POLITECNICO DI TORINO

Master's degree in Computer Engineering

Master's degree Thesis

Study and analysis of training strategies to improve the reliability of artificial neural networks.

Supervisor

Prof. Ernesto Sánchez Sánchez

Candidate

Gabriel Cerón Viveros

ACADEMIC YEAR 2020-2021

Abstract

In the latest years we have seen an increased us of machine learning applications due to the increasing computational power and the development of more advanced techniques to train and implement these programs. Machine learning applications are computer programs that can be trained with real world data to perform a task without explicit programming. One of the most popular and widely used machine learning algorithm is the artificial neural network (ANN) and specially the deep neural networks (DNNs) which have shown to perform even above human precision. The great performance of DNNs have found successful applications in various areas such as avionics, automotive and medical devices. Some of these areas are considered safety-critical because system failures can compromise human lives. For this reason, in the last decades there has been an increasing interest by the research community to understand and improve the reliability of these computing models.

In this work we present a study and analysis of different methods to train DNNs to improve their reliability. Specifically, we train a residual network of 18 layers (ResNet18) with different training parameters and optimizers to see how much the accuracy of these trained models decreases in the presence of faults. We perform fault injections at a software level using the PytorchFI library which works over the framework Pytorch. The fault model implemented is a weight fault that at the moment of being accessed it reads the value of zero instead of the real weight value. The faults are injected progressively for a total of 0.1% and 1% of the total trainable parameters of the ResNet18 (11 million). The results show that there are subtle relations between the reliability and the tested training parameters such as Batch Size and Weight Decay, these parameters show different behavior on each optimizer so specific values for the optimizers are recommended. However, the most important result in this work is obtained at the moment of comparing reliability between optimizers. After selecting the models which gave the best reliability from the fault injection campaigns, we observe from the experiments that the optimizer which performs better in terms of reliability is SGD, followed by Adagrad, then Rmsprop and finally Adam.

Another experiment was done with the SGD optimizer, we used the fault model bit-flip to see if there is a bit of the weight (float-64) which is more sensible to this kind of fault, to do so we injected 1k and 10k faults on random weights but applying the bit-flip in the same bit each time. The results showed that bits 54 and 62 are very sensible to this fault and, as these bits determine the exponent of the floating-point number, they drop the accuracy significantly.

Contents

1 Introduction 10
2 Background 11
2.1 Fault models 11
2.2 Pytorch 12
2.3 PytorchFI 13
2.4 Resnet18 15
2.4.1 Resnet18 architecture: 16
2.5 Optimizers 18
2.5.1 Stochastic Gradient Descent (SGD) 18
2.5.2 Adam 19
2.5.3 Adagrad 22
2.5.4 Rmsprop 23
2.6 Scheduler 24
2.6.1 Cosine Annealing 24
2.6.2 Reduce on Plateau 25
2.7 Cross Entropy Loss 26
2.8 Batch Size 27
2.9 Regularization and weight decay 28
2.10 CIFAR-10 30
2.11 Floating-point format (IEEE 754 - float64) 32
3 Experimental results and analysis 34
3.1 Selecting a scheduler 34
3.2 SGD optimizer 36
3.2.1 SGD -- Varying the weight decay 37
3.2.2 Bit fault injections 43
3.3 Adam optimizer 46
3.3.1 Adam – Varying the batch size 47
3.3.2 SGD -- Varying the batch size 52
3.3.3 Adam – Varying the weight decay 62

3.4 Adagrad optimizer 68
3.4.1 Adagrad – Varying the batch size 68
3.4.2 Adagrad – Varying weight decay 74
3.5 Rmsprop optimizer 79
3.5.1 Rmsprop -- Varying the batch size 79
3.5.2 Rmsprop – Varying the weight decay 82
3.6 Optimizer Comparison 87
4 Conclusions 96
5 References .... **Error! Bookmark not defined.**

List of Figures

Figure 1. Popularity of ML frameworks Pytorch and Tensorflow over time. 12
Figure 2. Multiple neuron perturbations performed in the YOLOv3 using the PytorchFI library. Left: original output. Right: output with injections. 13
Figure 3. PytorchFI error models. 14
Figure 4. PytorchFI runtime overhead. 15
Figure 5. 20-layer and 56-layer error difference. 15
Figure 6. ResNet basic block. 16
Figure 7. ResNet18 architecture 17
Figure 8. ResNet18 architecture details. 17
Figure 9. Error of multiple optimizers on the CIFAR-10 dataset... 19
Figure 10. Cost function of multiple optimizers on the MNIST dataset. 21
Figure 11. Weight updating formula.. 22
Figure 12. Iterative weight updating formula.. 22
Figure 13. Adagrad weight updating formula.. 22
Figure 14. Adagrad $\eta t'$ equation.. 22
Figure 15. Definition of Rmsprop algorithm... 23
Figure 16. Equations for cosine annealing scheduler. 25
Figure 17. Equation 2 for cosine annealing scheduler.. 25
Figure 18. Cross entropy loss definition. 27
Figure 19. Cross entropy cost function definition. 29
Figure 20. Regularization term. 29
Figure 21. Cross entropy cost function with regularization term.. 29
Figure 22. Generalized formula of L2 regularization 29
Figure 23. Gradient Descent Learning Rule for Weight Parameter. 30
Figure 24. Classes in the CIFAR-10 dataset and sample images. 31
Figure 25. 64 bit Floating point format (IEEE 754).. 33
Figure 26. Training Resnet18 with cosine annealing scheduler. 35
Figure 27. Training ResNet18 with reduce on plateau scheduler. 36
Figure 28. Training with SGD optimizer, WD = 0 37
Figure 29. Training with SGD optimizer, WD = 5e-7 38
Figure 30. Training with SGD optimizer, WD = 5e-5 38
Figure 31. Training with SGD optimizer, WD = 5e-3 39
Figure 32. Training with SGD optimizer, WD = 5e-1 40
Figure 33. SGD test accuracy vs 10k injections for different weight decay values 41
Figure 34. SGD test accuracy vs 10k injections for different weight decay values, minus mean 41
Figure 35. SGD test accuracy vs 100k injections for different weight decay values 42
Figure 36. SGD test accuracy vs 100k injections for different weight decay values, epoch 80 ... 43
Figure 37. Test accuracy vs bit number after 1k injections 44
Figure 38. Accuracy vs bit number after 10k injections. 45

Figure 39. Accuracy vs bit number after 1 injection. 46
Figure 40. Training with Adam optimizer, batch size = 64 47
Figure 41. Training with Adam optimizer, batch size = 128 48
Figure 42.. Training with Adam optimizer, batch size = 256 48
Figure 43. Training with Adam optimizer, batch size = 512 49
Figure 44. Training with Adam optimizer, batch size = 1024 49
Figure 45. Adam Test accuracy vs 10k injections for different batch size values, epoch 160...... 50
Figure 46. Adam Test accuracy vs 10k injections for different batch size values, epoch 160, minus mean.......... 51
Figure 47. Adam Test accuracy vs 100k injections for different batch size values, epoch 160.... 51
Figure 48. Adam Test accuracy vs 100k injections for different batch size values, epoch 160, minus mean.......... 52
Figure 49. Training with SGD optimizer, batch size = 64 53
Figure 50. Training with SGD optimizer, batch size = 128 54
Figure 51. Training with SGD optimizer, batch size = 256 54
Figure 52. Training with SGD optimizer, batch size = 512 55
Figure 53.Training with SGD optimizer, batch size = 1024 55
Figure 54. Accuracy vs 10k fault injections for SGD and different batch sizes. 56
Figure 55. Accuracy vs 10k fault injections for SGD and different batch sizes, minus mean 57
Figure 56. Accuracy vs 100k fault injections for SGD and different batch sizes. 58
Figure 57. Accuracy vs 100k fault injections for SGD and different batch sizes on epoch 80...... 59
Figure 58. Accuracy vs 100k fault injections for SGD and different batch sizes on epoch 160.... 59
Figure 59. Accuracy difference between epoch 80 and 160 vs 100k fault injections. 60
Figure 60. Accuracy vs 100k fault injections for SGD adding batch size 2048. 61
Figure 61. Accuracy vs 100k fault injections for different batch sizes, minus mean. 61
Figure 62. Training with Adam optimizer, WD = 0.......... 62
Figure 63. Training with Adam optimizer, WD = 5e-7 63
Figure 64. Training with Adam optimizer, WD = 5e-5 63
Figure 65. Training with Adam optimizer, WD = 5e-3 64
Figure 66. Training with Adam optimizer, WD = 5e-1 65
Figure 67. Adam test accuracy vs 10k injections for different WD values, epoch 120 65
Figure 68. Adam test accuracy vs 10k injections for different WD values, epoch 120, minus mean 66
Figure 69. Adam test accuracy vs 100k injections for different WD values, epoch 120 67
Figure 70. Adam test accuracy vs 100k injections for different WD values, epoch 120, minus mean 68
Figure 71. Training with Adagrad optimizer, batch size = 64 69
Figure 72. Training with Adagrad optimizer, batch size = 128 69
Figure 73. Training with Adagrad optimizer, batch size = 256 70
Figure 74. Training with Adagrad optimizer, batch size = 512 71
Figure 75. Training with Adagrad optimizer, batch size = 1024 71

Figure 76. Adagrad test accuracy vs 10k injections for different batch size values, epoch 120 .. 72
Figure 77. Adagrad test accuracy vs 10k injections for different batch size values, epoch 120, minus mean... 73
Figure 78. Adagrad test accuracy vs 100k injections for different batch size values, epoch 120 73
Figure 79. Adagrad test accuracy vs 100k injections for different batch size values, epoch 120, minus mean... 74
Figure 80. Training with Adagrad optimizer, WD = 0.. 75
Figure 81. Training with Adagrad optimizer, WD = 5e-7 75
Figure 82. Training with Adagrad optimizer, WD = 5e-5 76
Figure 83. Training with Adagrad optimizer, WD = 05e-3 76
Figure 84. Training with Adagrad optimizer, WD = 5e-1 77
Figure 85. Adagrad test accuracy vs 10k injections for different WD values, epoch 120 77
Figure 86. Adagrad test accuracy vs 10k injections for different WD values, epoch 120, minus mean 78
Figure 87. Adagrad test accuracy vs 100k injections for different WD values, epoch 120 78
Figure 88. Adagrad test accuracy vs 100k injections for different WD values, epoch 120, minus mean 79
Figure 89. Training with Rmsprop optimizer, batch size = 64 80
Figure 90. Training with Rmsprop optimizer, batch size = 128 80
Figure 91. Training with Rmsprop optimizer, batch size = 256 81
Figure 92. Training with Rmsprop optimizer, batch size = 512 81
Figure 93. Training with Rmsprop optimizer, batch size = 1024 82
Figure 94. Training with Rmsprop optimizer, WD = 0 83
Figure 95. Training with Rmsprop optimizer, WD = 5e-7 83
Figure 96. Training with Rmsprop optimizer, WD = 5e-5 84
Figure 97. Training with Rmsprop optimizer, WD = 5e-3 84
Figure 98. Training with Rmsprop optimizer, WD = 5e-1 85
Figure 99. Rmsprop accuracy vs 10k weight injections for different WD values, epoch100 85
Figure 100. Rmsprop accuracy vs 10k injections for different WD values, epoch 100, minus mean 86
Figure 101. Rmsprop test accuracy vs 100k injections for different weight decay values, epoch 100 86
Figure 102. Rmsprop test accuracy vs 100k injections for different weight decay values, epoch 100, minus mean.. 87
Figure 103. Optimizer reliability comparison. 88
Figure 104. Optimizer reliability comparison, minus mean 88
Figure 105. Optimizers weights standard deviation vs epochs 89
Figure 106. Optimizers weights mean vs Epochs 90
Figure 107. SGD weights magnitude vs epochs.. 91
Figure 108. SGD training for injections at accuracy plateau 92
Figure 110. Fault injections in accuracy plateau... 93

Figure 111. Fault injections in accuracy plateau, minus mean 93
Figure 112. Training with SGD to save all epochs 94
Figure 113. Injections on equally spaced epochs, minus mean 95

Acronyms

ANN – Artificial Neural Network

Adagrad – Adaptive Gradient

Adam – Adaptive Moments

BS – Batch Size

CIFAR -- Canadian Institute for Advanced Research

CNN – Convolutional Neural Network

DNN – Deep Neural Network

ILSVRC – ImageNet Large Scale Visual Recognition Challenge

MNIST -- Mixed National Institute of Standards and Technology

PytorchFI – Pytorch Fault Injector

RMSprop – Root Mean Square Propagation

ResNet – Residual Network

SGD – Stochastic Gradient Descent

SGDR -- Stochastic Gradient Descent with Warm Restarts

WD – Weight Decay

1 Introduction

The need of automation has inspired researchers and engineers to develop more and more complex computer systems which can perform tasks with high precision and a human like dexterity. Some of these systems are inspired in the way humans learn and for this reason are called artificial neural networks (ANNs). ANNs consist of a collection of connected nodes called artificial neurons and like the synapses in a biological brain, they can signal to communicate with other neurons. The signals enter in the first layer and travel through the network giving an output as a result in the last layer. The ANNs are trained with real world data to classify or predict outcomes and can execute their assigned task without explicit programming. After the training a test phase is used to see how the network would perform in a real world application, the precision of the network is measured in an accuracy metric from 0% to 100% (or error metric from 100% to 0%).

Since the neural networks where first proposed [1], their accuracy has been increasing very fast thanks to the current computational power and more advanced techniques of the algorithms and architectures of the networks. Some of these architectures that have led to great performances are named deep neural networks (DNNs), which consist of a neural network with many layers between the first and last layer. Some of this DNNs have demonstrated they can surpass the human-level performance, for example Kaiming et al. [2] achieve 4.94% top-5 test error on the ImageNet dataset, and the human-level performance is 5.1% according to Russakovsky et al. [3] . The great performance of DNNs have found successful applications in various areas such as avionics [4], robotics [5] , automotive [6] and medical devices [7]. Some of these domains are considered safety-critical areas because system failures can compromise human lives. For these reasons, in the last decades the research community has shown a growing interest in understanding the reliability of these computing models [8] [9] [10].

In this work we will use DNNs, specifically ResNet18, to analyze and study training strategies with the purpose of obtaining a higher reliability. The network will be trained over the CIFAR-10 dataset and the reliability will be measured by varying training parameters to obtain different models and later inject faults at a software level, to see how much the accuracy decreases in the presence of faults. The weight injections will be done progressively for a total of 0.1% and 1% of the total trainable parameters of the ResNet18 (11 million). The training parameters to vary include the batch size (BS), weight decay (WD) and specific optimizer parameters, different optimizers for the network will be tested as well. In the conclusions we will give some recommendations of how a neural network such as ResNet18 should be trained and which parameter values to use in order to obtain the maximum reliability possible.

2 Background

Many technologies and terms will be used during the analysis of the training strategies to improve the reliability of artificial neural networks. This section will describe the main terms and technologies used in this work.

2.1 Fault models

To understand the faults and the fault injection process that may happen in a neural network, we need to clarify the terms defect, fault, error and failure.

A defect in an electronic system is the difference between the intended design and the implemented hardware. Typical defects are due to the process, the material, the age of the device and the package [10].

A fault is an anomalous physical condition or a defect in the system that might occur at the architectural level of a system. In literature, different fault models have been proposed, by their temporal characteristics, physical faults can be mainly classified as permanent or transient. A permanent fault is an unrecoverable defect in the system, such as wires assuming fixed logic values at 0 (stuck-at-0) or 1 (stuck-at-1), being non-reversible, the fault is stable and fixed over time and affects all the system computations. A transient fault is a defect in the system that is present for a short period of time, it is also known as an intermittent fault or soft error, and it may be due to external perturbations, radiations or disturbances. An example of this fault is a bit-flip, which changes the value of a bit from 0 to 1 or from 1 to 0.

An error, also referred as a behavioral error for exhibiting at the behavioral level, is an unexpected system behavior, for example, due to the activation of physical fault. Neural networks are viewed as distributed systems consisting of two components: neurons and synapses (the communication channels connecting the neurons). As for neurons, the error of a synapse is also independent of that of other synapses or neurons. Therefore, we can distinguish between two typologies of errors at the behavioral level:

- Crash: Neurons or synapses completely stop their activity. A crashed synapse can be modeled as a synapse weighted by value 0. Contrarily, to model a crashed neuron, the dropout fault model is exploited where the output of a neuron is purposely set to 0.
- Byzantine: Neurons or synapses keep their activity but send arbitrary values, within their bounded transmission capacity [11].

An error affecting a single neuron or synapses may not lead to a failure. This is not only related to the intrinsic definition of an error, but also to the ANN property of being over-provisioned.

A failure in a system occurs when the network, due to the manifestation of errors, wrongly predicts the output.

In this work we will implement permanent faults, specifically crash errors in synapses at the behavioral level, which will result in the reading of the value of a weight as 0 every time it is accessed. These faults will be injected at software level implementing the library PytorchFI which works over the framework Pytorch.

2.2 Pytorch

Pytorch is an optimized tensor library primarily used for Deep Learning applications using GPUs and CPUs. It is an open-source machine learning library for Python, mainly developed by the Facebook AI Research team. It is one of the widely used Machine learning libraries, others being TensorFlow and Keras. [12] Following we have a comparison of these search terms in Google:

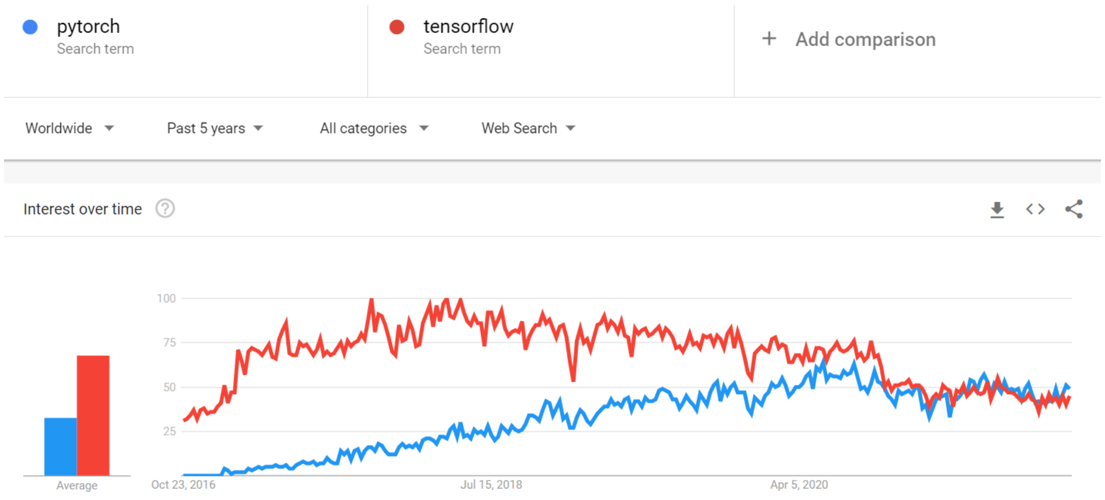


Figure 1. Popularity of ML frameworks Pytorch and Tensorflow over time.

Pytorch was second in popularity and this year (2021) it surpassed Tensorflow.

Why to choose Pytorch? Soon over, Google released its first version of TensorFlow in November 2015. TensorFlow not only became the default backend/engine for the Keras library, but also implemented a number of lower-level features that advanced deep learning practitioners and researchers needed to create state-of-the-art networks and perform novel research.

However, there was a problem, the TensorFlow API wasn’t very Pythonic, nor was it intuitive and easy to use. To solve that problem Pytorch, sponsored by Facebook and endorsed by Yann LeCun (AI researcher at Facebook), was released in September 2016. [13]

Pytorch solved much of the problems researchers were having with Keras and TensorFlow. While Keras is incredibly easy to use, by its very nature and design Keras does not expose some of the low-level functions and customization that researchers needed. [13]

2.3 PytorchFI

PytorchFI is a runtime perturbation tool for deep neural networks (DNNs), implemented for the popular Pytorch deep learning platform. PytorchFI enables users to perform perturbation on weights or neurons of a DNN during runtime. It is extremely versatile for dependability and reliability research, with applications including resiliency analysis of classification networks, resiliency analysis of object detection networks, training resilient models, and for DNN interpretability. [14]

Why to choose PytorchFI? With the recent advances of machine learning and artificial intelligence, deep neural networks (DNNs) have become a major player across various domains, from entertainment devices such as mobile phones to safety-critical applications such as self-driving cars and medical devices. These applications can be executed on a variety of different hardware platforms such as CPUs, GPUs or dedicated processors such as Google's TPUs, however there is mounting evidence that even tiny perturbations such as transient hardware errors or adversarial attacks can cause DNNs to output an incorrect value, this may have undesirable consequences for the application. To understand the effect perturbations may have on DNNs outcome, PytorchFI was developed. [15]

An example of a use case for PytorchFI is to simulate an error by performing a fault-injection on an object recognition model [14] :

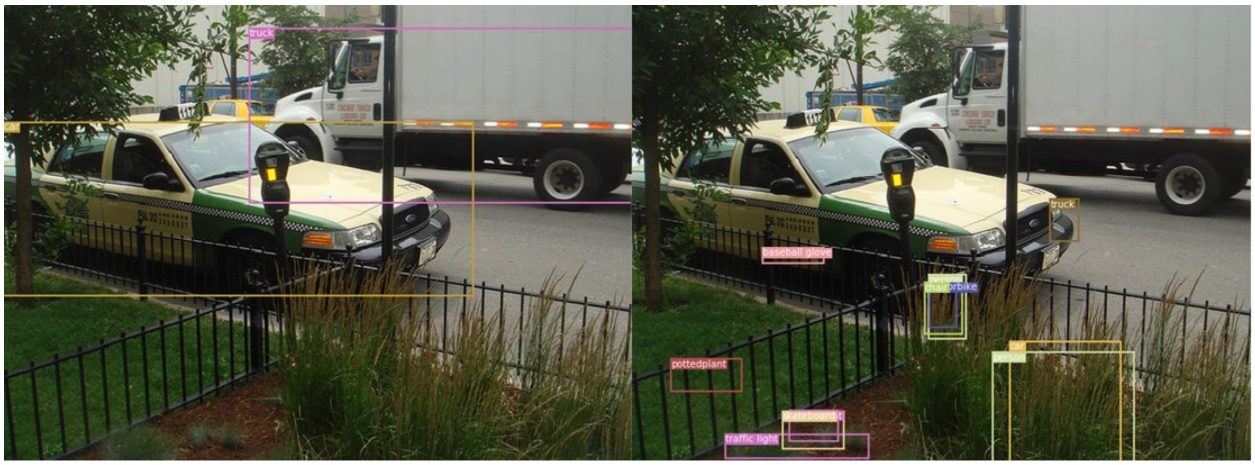


Figure 2. Multiple neuron perturbations performed in the YOLOv3 using the PytorchFI library. Left: original output. Right: output with injections.

In the figure above multiple neuron perturbations are performed with one error per layer in the YOLOv3 object detection network, changing the original neuron value with an erroneous value between -1 and 1. We can see in the graph at the right that the results are quite drastic, previously the network identified a single car and truck in the image however after perturbations the network not only misses the original objects but also identifies many incorrect objects.

How does it work? PytorchFI does not require the user to make any changes to their original model, rather it utilizes Pytorch's hook functionality to perform runtime perturbations on neurons, while weight perturbations are injected offline before the model starts running.

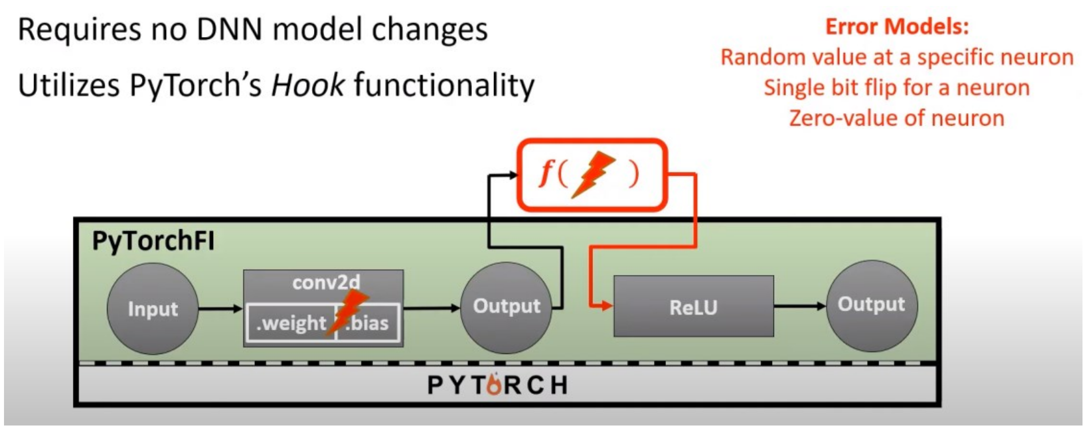

Requires no DNN model changes
Utilizes PyTorch's *Hook* functionality
Error Models:
Random value at a specific neuron
Single bit flip for a neuron
Zero-value of neuron
f( )
PyTorchFI
Input
conv2d
.weight
.bias
Output
ReLU
Output
PYTORCH

Figure 3. PytorchFI error models.

Essentially the hook implementation allows PytorchFI to modify the output of any specified neuron and then lets the execution continue seamlessly with the error injection in place, we can model various errors, such as random values for neurons, single bit flips or zeroing out a neuron, it can be thought as a function that transforms the original value into an erroneous value.

By using Pytorch's built-in hook API to perform perturbations, the authors found that the instrumentation cost is extremely low. In the following figure it is shown the runtime in seconds of running a single inference with and without PytorchFI perturbations across multiple networks and datasets, many runs were performed and averaged the runtime, and we can see that the average overhead of using PytorchFI for perturbations is negligible compared to the base runtime of the original error-free inference. [16]

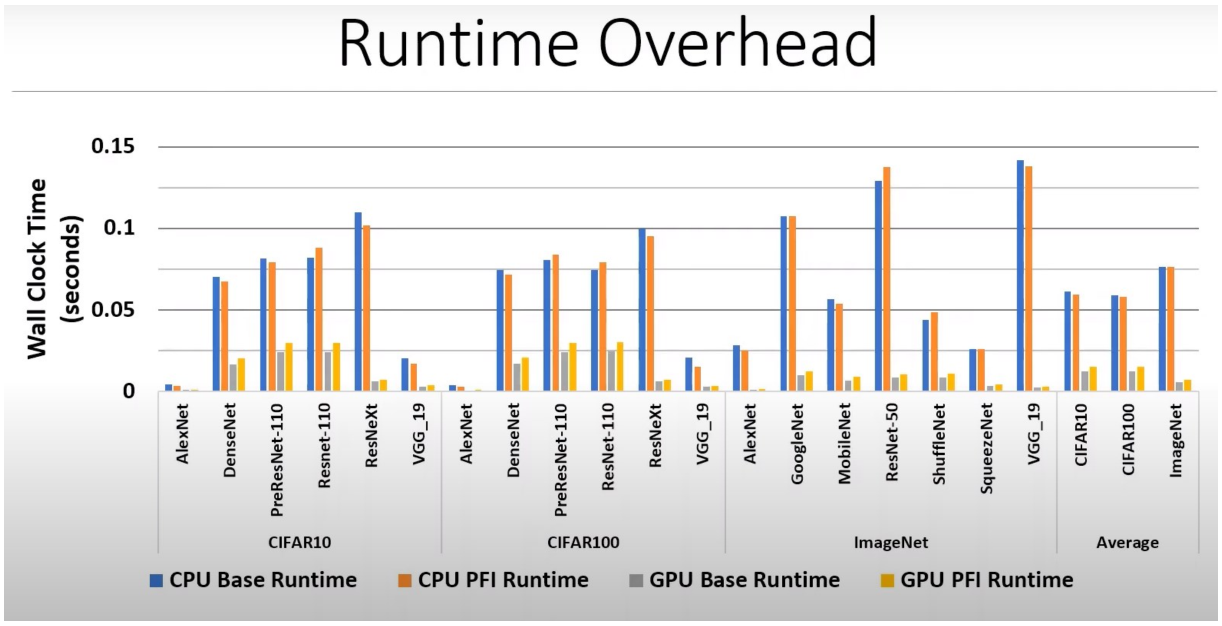


Figure 4. PytorchFI runtime overhead.

2.4 Resnet18

After the first CNN-based architecture (AlexNet) that won the ImageNet 2012 competition, every subsequent winning architecture uses more layers in a deep neural network to reduce the error rate. This works for a small number of layers, but when we increase the number of layers, there is a common problem in deep learning associated with that called Vanishing/Exploding gradient. This causes the gradient to become 0 or too large. Thus when we increase the number of layers, the training and test error rate also increases:

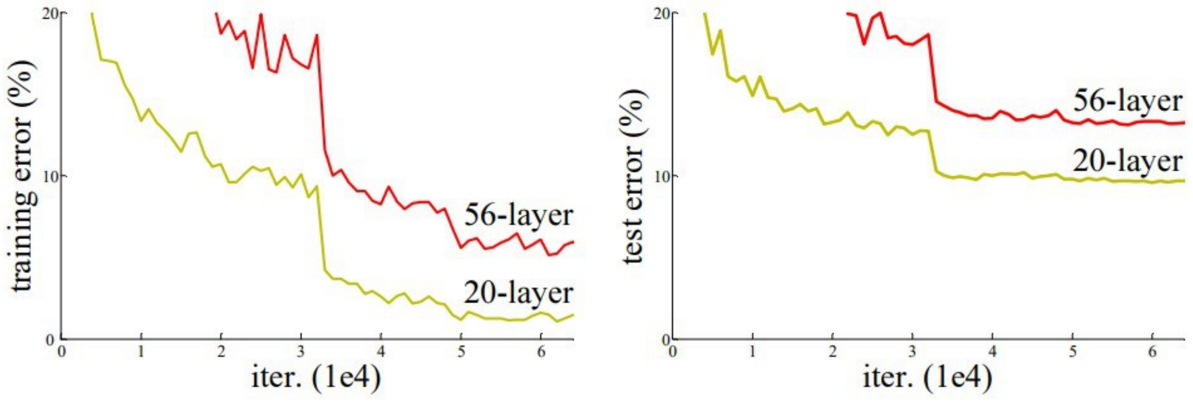


Figure 5. 20-layer and 56-layer error difference.

In the above plot, we can observe that a 56-layer CNN gives more error rate on both training and testing dataset than a 20-layer CNN architecture, if this was the result of over fitting, then we should have lower training error in 56-layer CNN but then it also has higher training error. After analyzing more on error rate the authors [17] were able to reach conclusion that it is caused by vanishing/exploding gradient.

In order to solve the problem of the vanishing/exploding gradient, a new architecture called Residual Network (Resnet) was proposed in 2015 by researchers at Microsoft Research. In this network we use a technique called skip connections. The skip connection skips training from a few layers and connects directly to the output.

The approach behind this network is instead of layers learning the underlying mapping, we allow the network fit the residual mapping [18]. Therefore we have the following diagram of a basic block of ResNet:

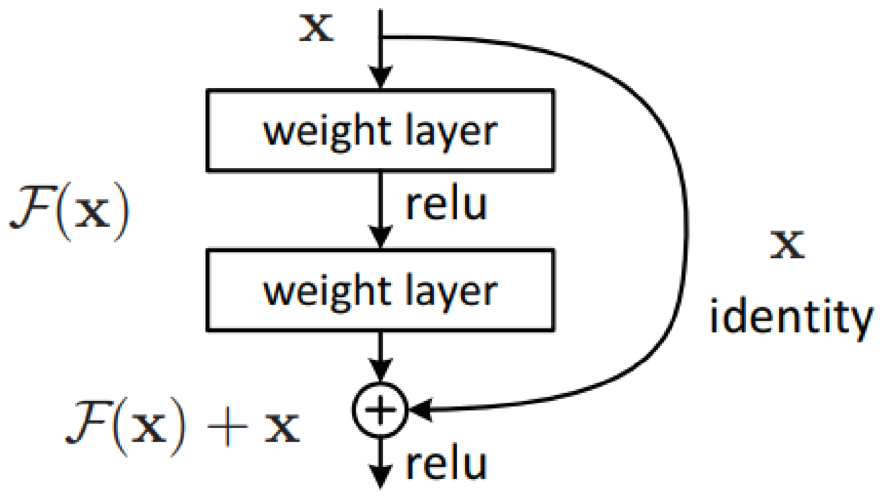

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 6. ResNet basic block.

