LeNet

Angela Landry
angelalandry@utexas.edu

Alex Cabrera
gac2827@my.utexas.edu

Abstract

This paper presents the results of five experiments conducted on a simple implementation of LeNet, a convolutional neural network architecture. Experiment 1 involved visualizing the kernel weights of each convolutional layer to understand the patterns the model was detecting. Experiment 2 tested the tolerance of the model to different nonlinearities and learning rates. Experiment 3 tested various combinations of parameters for the Maxpool layers to detect spatial patterns of different sizes and complexities. Experiment 4 explored the effects of adding or removing convolutional layers to improve the model's accuracy. Finally, Experiment 5 investigated the impact of including RNN layers in a classification model that does not have an intrinsic ordering. For the most part, the original implementation was efficient and accurate. We discovered some weaknesses in the model, particularly relating to nonlinearities that we found an improvement for.

1 Introduction

LeNet is one of the earliest successful convolutional models, which served as the basis for more complex work such as AlexNet and ResNet. For this project, we decided to trace back the steps modern research on convolutional layers, and derive performance improvements from the original model. This might make it easier to see the benefits of each technique we employ here.

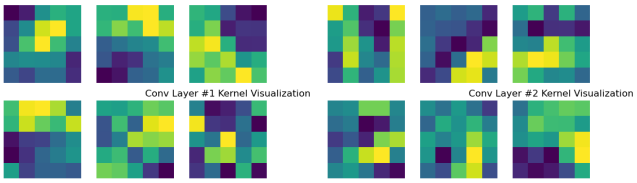
2 Experiments

2.1 Weight Visualization

Justification: We sought to understand different patterns the original model was detecting on its input. In particular, we wanted to see how the model chose to filter information on its different kernels on each convolutional layer. Thus, we visualized a subset of the model's kernels through a heat map for each value.

Set Up: After training the model, we detached the kernels from each convolutional layer and used the package matplotlib to correctly plot them in a 2D space.

Results:



(a) 1st Layer (b) 2nd Layer

Figure 1: Visualization for Different Kernels

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

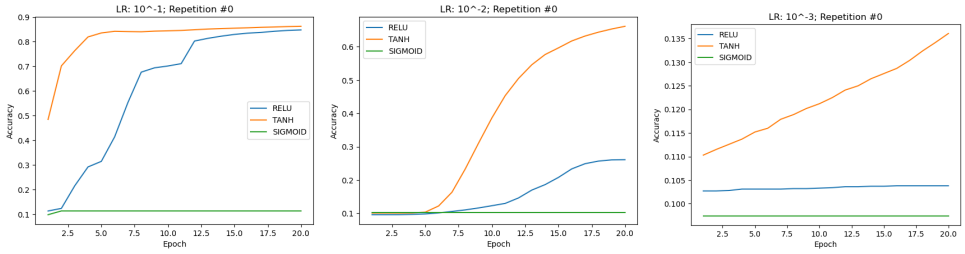
Visualizing different kernel weights did not, in most cases, reveal immediately identifiable, or meaningful, patters. However, it is possible to make an educated guess as to the function of a few of the kernels we analyzed. For example, kernel #2 in the second convolutional layer appears to be filtering out everything higher than the positive diagonal of the image, while kernel #3 in the first convolutional layer appears to filter out information in the top right corner, while detecting what is present in its bottom left corner.

2.2 Nonlinearities

Justification: As we were able to see in the various experiments we ran in class, the choice of what nonlinearity to use for a given model can significantly impact the model's ability to converge to a proper accuracy, and may force one to use specific parameters during training to avoid this from happening. Thus, for this experiment we decided to test changing the original model's ReLU activation function to other popular alternatives such as Tanh and Sigmoid, and check how these choices affect the model's ability to learn. In addition, we tested using different learning rates to avoid misrepresenting any nonlinearity which may work more properly under certain conditions.

Set Up: We replaced all nonlinearities in the model with our proposed alternatives, leaving the rest of the layers untouched. And as mentioned, we tested each of these model variations with different learning rates.

Results: We were able to arrive at different conclusions for each nonlinearity we tested.



