LeNet-5 Model Optimizers on MNIST Dataset

Sanchit Thakur
st976

Abstract

LeNet-5 is a pioneering convolutional neural network (CNN) model that revolutionized the field of deep learning and played a significant role in the advancement of image classification tasks. This model was specifically designed for handwritten digit recognition, serving as a breakthrough in computer vision research.LeNet-5's impact extends beyond handwritten digit recognition, as its architecture and principles have influenced the development of subsequent CNN models for various computer vision applications. Its success highlights the effectiveness of deep learning approaches in extracting meaningful features from visual data, paving the way for further advancements in image classification and related fields.
The topic chosen for this project is Topic 1. The aim of this project is to build a LeNet-5 model to be used on the MNIST dataset. In addition, different optimizers are used in the LeNet-5 model, and their performances are compared and evaluated. The different optimizers used are: SGD, AdaGrad, RMSprop. A detailed analysis of the model is provided in the report.

1 Introduction

LeNet-5 is a convolutional neural network (CNN) model that was developed by Yann LeCun and his colleagues in 1998. It was one of the pioneering models in the field of deep learning and played a crucial role in popularizing CNNs for image classification tasks. LeNet-5 was one of the earliest convolutional neural networks and promoted the development of deep learning. Since 1988, after years of research and many successful iterations, the pioneering work has been named LeNet-5.

The LeNet-5 model, despite being originally designed for handwritten digit recognition, has found applications in various computer vision tasks. Some of the notable applications of the LeNet-5 model are:

- Handwritten Character Recognition: LeNet-5's primary application is in recognizing and classifying handwritten characters, such as digits. Its architecture and training techniques make it effective for tasks like reading postal codes, recognizing checks, and automated form processing.
- Optical Character Recognition (OCR): LeNet-5 can be used in OCR systems to extract text from images or scanned documents. By training on large datasets of characters, LeNet-5 can accurately recognize printed or handwritten text in different fonts and styles.
- License Plate Recognition: The LeNet-5 model can be employed in license plate recognition systems, which are widely used in traffic monitoring, parking management, and law enforcement. It can help extract alphanumeric characters from license plate images and enable vehicle identification.
- Medical Image Analysis: LeNet-5 has been utilized in medical imaging applications, such as the classification of X-ray images, mammograms, and histopathological images. It aids in tasks like identifying abnormalities, detecting diseases, and assisting in medical diagnosis.
- Object Recognition: LeNet-5's convolutional architecture makes it suitable for object recognition tasks. It can classify objects in images, enabling applications like object detection, surveillance systems, and autonomous vehicles.
- Facial Recognition: LeNet-5 can be adapted for facial recognition tasks, where it can learn

to identify individuals from images or video streams. It has been employed in applications like biometric authentication, surveillance, and access control systems.

- Document Classification: LeNet-5's ability to recognize patterns and classify inputs makes it useful for document classification tasks. It can automatically categorize documents based on their content, enabling tasks like spam filtering, sentiment analysis, and topic classification.

- Gesture Recognition: LeNet-5 has been applied to gesture recognition tasks, where it can recognize hand movements or poses and associate them with specific actions. This is relevant in applications such as sign language translation, virtual reality, and human-computer interaction.

Overall, the versatility of LeNet-5's architecture and its ability to learn discriminative features make it applicable to a wide range of computer vision tasks, beyond its original purpose of handwritten digit recognition.

2 Related Work

Previous research has extensively investigated the performance of different optimizers in training machine learning and deep learning models. For instance, Wilson et al. (citation [1]) conducted a study involving a binary classification model to compare the generalization abilities of adaptive and non-adaptive optimization techniques. They observed that non-adaptive gradient descent methods achieved perfect test-error results in the binary classification task, while the adaptive methods converged to an incorrect solution with a 0.5 probability of accurate classification. Another related study by Soydaner (citation [2]) explored various adaptations of the adaptive gradient method, such as AdaGrad, AdaDelta, Adamax, Adam, Nadam, RMSprop, and AMSgrad. These adaptations were implemented and evaluated on four datasets, including MNIST, CIFAR-10, Kaggle Flowers, and Labeled Faces in the Wild (LFW), covering both supervised and unsupervised learning tasks.

3 Dataset

The dataset being used for this project is the MNIST Dataset. The MNIST dataset is a widely used benchmark dataset in the field of machine learning and computer vision. It stands for the Modified National Institute of Standards and Technology database. The dataset is composed of a collection of 70,000 grayscale images of handwritten digits, each measuring 28x28 pixels.