The advantage of adding this type of skip connection is because if any layer hurt the performance of architecture then it will be skipped. So, this results in training very deep neural network without the problems caused by vanishing/exploding gradient. [18]

An ensemble of deep residual networks achieved a 3.57% error rate on ImageNet which achieved 1st place in the ILSVRC 2015 classification competition. [19] [17]

2.4.1 Resnet18 architecture:

One of the smallest residual networks proposed by the authors and thus one easiest to train is the Resnet18, the ResNet 18 architecture consists of an initial convolutional layer of kernel size 7x7, 4 blocks of convolutional layers each one with 2 pairs of kernels 3x3, and a final fully connected layer. So, the sum of the individual layers equal to 18. We can see the architecture of resnet18 in the figure below [20]:

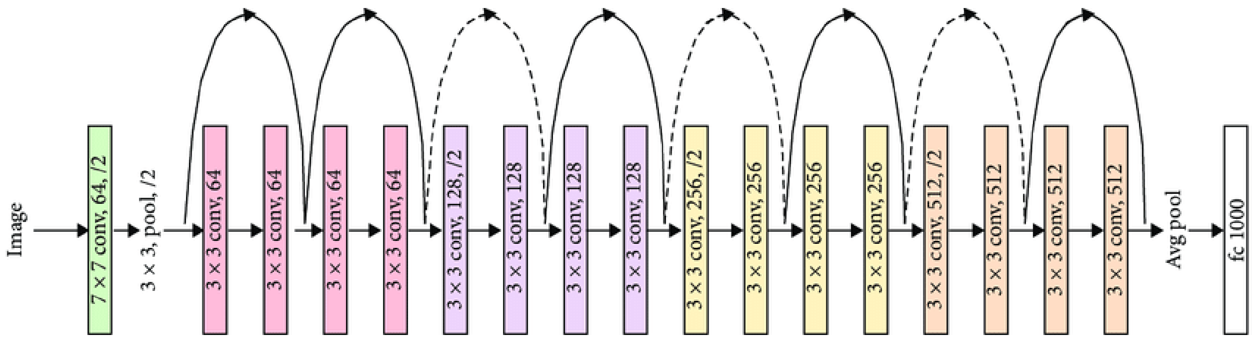
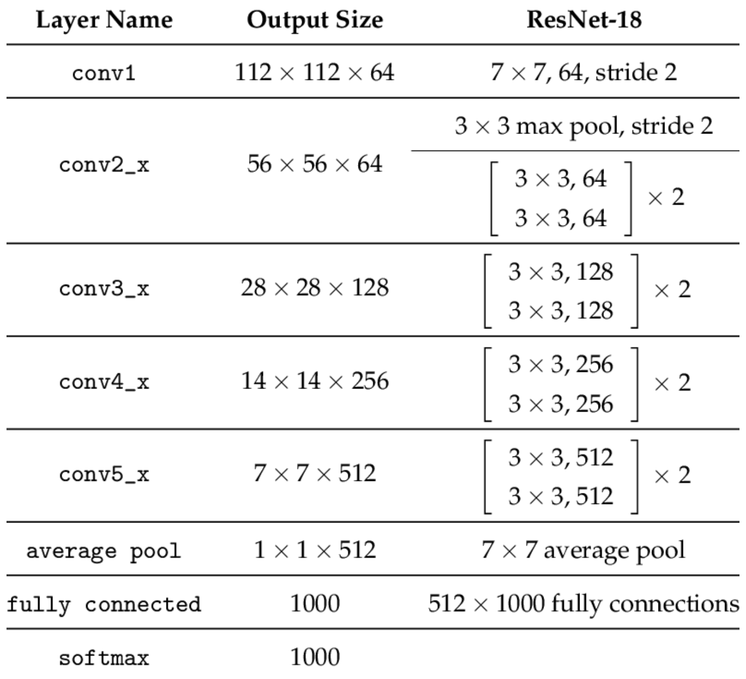


Figure 7. ResNet18 architecture

The architecture details and internal parameters are shown in the next figure [21]:



Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

Figure 8. ResNet18 architecture details.

2.5 Optimizers

The process of minimizing (or maximizing) any mathematical expression is called optimization. Optimizers are algorithms or methods used to change the attributes of the neural network such as weights and learning rate to reduce the losses, thus training the network and giving the results and outputs desired. We will study some of the most popular optimizers for neural networks to see how they may affect the reliability of the trained network.

2.5.1 Stochastic Gradient Descent (SGD)

Gradient Descent is the most basic but most used optimization algorithm. It’s used heavily in linear regression and classification algorithms. [22] Gradient descent follows the negative gradient of an objective function in order to locate the minimum of the function.

It is a simple and effective technique that can be implemented with just a few lines of code. It also provides the basis for many extensions and modifications that can result in better performance. The algorithm also provides the basis for the widely used extension called stochastic gradient descent (SGD), used to train deep learning neural networks.

It is technically referred to as a first-order optimization algorithm as it explicitly makes use of the first-order derivative of the target objective function. The first-order derivative, or simply the “derivative,” is the rate of change or slope of the target function at a specific point.

If the target function takes multiple input variables, it is referred to as a multivariate function and the input variables can be thought of as a vector. In turn, the derivative of a multivariate target function may also be taken as a vector and is referred to generally as the “gradient.”

The derivative or the gradient points in the direction of the steepest ascent of the target function for an input. Specifically, the sign of the gradient tells you if the target function is increasing or decreasing at that point.

- Positive Gradient: Function is increasing at that point.
- Negative Gradient: Function is decreasing at that point.

Gradient descent refers to a minimization optimization algorithm that follows the negative of the gradient downhill of the target function to locate the minimum of the function.

By definition, the optimization algorithm is only appropriate for target functions where the derivative function is available and can be calculated for all input values. This does not apply to all target functions, only so-called differentiable functions.

The main benefit of the gradient descent algorithm is that it is easy to implement and effective on a wide range of optimization problems.

Gradient descent refers to a family of algorithms that use the first-order derivative to navigate to the optima (minimum or maximum) of a target function.

There are many extensions to the main approach that are typically named for the feature added to the algorithm, such as gradient descent with momentum, gradient descent with adaptive gradients, and so on.

Gradient descent is also the basis for the optimization algorithm used to train deep learning neural networks, referred to as stochastic gradient descent, or SGD. In this variation, the target function is an error function and the function gradient is approximated from prediction error on samples from the problem domain. The SGD optimizer has shown to very convenient for the CIFAR-10 dataset showing low test error [23] :

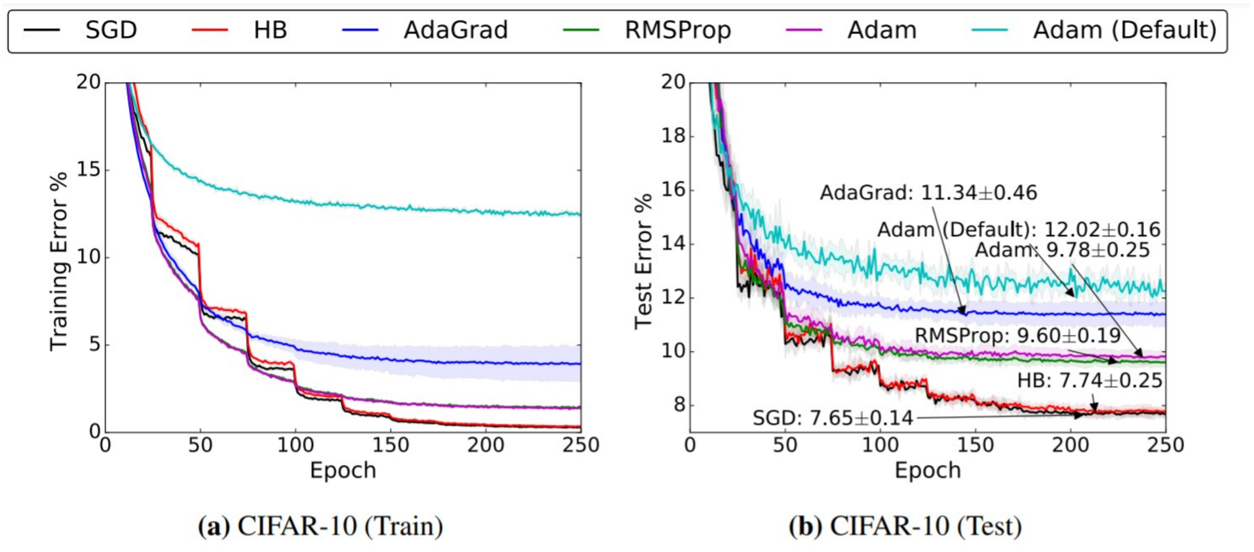


Figure 9. Error of multiple optimizers on the CIFAR-10 dataset.

2.5.2 Adam

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization“. [24]

Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds.

The authors describe Adam as combining the advantages of two other extensions of stochastic gradient descent. Specifically:

- Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
- Root Mean Square Propagation (Rmsprop) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Adam realizes the benefits of both Adagrad and Rmsprop. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in Rmsprop, Adam also makes use of the average of the second moments of the gradients (the uncentered variance).

Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

The initial value of the moving averages and beta1 and beta2 values close to 1.0 (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

Adam is a popular algorithm in the field of deep learning because it achieves good results fast. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods.

In the original paper, Adam was demonstrated empirically to show that convergence meets the expectations of the theoretical analysis. Adam was applied to the logistic regression algorithm on the MNIST digit recognition and IMDB sentiment analysis datasets, a Multilayer Perceptron algorithm on the MNIST dataset and Convolutional Neural Networks on the CIFAR-10 image recognition dataset. They conclude:

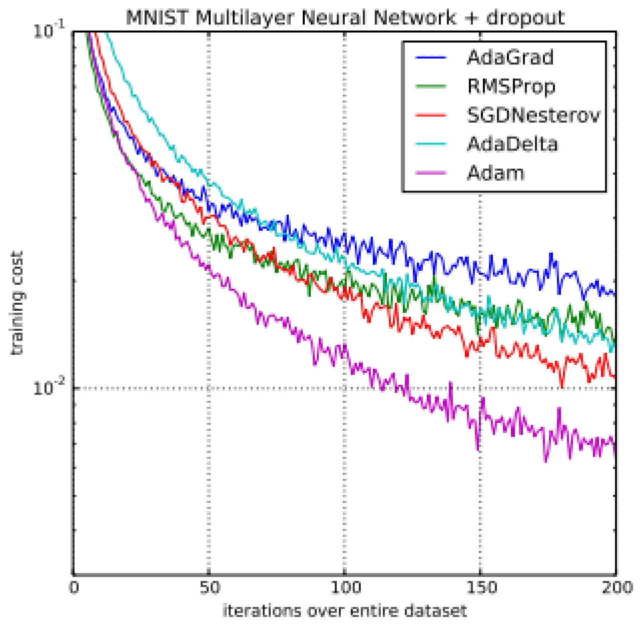


Figure 10. Cost function of multiple optimizers on the MNIST dataset.

Sebastian Ruder developed a comprehensive review of modern gradient descent optimization algorithms titled “An overview of gradient descent optimization algorithms” published first as a blog post, then a technical report in 2016. [25] The paper is basically a tour of modern methods. In his section titled “Which optimizer to use?“, he recommends using Adam.

The Adam configuration parameters in the Pytorch library are:

- params (iterable) – iterable of parameters to optimize or dicts defining parameter groups
- lr (float, optional) – learning rate (default: 1e-3)
- betas (Tuple[float, float], optional) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- eps (float, optional) – term added to the denominator to improve numerical stability (default: 1e-8)
- weight_decay (float, optional) – weight decay (L2 penalty) (default: 0)
- amsgrad (boolean, optional) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)

The Adam paper suggests:

Good default settings for the tested machine learning problems are alpha=0.001, beta1=0.9, beta2=0.999 and epsilon=10−8. [26]

2.5.3 Adagrad

Adagrad stands for Adaptive Gradient Optimizer. There were optimizers like Gradient Descent, Stochastic Gradient Descent, mini-batch SGD, all were used to reduce the loss function with respect to the weights. The weight updating formula is as follows:

$$(\mathrm{w})_{\text{new}} = (\mathrm{w})_{\text{old}} - \eta \frac{\partial \mathrm{L}}{\mathrm{dw}(\text{ old })}$$

Figure 11. Weight updating formula.

Based on iterations, this formula can be written as:

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w(t-1)}$$

Figure 12. Iterative weight updating formula.

Where: w(t) = value of w at current iteration, w(t-1) = value of w at previous iteration and η = learning rate.

In SGD, the value of η used to be the same for each weight, or say for each parameter. But in Adagrad Optimizer the core idea is that each weight has a different learning rate (η). This modification has great importance, in the real-world dataset, some features are sparse (for example, in Bag of Words most of the features are zero so it's sparse) and some are dense (most of the features will be noon-zero), so keeping the same value of learning rate for all the weights is not good for optimization. The weight updating formula for Adagrad looks like:

$$\mathrm{w}_{\mathrm{t}} = \mathrm{w}_{\mathrm{t}-1} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w}(\mathrm{t}-1)}$$

Figure 13. Adagrad weight updating formula.

Where alpha(t) denotes different learning rates for each weight at each iteration.

$$\eta'_{\mathrm{t}} = \frac{\eta}{\operatorname{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

Figure 14. Adagrad η'_t equation.

Here, η is a constant number, epsilon is a small positive value number to avoid divide by zero error if in case alpha(t) becomes 0 because if alpha(t) become zero then the learning rate will become zero which in turn after multiplying by derivative will make w(old) = w(new), and this will lead to small convergence.

Advantages of Adagrad [27]:

- No manual tuning of the learning rate required.
- Faster convergence
- More reliable

The available parameters in the Pytorch library for this optimizer are [28]:

- params (iterable) – iterable of parameters to optimize or dicts defining parameter groups
- lr (float, optional) – learning rate (default: 1e-2)
- lr_decay (float, optional) – learning rate decay (default: 0)
- weight_decay (float, optional) – weight decay (L2 penalty) (default: 0)
- eps (float, optional) – term added to the denominator to improve numerical stability (default: 1e-10)

2.5.4 Rmsprop

Rmsprop, which stands for Root Mean Square Propagation, is a gradient descent optimization unpublished extension, first described in Geoffrey Hinton's lecture notes for his Coursera course on neural networks, specifically Lecture 6e titled "Rmsprop: Divide the gradient by a running average of its recent magnitude." [29] [30]

Rmsprop was developed in order to overcome the short comings of the Adagrad algorithm. That is, Rmsprop does not decay the learning rate too quickly preventing convergence.

The similarities between the Adagrad and Rmsprop algorithms are immediately clear from the equations defining the Rmsprop algorithm:

$$s = \beta s + (1-\beta)\nabla_\theta J(\theta) \odot \nabla_\theta J(\theta)$$

$$\theta = \theta \ - \ \eta \nabla_\theta J(\theta) \oslash \sqrt{s+\epsilon}$$

Figure 15. Definition of Rmsprop algorithm.

where $\odot$ denotes the Hadamard product (element-wise multiplication) and $\oslash$ denotes Hadamard division (element-wise division).

Rmsprop makes use of exponential decay in order to manage the size of the vector s. Unlike Adagrad, Rmsprop decays the contribution of older gradients at each step. This prevents the magnitude of s from becoming so large that it prevents learning.

As a result of this exponential decay, the accumulated gradients in s are focused on recent gradients as opposed to all previous gradients.

The hyperparameter β is known as the decay rate. This hyperparameter is used to control the focus of the adaptive learning rate on more recent gradients.

Adagrad decays the learning rate too far before reaching the minima. Thus, it is unable to improve the loss after a given number of training steps. This results in the loss remaining constant until training is stopped.

In contrast to this Rmsprop is able to continually reduce the loss throughout the training process until the local minimum is reached. [31]

The parameters available in the Pytorch library for Rmsprop optimizer are:

- params (iterable) – iterable of parameters to optimize or dicts defining parameter groups
- lr (float, optional) – learning rate (default: 1e-2)
- momentum (float, optional) – momentum factor (default: 0)
- alpha (float, optional) – smoothing constant (default: 0.99)
- eps (float, optional) – term added to the denominator to improve numerical stability (default: 1e-8)
- centered (bool, optional) – if True, compute the centered Rmsprop, the gradient is normalized by an estimation of its variance
- weight_decay (float, optional) – weight decay (L2 penalty) (default: 0)

2.6 Scheduler

The scheduler schedules the change of the learning rate as epochs passes, it is important to use a scheduler to speed up the training process with a big learning rate and later increase the accuracy by reducing the learning rate. Some of the most common schedulers are Cosine Annealing and the Plateau Scheduler.

2.6.1 Cosine Annealing

Set the learning rate of each parameter group using a cosine annealing schedule, ηmax is set to the initial lr and Tcur is the number of epochs since the last restart in SGDR:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}.$$

Figure 16. Equations for cosine annealing scheduler.

When last_epoch=-1, sets initial lr as lr. Notice that because the schedule is defined recursively, the learning rate can be simultaneously modified outside this scheduler by other operators. If the learning rate is set solely by this scheduler, the learning rate at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

Figure 17. Equation 2 for cosine annealing scheduler.

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts [32]. Note that this only implements the cosine annealing part of SGDR, and not the restarts. [33]

The parameters of this scheduler that are found in the Pytorch framework are:

- optimizer (Optimizer) – Wrapped optimizer.
- T_max (int) – Maximum number of iterations.
- eta_min (float) – Minimum learning rate. Default: 0.
- last_epoch (int) – The index of last epoch. Default: -1.
- verbose (bool) – If True, prints a message to stdout for each update. Default: False.

2.6.2 Reduce on Plateau

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a ‘patience’ number of epochs, the learning rate is reduced. [34]

The parameters of this scheduler that are found in the Pytorch framework are:

- optimizer (Optimizer) – Wrapped optimizer.
- mode (str) – One of min, max. In min mode, lr will be reduced when the quantity monitored has stopped decreasing; in max mode it will be reduced when the quantity monitored has stopped increasing. Default: ‘min’.

- factor (float) – Factor by which the learning rate will be reduced. new_lr = lr * factor. Default: 0.1.
- patience (int) – Number of epochs with no improvement after which learning rate will be reduced. For example, if patience = 2, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn't improved then. Default: 10.
- threshold (float) – Threshold for measuring the new optimum, to only focus on significant changes. Default: 1e-4.
- threshold_mode (str) – One of rel, abs. In rel mode, dynamic_threshold = best * (1 + threshold) in 'max' mode or best * (1 - threshold) in min mode. In abs mode, dynamic_threshold = best + threshold in max mode or best - threshold in min mode. Default: 'rel'.
- cooldown (int) – Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.
- min_lr (float or list) – A scalar or a list of scalars. A lower bound on the learning rate of all param groups or each group respectively. Default: 0.
- eps (float) – Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: 1e-8.
- verbose (bool) – If True, prints a message to stdout for each update. Default: False.

2.7 Cross Entropy Loss

Also called logarithmic loss, log loss or logistic loss. Each predicted class probability is compared to the actual class desired output 0 or 1 and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. The penalty is logarithmic in nature yielding a large score for large differences close to 1 and small score for small differences tending to 0.

Cross-entropy loss is used when adjusting model weights during training. The aim is to minimize the loss, i.e, the smaller the loss the better the model. A perfect model has a cross-entropy loss of 0. [35]

Cross-entropy is defined as:

$$L_{\mathrm{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

Figure 18. Cross entropy loss definition.

2.8 Batch Size

Neural networks are trained using the stochastic gradient descent optimization algorithm. This involves using the current state of the model to make a prediction, comparing the prediction to the expected values, and using the difference as an estimate of the error gradient. This error gradient is then used to update the model weights and the process is repeated.

The more training examples used in the estimate, the more accurate this estimate will be and the more likely that the weights of the network will be adjusted in a way that will improve the performance of the model. The improved estimate of the error gradient comes at the cost of having to use the model to make many more predictions before the estimate can be calculated, and in turn, the weights updated.

Alternately, using fewer examples results in a less accurate estimate of the error gradient that is highly dependent on the specific training examples used.

This results in a noisy estimate that, in turn, results in noisy updates to the model weights, e.g. many updates with perhaps quite different estimates of the error gradient. Nevertheless, these noisy updates can result in faster learning and sometimes a more robust model.

The number of training examples used in the estimate of the error gradient is a hyperparameter for the learning algorithm called the "batch size".

A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated.

Historically, a training algorithm where the batch size is set to the total number of training examples is called "batch gradient descent" and a training algorithm where the batch size is set to 1 training example is called "stochastic gradient descent" or "online gradient descent."

A configuration of the batch size anywhere in between (e.g. more than 1 example and less than the number of examples in the training dataset) is called "minibatch gradient descent."

For shorthand, the algorithm is often referred to as stochastic gradient descent regardless of the batch size. Given that very large datasets are often used to train deep learning neural networks, the batch size is rarely set to the size of the training dataset.

Smaller batch sizes are used for two main reasons:

- Smaller batch sizes are noisy, offering a regularizing effect and lower generalization error.
- Smaller batch sizes make it easier to fit one batch worth of training data in memory (i.e. when using a GPU).

Nevertheless, the batch size impacts how quickly a model learns and the stability of the learning process. It is an important hyperparameter that should be well understood and tuned by the deep learning practitioner. [36]

2.9 Regularization and weight decay

Neural Networks are very good at approximating functions be linear or non-linear, and are also very good when extracting features from the input data. This capability makes them perform very well over a large range of tasks be it computer vision domain or language modelling.

However, their power of being great function approximators sometimes causes them to overfit the dataset by approximating a function which will perform extremely well on the data on which it was trained on but fails when tested on a data it hasn't seen before. To be more technical, the neural networks learn weights which are more specialized on the given data and fails to learn features which can be generalized.

To solve the problem of overfitting, a class of techniques known as Regularization is applied to reduce the complexity of the model and constraint weights in a manner which forces the neural network to learn generalizable features.

Regularization may be defined as any change we make to the training algorithm in order to reduce the generalization error but not the training error. There are many regularization strategies. Some put extra constraints on the models such as adding constraints to parameter values while some add extra terms to the objective function which can be thought as adding indirect or soft constraints on the parameter values. If we use these techniques carefully, this can lead to improved performance on the test set. In the context of deep learning, most regularization techniques are based on regularizing the estimators.

While regularizing an estimator, there is a tradeoff where we have to choose a model with increased bias and reduced variance. An effective regularizer is one which makes a profitable trade, reducing variance significantly while not overly increasing the bias.

L2 regularization belongs to the class of regularization techniques referred to as parameter norm penalty. It is referred to this because in this class of techniques, the norm of a particular parameter mostly weights are added to the objective function being optimized. In L2 norm, an extra term often referred to as regularization term is added to the cost function of the network.

For example, the cross-entropy cost function which is defined as shown below.

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right]$$

where y_j is the true class label and a_j^L is the predicted class label of layer L

Figure 19. Cross entropy cost function definition.

To apply L2 regularization to any network having cross-entropy loss, we add the regularizing term which is the following:

$$\frac{\lambda}{2n}\sum_w w^2.$$

Figure 20. Regularization term.

Now putting this together to form the final equation of L2 regularization applied to the cross-entropy loss function given by:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2.$$

Figure 21. Cross entropy cost function with regularization term.

The above example showed L2 regularization applied to cross-entropy loss function but this concept can be generalized to all the cost-functions available. A more general formula of L2 regularization is given below.

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2,$$

Figure 22. Generalized formula of L2 regularization

Where Co is the unregularized cost function and C is the regularized cost function with the regularization term added to it.

We don't consider the bias of the network when regularizing the network because regularizing a bias can introduce a significant amount of underfitting.

Let's try to understand the working of L2 regularization based on the gradient of the cost function.

If we take the partial derivative or gradient of equation presented below for all weights in the network, the learning rule for weights for gradient descent hence become:

$$\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \end{aligned}$$

Figure 23. Gradient Descent Learning Rule for Weight Parameter.

The above weight equation is similar to the usual gradient descent learning rule, except the now we first rescale the weights w by (1−(η*λ)/n). This term is the reason why L2 regularization is often referred to as weight decay since it makes the weights smaller. Hence you can see why regularization works, it makes the weights of the network smaller. The smallness of weights implies that the network behavior won't change much if we change a few random inputs here and there which in turn makes it difficult for the regularized network to learn local noise in the data. This forces the network to learn only those features which are seen often across the training set.

As we add the regularization term to the cost function we are actually increasing the value of the cost function. Hence, if the weights will be larger it will also make the cost to go up and the training algorithm will try to bring the weights down by penalizing the weights forcing them to take smaller values thereby regularizing the network. L2 Regularization and Weight Decay are not the same things but can be made equivalent for SGD by a reparameterization of the weight decay factor based on the learning rate. [37] [38]

2.10 CIFAR-10

The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. [39] [40]

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

CIFAR-10 is a set of images that can be used to teach a computer how to recognize objects. Since the images in CIFAR-10 are low-resolution (32x32), this dataset can allow researchers to quickly try different algorithms to see what works. Various kinds of convolutional neural networks tend to be the best at recognizing the images in CIFAR-10. CIFAR-10 is a labeled subset of the 80 million tiny images dataset. [41]

Here are the classes in the dataset, as well as 10 random images from each [42]:

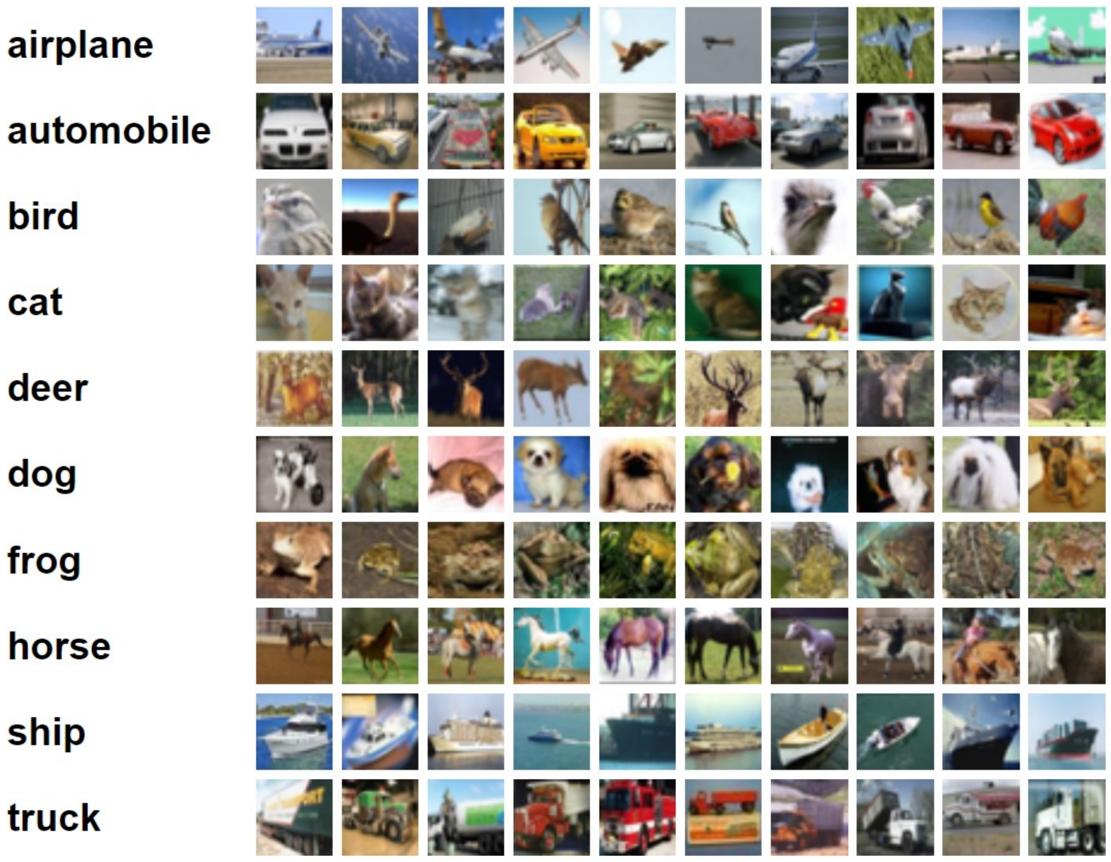


Figure 24. Classes in the CIFAR-10 dataset and sample images.

2.11 Floating-point format (IEEE 754 - float64)

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units use the IEEE 754 standard.

IEEE 754-2008, published in August 2008, includes nearly all of the original IEEE 754-1985 standard, plus the IEEE 854-1987 Standard for Radix-Independent Floating-Point Arithmetic. The current version, IEEE 754-2019, was published in July 2019. [43] It is a minor revision of the previous version, incorporating mainly clarifications, defect fixes and new recommended operations.

Double-precision floating-point format (sometimes called FP64 or float64) the format usually occupying 64 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point.

Floating point is used to represent fractional values, or when a wider range is needed than is provided by fixed point (of the same bit width), even if at the cost of precision. Double precision may be chosen when the range or precision of single precision would be insufficient.

In the IEEE 754-2008 standard, the 64-bit base-2 format is officially referred to as binary64;

Double-precision binary floating-point is commonly used due to its wider range over single-precision floating point, in spite of its performance and bandwidth cost. It is commonly known simply as double. The IEEE 754 standard specifies a binary64 as having:

- Sign bit: 1 bit
- Exponent: 11 bits
- Significand precision: 53 bits (52 explicitly stored)

The sign bit determines the sign of the number (including when this number is zero, which is signed).

The exponent field is an 11-bit unsigned integer from 0 to 2047, in biased form: an exponent value of 1023 represents the actual zero. Exponents range from −1022 to +1023 because exponents of −1023 (all 0s) and +1024 (all 1s) are reserved for special numbers.

The 53-bit significand precision gives from 15 to 17 significant decimal digits precision (2−53 ≈ 1.11 × 10−16). If a decimal string with at most 15 significant digits is converted to IEEE 754 double-precision representation, and then converted back to a decimal string with the same number of digits, the final result should match the original string. If an IEEE 754 double-precision number is converted to a decimal string with at least 17 significant digits, and then converted back to double-precision representation, the final result must match the original number. [43]

The format is written with the significand having an implicit integer bit of value 1 (except for special data, see the exponent encoding below). With the 52 bits of the fraction (F) significand appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, 53 log10(2) ≈ 15.955). The bits are laid out as follows:

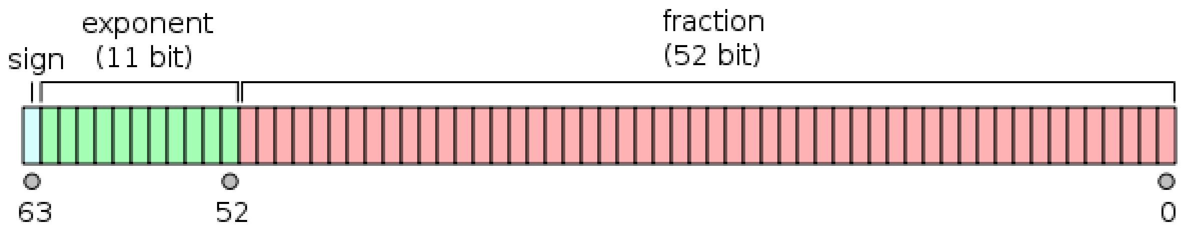


Figure 25. 64 bit Floating point format (IEEE 754)

3 Experimental results and analysis

The aim of this project is to find the right parameters and strategies to train a neural network to increase the reliability in the presence of faults. As mentioned before we are going to implement the neural network Resnet18, this because ResNets have shown great performance in image recognition tasks [44], and because it is one of the smallest ResNets, small enough to perform many experiments with it.