(a) lr=1e-1 (b) lr=1e-2 (c) lr=1e-3

Figure 2: Accuracy vs Epoch for Nonlinearity Models (Only 1 iteration shown)

- Tanh nonlinearities appear to consistently reach a high accuracy in a shorter amount of time than either of the other two being tested, and doing so more consistently across the different learning rates we tested.
- Relu nonlinearities (from the original model) appear perfectly capable of reaching high accuracy levels with moderate speed (as measured in number of epochs). However, they also appear susceptible to vanishing gradients and thus getting stuck at low learning rates on subsequent iterations.
- Sigmoids show a very slow, almost non-existent, increase in accuracy across epochs for whichever learning rate is picked. We note that there may be a training parameter we did not consider for our experiments which might be affecting the performance of this model variation.
- Performance results appear random and inconclusive with a very low learning rate, as on each repetition that was ran the results appeared wildly random and different from one another.

2.3 Maxpool Tweaks

Justification: The Maxpool layer can detect spatial patterns of different sizes and complexities depending on the parameters chosen. We chose to focus on the kernel size, stride, and padding. We sought to find a combination of parameters that resulted in a higher accuracy than the original model.

Set Up: We changed the model to accept kernel size, stride, and padding parameters. We compared six different combinations of parameters against the original.

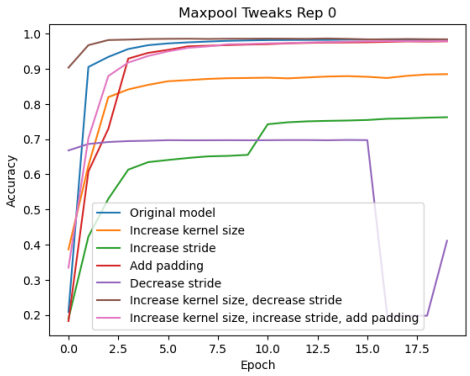
Results: From the figure below we observe that none of the model consistently performed well across all iterations. The original LeNet model performed very well on the first and third iterations. It's poor performance in the second iteration was likely due to the nonlinearity rather than the Maxpool parameters. Other parameter combinations that performed well for two of the three iterations were:

- Adding padding: This is intuitive because it can prevent the output of the Maxpool layer from being smaller than the input. It starts with a relatively low accuracy because there are more weights to learn. From our experiment, adding padding appears to have similar but slightly worse performance than the original model.

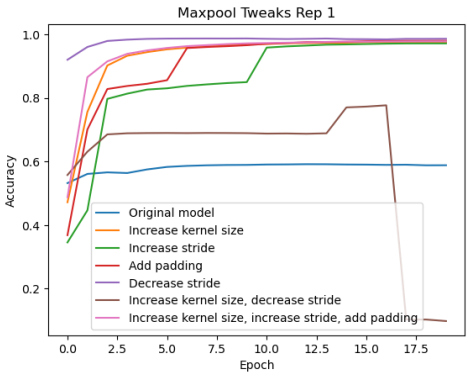
- Decreasing stride: For the two iterations where decreasing stride was successful, it had among the best performance. It started with a very high accuracy and converged quickly. Decreasing the stride increases the complexity of the model because it must perform more operations. However it allows the model to detect more detailed spatial patterns.

- Increasing kernel size and decreasing stride: By increasing kernel size and decreasing stride together, the model is able to detect larger and more detailed spatial patterns. Since this model's performance is very similar to only decreasing the stride, it appears that this added complexity is not needed to obtain good results.

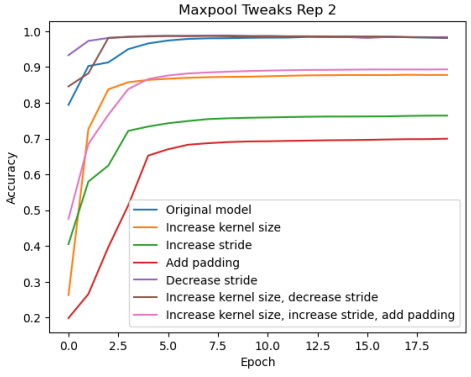
- Increasing kernel size, increasing stride, and adding padding: This choice of parameters allows the model to detect larger and more detailed spatial patterns without shrinking the output of the Maxpool layer. The added complexity seems to do more harm than good as the model takes longer to converge than previous two options.