The MNIST dataset is split into two main parts: a training set and a test set. The training set consists of 60,000 images, while the test set contains 10,000 images. This division allows researchers and practitioners to evaluate and compare the performance of machine learning models on unseen data.

Each image in the MNIST dataset corresponds to a single handwritten digit from 0 to 9. The digits are centered within the image and normalized to have a fixed size. The grayscale pixel values range from 0 to 255, where 0 represents white and 255 represents black.

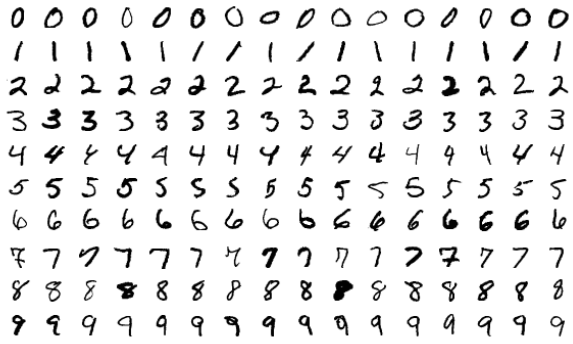

Figure 1: The MNIST dataset

The MNIST dataset is commonly used for image classification tasks, particularly for evaluating the performance of algorithms and models on handwritten digit recognition. It has served as a standard benchmark for testing and comparing the accuracy of various machine learning and deep learning models.

Due to its simplicity and accessibility, the MNIST dataset has played a crucial role in the development and evaluation of many fundamental algorithms and architectures in the field of computer vision, including the LeNet-5 model. However, it is important to note that the MNIST dataset is relatively small and simplistic compared to real-world scenarios, and models achieving high accuracy on MNIST may not necessarily

generalize well to more complex tasks or datasets.

4 Method Description

For this project, the LeNet-5 model was built according to the description in the next section. The MNIST dataset was loaded and split into test and train data, followed by the preprocessing: normalization, one-hot encoding. Five different optimizers were used in the model, and the performance of each was measured on the basis of the testing accuracy. These optimizers are:

4.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used in machine learning and deep learning to train models. It is particularly effective when dealing with large datasets or complex models due to its efficiency and ability to handle noisy or non-convex objective functions. The basic idea behind SGD is to update the model's parameters iteratively by computing and applying gradients on a subset of training examples (known as mini-batches) rather than the entire dataset. This approach introduces randomness into the optimization process. The steps involved in the SGD optimization algorithm are as follows:

- Initialization: The model's parameters, such as weights and biases, are initialized randomly or with predetermined values.
- Mini-Batch Selection: From the training dataset, a mini-batch of examples is randomly sampled without replacement. The size of the mini-batch is typically chosen to balance computational efficiency and model convergence.
- Gradient Computation: For each example in the mini-batch, the gradient of the loss function with respect to the model's parameters is computed. This gradient represents the direction and magnitude of the steepest descent for the current mini-batch.
- Parameter Update: The model's parameters are updated using the computed gradients. The update rule is generally defined as: parameter = parameter - learning_rate * gradient, where the learning_rate is a hyperparameter that determines the step size of the update.
- Iteration: Steps 2-4 are repeated for a fixed number of iterations (epochs) or until convergence criteria are met. In each iteration, a new mini-batch is sampled, gradients are computed, and parameters are updated.

4.2 AdaGrad

AdaGrad (Adaptive Gradient) is an optimization algorithm used in machine learning and deep learning to adjust the learning rate for each parameter during training adaptively. It was introduced by Duchi et al. in 2011 as a way to address the challenges of choosing an appropriate learning rate for different parameters in stochastic gradient descent (SGD).

The key idea behind AdaGrad is to perform larger updates for infrequent and sparse features and smaller updates for frequent features. It achieves this by scaling the learning rate individually for each parameter based on the historical squared gradients accumulated over time. The steps involved in the AdaGrad optimization algorithm are as follows:

- Initialization: Initialize the model's parameters, such as weights and biases, and initialize an accumulator variable for each parameter. The accumulators are initialized to zero.
- Mini-Batch Gradient Computation: Select a mini-batch of training examples and compute the gradients of the loss function with respect to the model's parameters.
- Accumulate Squared Gradients: For each parameter, add the squared value of the gradient to its corresponding accumulator. This accumulator keeps track of the sum of squared gradients for each parameter throughout training.
- Update Parameters: For each parameter, compute the scaled gradient update using the accumulated squared gradients. The update rule is as follows: parameter = parameter - (learning_rate / sqrt(accumulator + epsilon)) * gradient, where learning_rate is the initial learning rate, the accumulator is the sum of squared gradients, and epsilon is a small constant (e.g., 1e-8) added for numerical stability to avoid division by zero.
- Iteration: Continue iterating over mini-batches, accumulating gradients, and updating parameters until convergence or a predetermined number of iterations.