For the fault injections we are implementing PytorchFI, a fault injection library developed by researchers at the University of Illinois and NVIDIA released in the year 2020, which works over the Pytorch framework to perform fault injections on DNNs. For this project we will implement weight injections because they are common type of fault in real environments and the library overhead time of implementation for this kind of fault shows to be low in practice. Initially the weight injections of the library were not working properly for the model Resnet18 and its layers, so we had to modify the code and create a custom version of the library to make it work for our purposes.

The number of weight injections we are going to use corresponds to a percentage of the total weights the Resnet18 has. The Resnet18 has a total of 11 million parameters, so we will inject a percentage of 0.1% and 1%. The weights where the faults are going to be injected are selected randomly across all the layers of the Resnet18 and the original value of the weight is replaced by zero. We shall also note that for each injection experiment we make from 5 to 10 trials and set as the final value the average of the outputs, making the final outcome more reliable.

First, we have to train our neural network with CIFAR 10, and for the training of our model we need an optimizer, the Pytorch framework offers many optimizers to select from, some of the most common are SGD and Adam optimizer, so we are going to implement them among other optimizers. The optimizers will be implemented with recommended or default parameters, and varying one parameter at a time to see if there is a relationship between its value and the reliability. Second, we need a cost function to be used as a criterion of improvement for our optimizer, for this purpose we will implement the cross entropy loss function. Third, we need a scheduler, the scheduler schedules the change of the learning rate used by the optimizer, again Pytorch offers many schedulers to select from.

3.1 Selecting a scheduler

A common scheduler is the cosine annealing schedule [45], this scheduler uses a cosine function to modify the learning rate value, first decreasing it and then increasing it again following a cosine wave form. In the next image we can see the results of training the model with cosine annealing scheduler with a T_max of 200 and the SGD optimizer:

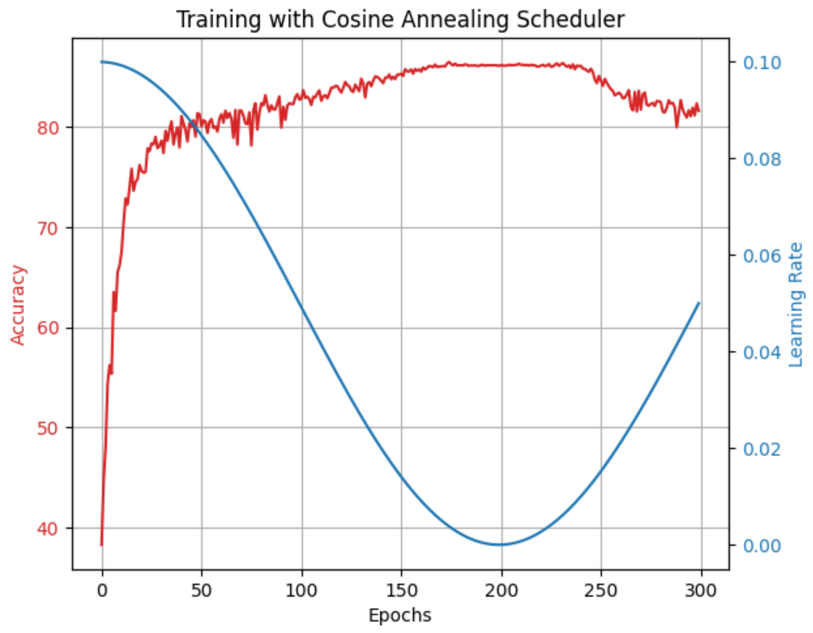


Figure 26. Training Resnet18 with cosine annealing scheduler.

We can see from the training how the learning rate varies in a cosine form, and this learning rate is inversely related to the accuracy, when the learning rate decreases the accuracy improves, and when the learning rate increases the accuracy decrease. The maximum test accuracy reached in this training is 84.32%. The learning rate starts at 0.1 and its minimum value is zero.

Now we will use the ReduceOnPlateu scheduler, which reduces the learning rate by a factor when the loss doesn't improve. We can modify its parameters to change when we want the learning rate to be updated, this is, changing the threshold when loss does not improve, other parameters are the minimum value of the learning rate and the patience, which is how many epochs without learning rate improvement shall pass to update its value. The selected parameters are shown below:

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,patience=3,threshold=1e-3)
```

The following is a graph with that scheduler configuration and the SGD optimizer:

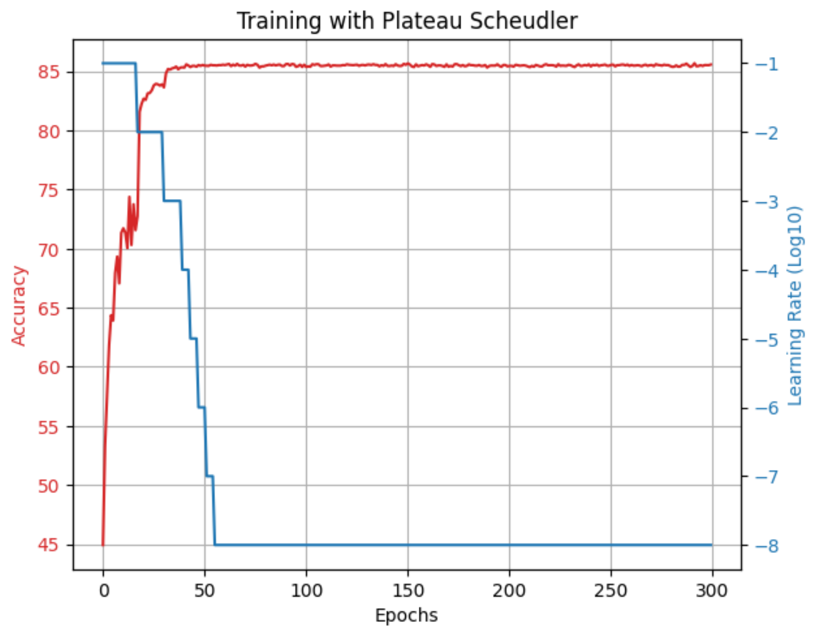


Figure 27. Training ResNet18 with reduce on plateau scheduler.

We can see again that the learning rate is inversely related to the accuracy, so when the learning rate starts decreasing the accuracy increases, this is because the optimizer takes smaller steps and thus more detailed and accurate steps towards the loss function minima. The maximum accuracy reached with this scheduler is 85.64.

From the figures above we can see that the accuracy in the plateau scheduler converges faster to its maximum value, this is in less epochs. This indicates that this scheduler trains the model faster as the learning rate changes whenever the loss value doesn't improve, and also we get a higher maximum accuracy for the plateau scheduler. For these reasons we are going to continue training our models and injecting faults with this scheduler.

3.2 SGD optimizer

First we will train our models with the SGD optimizer which has shown to give good results on the CIFAR-10 dataset [23], we will initially use it with its recommended parameters [45], thus the SGD optimizer definition in python and its parameters are as follows:

optimizer = optim.SGD(net.parameters(), lr=0.1,momentum=0.9, weight_decay=5e-4)

We will vary some training parameters and see how these changes affect the reliability of the model, we will search parameter values that at some point increases the reliability, this is, it is less affected by fault injections.

3.2.1 SGD -- Varying the weight decay

The Pytorch SGD optimizer has weight decay as one of its parameters, weight decay is a regularization term that is used to let the neural network learn from the dataset in a more generalized form by reducing overfitting (see section: "Regularization and weight decay"). We will vary this parameter in the training of the model and search for the value that would provide the best reliability results.

For this purpose, we will train the Resnet18 with the following weight decay (wd) values: 0,5e-7,5e-5,5e-3,5e-1. The batch size will remain 128, the accuracy shown in the following figures is the test accuracy.

For wd=0 we have no regularization, so the model may be prone to overfitting, after training the model with this value we obtain the following figure:

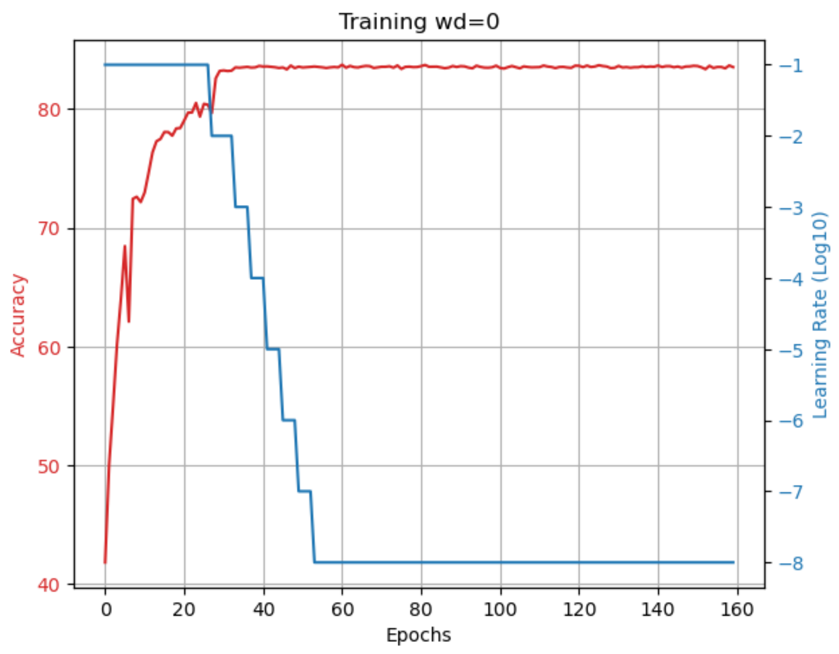


Figure 28. Training with SGD optimizer, WD = 0

The maximum test accuracy obtained from the training is: 83.74%. We can also see that the model reaches this maximum around epoch 30, training above this epoch doesn't improve the network accuracy.

Next, we train our model with a small value of weight decay, the value of 10e-7, the results of the training are shown in the following graph:

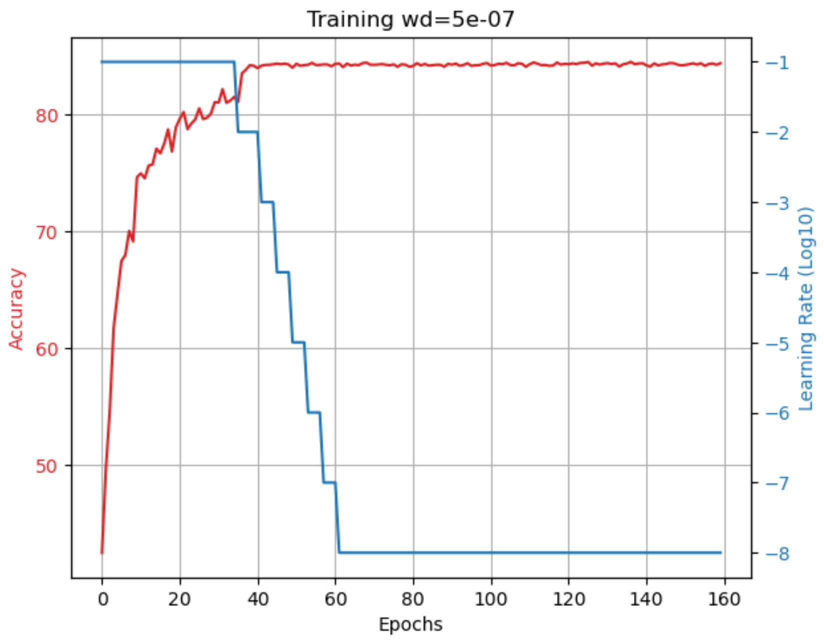


Figure 29. Training with SGD optimizer, WD = 5e-7

This time we obtain a higher maximum test accuracy, with the value of 84.52%, showing that the regularization does its work and improve the generalization of the model. Also we can observe from the figure that the maximum value of test accuracy converges later than if we don't use any regularization, this time around the epoch number 40.

Then we train the model with a weight decay of 5e-5, the results are the following:

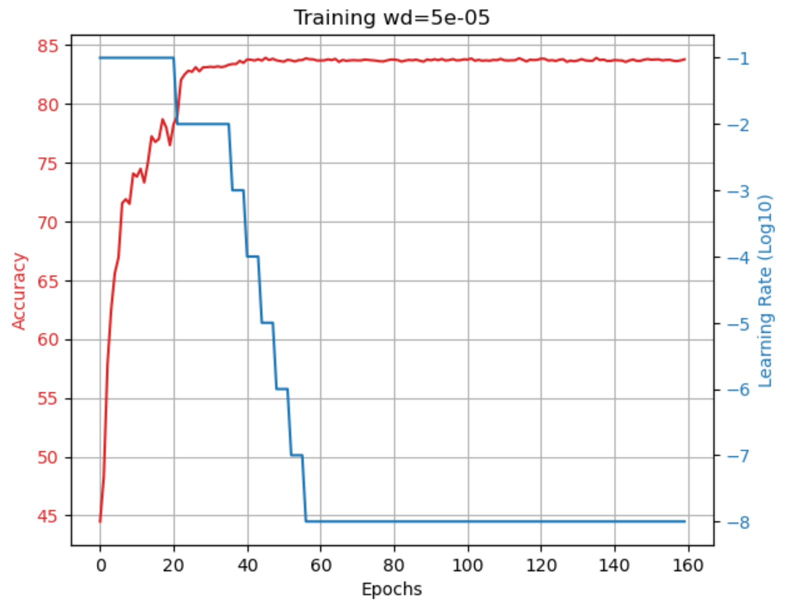


Figure 30. Training with SGD optimizer, WD = 5e-5

The maximum test accuracy reached by the weight decay for 5e-5 is 83.88%. Which is lower than the previous weight decay value of 5e-7, and again this value is reached around epoch 40.

The next value is 5e-3, the results are shown below:

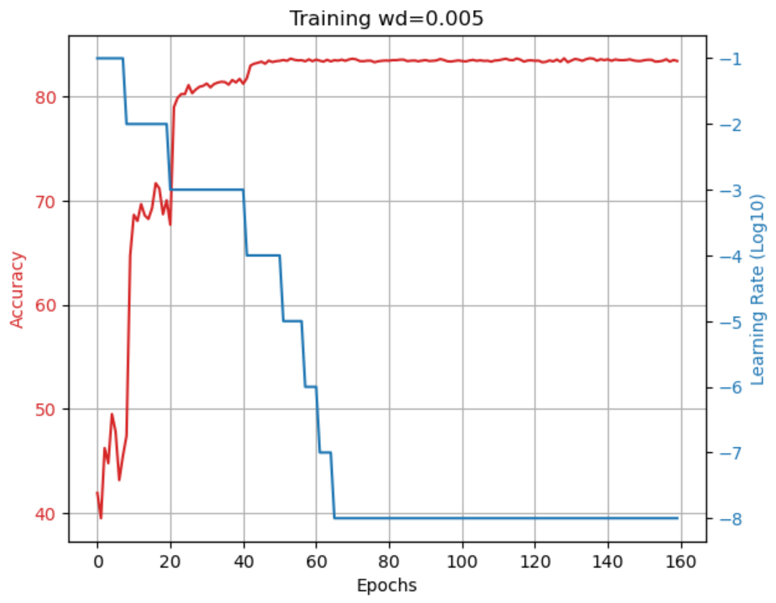


Figure 31. Training with SGD optimizer, WD = 5e-3

We find a similar behavior to the precedent trained models with regularization, however this time the maximum test accuracy is lower than the model trained without regularization, with wd=5e-3 we find a maximum test accuracy of 83.64% and the convergence to this value reached around the 40th epoch, compared with the no regularized model accuracy of 83.74% converging at epoch 30.

Then we used a bigger value of weight decay with a value of 5e-1, but this time we found some interesting results:

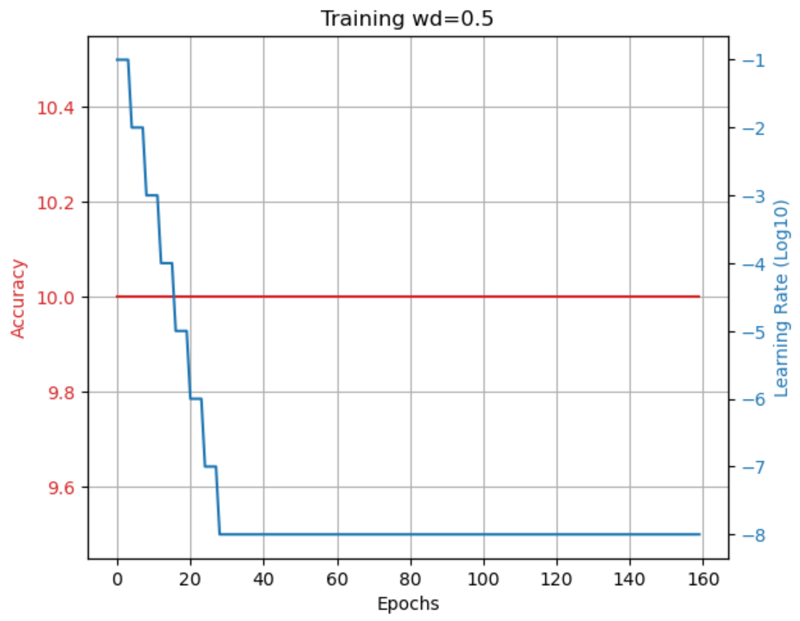


Figure 32. Training with SGD optimizer, WD = 5e-1

This time, with a big regularization value, the model is not capable of training and the test accuracy does not increase, it has a constant accuracy value of 10, which means that the models inferences are selecting the output randomly, as we have 10 classes the random output accuracy is 10%. The big value of weight decay affects the training enough to not let the model classify correctly. For this reason we don't keep increasing the weight decay.

Now we are interested in the reliability at the moment of varying the weight decay, to see if some of these values improve its resilience to faults. Therefore, we take each of the models trained with decay values (except 5e-1 as it was not able to train) and inject a number of random weight faults among the layers. First, a total of 10k injections with a step of 1k:

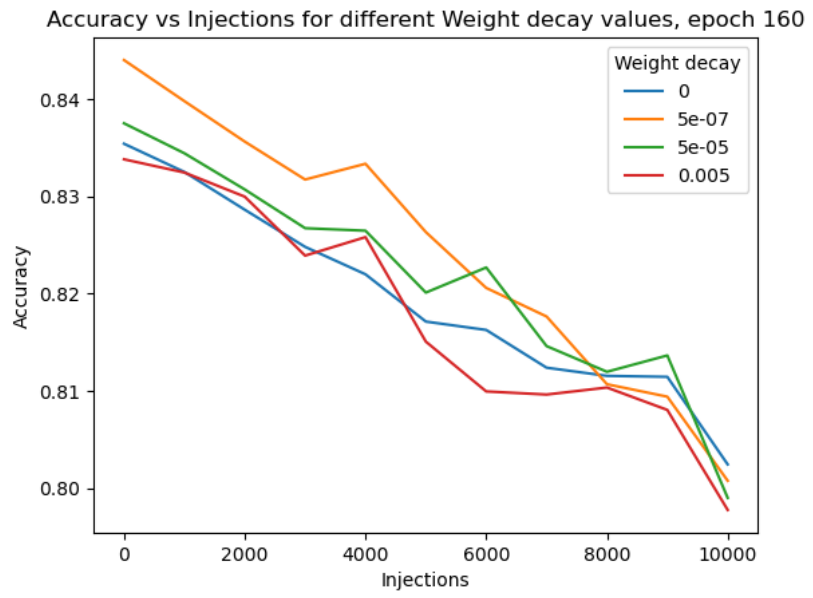


Figure 33. SGD test accuracy vs 10k injections for different weight decay values

We can observe that, as shown from the training, a high value of WD result in a lower accuracy, it looks like a high WD also leads to a lower reliability, however this may seem to happen in the graph because the models start at different accuracies. So, we create a graph where all the models start at the same accuracy by subtracting the mean value:

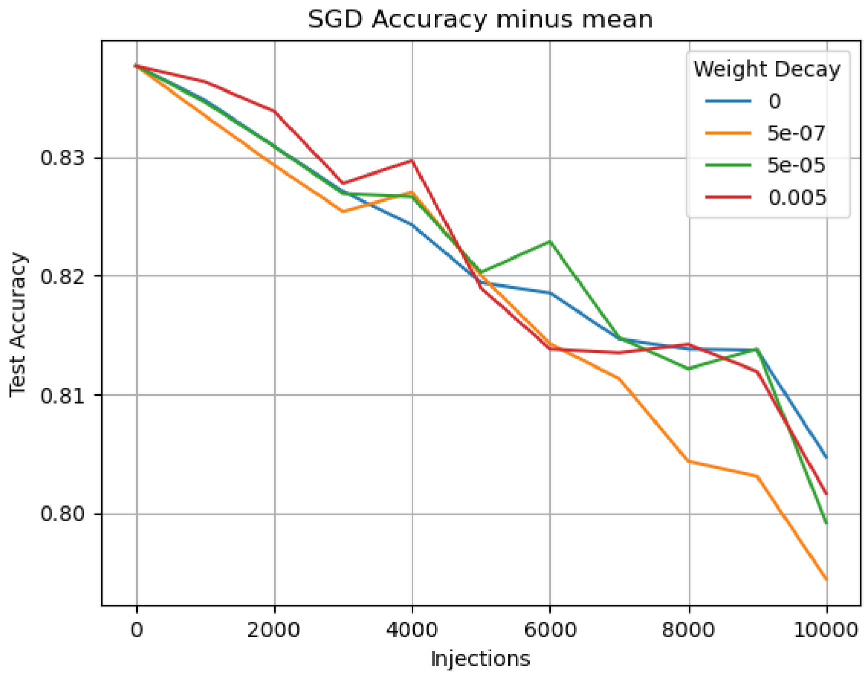


Figure 34. SGD test accuracy vs 10k injections for different weight decay values, minus mean

We can observe from the graph that starting from the same initial accuracy, there is not a clear winner in terms of reliability.

Now, we continue doing experiments for a greater range of injection numbers, a total of 100k injections with a step of 10k:

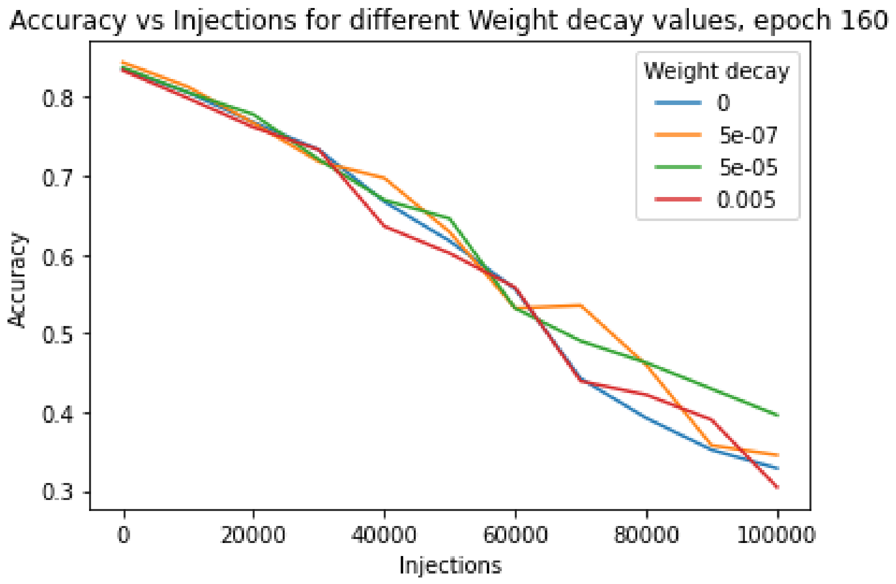


Figure 35. SGD test accuracy vs 100k injections for different weight decay values

We can see from the graph that again there is not a clear model that has a better performance in terms of reliability, all the models behave similarly at the first 20k injections, and then their accuracies start reducing with the same trend.

The previous experiments in WD where done in epoch 160, as we did for the batch size, we also inspect the reliability at epoch 80 to see if there are any changes:

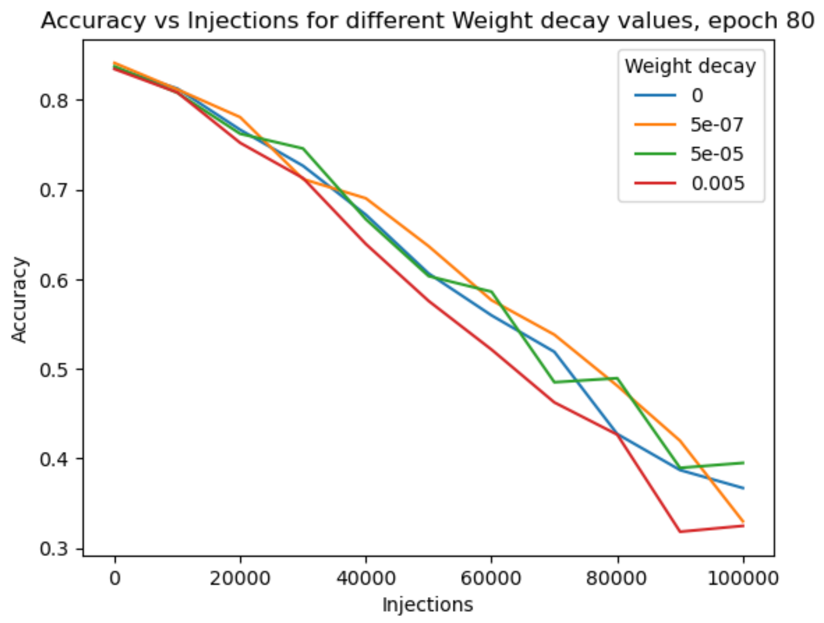


Figure 36. SGD test accuracy vs 100k injections for different weight decay values, epoch 80

We can observe that at epoch 80 the reliability of WD=5e-3 is lower than the other WD values for the most part of the graph, this subtle difference may let us recommend a high value of weight decay to increase the reliability of the model trained with SGD and after 80 epochs.

3.2.2 Bit fault injections

Another type of common faults is the bit-flip, this kind of fault characterizes by changing the value of one bit in memory, flipping it from 1 to 0 or from 0 to 1. We will study this kind of faults to see how a model is affected by this kind of fault.

First of all, we shall know that Pytorch tensors store their values in the format float 64 of the IEEE 754 (see IEEE 754 - float64 section in background), the bit 0 to 51 are used for the fraction or the precision of the floating point, the bits 52 to 62 are in charge of the exponent, and lastly the bit 63 is assigned to the floating point sign.

Our objective will be to add a bit-flip fault to many weights in the network, but adding all the faults into a specific bit, hence, we are going to add at first 1k bit-flip injections to random weights but all faults in the bit 0, then add 1k bit-flip faults to random weights in the bit 1 and so on up to bit 63. We want to see if there is a bit which is more sensible to this kind of fault and if it will change in a greater degree the accuracy of the model.

The results of injecting 1k faults in each bit are shown below:

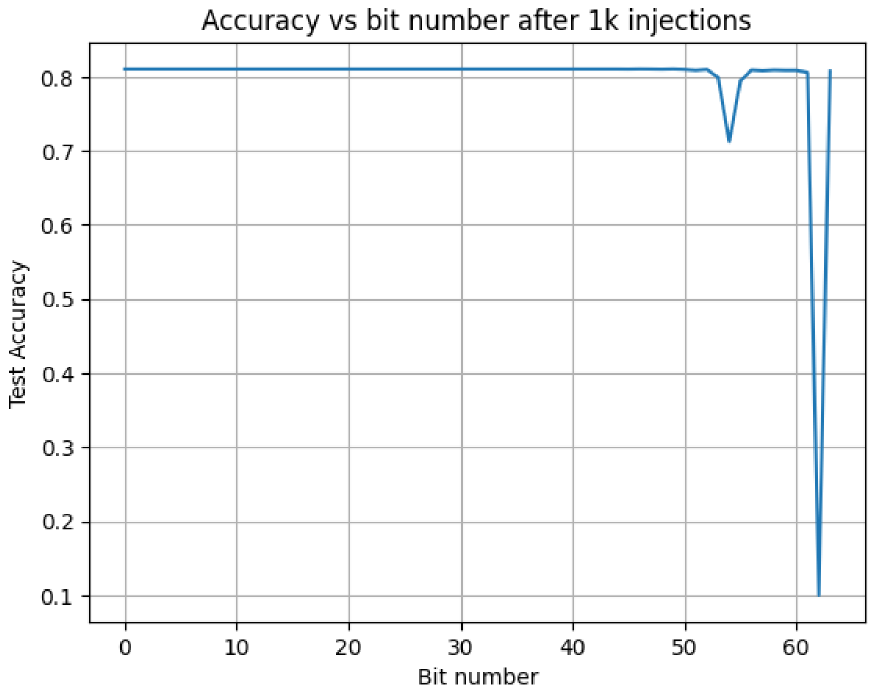


Figure 37. Test accuracy vs bit number after 1k injections

As we can observe there are some bits that are more significant for the reliability of the network, more in detail, the accuracy in the first 51 bits remain the same as the original model (0.8102), this is because these bits are used as the fraction value of the float and a changes in this value are not very significant to the network reliability, however after bit 51 the accuracy decreases to 0.8, this because changing the exponent affects more the numerical value of the floating point and hence lowering the accuracy of the hole network. And this changes are more significant in bits 54 and bit 62, there is a little depression around bit 54, which reaches a low test accuracy of 0.7124, and for bit 62 the accuracy decreases to 0.1, which means that this model's output is practically random.

Let's continue experimenting with bit-flip injection, this time increasing the number of injections to 10k on each bit, the results are the following:

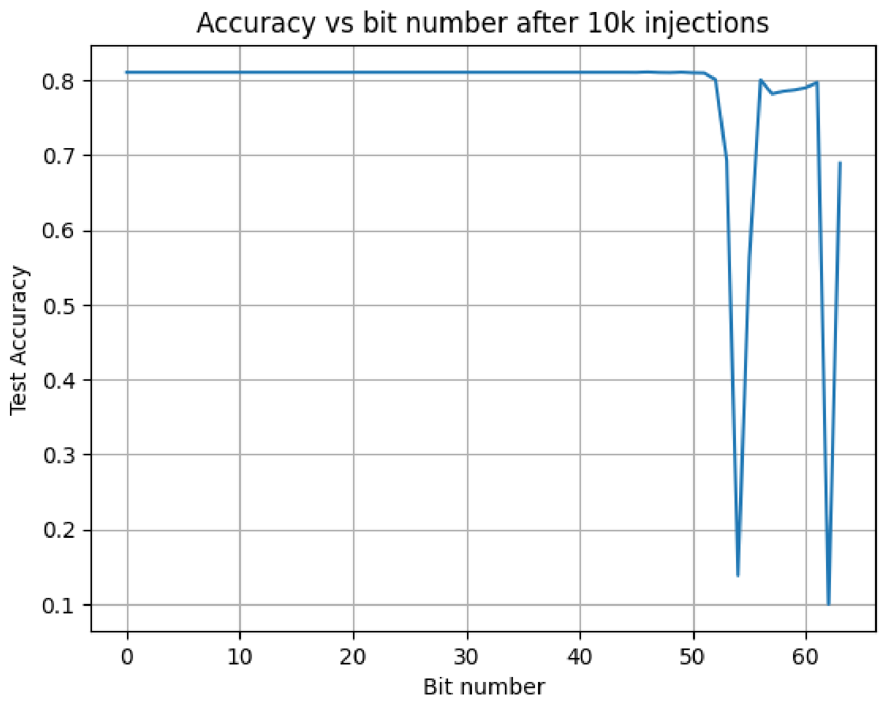


Figure 38. Accuracy vs bit number after 10k injections.

We can observe that the pattern of low test accuracy on high bit number (52 and above) continues, again this because these bits are related to the exponent of the floating point. In the last figure we can observe that the accuracy for the depression around bit 54 continues to decrease as we increase the number of faults, specifically the accuracy values for these bits are 0.6922, 0.1381, 0.5617 for bits 53,54,55. Another difference with respect to the previous graph is the accuracy of the bit 63, this time it decreases from the original value to a value of 0.68, showing that the bit related to the sign of the weight's floating point also plays an important role in the network reliability.

The accuracy for bit 62 keeps the same of previous graph of 0.1, behaving as a random output. So if we increase the faults the bit 62 accuracy continues to be 0.1, now let's see what happens if we reduce the number of injections to 1 injection on one weight on every bit.

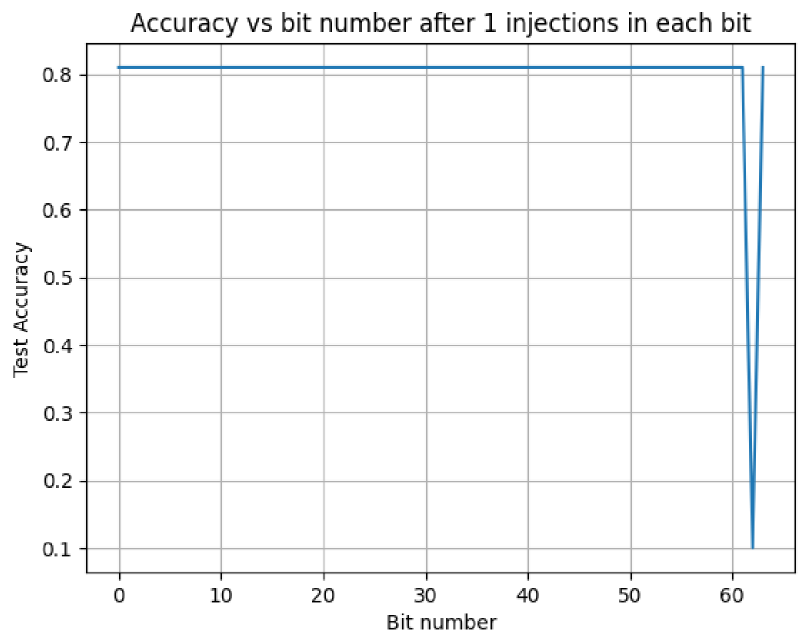


Figure 39. Accuracy vs bit number after 1 injection.

The accuracy remains the same in all the bits, except for bit 62 which is dropped again to 0.1. Only one injection in bit 62 in a random weight is enough to make the whole network behave as if it was untrained or giving random outputs. Reviewing the code to see what's happening in the operations with this bit-flip, we found that as this the most significant bit for the exponent, when it is bit-flipped the resulting value is taken by a Pytorch tensor as infinite, making all the following operations to be infinite and resulting in a random output. If this conversion from a numpy float to a Pytorch tensor value is not done in the code, a fault injection in this bit results in an overflow error and leads the program to crash. From this experiments we remark the importance on good reliability practices and methods for the bit 62, and also bit 54.

3.3 Adam optimizer