(a) Iteration 1



(b) Iteration 2



(c) Iteration 3

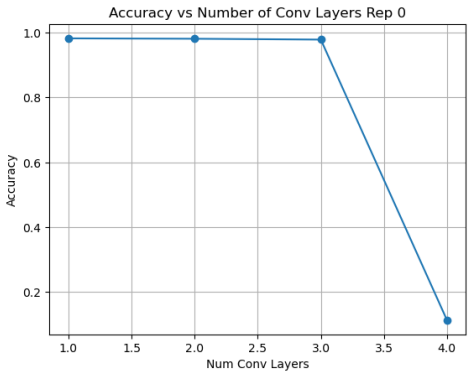
Figure 3: Accuracy vs Epoch for Various Maxpool Parameters

2.4 Deepen the Model

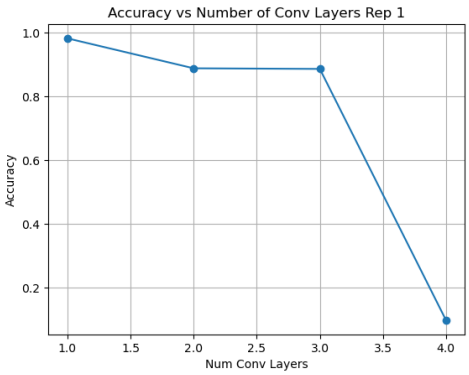
Justification: We sought to test the importance of depth on the model on its accuracy. We would expect a model with more layers to be at least as accurate as its counterpart with fewer layers. However we would also expect a deeper model to take longer to train, so we decided to also consider run time in deciding whether it made sense to add additional convolutional layers.

Set Up: The LeNet implementation we used as our base model had two convolutional layers, so we decided to test models with one, two, three, and four convolutional layers. Because the size of the output shrank after each Maxpool layer, we had to adjust the parameters of the Maxpool layers for the models with three and four convolutional layers. This ensured that the outputs were large enough for each subsequent convolutional layer.

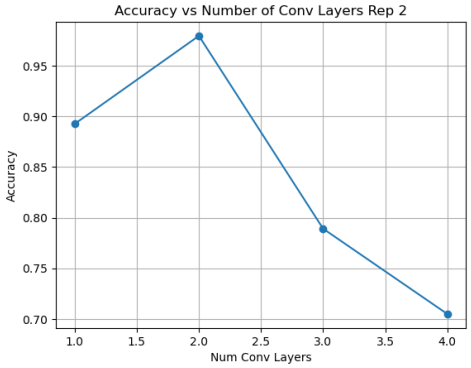
Results: From the figure below we see that the models with 1, and 2 convolutional layers consistently had high accuracies while the model with 4 convolutional layers performed very poorly. The model with 3 convolutional layers had inconsistent results.



(a) Iteration 1



(b) Iteration 2



(c) Iteration 3

Figure 4: Accuracy vs Epoch for Various Maxpool Parameters

Table 1: Model Execution Time

Number of Convolutional Layers	Training Time (sec)
1	128
2	158
3	593
4	735

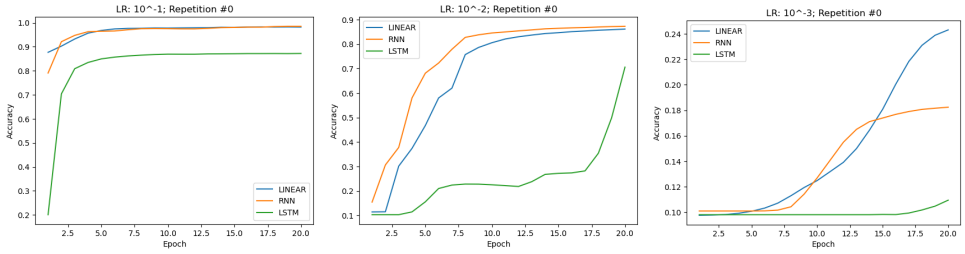
Considering both the accuracies and run time, the model with two convolutional layers is the most obvious choice. The model with only one convolutional layer also performs relatively well and the time is slightly faster however we should prioritize an improvement in accuracy over a slight improvement in training time. We would not want to add convolutional layers to the model because it took many times longer to train and the accuracy was similar or worse. Since a deeper model should be able to have at least equal performance as an equivalent model with fewer layers, this decrease in accuracy can be attributed to vanishing gradients. Thus we were not able to find an improvement over the initial model by adding convolutional layers.