4.3 Adam

learning and deep learning. It combines the concepts of both momentum optimization and RMSProp to provide adaptive learning rates and faster convergence.

The key idea behind Adam is to maintain a per-parameter learning rate that adapts based on the estimates of both the first moment (the mean) and the second moment (the uncentered variance) of the gradients. This enables Adam to automatically adjust the learning rate for each parameter and perform effective optimization in different directions of the parameter space. the Adam optimizer combines the concepts of momentum optimization and RMSProp to adaptively adjust the learning rate for each parameter during training. The Adam optimizer works in the following way:

- Initialize parameters and accumulators for first and second moment estimates.
- Compute gradients for a mini-batch of training examples.
- Update first moment estimate and second moment estimate using exponential moving averages.
- Perform bias correction to account for initial biases in the moment estimates.
- Update parameters using the bias-corrected moment estimates and a scaled learning rate.
- Repeat steps 2-5 for multiple mini-batches until convergence or a fixed number of iterations.

By estimating the first and second moments of the gradients, Adam adapts the learning rate for each parameter based on the gradient magnitudes and directions. This allows for efficient optimization with automatic adaptation to different parameter spaces.

4.4 RMSProp

RMSProp (Root Mean Square Propagation) is an optimization algorithm used in machine learning and deep learning. It is an extension of the AdaGrad optimizer and aims to address some of its limitations by introducing a decay factor to mitigate the diminishing learning rates over time.

The main idea behind RMSProp is to adaptively adjust the learning rate for each parameter based on the magnitudes of recent gradients. It achieves this by maintaining a moving average of the squared gradients for each parameter. This moving average acts as an estimate of the second moment of the gradients and helps in scaling the learning rates. The steps involved in the RMSProp optimization algorithm are as follows:

- Initialization: Initialize the model's parameters, such as weights and biases, and initialize an accumulator variable for each parameter. The accumulators are initialized to zero.
- Mini-Batch Gradient Computation: Select a mini-batch of training examples and compute the gradients of the loss function with respect to the model's parameters.
- Update Accumulators: For each parameter, compute the squared value of the gradient and update the accumulator using an exponential moving average. The update rule is as follows: accumulator = decay_rate * accumulator + (1 - decay_rate) * gradient$\hat{2}$, where decay_rate is a hyperparameter between 0 and 1 that controls the decay rate of the moving average.
- UUpdate Parameters: For each parameter, compute the scaled gradient update using the accumulated squared gradients. The update rule is as follows: parameter = parameter - (learning_rate / sqrt(accumulator + epsilon)) * gradient, where learning_rate is the initial learning rate, accumulator is the moving average of squared gradients, and epsilon is a small constant (e.g., 1e-8) added for numerical stability to avoid division by zero.
- Iteration: Continue iterating over mini-batches, updating the accumulators and parameters until convergence or a predetermined number of iterations.

4.5 AdaDelta

The AdaDelta optimizer is an adaptive learning rate optimization algorithm that aims to address the limitations of other methods, such as the need for manual learning rate tuning and the sensitivity to noisy or sparse gradients. It achieves this by dynamically adapting the learning rate based on the accumulated past gradients and parameter updates.

The key idea behind AdaDelta is that it eliminates

the need for a fixed learning rate by using accumulated past gradients and parameter updates to determine the effective learning rate. This allows the optimizer to automatically adapt the learning rate to different parameter updates and helps in stabilizing the optimization process. Additionally, AdaDelta performs well in situations where there are sparse or noisy gradients, as it adjusts the learning rate based on the historical information accumulated during training.

AdaDelta dynamically adapts the learning rate based on the accumulated squared gradients and squared parameter updates. It addresses the need for manual learning rate tuning and helps handle noisy or sparse gradients effectively. By using the ratio of the RMS values, it allows for adaptive learning rates at different stages of training and improves optimization stability.

5 Model Description

The LeNet-5 architecture consists of a series of convolutional and pooling layers, followed by fully connected layers. The input to LeNet-5 is a grayscale image of size 32x32 pixels. The convolutional layers utilize small filters to extract local features from the input image, while the pooling layers reduce spatial dimensions and capture translational invariance.