The Adam optimizer is different from SGD (see section "Optimizers"). A learning rate is maintained for each network weight (parameter) and separately adapted as learning happens. Adam realizes the benefits of both Adagrad and Rmsprop.

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in Rmsprop, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Therefore, Adam stands for Adaptive moments.

To test the reliability of the Resnet18 with the Adam optimizer, we will start training this model with different parameters, starting from the recommended parameters by Pytorch:

optimizer = optim.Adam(net.parameters(), lr=0.1, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)

3.3.1 Adam – Varying the batch size

We will train different models with different values of batch size. The values we will use are 64, 128, 256, 512 and 1024, using the Adam optimizer, we will train a total of 160 epochs expecting to reach the maximum accuracy value in this range of epochs.

First, the training for BS of 64:

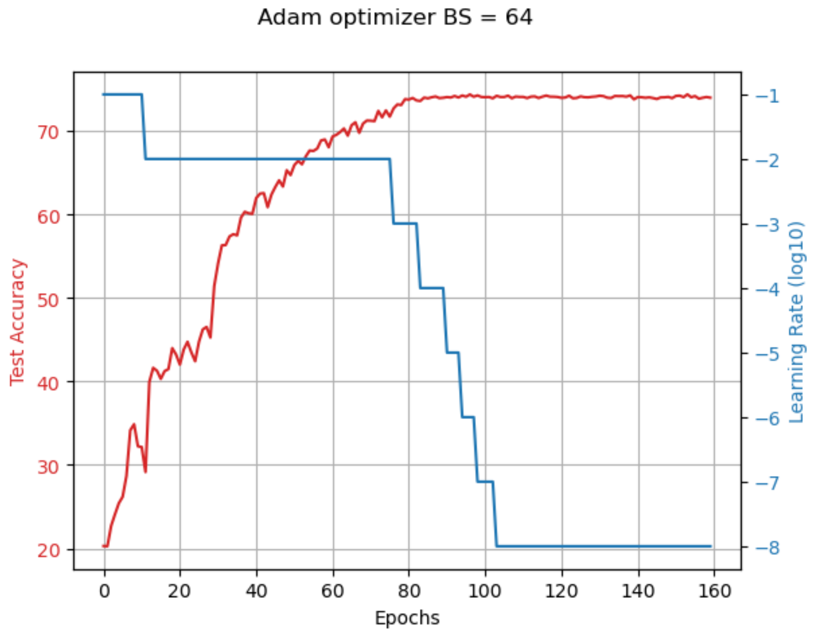


Figure 40. Training with Adam optimizer, batch size = 64

The maximum accuracy reached by this model is 74.37%, not a very good accuracy. We can see it reaches this maximum around epoch 85.

Next, the training for BS=128:

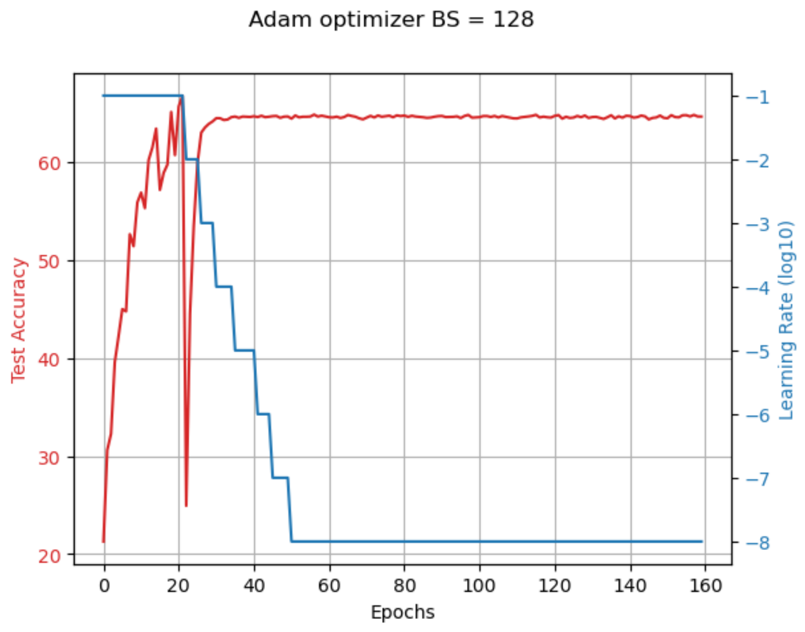


Figure 41. Training with Adam optimizer, batch size = 128

The maximum accuracy reached is 66.79%, it is a low accuracy, we can see that the model training present difficulties at epoch 20, it is able to recover but the accuracy does not improve further.

Next, training the model for a BS of 256:

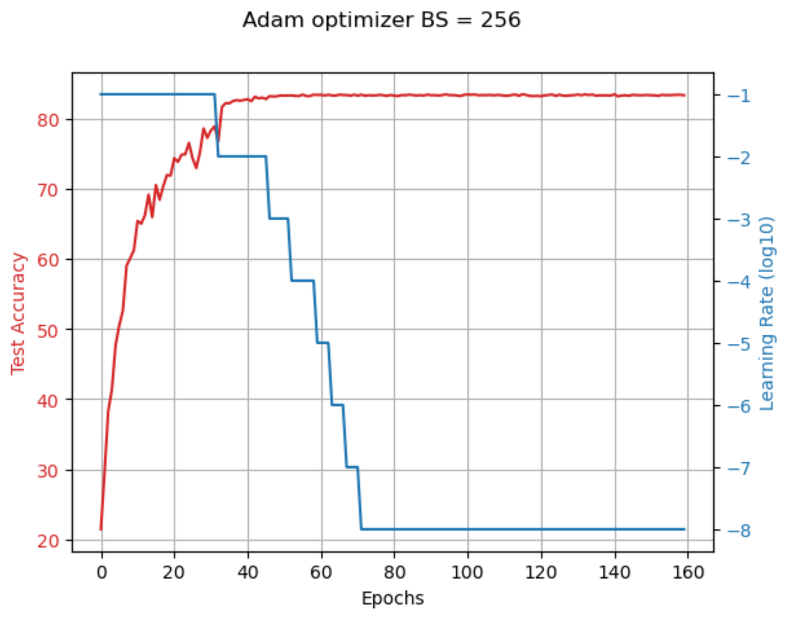


Figure 42.. Training with Adam optimizer, batch size = 256

Here the maximum accuracy is 83.47%, it is a good accuracy, similar to the ones reached by the best of the SGD models, also this model is trained very fast, it reaches it maximum around epoch 40. Let's continue to see how the BS affects the model training, for BS =512:

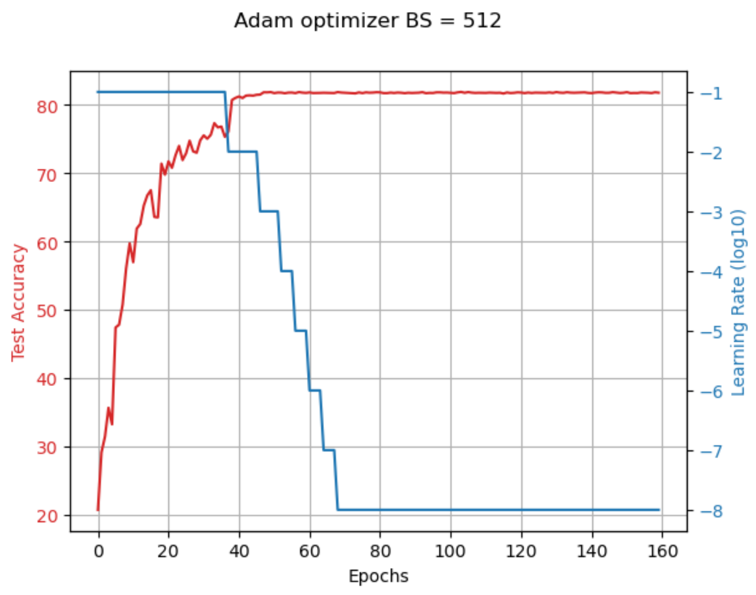


Figure 43. Training with Adam optimizer, batch size = 512

The maximum accuracy reached by this model is 81.89%, a little lower than the BS of 256, but the model performs good in general.

For BS = 1024:

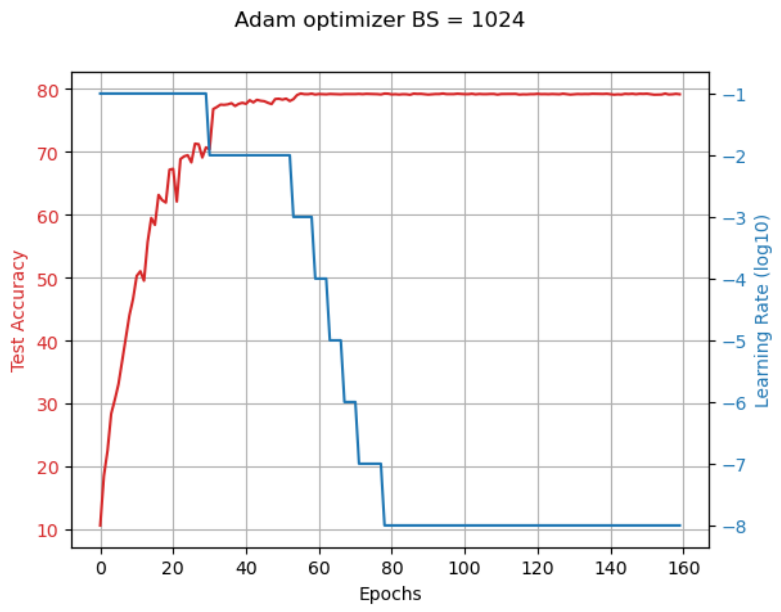


Figure 44. Training with Adam optimizer, batch size = 1024

The maximum accuracy reached is 79.26%. This accuracy is not as good as the ones of BS 256 and 512.

We can conclude from the training that only BS 128 has issues in the training and the best accuracy reached is 83.47% by BS 256. The previous analysis of the training is important to know if the models reach its maximum test accuracy within the selected number of epochs. Now let's compare the reliability of this models and see if the BS has an effect on it.

First, we will inject 10k weight faults in random weights among all the layers, the same way we did in SGD section. The results are show below:

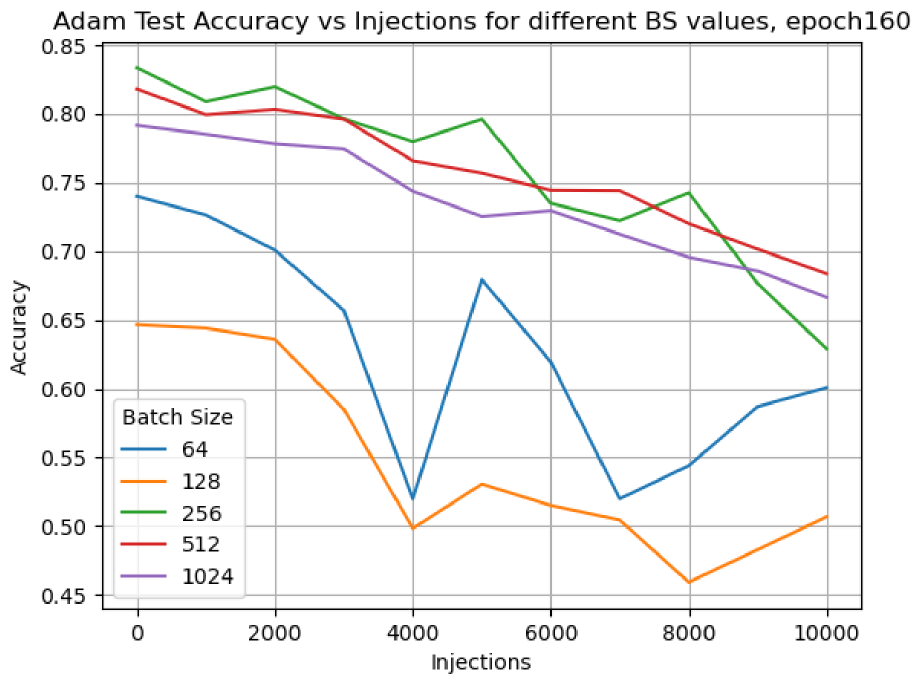


Figure 45. Adam Test accuracy vs 10k injections for different batch size values, epoch 160

We can observe the differences on accuracy, more than anything in the batch sizes of 64 and 128 which are lower. We can also observe that at 4k injections the accuracy is even lower on those BS, the same occurs on 8k injections. Now we will subtract the mean value of all the accuracies to remove the offset and see more in detail the comparison of reliability:

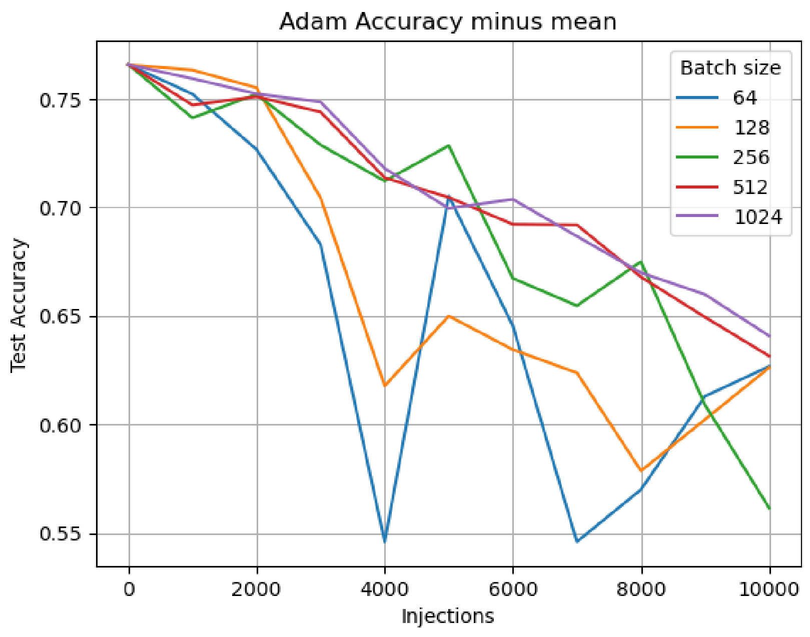


Figure 46. Adam Test accuracy vs 10k injections for different batch size values, epoch 160, minus mean

We can see that by equaling the initial accuracy, the models that perform the better in terms of reliability are those with BS higher than 256, so for this optimizer a high value of batch size is recommended as they seem to be more robust and stable in presence of faults.

Now we do the same experiment for 100k injections:

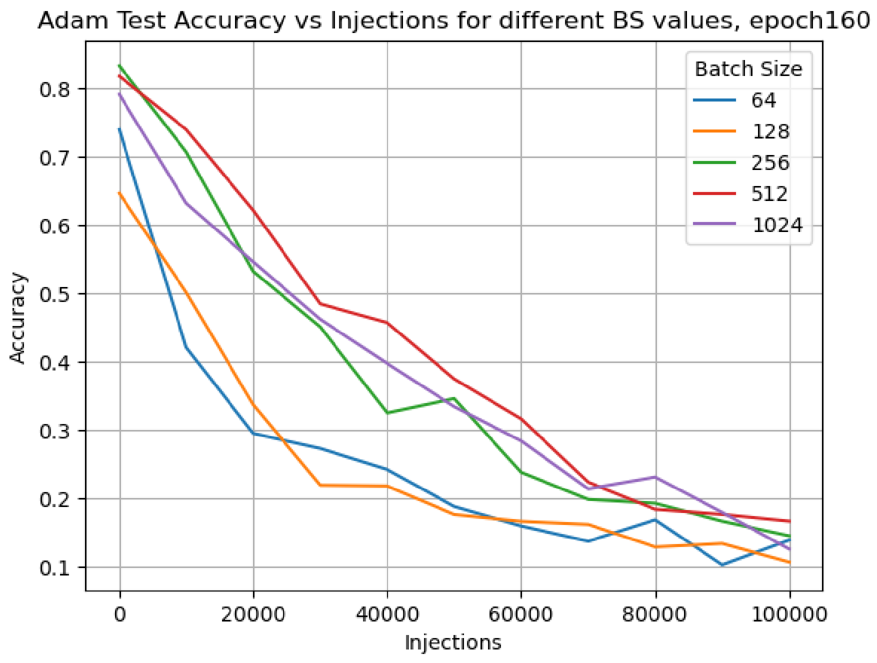


Figure 47. Adam Test accuracy vs 100k injections for different batch size values, epoch 160

As we can see we have similar results that the 10k injections, again BS equal or higher than 256 have a better reliability. Lets see the models after subtracting the mean to see a better comparison:

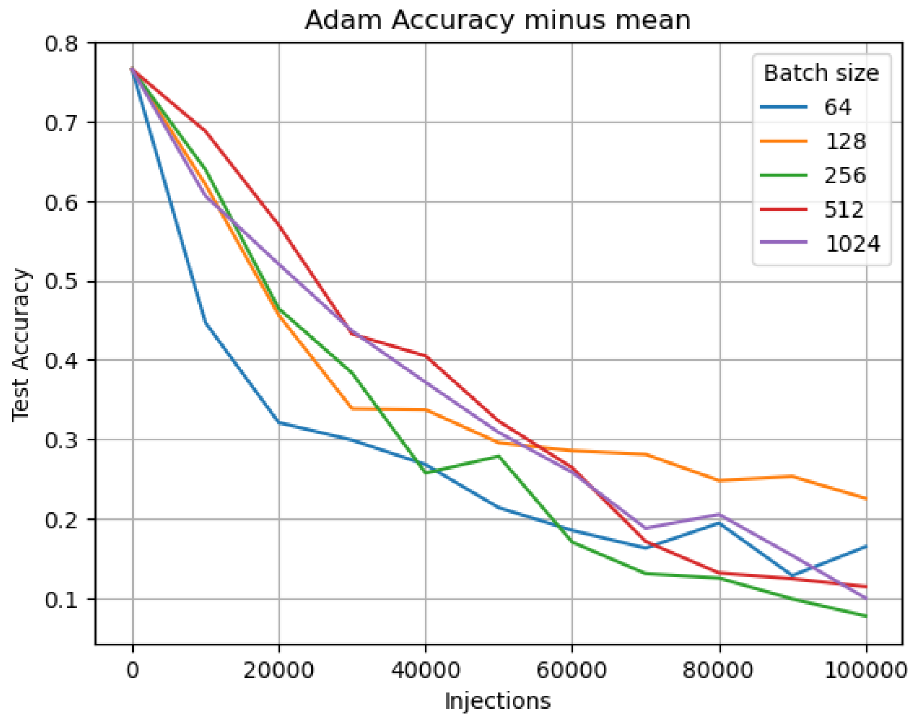


Figure 48. Adam Test accuracy vs 100k injections for different batch size values, epoch 160, minus mean

We observe that the BS of 64 has the lowest reliability, but BS of 128 increases it’s reliability to the point of comparing with bigger batch sizes, so the only batch size that seem may have an impact on the reliability of the network for this optimizer is 64.

However when subtracting the mean we have to cautious because after many injections the accuracy of all models will converge to 0.1, so this method to spot the models with best reliability by equaling the accuracies only works when we inject a small number of faults.

3.3.2 SGD -- Varying the batch size

One of the parameters we can vary or modify in the training of the model is the batch size. In particular, the training batch size is the number of elements we will use to update the weights of our model with the optimizer. The batch size is used for many reasons, one of them because minibatches use less memory, since you train the network using fewer samples. Another one is the training speed, typically networks train faster with minibatches, that's because we update the weights after each propagation. And finally, because large minibatch sizes lead to a worse accuracy [46].

We will select the following number of batches to train our networks: 64,128,256,512 and 1024.

The first value is BS=64, the training results are shown below:

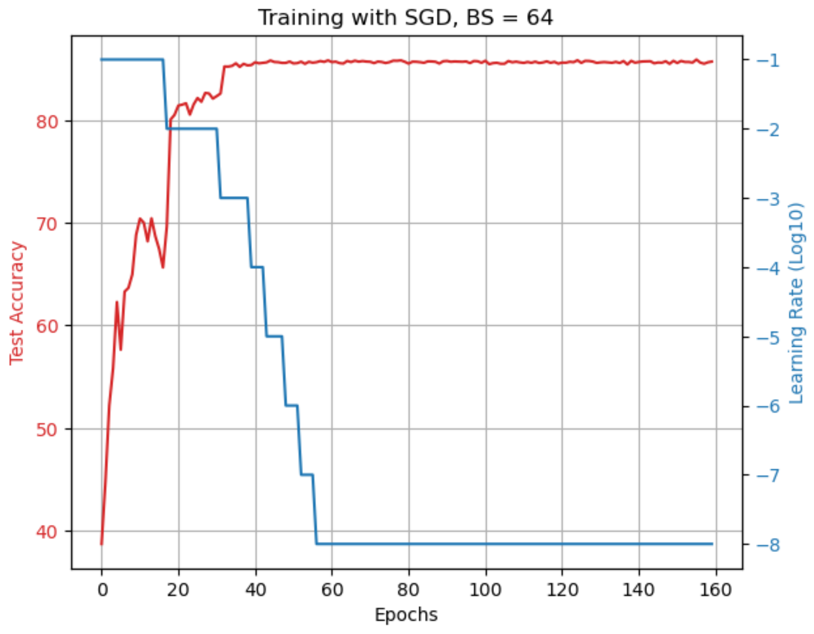


Figure 49. Training with SGD optimizer, batch size = 64

The maximum accuracy reached for this model is: 85.92%, and we can see that this maximum is reached around epoch 40, we can also observe that there are steps of the learning rate that significantly improve the accuracy of the model, showing the importance of having a good scheduler, the purpose of this scheduler is to have a big learning rate at the beginning so the model updates the weights faster and later reduce the learning rate value to give smaller but more accurate steps towards the loss function minima. Also, as we can see from the graph, after some steps of the scheduler the accuracy does not improve any more, meaning we reached a maximum accuracy for this model.

The following graph is for BS = 128:

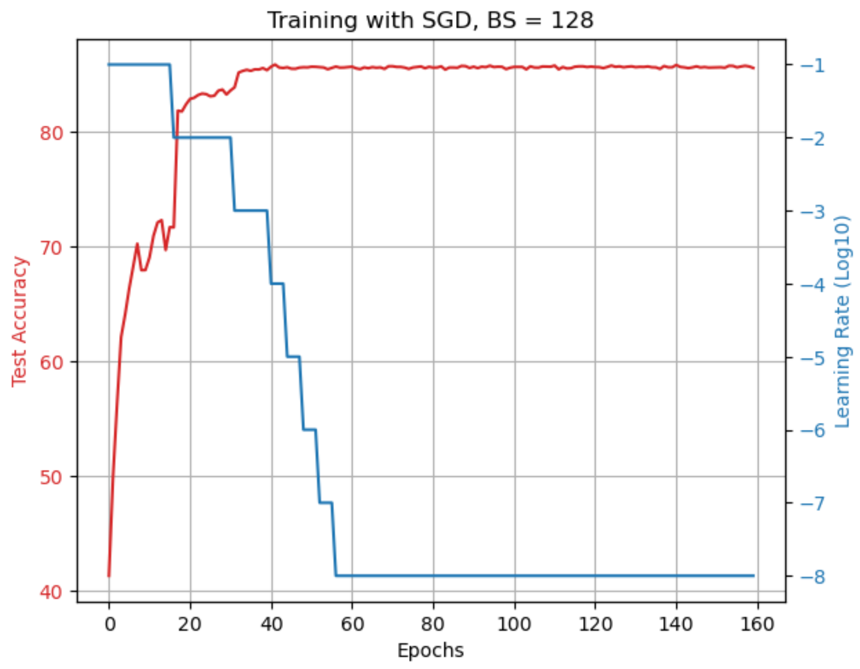


Figure 50. Training with SGD optimizer, batch size = 128

We reach a maximum test accuracy of 85.82%, a little below from BS=64. However the training behavior of the model is similar to the previous one.

Next for BS=256:

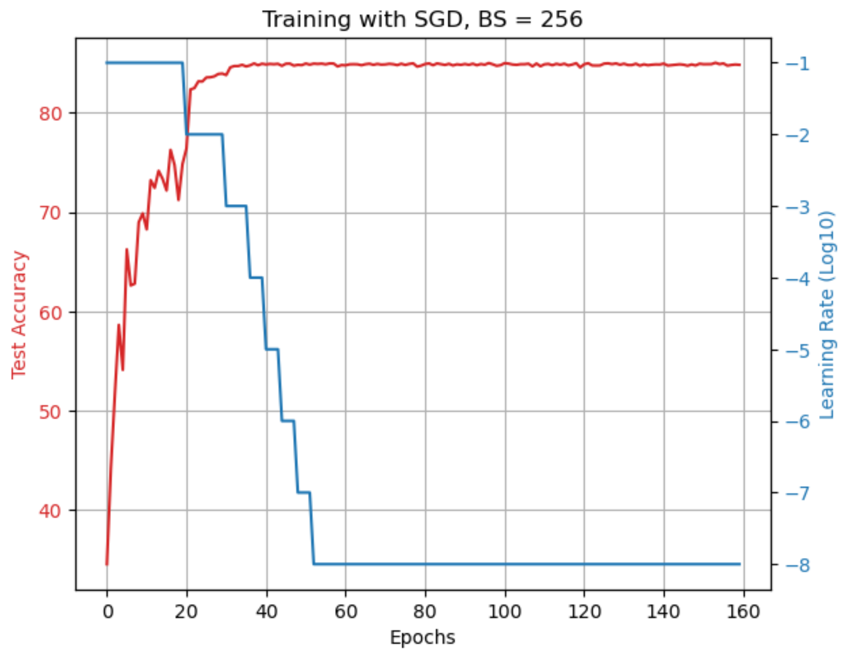


Figure 51. Training with SGD optimizer, batch size = 256

The maximum accuracy reached is 85.03%, a value below the previous ones, this maximum is reached around epoch 40.

Next for BS = 512:

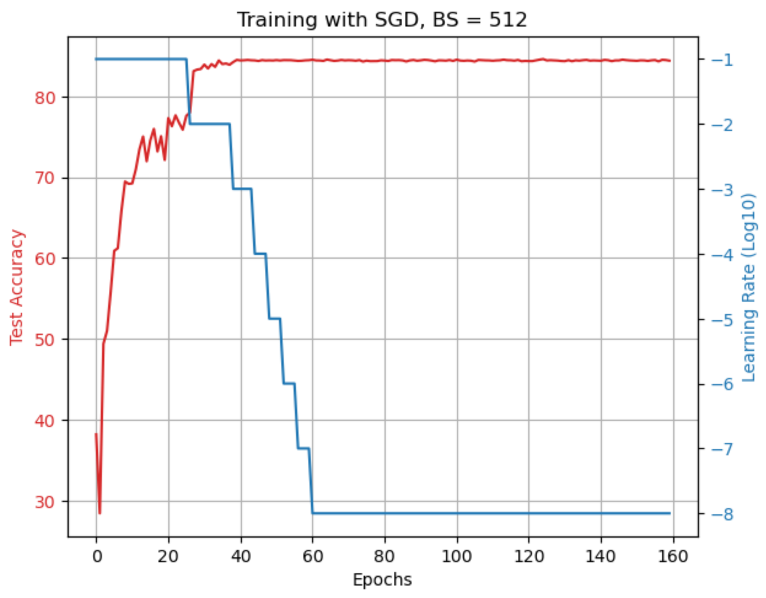


Figure 52. Training with SGD optimizer, batch size = 512

The maximum accuracy reached is 84.61%, it is reached around epoch 40.

Finally let’s see the training results for BS=1024:

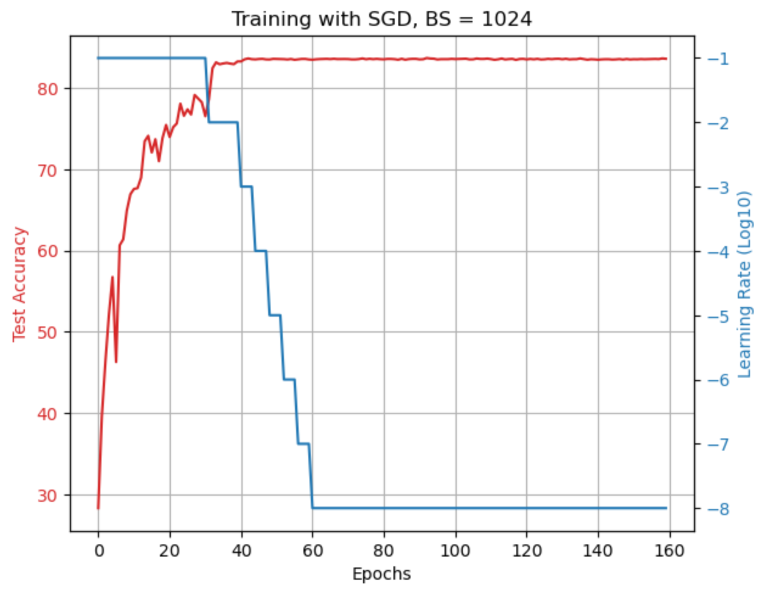


Figure 53.Training with SGD optimizer, batch size = 1024

The maximum accuracy reached is 83.72%, this time around epoch 45.

We can conclude from the graphs that the training is similar for all the batch sizes, however the accuracy is reduced while the batch size is increased.

Now that we have seen that all the models are correctly trained and reached their maximum accuracy, we proceed to do the weight fault injections. The results of the 0.1% weight injections, i.e., 10.000 injections are shown below with a step of 1000:

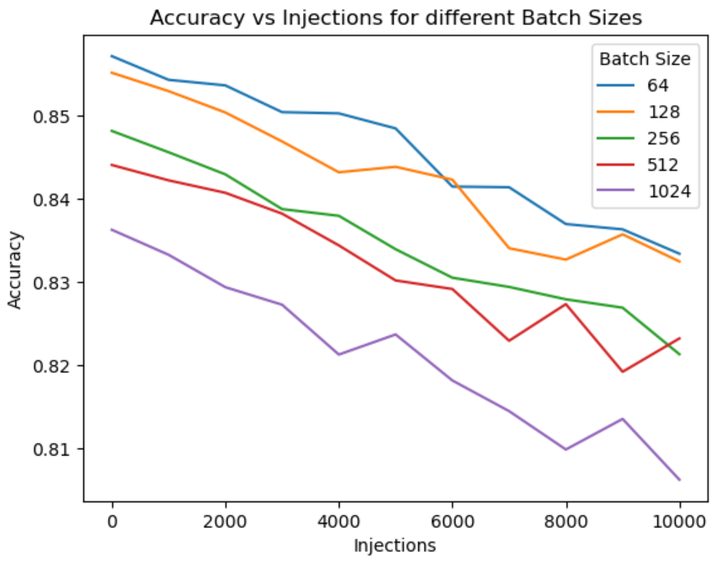


Figure 54. Accuracy vs 10k fault injections for SGD and different batch sizes.

We can see from the figure that there are some differences in the initial accuracies, we will remove this offset by subtracting the mean value on all the graphs, the results are shown below:

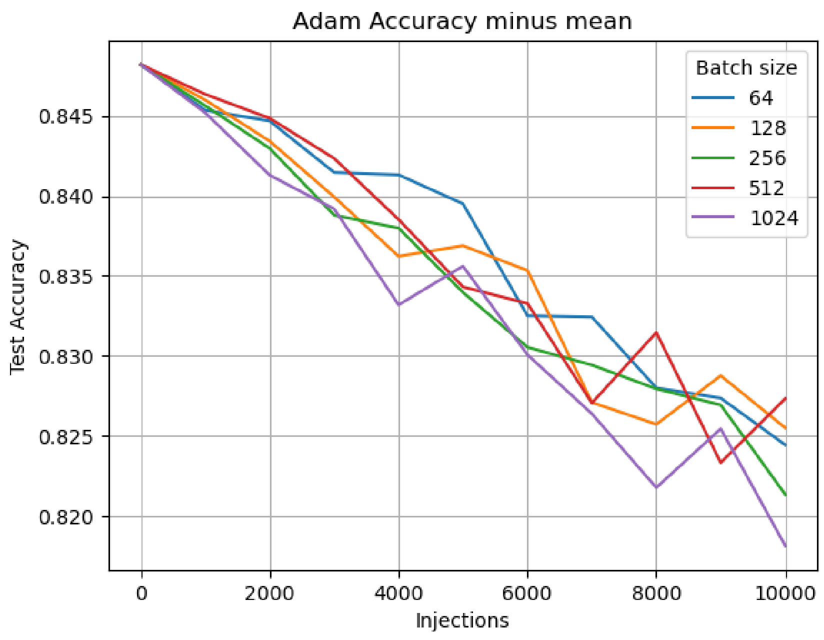


Figure 55. Accuracy vs 10k fault injections for SGD and different batch sizes, minus mean

As we can see from the figure, the best performance, this is, the model trained with the batch size that is less affected by the fault injections, is the model trained with batch size of 64, and the model which is most affected by the injections is the model trained with BS = 1024. However this difference is subtle, so we increase the number of fault injections to see how the models are affected in a wider range of injections. For 1% of total injections, i.e. 100.000 injections, the results are shown below:

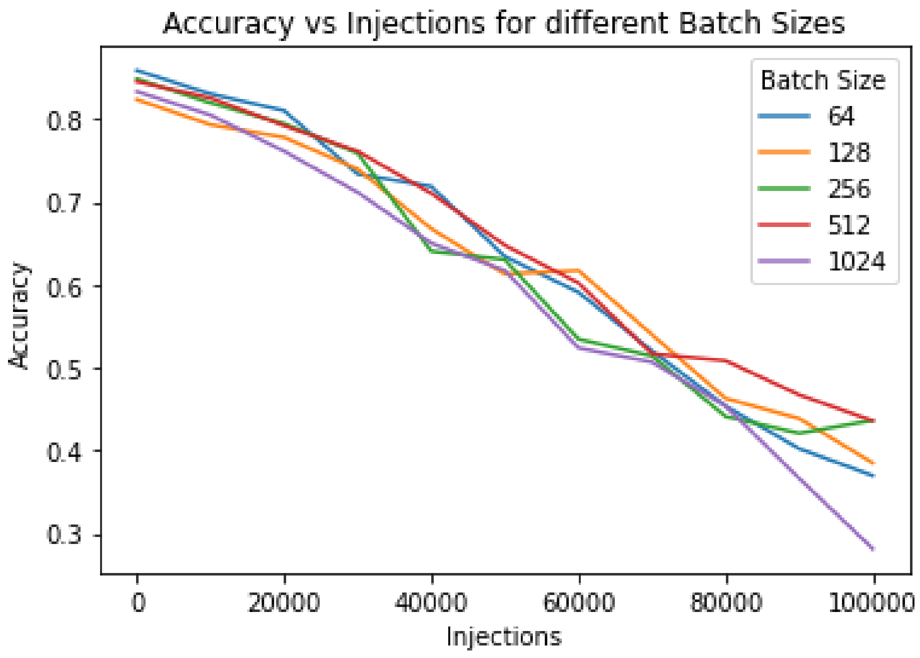


Figure 56. Accuracy vs 100k fault injections for SGD and different batch sizes.

The first 20k injections behave similarly or equally to the graph of 0.1% injections, after this, we can see that the model trained with batch size of 1024 behaves a little worse compared with the others in terms of reliability.

The previous experiments where all done with models where the accuracy was the best after 160 epochs of training, this is, from the 160 epochs we selected the models which had best accuracy. This lead to the selection of models of different epochs. But we also wanted to examine if the reliability was affected as the epochs increase. For that purpose we first selected the epoch 80, because as we saw from the training figures that all the models have already reached their maximum accuracy around this epoch. The results for epoch 80 are shown below:

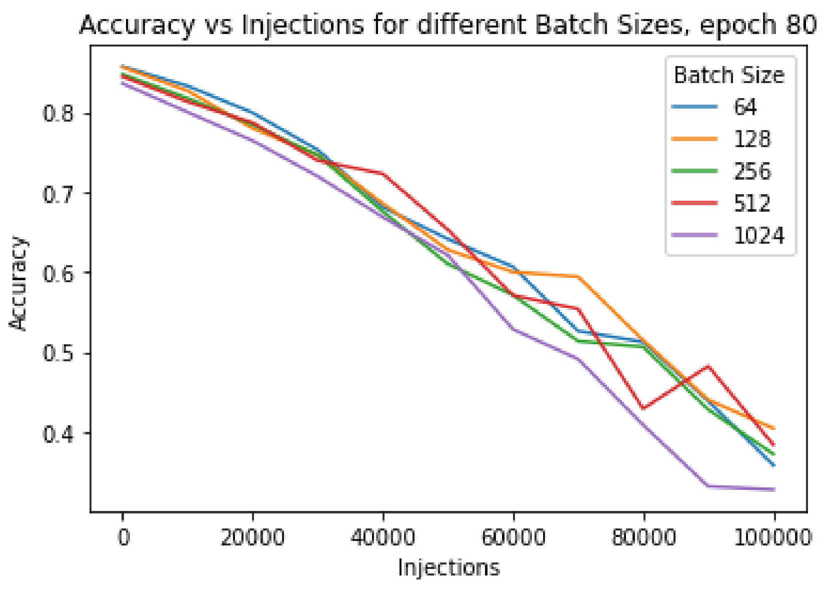


Figure 57. Accuracy vs 100k fault injections for SGD and different batch sizes on epoch 80

Here we can see again that the model with the batch size that performs the worst or that is most affected by the fault injections is the one with batch size 1024. We will repeat the same experiment of injections but now for epoch 160 to see if with the pass of the epochs the reliability changes:

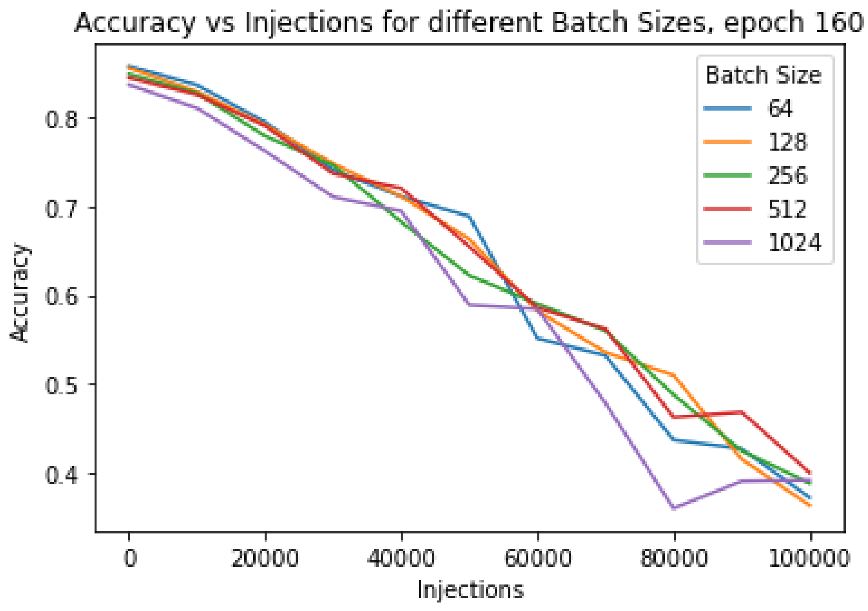


Figure 58. Accuracy vs 100k fault injections for SGD and different batch sizes on epoch 160

We can identify again that the model with batch size equal to 1024 is the one that behaves the worst. Now we would like to see if there was any difference between the reliability in these two selected epochs, to do so, we subtract the accuracy value from both epochs to obtain their difference, obtaining the following graph:

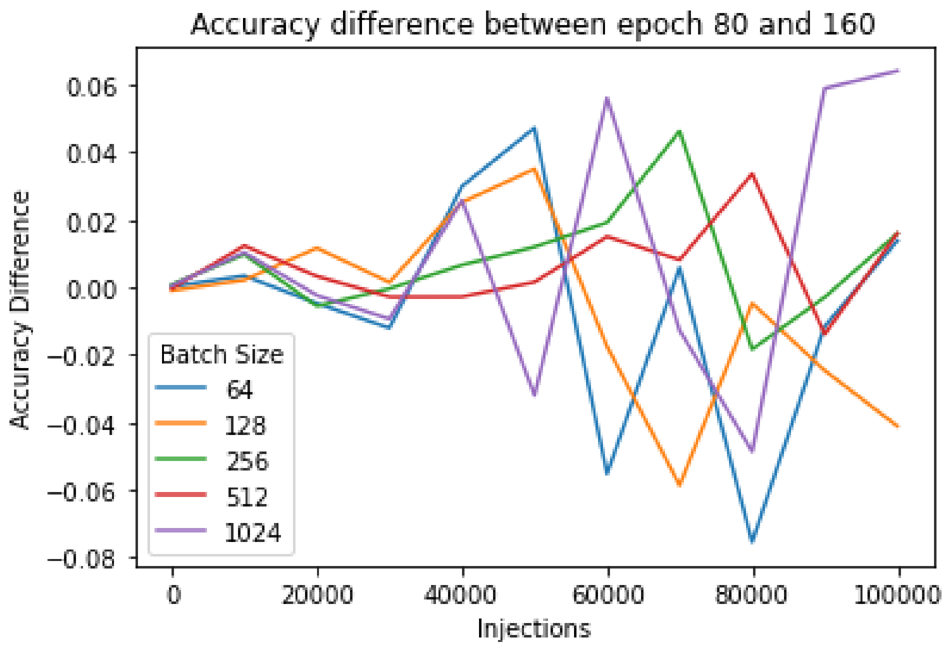


Figure 59. Accuracy difference between epoch 80 and 160 vs 100k fault injections.

We can observe that the accuracy difference value is almost random between the epoch 80 and epoch 160, it doesn’t tend to be more positive or more negative, however we can see that this randomness increases while the number of injections increases, so the more fault injections we have the more unpredictable the model accuracy is, this is probably caused because the more injections we have the more probable an important weight for the network is selected as a fault injection resulting in a more affected model and a reduced accuracy.

