler of the last restart.
η_{min}	1^{-6}	Sets the minimum learning rate for the scheduler.
E_{max}	300	Sets the total training duration, used to calculate the decay period of the maximum learning rate.
$E_{restart}$	50	Sets the restart period in epochs, which defines the initial cycle length (T_0) of the scheduler.

TABLE V
THRESHOLDS AND PARAMETERS FOR ADAPTIVE NOISE INJECTION.

Parameter	Value	Description
$MinEpochs_{beforeflag}$	70	Min epochs before adaptive flag checks begin.
$Windowsize_{flag}$	10	History window (in epochs) for flag evaluation.
$ValPlateau\Delta$	1e-4	Min improvement to not be a validation loss plateau.
$NEpochsOverfit$	3	Epochs validation loss must increase for the overfit flag.
$Gradient_{plateau}$	0.025	Relative improvement threshold for gradient norm.
$LowWeightupdate$	5e-4	Threshold for the low weight update norm flag.
σ_{min}	0.01	The minimum noise magnitude to apply.
σ_{max}	0.05	The maximum noise magnitude to apply.
$ConsecutiveTrigger$	10	Consecutive noisy epochs to trigger a cooldown.
CD_{min}	3	The minimum duration of the cooldown period.
CD_{max}	10	The maximum duration of the cooldown period.

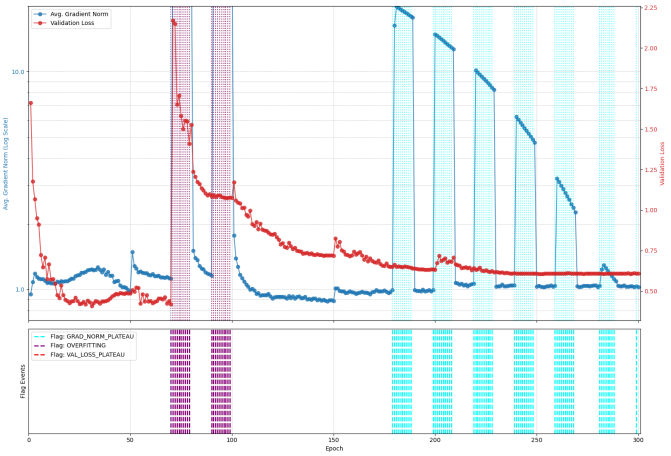

Fig. 1. Example of Gradient and Validation Loss evolution during training (Resnet18, Gaussian noise distribution, SGD optimizer)

REFERENCES

[1] huyvnphan, "PyTorch_CIFAR10," GitHub repository, 2020.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009, citing "The CIFAR-10 dataset".

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html