With its six filters in the first convolutional layer and 16 filters in the second, LeNet-5 progressively learns hierarchical representations of the input image. The model then applies average pooling after each convolutional layer, further reducing the size of feature maps. The fully connected layers, comprising 120 and 84 neurons, respectively, leverage the extracted features to make high-level predictions.

The output layer of LeNet-5 consists of 10 neurons, representing the possible classes in the dataset, and employs the softmax activation function to generate class probabilities. During training, the model optimizes its parameters using gradient-based techniques such as backpropagation and stochastic gradient descent.

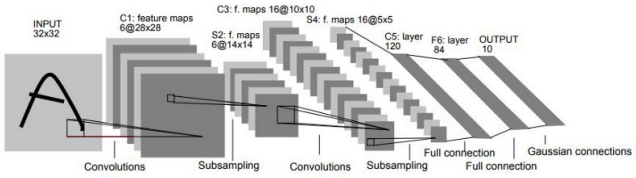

Figure 2: Architecture of the LeNet-5 model

Here is a brief overview of the Lenet-5 architecture:

- Input Layer: The LeNet-5 model takes grayscale images of size 32x32 as input.
- Convolutional Layer 1: The first convolutional layer has 6 filters of size 5x5. Each filter convolves over the input image to produce feature maps.
- Average Pooling Layer 1: After each convolutional layer, LeNet-5 applies average pooling with a 2x2 window and a stride of 2. This layer reduces the spatial dimensions of the feature maps and helps in capturing translational invariance.
- Convolutional Layer 2: The second convolutional layer has 16 filters of size 5x5. It takes the pooled feature maps from the previous layer as input.
- Average Pooling Layer 2: Similar to the first pooling layer, the second pooling layer performs average pooling with a 2x2 window and a stride of 2.
- Fully Connected Layer 1: This layer has 120 neurons and is fully connected to the output of the second pooling layer. It applies the rectified linear unit (ReLU) activation function.
- Fully Connected Layer 2: The second fully connected layer consists of 84 neurons and also applies the ReLU activation function.
- Output Layer: The final fully connected layer consists of 10 neurons, corresponding to the 10 possible classes in the dataset (e.g., digits from 0 to 9 in the original LeNet-5). It employs the softmax activation function to produce the class probabilities.

6 Results

Once the model was built, it was run on different optimizers, as explained above. The model was trained over 50 epochs, and then it was tested using the test data. The testing accuracy obtained for each optimizer is shown below:

Optimizer	Test Accuracy
Adadelta	0.85
SGD	0.858
Adagrad	0.94
RMSProp	0.98
Adam	0.99

Table 1: Test Accuracy rates for different optimizers

A graph was then plotted to compare the values of the test accuracy rates for each optimizer, which is shown below.

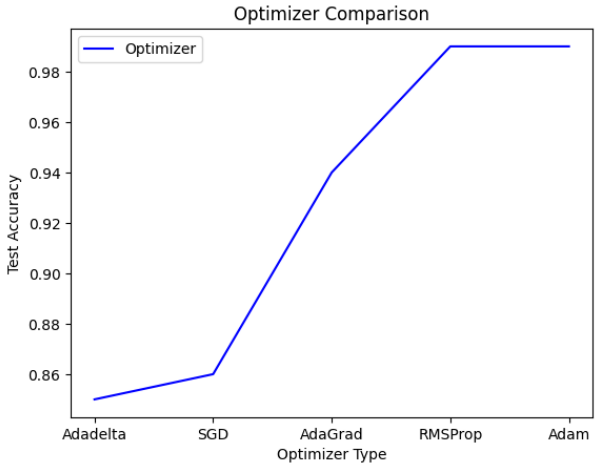

Figure 3: Comparison of test accuracy rate for different optimizers

It was observed that the SGD optimizer had a higher accuracy than Adadelta. Since the MNIST dataset is relatively small, consisting of 60,000 training images and 10,000 test images of handwritten digits, the SGD optimizer, being a simple optimization algorithm, can work well on smaller datasets as it quickly updates the model parameters based on individual or mini-batches of examples. AdaDelta, with its adaptive learning rate scheme and accumulated gradients, may not provide a significant advantage in such cases. Also, MNIST has a relatively smooth optimization landscape without many local optima, which can benefit SGD. AdaDelta's adaptive learning rate might be more advantageous in complex optimization landscapes with irregular surfaces and many local optima.