As we saw previously, the batch size 1024 behaved poorly in reliability in almost all the previous experiments, we wanted to see if there was a direct relationship between the batch size and the reliability. We did this by adding a model trained with a batch size of 2048 so we can see if by increasing the batch size the reliability decreases, the results are the following:

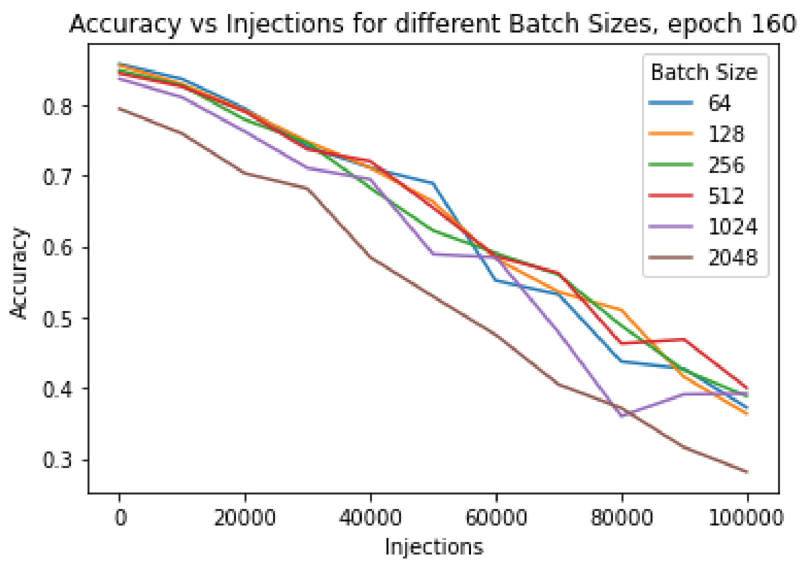


Figure 60. Accuracy vs 100k fault injections for SGD adding batch size 2048.

Previous studies have shown that when we increase the batch size too much the accuracy and quality of the model decreases [46], and this is what we are obtaining in the graph, the model trained with batch size 2048 has a lower accuracy than the other models. However, we want compare the model with the others model accuracy, so what we do is remove the accuracy offset by taking the mean of all the models and subtract it to each one, with this change we obtain a graph as if all the models would start with the same accuracy. The result is shown below:

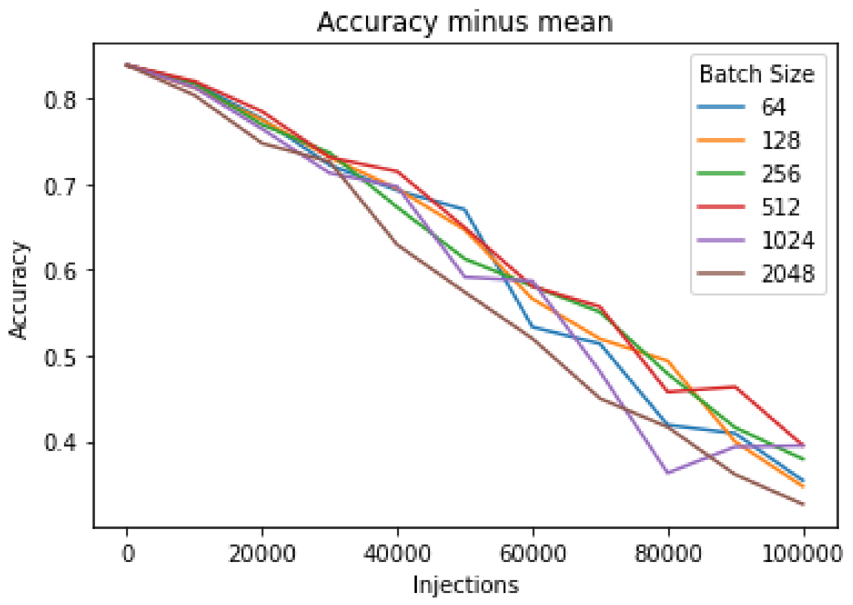


Figure 61. Accuracy vs 100k fault injections for different batch sizes, minus mean.

The result is that by increasing the batch size to 2048 we not only lose accuracy but also the reliability decreases, the model with batch size of 2048 performs the worst compared to the others for almost all the graph, thus having a low batch size (<1024) may increase slightly the reliability.

3.3.3 Adam – Varying the weight decay

We will vary the weight decay (WD) to see if the network reliability improves. We will give the WD the same values we did with SGD optimizer: 0,5e-7,5e-5,5e-3 and 5e-1. As we will see the training with values of WD higher that 5e-1, the network does not learn to classify the images. Studying the training of the models is important to see if all the models reach the maximum test accuracy within the selected number of epochs, also if it trains correctly or not (WD=5e-1) with the purpose of doing the following fault injections and compare their reliability. This time we will use 120 epochs to train the models as we have seen from previous trainings that this may be an enough number of epochs to let the model reach its maximum.

The training with Adam Optimizer for WD=0:

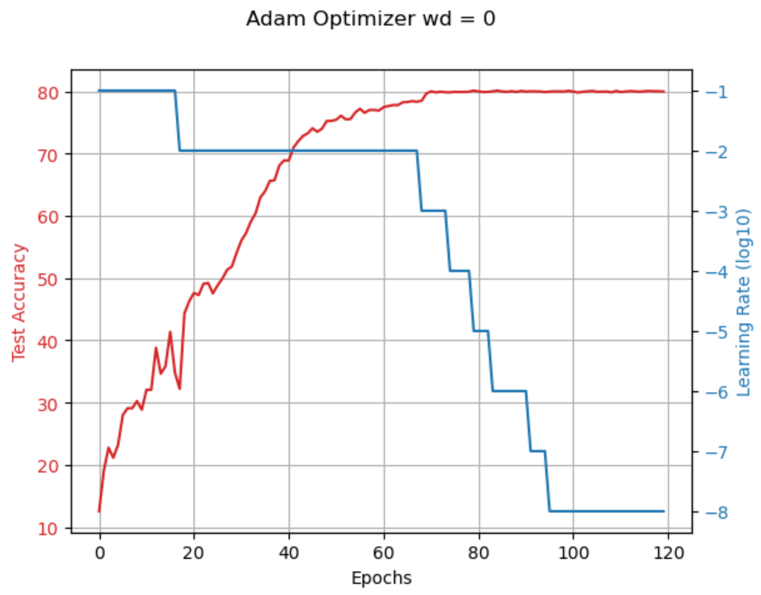


Figure 62. Training with Adam optimizer, WD = 0

The maximum accuracy is 80.13%, not a bad accuracy value, and it is reached around epoch 70.

The training for WD = 5e-7:

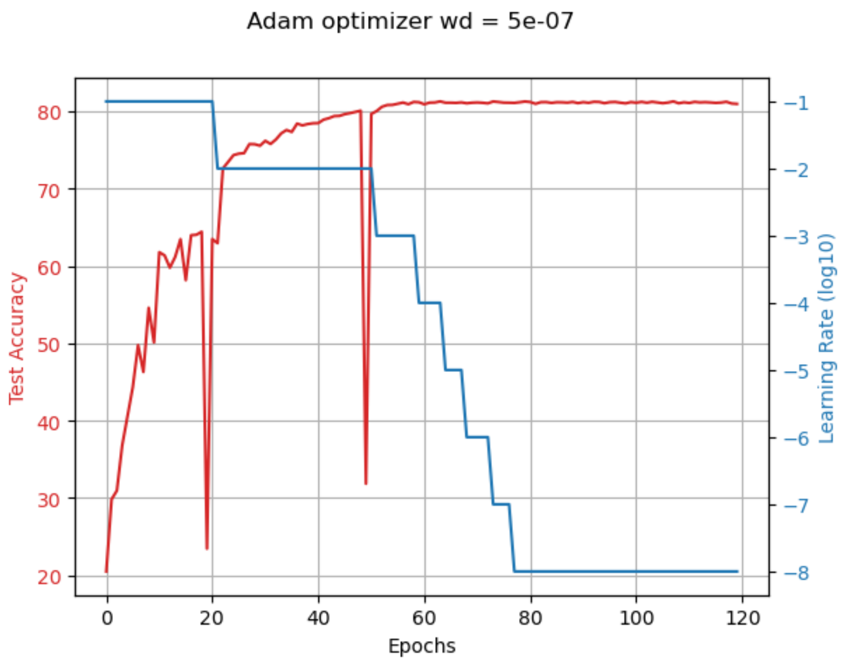


Figure 63. Training with Adam optimizer, WD = 5e-7

The maximum accuracy is 81.3%, higher than with no regularization even if it presents some falls in epochs 20 and 30. The maximum value is reached around epoch 60.

The training for WD = 5e-5:

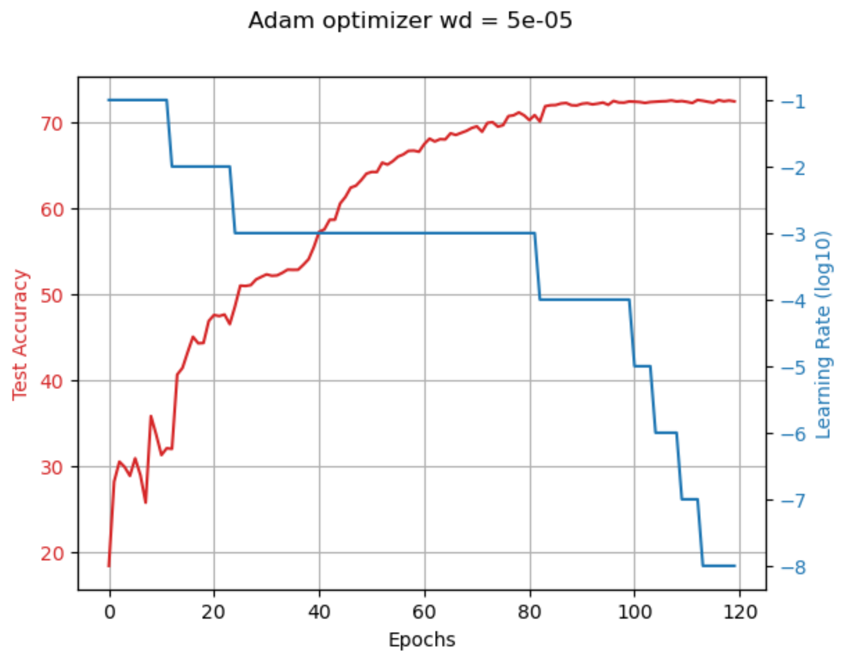


Figure 64. Training with Adam optimizer, WD = 5e-5

The maximum accuracy is 72.58%, being a low accuracy compared with the previous one, this value is reached around epoch 100.

Next, training for WD = 5e-3:

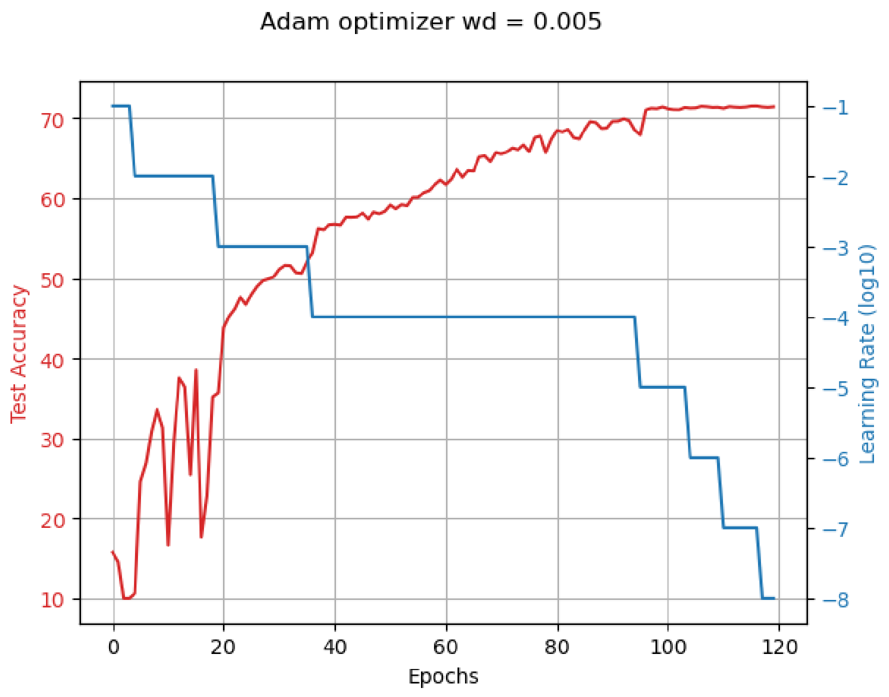


Figure 65. Training with Adam optimizer, WD = 5e-3

The maximum accuracy is 71.54%, reached at epoch 100. Again, obtaining a low value of accuracy.

Finally, the training for WD= 5e-1:

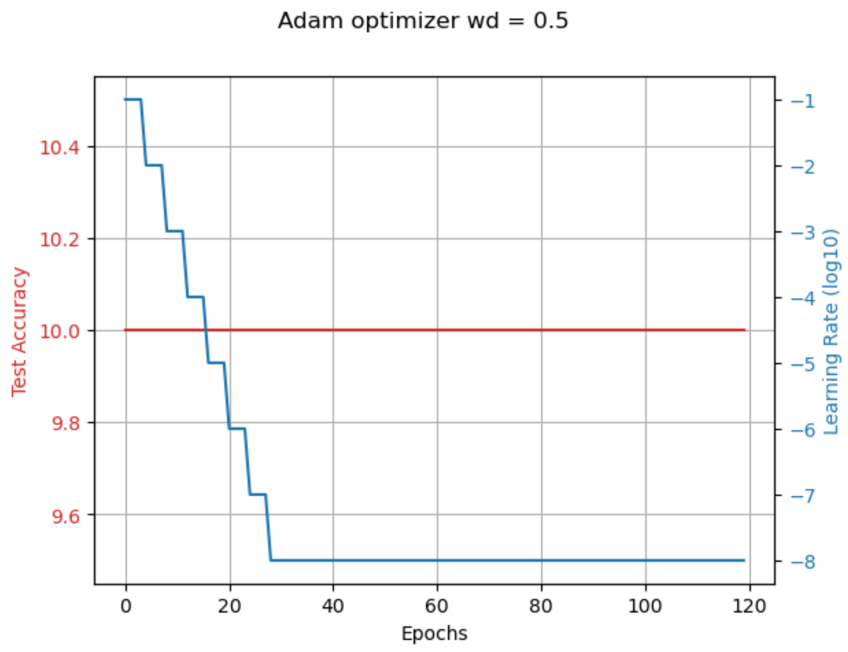


Figure 66. Training with Adam optimizer, WD = 5e-1

This time the model, similarly as what happened with SGD, with big values of WD it is not able to learn to classify, obtaining an accuracy of 10.0% which is equal to a random output.

Now we will test the reliability of the trained networks by injecting random weight faults among all network layers. First, we will inject 10k weight injections in steps of 1k.

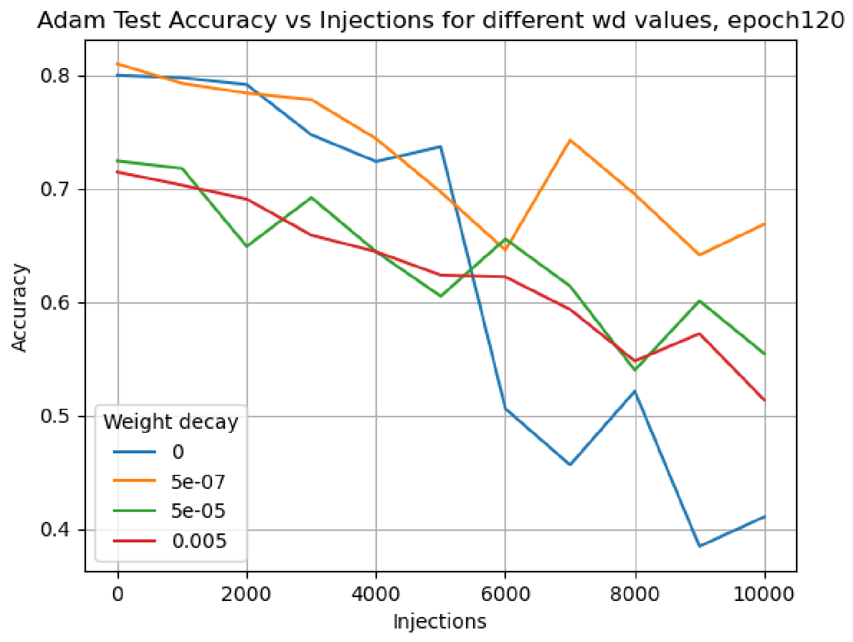


Figure 67. Adam test accuracy vs 10k injections for different WD values, epoch 120

We can observe that there is a difference in the initial accuracy, smaller values of WD lead to a better initial accuracy, and we can also find differences in terms of reliability. However, as the initial accuracy of the networks is different, this is, the models with zero injections have different accuracies, then we cannot compare the reliability accurately, for this purpose we will subtract the mean value of the accuracies, the results are shown below:

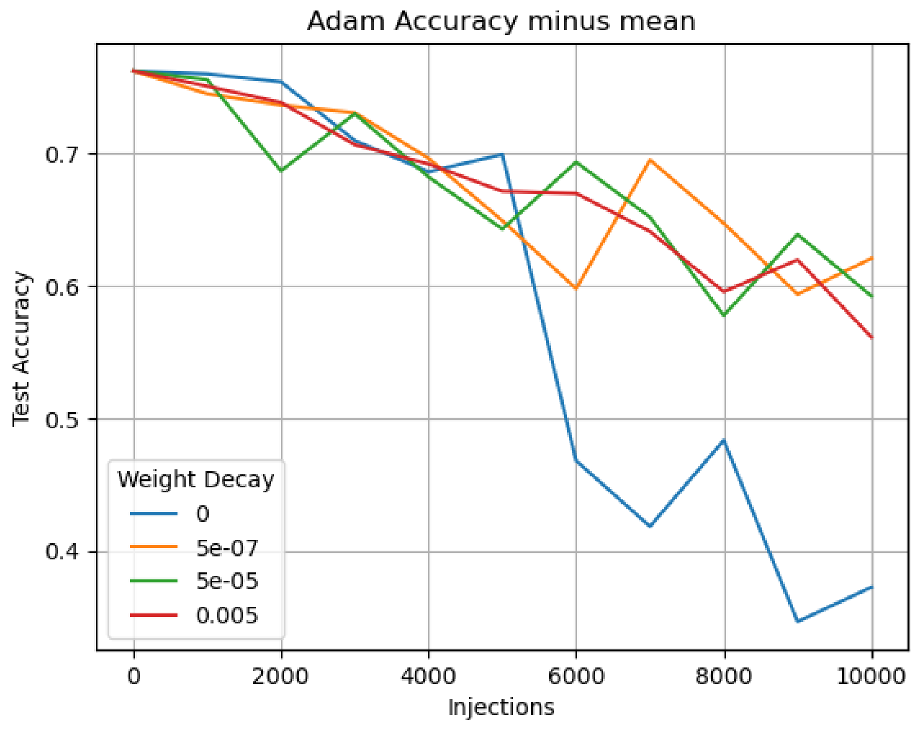


Figure 68. Adam test accuracy vs 10k injections for different WD values, epoch 120, minus mean

As we can see here and from the previous graph there is a model which performs worse than the others, and this is the model with no weight decay or WD equal to 0. We would recommend if we want an application with good reliability to not set a WD equal to 0 for this optimizer, however we first want to see how the model behaves with more injections, this time with 100k injections in steps of 10k.

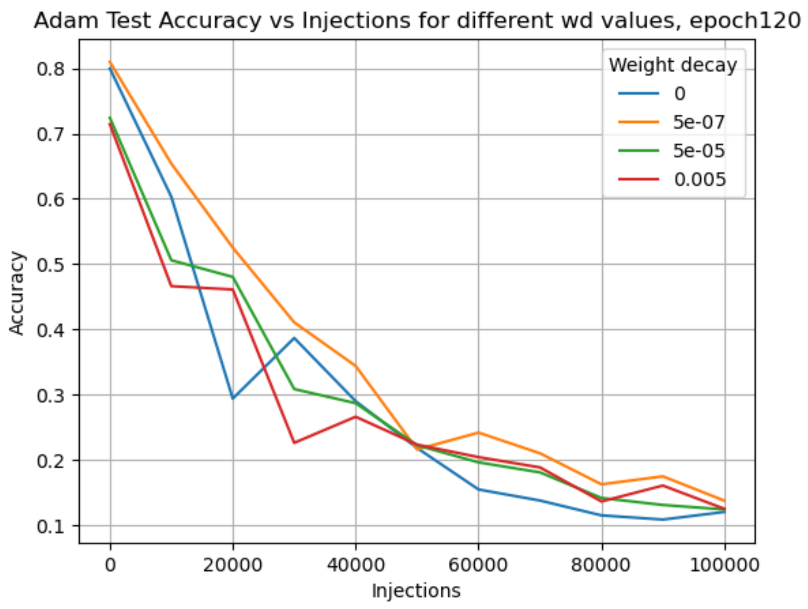


Figure 69. Adam test accuracy vs 100k injections for different WD values, epoch 120

Here we can see that for the firs 10k injections the reliability is similar to the previous graphs, however this time at 10k injections the accuracy for WD=0 remains above the models with high WD and goes below in 20k injections. So because of these variations, we cannot conclude that a low WD or a WD=0 would lead to a low reliability.

We can also notice something from the graph and it is that with Adam optimizer the accuracy is reduced in an exponential form rather than the linear form we had with SGD. We will review this behavior in a following section.

We also compare the models subtracting the mean:

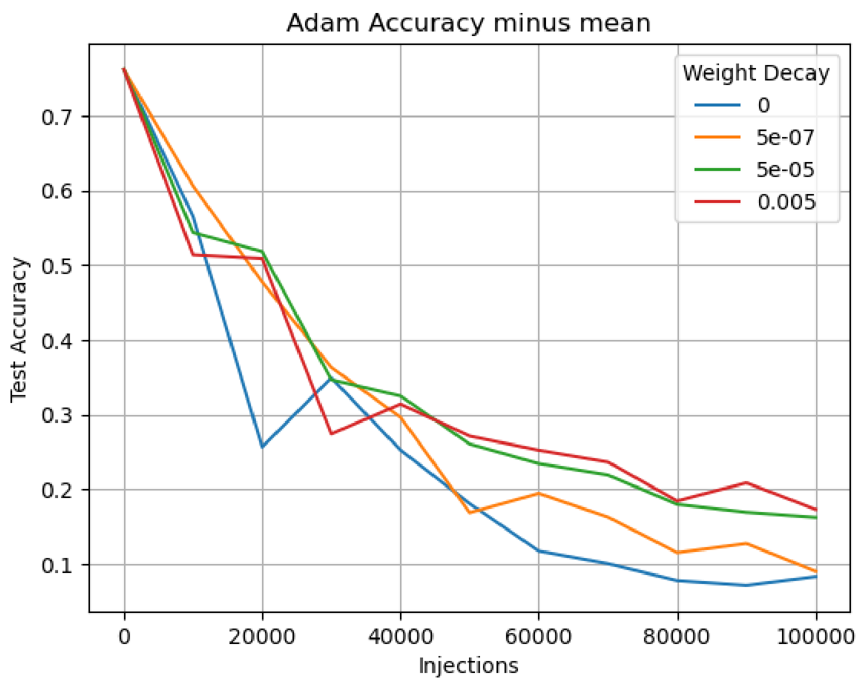


Figure 70. Adam test accuracy vs 100k injections for different WD values, epoch 120, minus mean

We confirm our observations made in the previous graph. Again we should be careful when comparing the models this way as they will converge to 0.1.

3.4 Adagrad optimizer

One of the advantages of Adagrad over SGD is that each weight has its own learning rate, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates. [47]

We will implement this algorithm and see how changing the training parameters affect the reliability of it. We will start from the parameters recommended by Pytorch:

optimizer = torch.optim.Adagrad(net.parameters(), lr=0.1, lr_decay=0, weight_decay=0, initial_accumulator_value=0, eps=1e-10)

3.4.1 Adagrad – Varying the batch size

As we did with other optimizers, we will vary the batch size parameter to see if there is a value that gives a better reliability performance. We will train the ResNet18 with the BS values of: 64, 128, 256, 512 and 1024.

First training the network with BS=64:

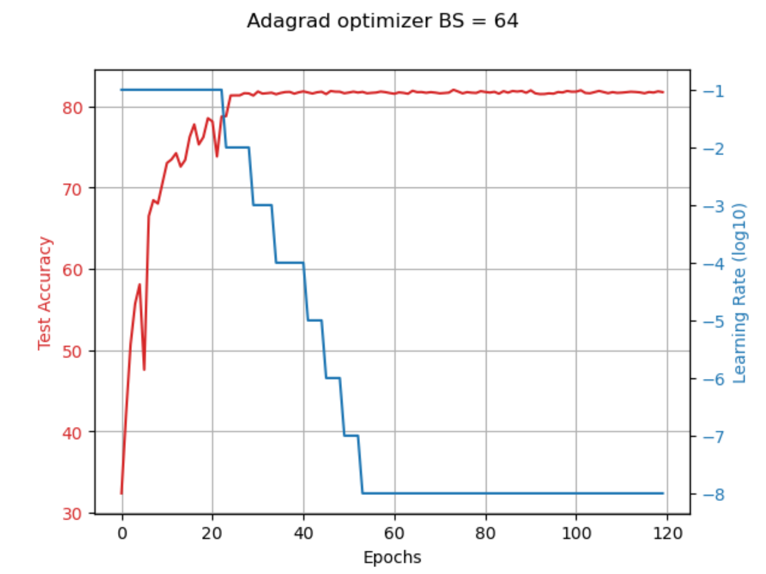


Figure 71. Training with Adagrad optimizer, batch size = 64

The maximum accuracy value obtained is 82.01%, the maximum is reached around epoch 30.

Next training the network for BS=128:

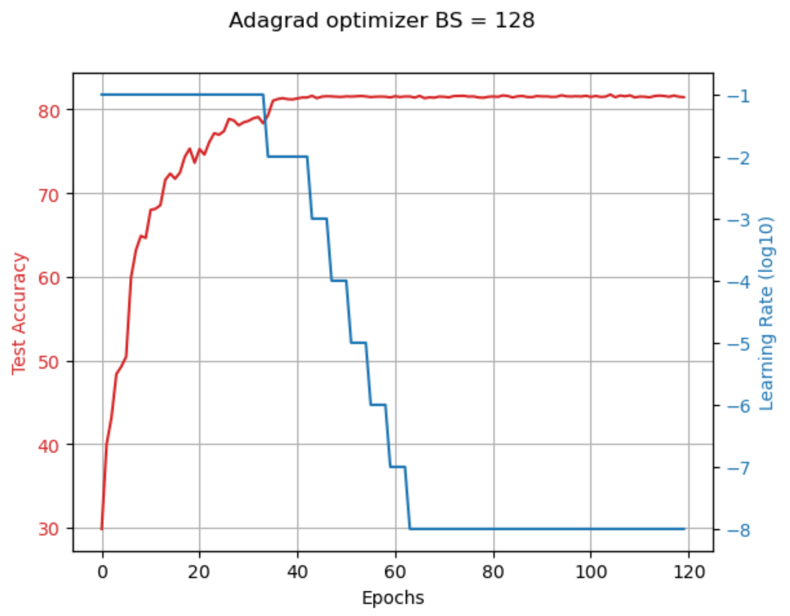


Figure 72. Training with Adagrad optimizer, batch size = 128

This time the accuracy is reduced to 81.78%, this maximum value is reached around epoch 40.

Next training for BS=256:

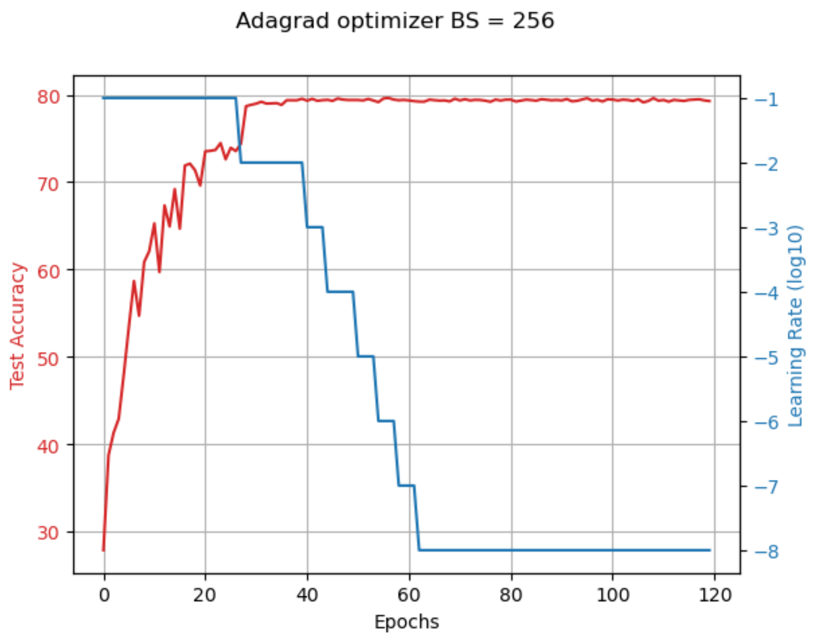


Figure 73. Training with Adagrad optimizer, batch size = 256

The accuracy is reduced again: 79.63%, it is reached around epoch 30.

Next training for BS = 512:

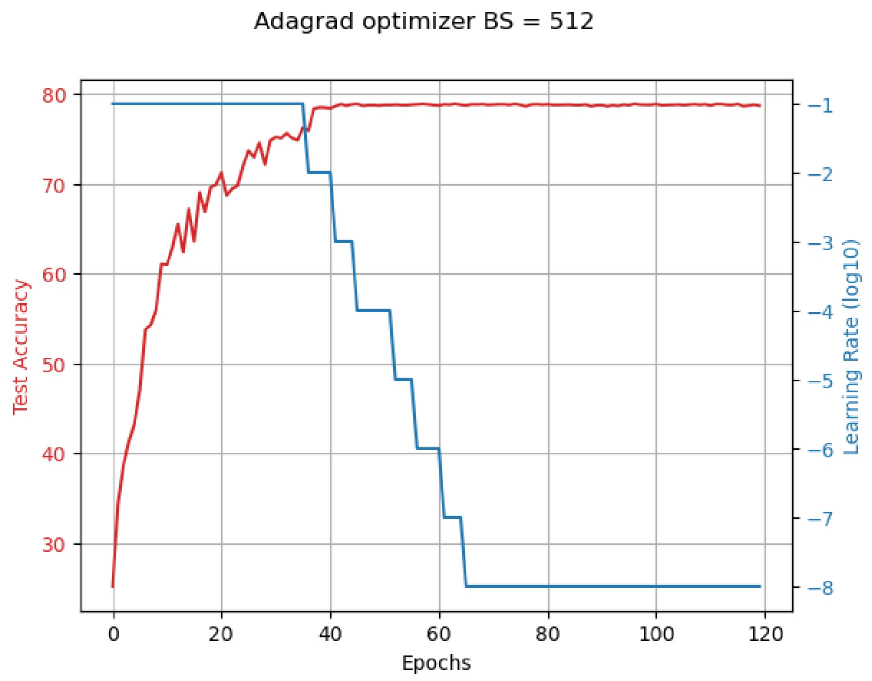


Figure 74. Training with Adagrad optimizer, batch size = 512

The accuracy keeps decreasing as we increase the batch size, this time to 78.9%. This maximum value is reached around epoch 40.

Finally training for BS = 1024:

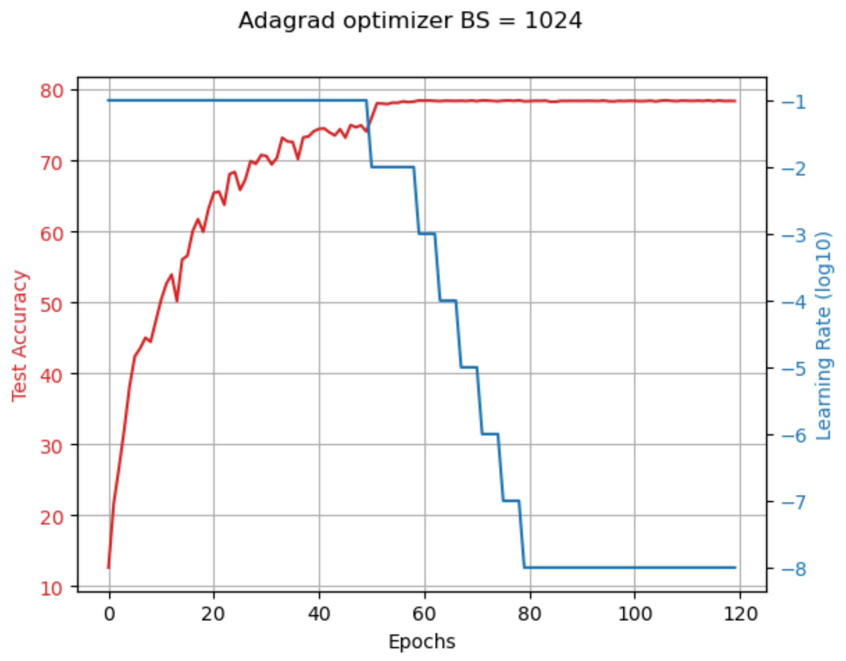


Figure 75. Training with Adagrad optimizer, batch size = 1024

The maximum accuracy reached is 78.47%. It is reached by epoch 50.

From the training we can observe that the accuracy for Adagrad optimizer decreases as the batch size increase, so a small batch size is recommended in terms of accuracy. Also the model is trained faster with small batch sizes.

Now lets see how the BS affect the reliability for Adagrad optimizer. We will implement first a total of 10k injections in steps of 1k:

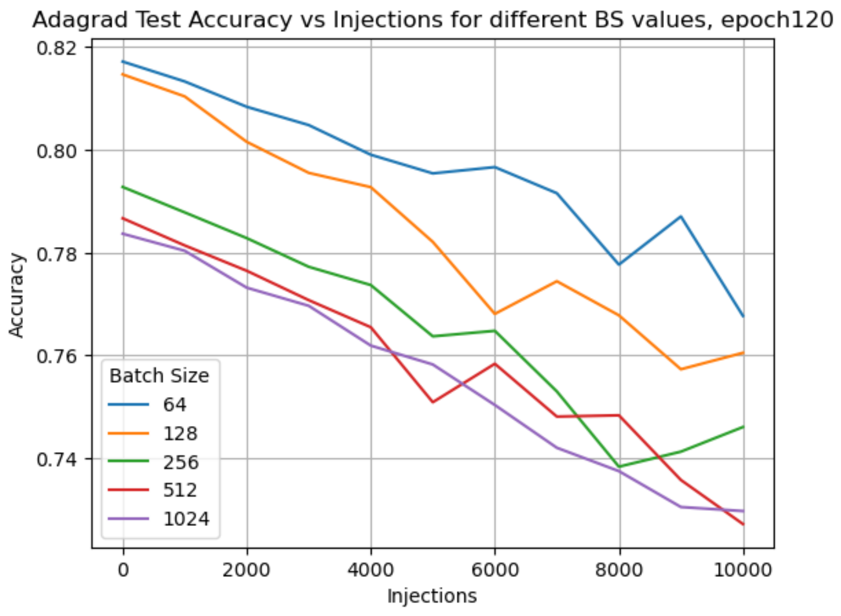


Figure 76. Adagrad test accuracy vs 10k injections for different batch size values, epoch 120

As we can see there is a big difference between accuracies so the comparison between reliabilities is hard to see, thus we subtract the mean from the graphs:

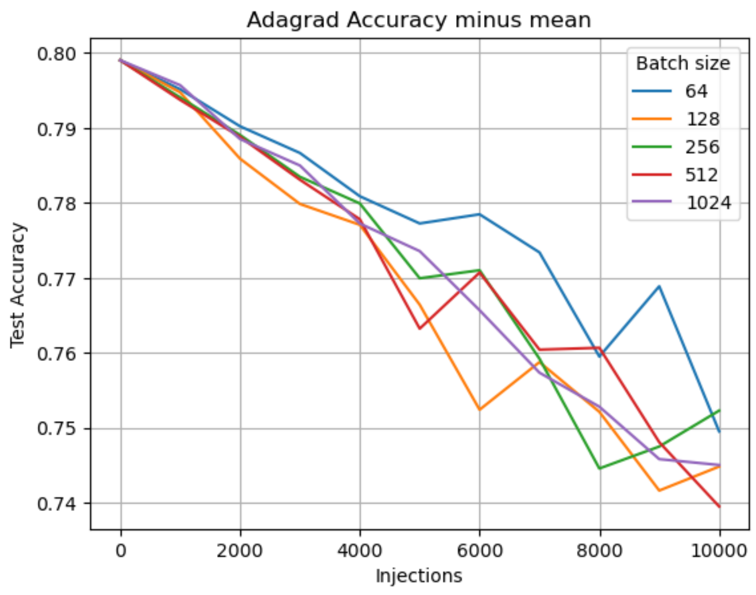


Figure 77. Adagrad test accuracy vs 10k injections for different batch size values, epoch 120, minus mean

We can see that the BS of 64 has a better performance in terms of reliability, lets see what happens in the range for the total of 100k injections:

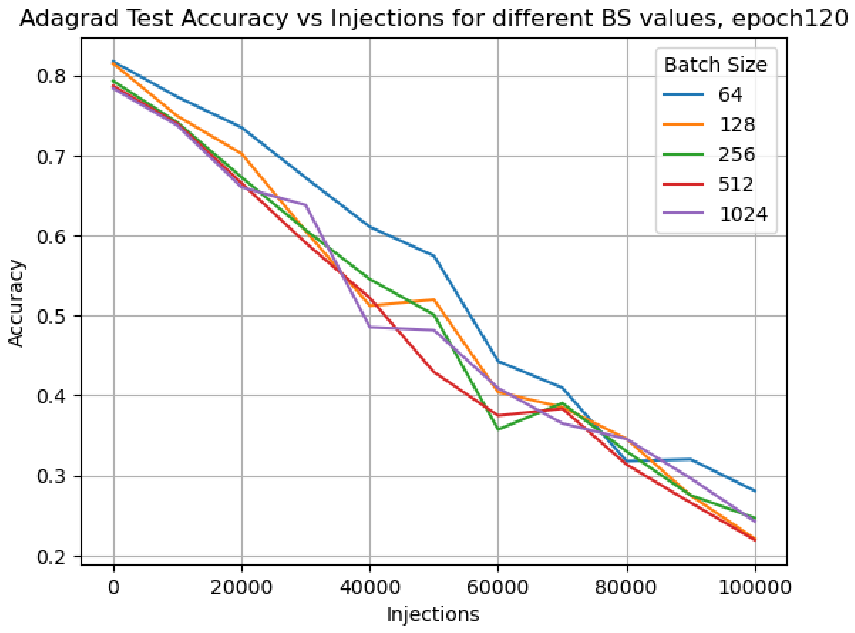


Figure 78. Adagrad test accuracy vs 100k injections for different batch size values, epoch 120

Again we can see that the BS of 64 is the model which performs the best compared with the others, lets subtract the mean value of the graphs to compare the reliability better:

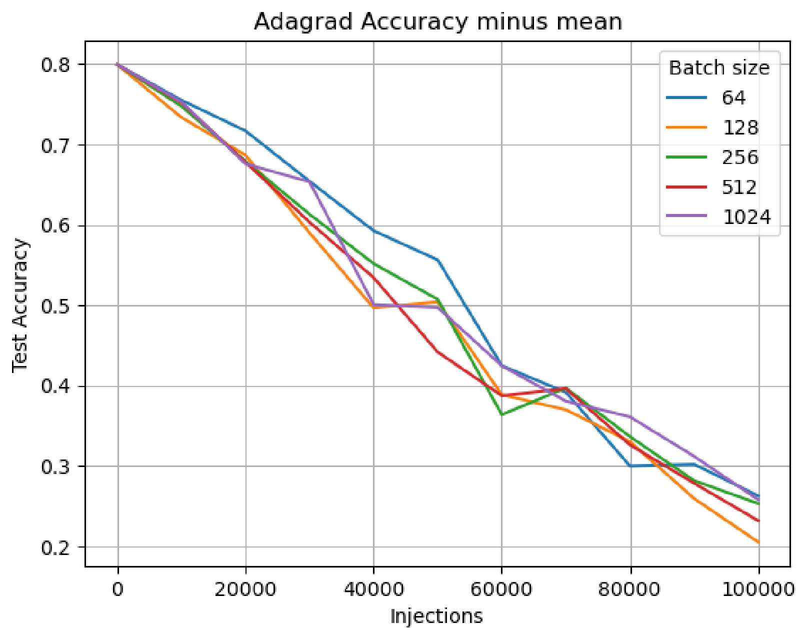


Figure 79. Adagrad test accuracy vs 100k injections for different batch size values, epoch 120, minus mean

When starting from the same accuracy we can see that the batch size of 64 performs the better in reliability for the most part of the graph.

3.4.2 Adagrad – Varying weight decay

Now we will vary the weight decay to see if there is a relationship between it and the reliability.

First we show the training for a WD=0:

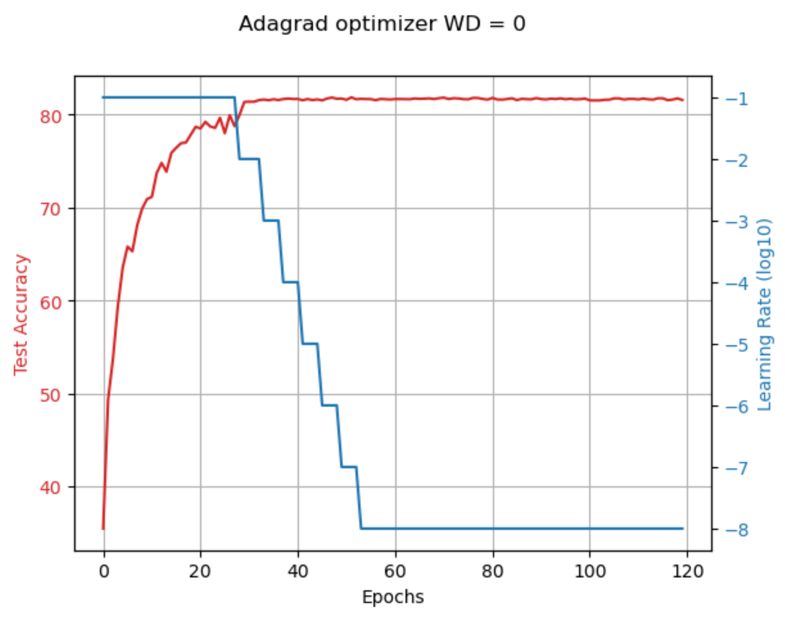


Figure 80. Training with Adagrad optimizer, WD = 0

We obtain a maximum test accuracy value of 81.85%, it is reached by epoch 30. Next we train the model for WD = 5e-7:

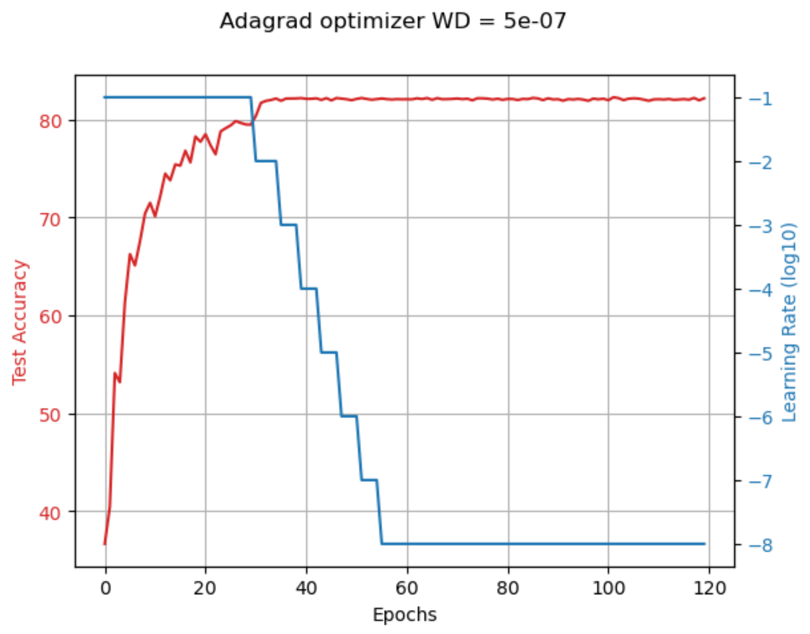


Figure 81. Training with Adagrad optimizer, WD = 5e-7

The maximum accuracy is 82.25%, a little bit higher than WD=0, it is reached by epoch 35. Next, training the model for WD= 5e-5:

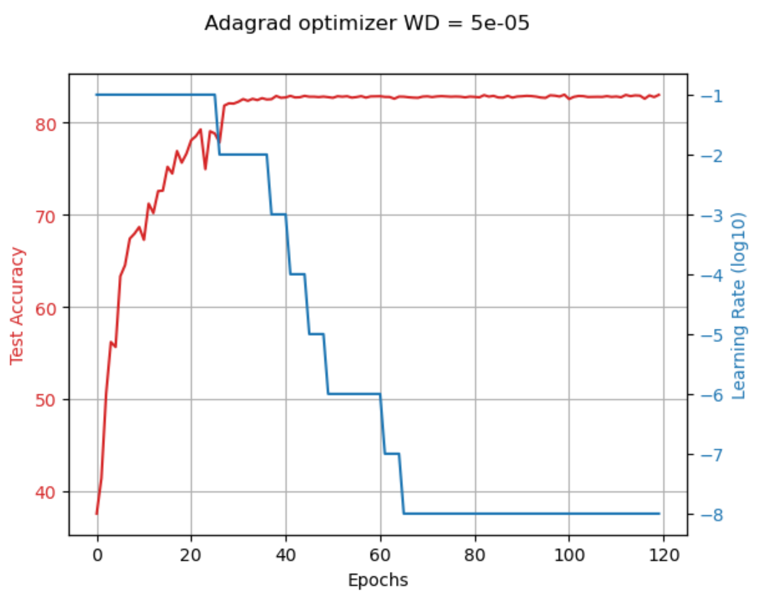


Figure 82. Training with Adagrad optimizer, WD = 5e-5

The maximum test accuracy is 83.04%, higher than WD = 5e-07, it is reached around epoch 30. Next, training for WD = 5e-3:

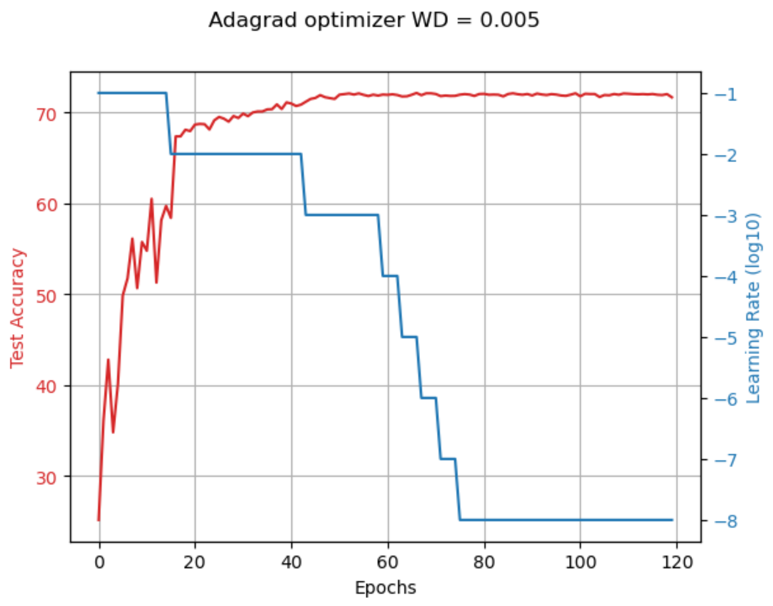


Figure 83. Training with Adagrad optimizer, WD = 05e-3

This time the accuracy reduces to 72.14%, it is reached around epoch 60. Finally for WD=5e-1:

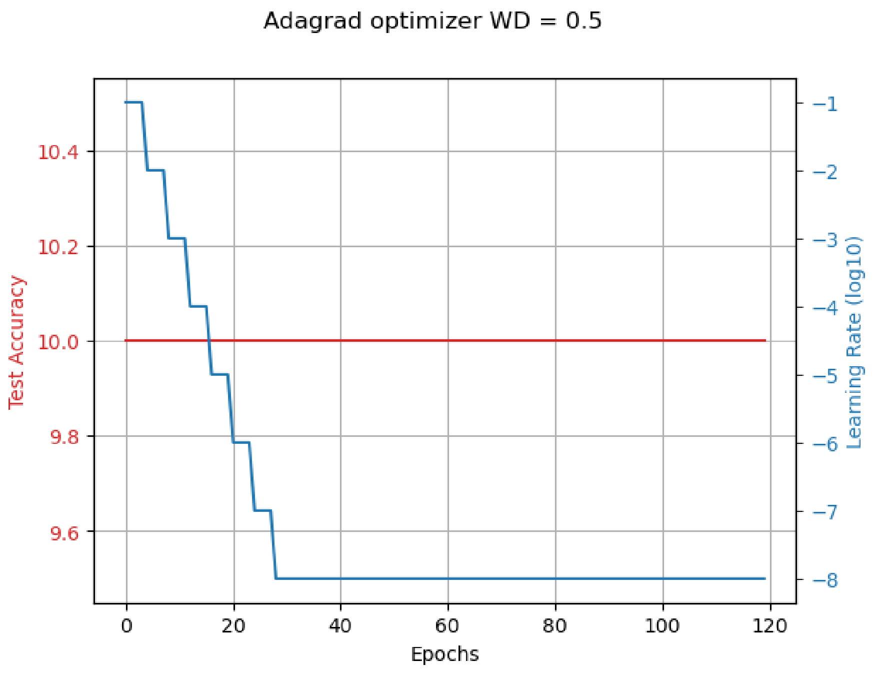


Figure 84. Training with Adagrad optimizer, WD = 5e-1

This time for a big value of WD the model is not able to learn to classify the images, the accuracy is a constant 10%.

Now let's inject the weight faults over each model (not WD=5e-1 because it didn't train) and see how the reliability is affected. First we will inject a total of 10k faults:

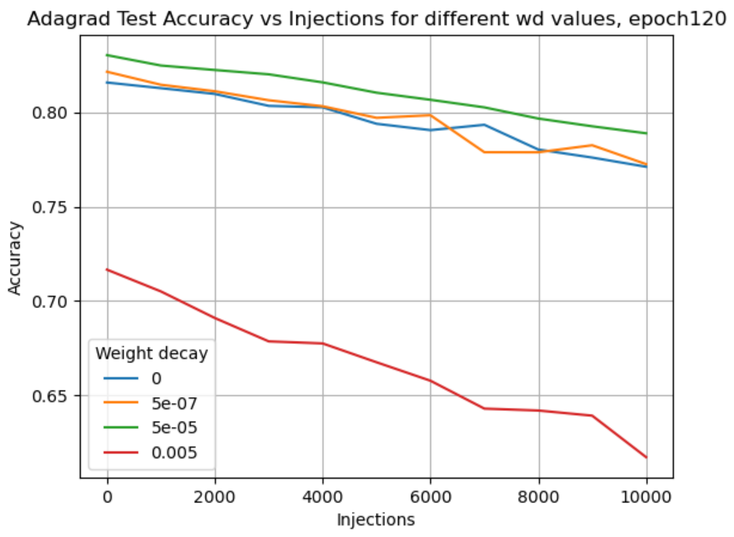


Figure 85. Adagrad test accuracy vs 10k injections for different WD values, epoch 120

We can observe that we have good accuracy and good reliability for small values of WD, but to get a better comparison we subtract the mean value from the graphs:

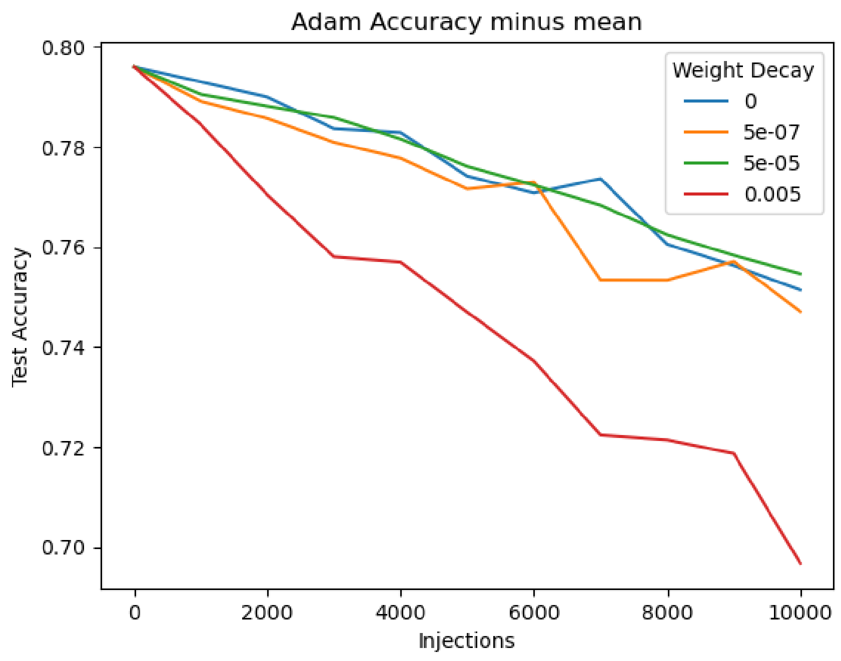


Figure 86. Adagrad test accuracy vs 10k injections for different WD values, epoch 120, minus mean

We can observe that the reliability is not good for big values of WD, i.e. 5e-3. We should keep a small value of WD if we want to achieve good accuracy and reliability. Now we inject more faults to see how the models perform with a greater number of injections.