2.5 Recursion

Justification: RNNs were created and are mostly used for data with an intrinsic ordering. In this experiment, we sought to measure the effect of using this type of layer with data that has an arbitrary order with no meaning in between data points. We initially hypothesized that this change would negatively impact the model's performance and accuracy due to the recursive layer accepting useless information after each iteration. We also test using an LSTM to see how the "forget" gate affected performance, and maybe even improve the model's accuracy.

Set Up: We replaced one of the model's linear layers with pytorch's default RNN and LSTM implementations, and graphed the model's accuracy after each epoch with different learning rates.

Results:



(a) lr=1e-1 (b) lr=1e-2 (c) lr=1e-3

Figure 5: Accuracy vs Epoch for Recursion Models (Only 1 iteration shown)

- At a high learning rate (lr=1e-1), using the model with an RNN does not appear to significantly impact either performance or accuracy in all the runs we tested. However, using the model with an LSTM does appear to occasionally degrade the accuracy that the model can achieve, with two out of three runs converging to a lower accuracy than both the RNN and original Linear model.
- Partially lowering the learning rate (to lr=1e-2) does not seem to greatly affect the original model compared to the RNN model. Both seem to maintain somewhat close performance with the third run being an outlier for the RNN model. However, it is clear that the LSTM model suffers from performance issues, with the model growing its accuracy slower than both RNN and the original.
- Forcing a really low learning rate (lr=1e-3) again maintains the performance similarities of the Linear and RNN models. However, with such a low learning rate the LSTM model struggles to increase its accuracy after all iterations, seemingly stuck converging to a low initial accuracy.

3 Acknowledgements:

Credit to GitHub user ChawDoe for the pytorch LeNet implementation that was used as the base for this project.

4 Contributions:

- Angela Landry
  - Code for experiment 1, 3, 4
  - Presentation for experiments 3, 4 and introduction
  - Write-up for experiments 3, 4
  - Abstract + Image formatting
- Alex Cabrera
  - Code for experiments 2, 5
  - Presentation for experiments 1, 2, 5
  - Write-up for experiments 1, 2, 5
  - Introduction + Image formatting

Model Name: LeNet
Description:
LeNet is a convolutional neural network (CNN) that has been designed to classify images. The network consists of two main parts: an input layer and an output layer. The input layer is responsible for processing the image data and extracting features using convolutional and pooling layers. The output layer is responsible for performing the classification task using fully connected layers.
Repo: https://github.com/ChawDoe/LeNet5-MNIST-PyTorch
Model Architecture:

- Input Layer:
    - Conv2d layer with 1 input channel and 6 output channels
    - ReLU activation function
    - MaxPool2d layer with kernel size 2
    - Conv2d layer with 6 input channels and 16 output channels
    - ReLU activation function
    - MaxPool2d layer with kernel size 2
- Output Layer:
    - Fully connected layer with 256 input features and 120 output features
    - ReLU activation function
    - Fully connected layer with 120 input features and 84 output features
    - ReLU activation function
    - Fully connected layer with 84 input features and 10 output features
    - ReLU activation function

Training:

- Loss function: Cross Entropy Loss
- Optimizer: Stochastic Gradient Descent (SGD) with learning rate 0.01
- Epochs: 20
- Batch Size: 256

Performance:

- Accuracy: 99%

Resources:

- PyTorch library was used to implement the model
- The MNIST dataset was used for training and testing the model

Dataset Name: MNIST

Description:

The MNIST dataset is a collection of 70,000 images of handwritten digits (0-9) that are commonly used for training and testing machine learning models. Each image is a 28x28 pixel grayscale image and is labeled with its corresponding digit.

Dataset Composition:

- Training Set: 60,000 images
- Test Set: 10,000 images

Data Format:

- Image data: 28x28 pixel grayscale images
- Image labels: integers (0-9) representing the digit in the image

Data Source:

The MNIST dataset is publicly available and can be downloaded from the following website: http://yann.lecun.com/exdb/mnist/

Potential Use Cases:

- Image Classification
- Deep Learning Model Training
- Computer Vision Research

Limitations:

- MNIST is a relatively small dataset compared to more modern datasets like CIFAR-10 and ImageNet
- The images in MNIST are centered and well-aligned, which may not be representative of real-world images.

License:

The MNIST dataset is publicly available and can be used for any purpose, including commercial purposes.