As for why Adagrad performed better than SGD, MNIST is a relatively simple and well-behaved dataset, but it can still have sparse gradients for certain features or examples. AdaGrad handles sparse gradients effectively by maintaining separate learning rates for each parameter based on their accumulated historical gradients. This adaptability to sparse gradients can contribute to better performance compared to SGD. AdaGrad adapts the learning rate for each parameter based on the historical gradients. It assigns larger learning rates to infrequent and important features, which can be beneficial for datasets like MNIST where certain features may carry more discriminative information. This adaptive learning rate can enable AdaGrad to converge faster and achieve better results compared to a fixed learning rate used in SGD.

RMSprop incorporates a momentum term that allows it to have a smoother and more controlled update process compared to AdaGrad. This momentum helps to accelerate the optimization process and overcome potential local minima. The combination of adaptive learning rates and momentum in RMSprop may have allowed it to converge faster and achieve better results compared to AdaGrad. While both AdaGrad and RMSprop adapt the learning rates, RMSprop introduces an additional refinement by incorporating an exponentially weighted moving average of the squared gradients. This refinement helps to balance the learning rates across different parameters and iterations. In the case of the MNIST dataset, this refined adaptation may have provided better updates and improved performance compared to the more straightforward adaptation in AdaGrad. The MNIST dataset is relatively small and well-behaved, which can benefit from adaptive learning rates and refined adaptation techniques. The combination of these factors in RMSprop may have allowed it to better navigate the optimization landscape and find better solutions compared to AdaGrad.

The Adam optimizer gave the best performance at a test accuracy of 99%. This could be because Adam combines the benefits of adaptive

learning rates from optimizers like AdaGrad and RMSprop. It dynamically adjusts the learning rate for each parameter based on estimates of both the first-order (gradient) and second-order (gradient squared) moments. This adaptability allows Adam to automatically tune the learning rate for different parameters, which can be advantageous for complex datasets like MNIST. Adam is known for being relatively robust to the choice of initial parameter values. It can handle a wide range of initializations, making it less sensitive to the selection of initial conditions. This robustness can be beneficial for the LeNet model on MNIST dataset, as it allows Adam to converge to good solutions even with suboptimal initializations.

7 Conclusion

In this project, a LeNet-5 model was implemented on the MNIST dataset. The model was also trained using various optimizers like SGD, Adagrad, RMSProp, Adam and Adadelta. Each of these optimizers was evaluated for their performance using the test accuracy rate obtained during the testing phase. It was found that the Adam optimizer performed the best out of all the other optimizers, producing a test accuracy rate of 99%.

In conclusion, the Lenet model has proven to be a highly effective solution for the MNIST dataset classification task. Through its unique architecture, including convolutional and pooling layers, Lenet successfully extracts meaningful features from the input images and achieves excellent accuracy in digit recognition. By leveraging the power of convolutional neural networks (CNNs), Lenet was able to learn hierarchical representations, enabling it to discern complex features and classify digits accurately. Furthermore, the Lenet model's relatively small number of parameters and efficient architecture make it computationally efficient, allowing for faster training and inference times. This advantage makes it an attractive choice for real-time applications where speed is crucial. Therefore, the Lenet model has proven to be a robust and accurate solution for the MNIST dataset classification task. Its ability to extract and leverage meaningful features from images, combined with its computational efficiency, makes it a valuable tool for various image recognition tasks. It is believed that with further advancements in deep learning, convolutional neural networks will undoubtedly continue to build upon the success of models like Lenet, leading to even more remarkable achievements in computer vision applications.

References

1. Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. 2018. The marginal value of adaptive gradient methods in machine learning. (May 2018). Retrieved May 10, 2023 from https://arxiv.org/abs/1705.08292

2. Derya Soydaner. 2020. A comparison of optimization algorithms for Deep Learning. (July 2020). Retrieved May 10, 2023 from https://arxiv.org/abs/2007.14166

3. Shipra Saxena. 2021. The architecture of lenet-5. (March 2021). Retrieved May 10, 2023 from https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/

4. Anon. 2023. Lenet. (May 2023). Retrieved May 10, 2023 from https://en.wikipedia.org/wiki/LeNet

5. Towards AI Team. 2023. The architecture and implementation of lenet-5. (January 2023). Retrieved May 10, 2023 from https://towardsai.net/p/deep-learning/the-architecture-and-implementation-of-lenet-5

6. Sanket Doshi. 2020. Various optimization algorithms for training neural network. (August 2020). Retrieved May 10, 2023 from https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

7. Keras Team. Keras documentation: Optimizers. Retrieved May 10, 2023 from https://keras.io/api/optimizers/