For 100k injections we obtain the following results:

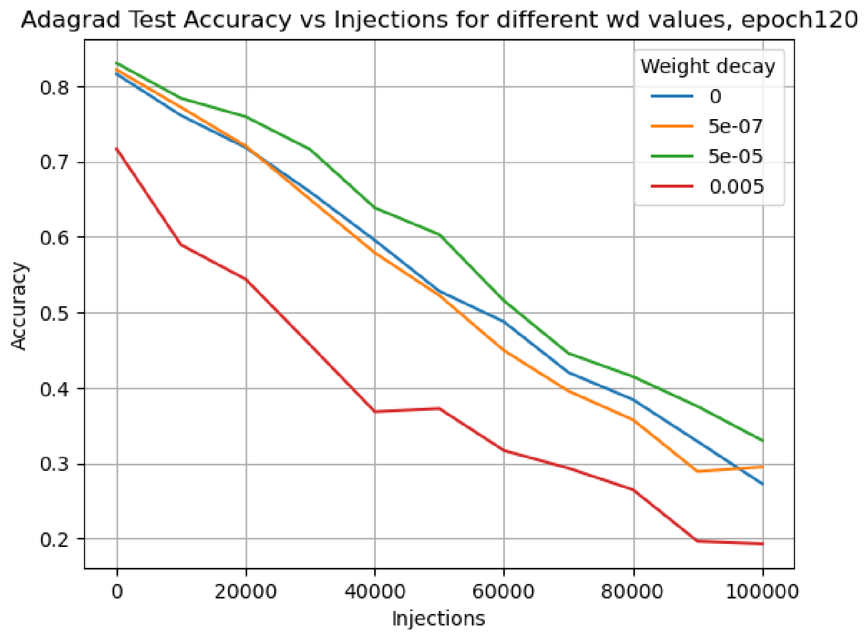


Figure 87. Adagrad test accuracy vs 100k injections for different WD values, epoch 120

As we can observe low values of WD lead to better accuracy, again we subtract the mean to compare the models better in terms of reliability:

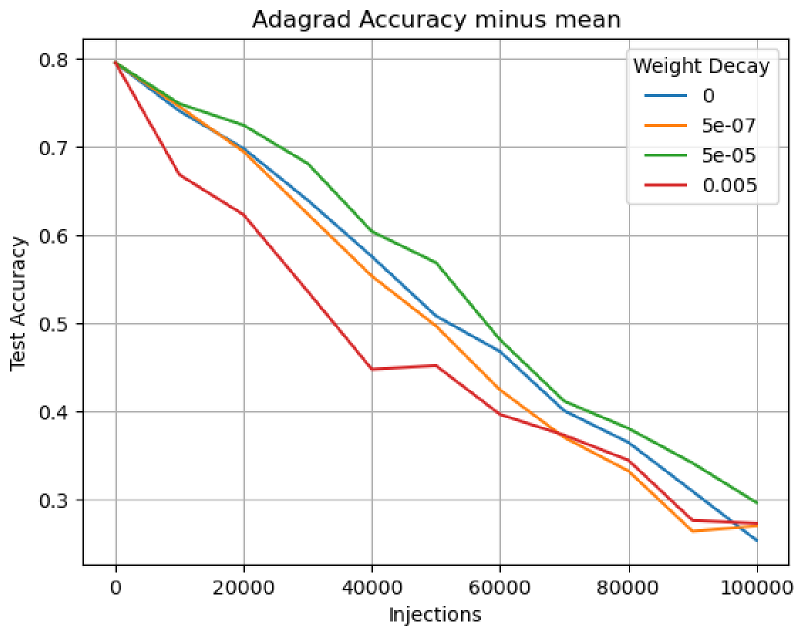


Figure 88. Adagrad test accuracy vs 100k injections for different WD values, epoch 120, minus mean

We can observe that low values of WD also result in a better reliability, mostly for the first thousands of injections.

3.5 Rmsprop optimizer

Rmsprop stands for Root Mean Square Propagation, it was developed in order better the results offered by Adagrad algorithm, that is, Rmsprop does not decay the learning rate too quickly preventing convergence. We are going to implement Rmsprop and study it's reliability. We will vary some of the its common training parameters starting from the default parameter values recommended in the Pytorch library:

optimizer=torch.optim.RMSprop(net.parameters(), lr=0.1, alpha=0.99, eps=1e-08, weight_decay=wd, momentum=0, centered=False)

3.5.1 Rmsprop -- Varying the batch size

We will vary the batch size training parameter, first we will show the training graphs and accuracy with the pass of the epochs to check that the models trained successfully. First for a BS of 64:

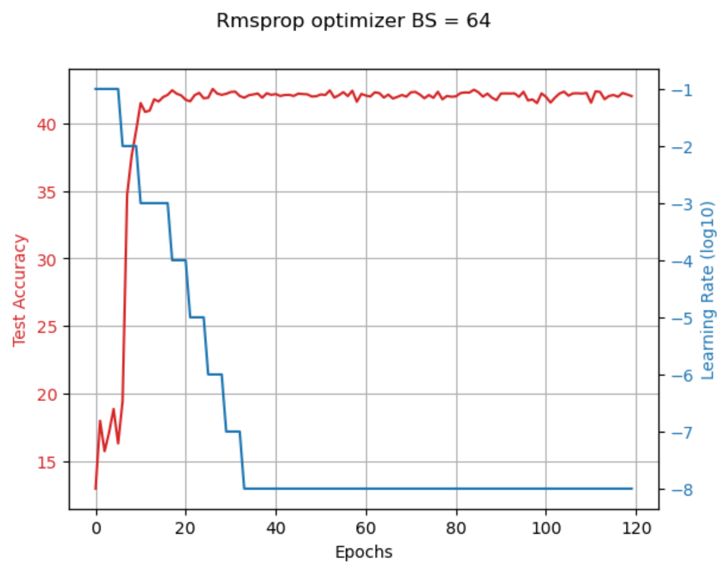


Figure 89. Training with Rmsprop optimizer, batch size = 64

We obtain a low accuracy compared with the accuracies reached by other optimizers, a maximum of 42.55% is obtained for BS = 64. Next for BS = 128:

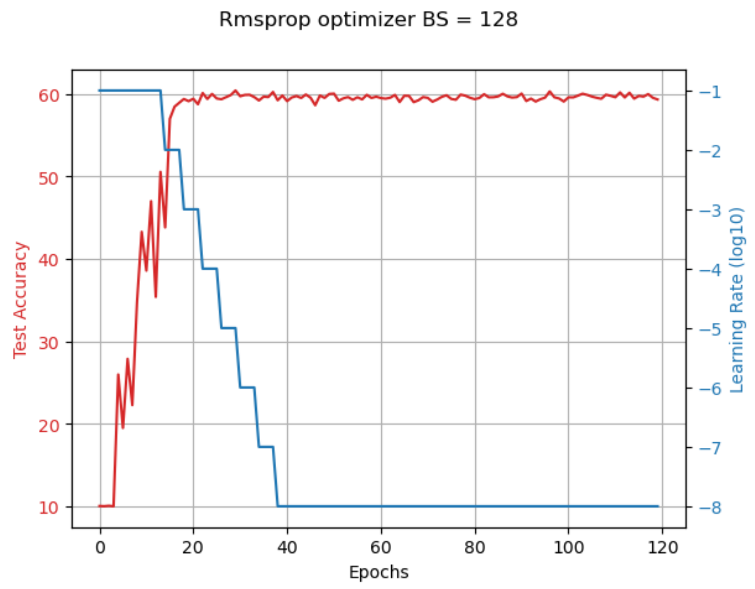


Figure 90. Training with Rmsprop optimizer, batch size = 128

The maximum accuracy is 60.39%, it is better than the previous but still not very good. Next for BS = 256:

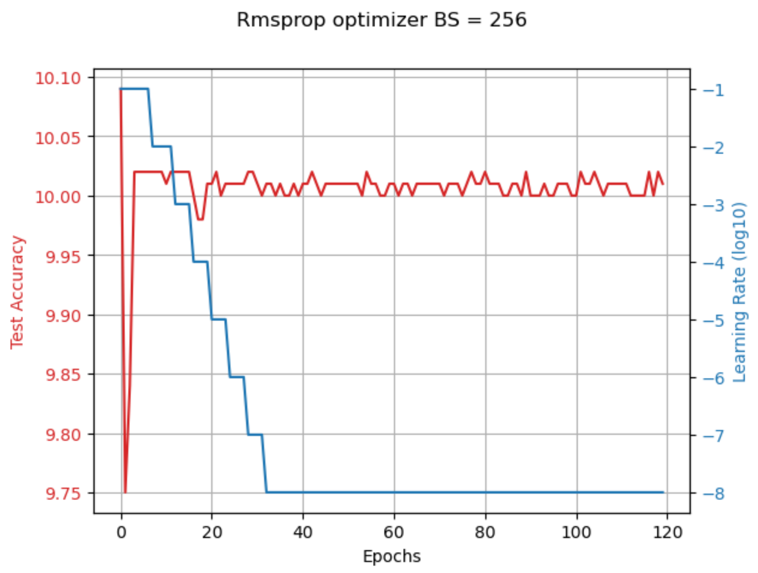


Figure 91. Training with Rmsprop optimizer, batch size = 256

For this BS the model is not able to train and the maximum test accuracy reached as shown in the graph is 10.09%. Next, training for BS = 512:

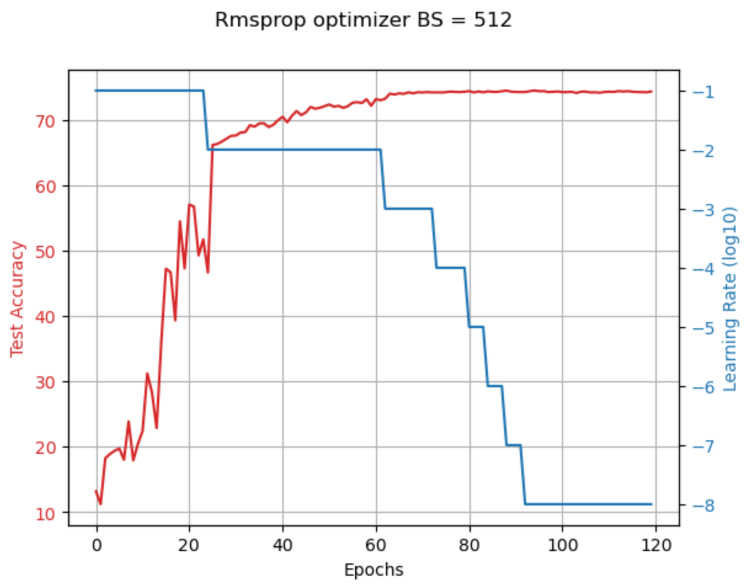


Figure 92. Training with Rmsprop optimizer, batch size = 512

The accuracy increases but still not the best, a maximum of 74.45% is reached. Next for BS=1024:

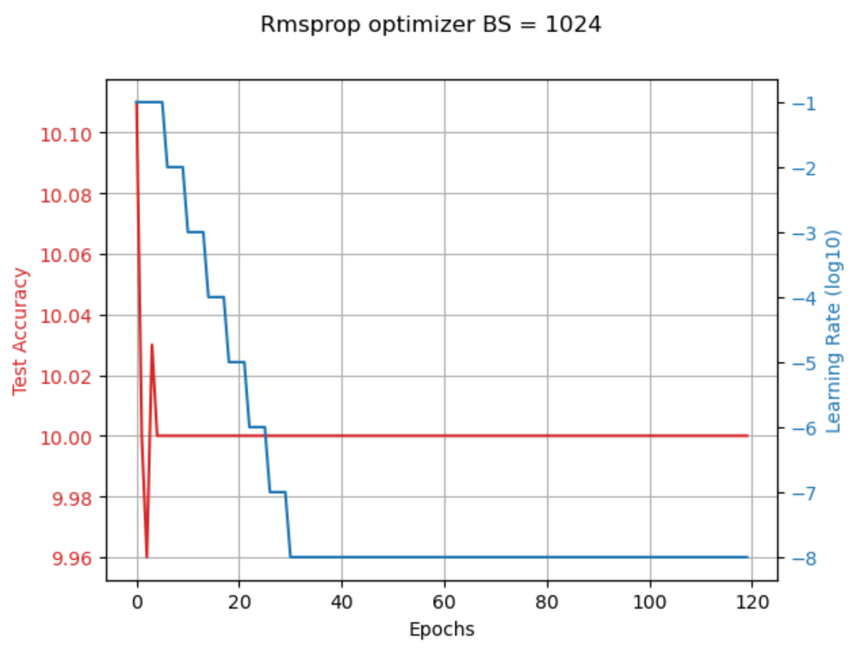


Figure 93. Training with Rmsprop optimizer, batch size = 1024

Again the model is not able to train. As most of the models were not able to train very well, we will not do a fault injection campaign for this parameter.

3.5.2 Rmsprop – Varying the weight decay

We will vary the parameter weight decay for the rmsprop optimizer and see how it is related to reliability. First for a WD of 0:

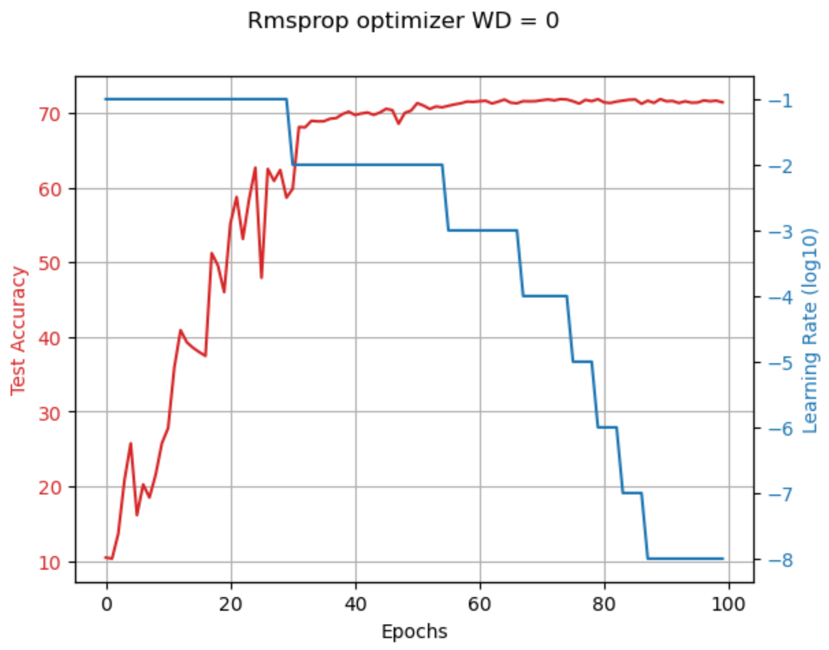


Figure 94. Training with Rmsprop optimizer, WD = 0

The maximum accuracy reached is 71.83%, not as high as the ones reached by other optimizers.
Next WD=5e-7:

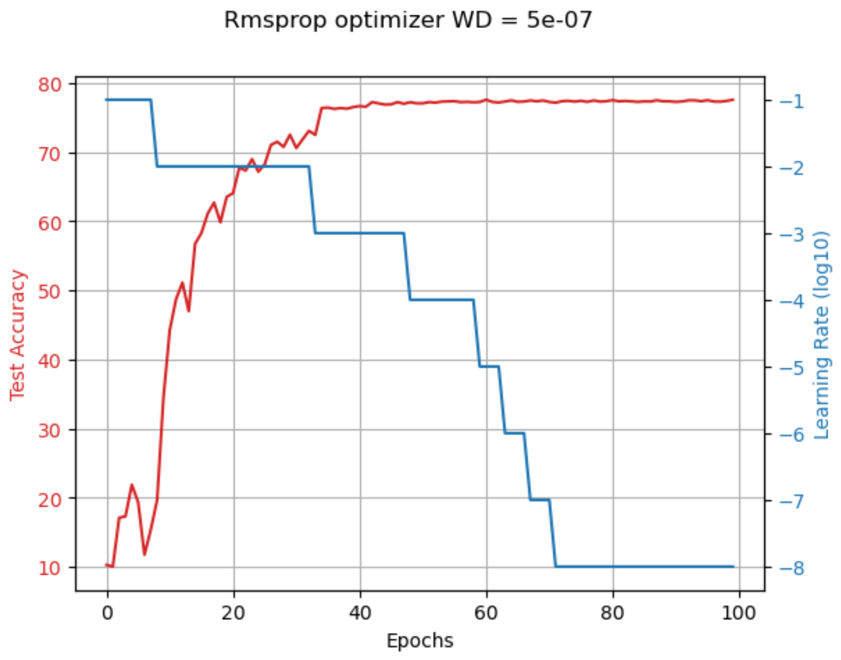


Figure 95. Training with Rmsprop optimizer, WD = 5e-7

This time the accuracy is 77.59% a decent value for this optimizer if compared with the best accuracies obtained by other optimizers (>80%).

Next, training for WD= 5e-5:

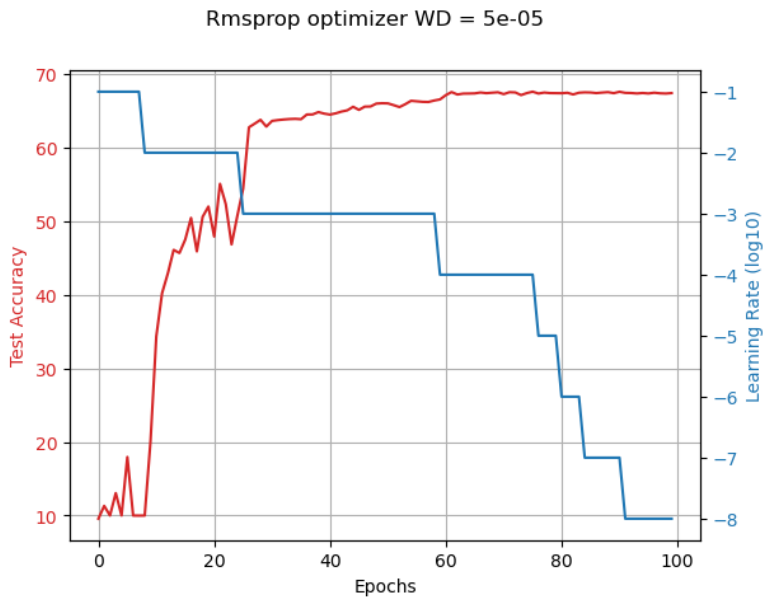


Figure 96. Training with Rmsprop optimizer, WD = 5e-5

The maximum accuracy reached is 67.6, not so high. Next WD is 5e-3:

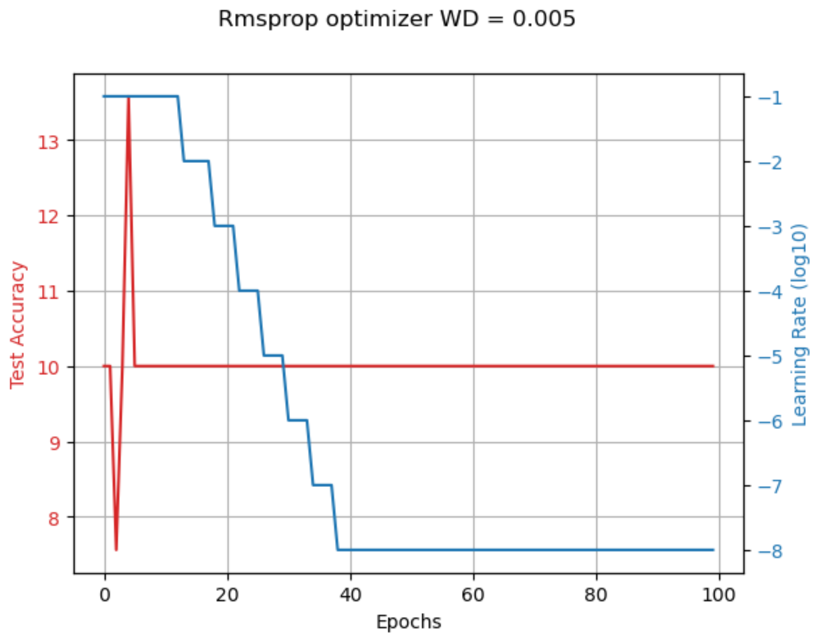


Figure 97. Training with Rmsprop optimizer, WD = 5e-3

The model is not able to train, the max accuracy reached is 13.58.

Next, training the model for WD = 5e-1:

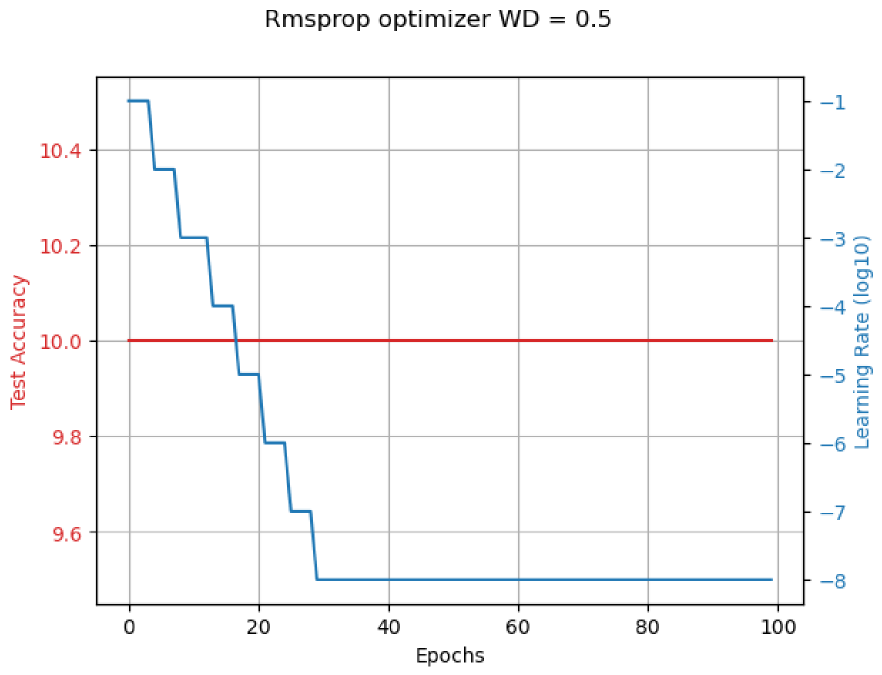


Figure 98. Training with Rmsprop optimizer, WD = 5e-1

Again, for this value of WD the model is not trained and the accuracy is 10%.

Next we will do the fault injection process for the models which trained well. First 10k injections with a step of 1k:

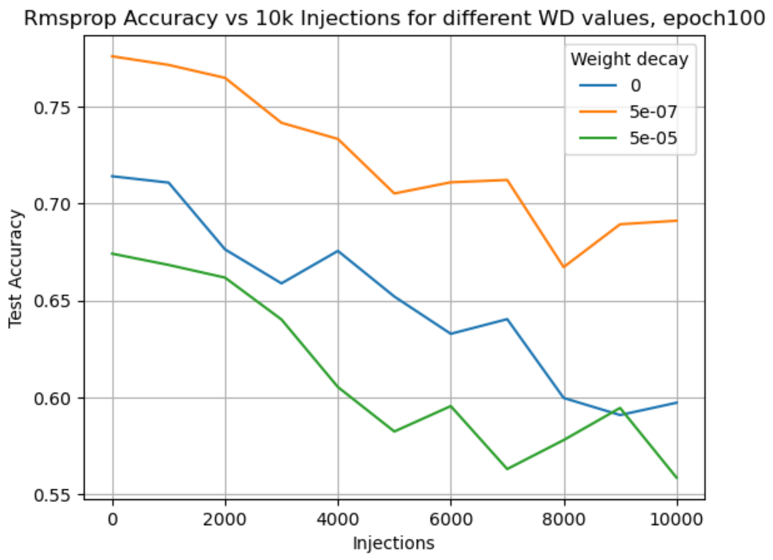


Figure 99. Rmsprop accuracy vs 10k weight injections for different WD values, epoch100

As we can observe the graphs start at different accuracy, just as the results of the training, but there is no clear tendency for accuracy. Let's see the reliability at the moment of subtracting the mean:

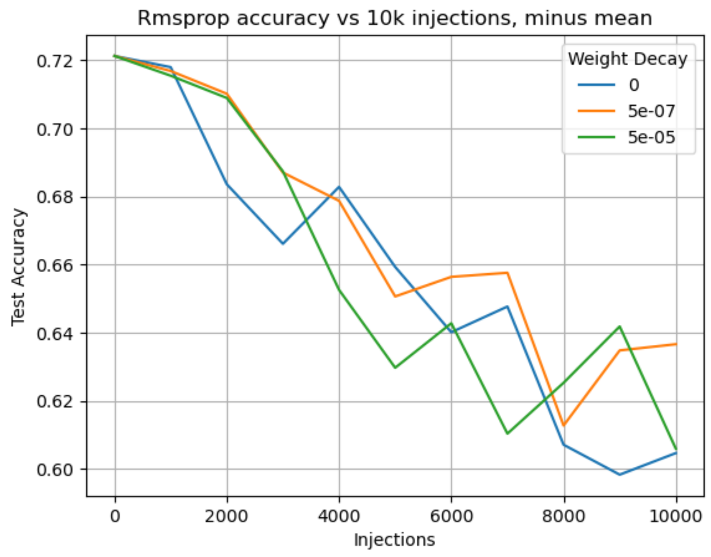


Figure 100. Rmsprop accuracy vs 10k injections for different WD values, epoch 100, minus mean

There is no clear relation between the weight decay and the reliability. Now we will increase the number of injections, for a total of 100k fault injections:

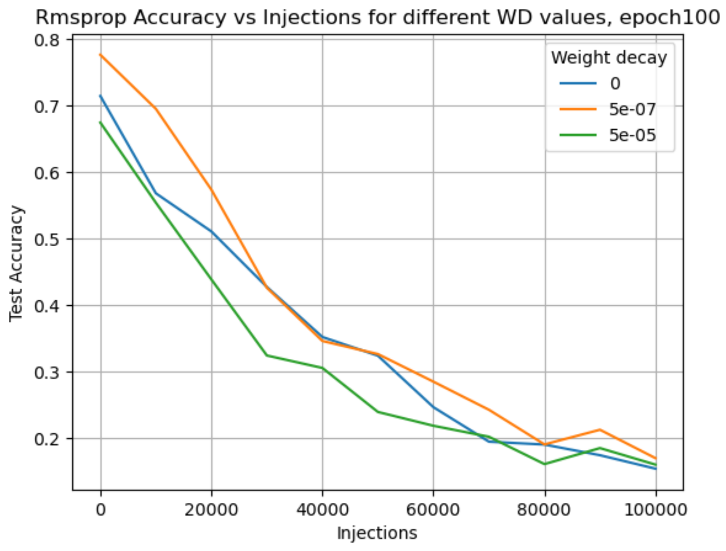


Figure 101. Rmsprop test accuracy vs 100k injections for different weight decay values, epoch 100

Although the accuracies are different, the reliability seems to be similar between the models, we will subtract the mean for 100k injections to see better how the models perform on reliability:

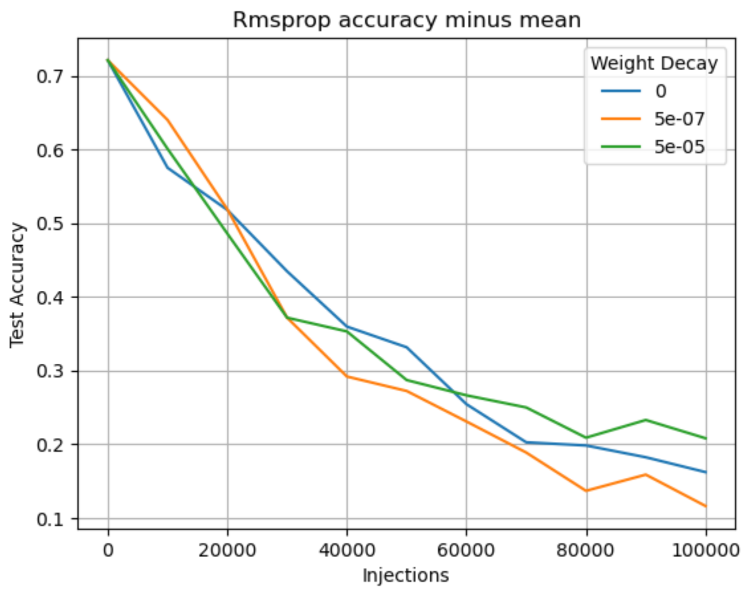


Figure 102. Rmsprop test accuracy vs 100k injections for different weight decay values, epoch 100, minus mean

As we can observe there is no clear winner of which WD value to use to improve the reliability for this optimizer.

3.6 Optimizer Comparison

After the injection campaigns done for each optimizer, we compare the reliability between them. We will select the models which reached the best reliability in the training process. Thus, the selected models are:

- SGD: BS = 128 (non specified parameters are the stablished by default)
- Adam: BS = 256
- Adagrad: BS = 64
- Rmsprop: WD = 5e-7

Now we graph this models in a single figure, we must remember that all the previous experiments and thus the following figures were drawn based on an average of 5 to 10 trials:

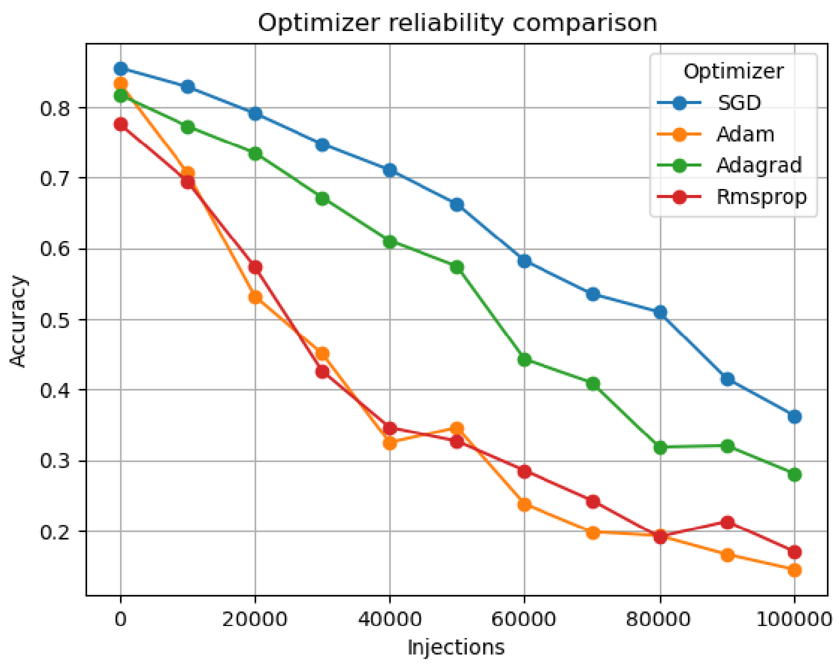


Figure 103. Optimizer reliability comparison.

As we can observe there are models trained with optimizers that are more resilient to faults as their lines are higher in the figure, we can subtract the mean from the graphs to see a better comparison of reliability:

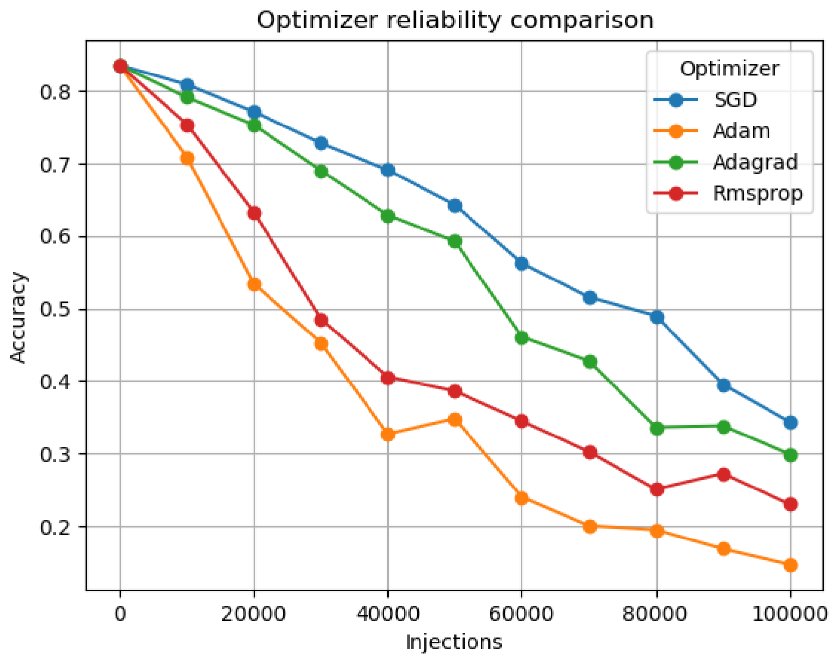


Figure 104. Optimizer reliability comparison, minus mean

We can observe from the figure that the optimizer which presents the best reliability is SGD, followed by Adagrad, then Rmsprop and Adam. More than anything a big gap can be observed between the optimizers SGD, Adagrad and Adam, Rmsprop.

Now we will try to understand why is this happening, we will check some statistics of the weights of the models, for example the weights distribution, this is the standard deviation:

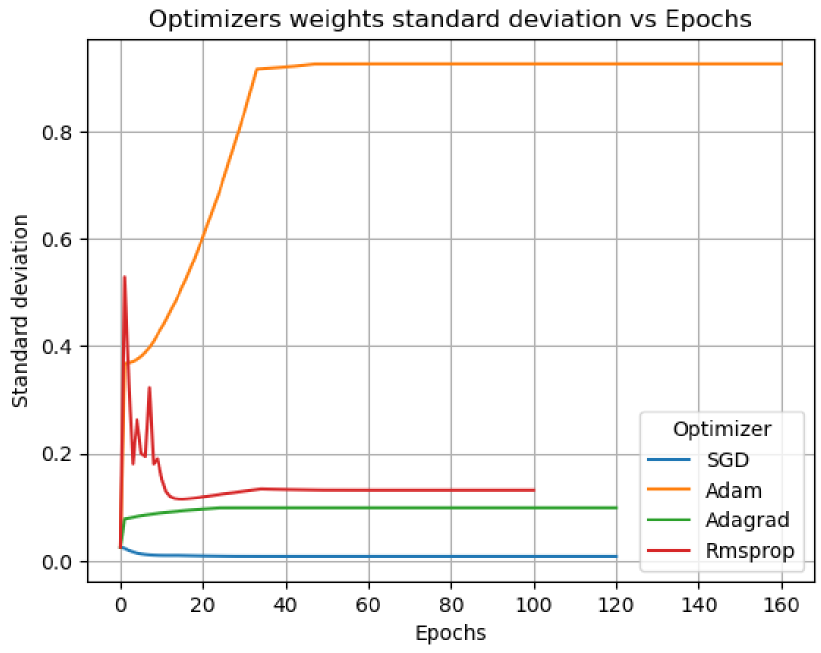


Figure 105. Optimizers weights standard deviation vs epochs

As we can observe, the low standard deviation of the weights of the models follow the order of reliability obtained in the Figure 104, first the optimizer SGD, then the optimizers Adagrad, Rmsprop and Adam, meaning that there is probably a relationship between the standard deviation and the reliability and that the weights are probably better distributed in the SGD models.

Another metric we can inspect is the mean value of the weights:

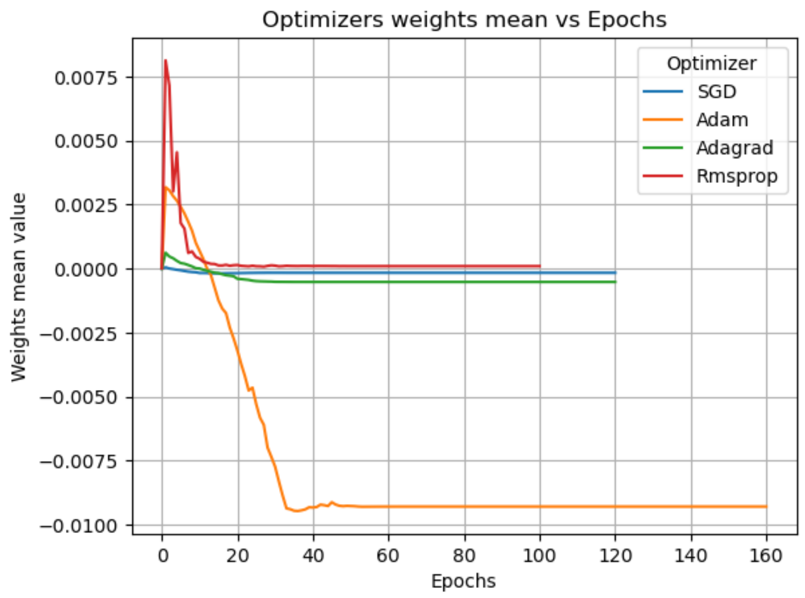


Figure 106. Optimizers weights mean vs Epochs

This time a small value of mean does not follow the same order of reliability for different optimizers, as SGD and Rmsprop have the smaller value of mean followed by Adagrad and Rmsprop. However, the mean may not be an accurate metric because the positive values of the weights counter the negative ones, thus another metric such as magnitude may give us a better insight of the behavior as we will see next.

The reliability may also be related to the quantity of important weights to the network, weights that if modified from its original value are more susceptible to lead to an erroneous output, weight importance has been studied in other areas such as weight pruning [48] and one of the metrics used to identify important weights is the magnitude of the weights [48] [49]. Thus, we are going to analyze the magnitude in our selected models and see if there is any relation with the reliability, first we see how the magnitude behaves in the training of a model, for example with the SGD optimizer:

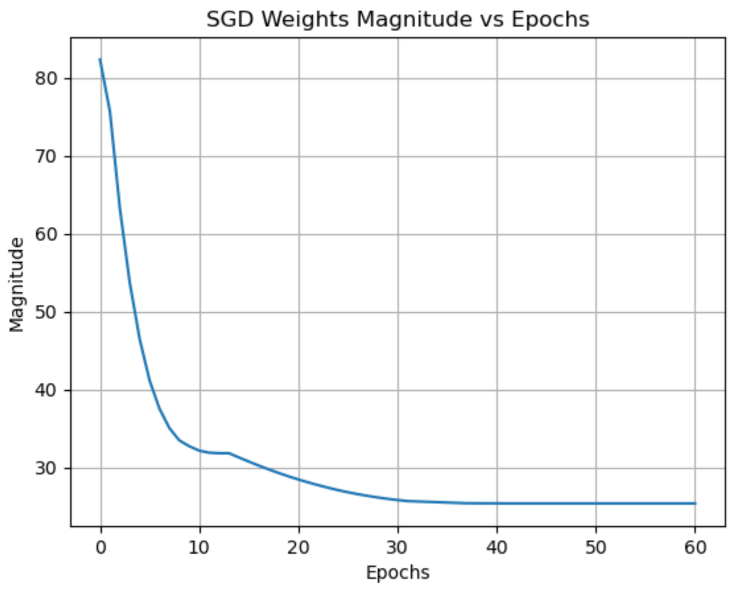


Figure 107. SGD weights magnitude vs epochs

We can see that the magnitude of the weights diminishes with the pass of epochs.

Then we take the total magnitude of the weights for epoch 80 for our selected models and the results are:

- SGD: 25.9687
- Adam: 3069.0245
- Adagrad: 325.2283
- Rmsprop: 434.9624

As we can see, the optimizer which gives the smallest magnitude is SGD, followed by Adagrad, Rmsprop and Adam. This is the same order of reliability obtained in Figure 104 (Optimizer reliability comparison), which may indicate a direct relationship between the total magnitude and the reliability.

To the moment we suspect a possible relationship between the reliability and the standard deviation and magnitude of the weights, a smaller value of standard deviation or magnitude may lead to a higher reliability. To confirm or refute this supposition we will do another couple of experiments. As we saw from the previous graphs (Figure 105 and Figure 107), for the SGD optimizer the standard deviation and magnitude of the weights decrease as the epochs increase, so we are going to train the network with SGD optimizer and select different epochs from it to obtain models with different standard deviation and magnitude, we expect to obtain in the lower epochs a higher standard deviation and magnitude and a lower reliability, and in the higher epochs a lower standard deviation and magnitude and a higher reliability.

The epochs we will select to do the fault injections and test the reliability will be first at the moment where we have an accuracy plateau and the learning rate needs to be updated probably caused by a local minima, so after training the network we obtain the following graph:

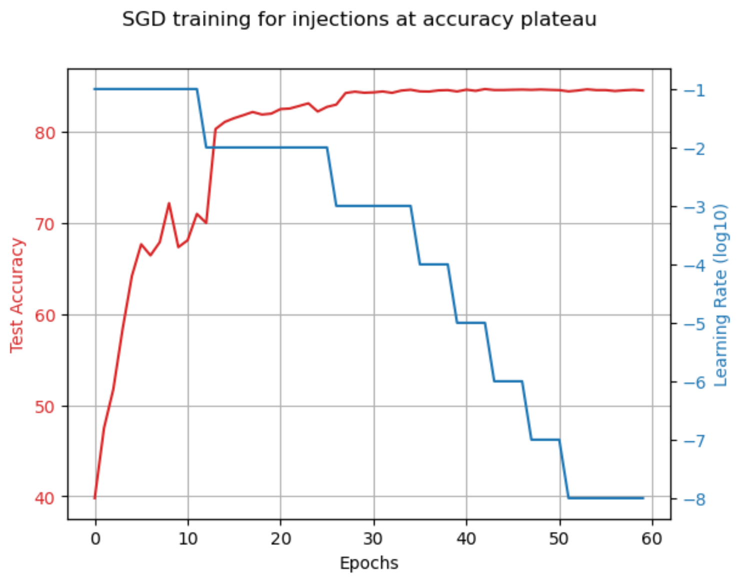


Figure 108. SGD training for injections at accuracy plateau

From the figure we observe that we have a first plateau at epoch 13 as the learning rate changes in this epoch, then it changes again at epoch 27 and then it changes many times but the accuracy does not improve, so the epochs we will select for the fault injections are 13,27 and the last learning rate change which is in epoch 52.

After performing the fault injections in these epochs we obtain the following graph:

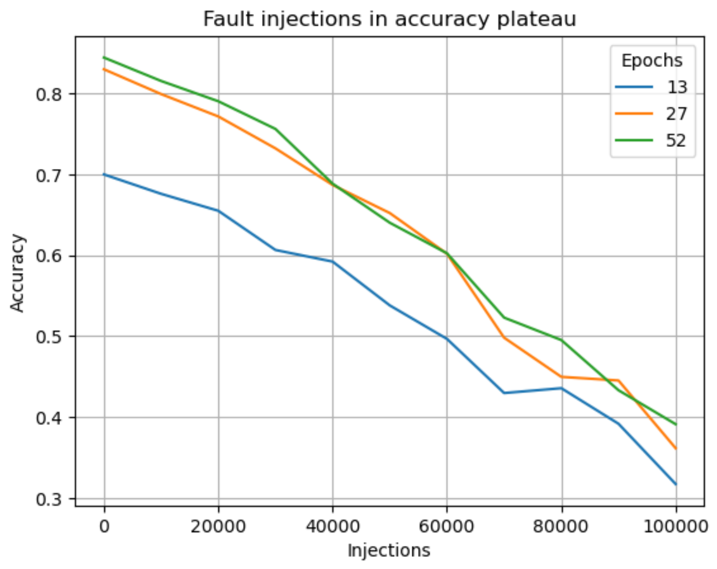


Figure 109. Fault injections in accuracy plateau

As expected the models start from different accuracies, so we overlap the graphs by subtracting the mean:

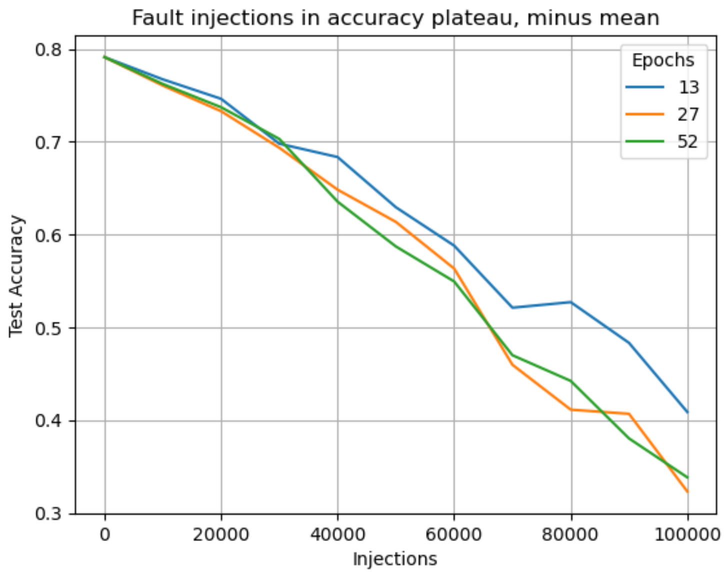


Figure 110. Fault injections in accuracy plateau, minus mean

We can see that the epoch that presents a better reliability is the epoch 13. After calculating the standard deviation and the magnitude for the same epochs we obtain the following results:

- Epoch 13: Standard deviation: 0.009567; Magnitude: 31.7161
- Epoch 27: Standard deviation: 0.007930; Magnitude: 26.2929

- Epoch 52: Standard deviation: 0.007794; Magnitude: 25.8414

As we can see the Standard Deviation and Magnitude actually decreases with the pass of epochs, but the reliability does not necessarily increases as shown in Figure 110, which refutes our supposition.

Secondly we will save all the models for all epochs and then select some epochs to perform the same experiment, so after training the network with SGD we have the following graph:

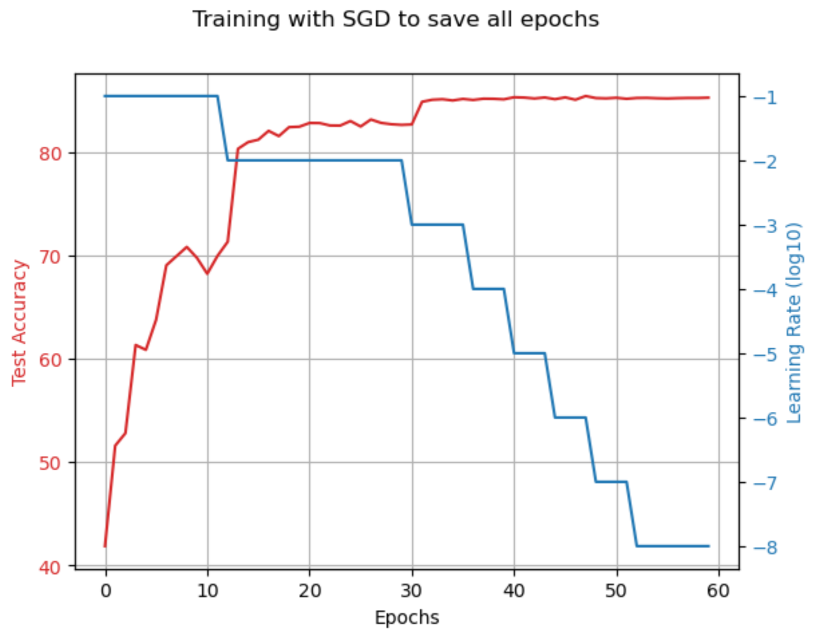


Figure 111. Training with SGD to save all epochs

Now we will select equally spaced epochs to perform the fault injections, so we select epochs 10, 20, 30 and 40. The result of the fault injections and the overlapped graphs is shown below:

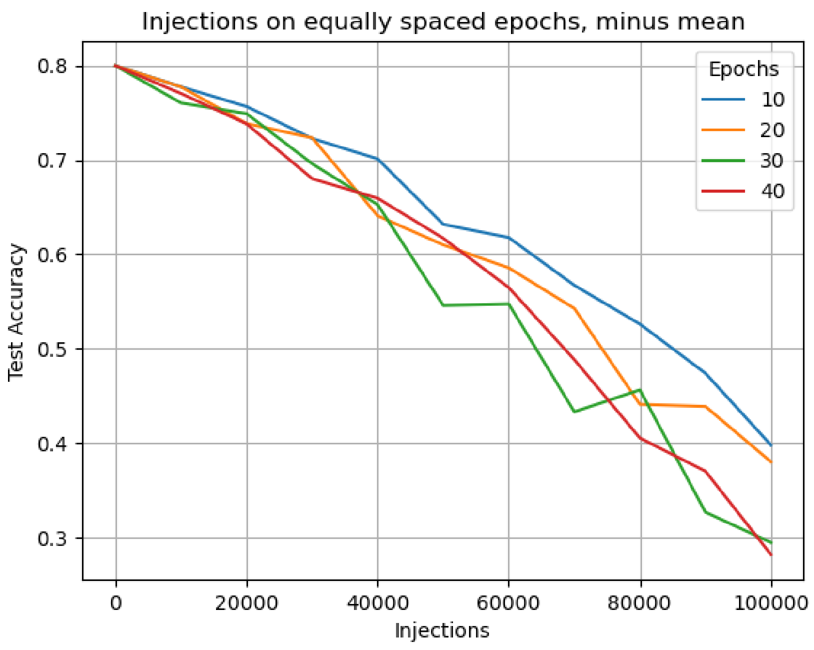


Figure 112. Injections on equally spaced epochs, minus mean

We can observe that the epoch which perform the best in terms of reliability is the epoch 10. Now we check the standard deviation and the magnitude of these epochs:

- Epoch 10: Standard deviation: 0.009700; Magnitude: 32.1561
- Epoch 20: Standard deviation: 0.008584; Magnitude: 28.4568
- Epoch 30: Standard deviation: 0.007788; Magnitude: 25.8198
- Epoch 40: Standard deviation: 0.007659; Magnitude: 25.3913

As we can see the standard deviation and magnitude decrease as the epochs pass, but from the Figure 112 we observe that the epoch 10 has a better reliability and there is not a clear order or pattern of reliability, which refutes again our supposition.

4 Conclusions

A study and analysis of training methods to increase the reliability of neural networks have been discussed in this work. We trained many models with a ResNet18 architecture with different parameter values, then performed fault injection campaigns to see relationships between these parameter values and reliability of the models. We mainly used 4 optimizers for the training of the networks and obtained the following results.

First, for the SGD optimizer, there is a small trend for the batch size and the reliability, bigger batch sizes resulted in a lower reliability, there was a subtle worse performance with batch sizes of 1024 and 2048 compared with those of inferior batch size. However, the batch size was inversely related to the accuracy of the models. The value of weight decay in SGD seemed to not affect the reliability. Also an experiment with bit-flip was done to see if there are bits of the weights (float-64) which are more sensitive to this kind of fault. We found that bits 54 and 62 are very sensitive to bit-flips as they drop the accuracy significantly, indeed a single bitflip on the bit 62 may lead to failures by corrupting the hole inference or crashing the program.

Second, for the Adam optimizer, the reliability and accuracy seemed to be lower on lower batch sizes, batch size of 64 and 128 showed a lower performance compared with bigger batch sizes. Small values of weight decay like 0 or 5e-7 showed to be good for accuracy, but they didn't prove to be better for the reliability.

Third, for the Adagrad optimizer, from the training of the models we could observe that the accuracy decreases as the batch size increase, thus small values of batch size like 64 or 128 are recommended in terms of accuracy. When starting from the same accuracy a batch size of 64 showed better results in reliability. Also, low values of weight decay, specifically lower or equal than 5e-5 resulted in a better reliability and accuracy.

Fourth, for the Rmsprop optimizer, as most of the models with different batch sizes were not able to train, we couldn't apply the fault injections and study the reliability. However, even if low values of weight decay (<5e-5) showed to be better for accuracy, there was not a stable trend for the reliability.

The previous optimizers were then compared between them and we obtained probably the most important results in this work. As we can observe from Figure 104, there are big differences in the reliability when using different optimizers, the best optimizer if we want to achieve a good reliability of the neural network is SGD, then Adagrad, and after a gap, Rmsprop and then Adam. This is a continuous and stable trend and the gaps are significant, so selecting an optimizer should be a priority if we want our application be resilient to faults.

5 References

[1] Stanford, "Neural Networks," 2000. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html.

[2] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *2015 IEEE International Conference on Computer Vision (ICCV),* pp. 1026-1034, 2015.

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang and A. Karpathy, "ImageNet Large Scale Visual Recognition," *International Journal of Computer Vision,* vol. 115, p. 211–252, 2015.

[4] A. N. Seidman., "Neural networks and digital avionics," *In 9th IEEE/AIAA/NASA Conference on Digital Avionics,* 1990.

[5] S. K. a. J. Hwang, "Neural network architectures for robotic applications," *IEEE Transactions on Robotics,* 1989.

[6] A. Luckow, M. Cook, N. Ashcraft, E. Weill, E. Djerekarov and B. Vorster, "Deep learning in the automotive industry: Applications and tools," *IEEE International Conference on Big Data,* 2016.

[7] H. Mechlih and M. M. Mechlih, "Neural network based medical decision making using wearable technology," *2015 12th Learning and Technology Conference,* 2015.

[8] C. Alippi, V. Piuri and M. Sami, "Sensitivity to errors in artificial neural networks: a behavioral approach.," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications,* vol. 42, no. 6, pp. 358-361, 1995.

[9] S. B. a. V. Piuri, "High performance fault-tolerant digital neural networks," *IEEE Transactions on Computers,* vol. 47, pp. 357-363, 1998.

[10] A. Ruospo, L. M. Luza, A. Bosio, L. Dilillo, R. Mariani, M. Traiola and E. Sanchez, "A Survey on Artificial Neural Networks Reliability," *Journal of Systems Architecture: the EUROMICRO Journal,* vol. 104, no. C, p. 28, 2021.

[11] C. Allen and C. Stevens, "An evaluation of causes for unreliability of synaptic transmission.," *Proceedings of the National Academy of Sciences of the United States of America,* vol. 91, 1994.

[12] N. Jwalapuram, "A Gentle Introduction to PyTorch Library for Deep Learning," 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/04/a-gentle-introduction-to-pytorch-library/.

[13] A. Rosebrock, "What is PyTorch?," 2021. [Online]. Available: https://www.pyimagesearch.com/2021/07/05/what-is-pytorch/.

[14] PytorehcFI, "Github," 2020. [Online]. Available: https://github.com/pytorchfi/pytorchfi.

[15] A. Mahmoud, "PyTorchFI - A Runtime Perturbation Tool for DNNs," 2020. [Online]. Available: https://www.youtube.com/watch?v=A2LbJZeC5k8&ab_channel=VarunChandrasekaran.

[16] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio and S. K. S. Hari, "PyTorchFI: A Runtime Perturbation Tool for DNNs," 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), 2020.

[17] X. Z. S. R. J. S. Kaiming He, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), p. 12, 2015.

[18] GeeksForGeeks, "Residual Networks (ResNet) – Deep Learning," [Online]. Available: https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/.

[19] C. Shorten, "Introduction to ResNets," 2019. [Online]. Available: https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4.

[20] K. Almezhghwi and S. Serte, "Improved Classification of White Blood Cells with the Generative Adversarial Network and Deep Convolutional Neural Network," Hindawi, Computational Intelligence and Neuroscience, 2020.

[21] P. Napoletano, F. Piccoli and R. Schettini, " Anomaly detection in nanofibrous materials by CNN-based self-similarity," Sensors, vol. 18, p. 209, 2018.

[22] S. Doshi, "Various Optimization Algorithms For Training Neural Network," [Online]. Available: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6.

[23] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro and B. Recht, "The Marginal Value of Adaptive Gradient Methods," NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems, p. 4151–4161, 2017.

[24] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," Conference paper at ICLR 2015, 2014.

[25] S. Ruder, "An overview of gradient descent optimization algorithms," Arxiv, 2016.

[26] J. Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning," Machine Learning Mastery, 2021. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[27] GeeksForGeeks, "Intuition behind Adagrad Optimizer," [Online]. Available: https://www.geeksforgeeks.org/intuition-behind-adagrad-optimizer/.

[28] Pytorch, "ADAGRAD," 2019. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.Adagrad.html.

[29] J. Brownlee, "Gradient Descent With RMSProp from Scratch," Machine Learning Mastery, 2021. [Online]. Available: https://machinelearningmastery.com/gradient-descent-with-rmsprop-from-scratch/.

[30] G. Hinton, N. Srivastava and K. Swersky, "Lecture 6a, Overview of mini-batch gradient descent," [Online]. Available: https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf.

[31] Machienlearningjourney, "RMSProp," 2021. [Online]. Available: https://machinelearningjourney.com/index.php/2021/01/06/rmsprop/.

[32] F. H. Ilya Loshchilov, "SGDR: Stochastic Gradient Descent with Warm Restarts," ICLR 2017 conference paper, 2016.

[33] Pytorch, "COSINEANNEALINGLR," 2019. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR.html#torch.optim.lr_scheduler.CosineAnnealingLR.

[34] Pytorch, "REDUCE LR ONPLATEAU," 2019. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html#torch.optim.lr_scheduler.ReduceLROnPlateau.

[35] K. E. Koech, "Cross-Entropy Loss Function," Towards Data Science, [Online]. Available: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e.

[36] J. Brownlee, "How to Control the Stability of Training Neural Networks With the Batch Size," Machine Learning Mastery, 2020. [Online]. Available:

https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/.

[37] D. Mishra, "Weight Decay == L2 Regularization?," 2020. [Online]. Available: https://towardsdatascience.com/weight-decay-l2-regularization-90a9e17713cd.

[38] A. Courville, I. Goodfellow and Y. B. a. Aaron, "Deep Learning," Genetic Programming and Evolvable Machines, vol. 19, p. 305–307, 2016.

[39] P. Eckersley and Y. Nasser, "AI Progress Measurement," Electronic Frontier Foundation, 2017.

[40] www.kaggle.com, " Popular Datasets Over Time | Kaggle," 2017. [Online]. Available: https://www.kaggle.com/benhamner/popular-datasets-over-time/code.

[41] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.

[42] A. Krizhevsky, "The CIFAR-10 dataset," [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[43] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic.," IEEE Standards, 2019.

[44] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[45] L. Kuang, "Pytorch CIFAR," 2017. [Online]. Available: https://github.com/kuangliu/pytorch-cifar.

[46] D. Mishkin, N. Sergievskiy and J. Matas, "Systematic evaluation of CNN advances on the ImageNet," Computer Vision and Image Understanding, vol. 161, pp. 11-19, 2016.

[47] Keras, "Adagrad," 2021. [Online]. Available: https://keras.io/api/optimizers/adagrad/.

[48] D. Blalock, J. J. G. Ortiz, J. Frankle and J. Guttag, "What is the State of Neural Network Pruning?," Published in Proceedings of Machine Learning and Systems 2020 (MLSys 2020), 2020.

[49] M. Li, Y. Sattar, C. Thrampoulidis and S. Oymak, "Exploring Weight Importance and Hessian Bias in Model Pruning," Arxiv, p. 28, 2020.