Project Topic:

Distributed Learning

Team Member Information

Anzhe Cheng (anzheche@usc.edu)

Zhenkun Wang (zhenkunw@usc.edu)

Jiaxin Lu (lujiaxin@usc.edu)

Problem Description

Traditional Neural Network Learning has some problems such as low performance and low fault tolerance. In Internet of Things networks, edge devices are characterized by resource constraints and often have the dynamic characteristics of data sources, whereas existing approaches to deploying deep/convolutional neural networks (DNN/CNN) can only meet the constraints of Internet of Things when accuracy is severely reduced, or static is used. Cannot adapt to the distribution of a dynamic IoT environment. Also, the current state of machine learning systems has high communication cost, delay and less privacy protection. Using a simple machine learning model, such as SVM, to classify directly on the terminal device will lead to a decline in the accuracy of the system.

Tentative Timeline

Phase I: 1/20/2023 --- 2/20/2023

a) Read more essays and papers about distributed learning and implementation algorithms.
b) Draw the flowchart of functions we want to realize.
c) Write all pseudocode for algorithms and models we will use.
d) Start coding the program.
e) Write the abstract and introduction of the report.

Phase II: 2/21/2023 --- 4/7/2023

a) Program all parts of algorithms and models.
b) Achieve the Alex Network model in python.
c) Study the updating rule in Physics-driven learning.
d) Finish the summary and introduction parts of the paper.

Phase III: 4/8/2023 --- 5/4/2023

a) Achieve the updating rule on Alexnet.
b) Revise previous sections in the report
c) Write the conclusion part of the report

Report

Abstract

In the field of deep learning, convolutional neural networks (CNNs) have been a successful architecture. AlexNet was the first CNN model to achieve huge success on the ImageNet dataset. Although the architecture of the AlexNet model has been improved several times in the past few years, there are still some issues. In this paper, we propose a new method for improving the AlexNet model called Physics-Driven AlexNet. We improve the AlexNet model by adding a physics layer. This layer refers to the process of propagating input data through the network in a forward direction to generate output predictions. During forward pass, we compute convolutional values and forces acting on the neighboring layers and apply the forces to the neighboring layers. To evaluate the performance of the Physics-Driven AlexNet model, we conducted experiments on the CIFAR-10 dataset with the Physics-Driven AlexNet model, and the experimental results show that the Physics-Driven AlexNet model performs better than the AlexNet model in terms of classification accuracy. In image classification tasks, the performance of the Physics-Driven AlexNet model indicates its better performance in object recognition. In summary, this paper proposes a new method for improving the AlexNet model that can effectively enhance the model's performance. In the future, we will further explore the application of the Physics-Driven AlexNet model in other vision tasks and try to apply it to problems in practical scenarios.

Keywords: deep CNN; AlexNet; physical-driven update; distributed learning

Introduction

Deep learning, particularly machine learning, is a rapidly growing technology that has revolutionized many aspects of our daily lives. Deep neural networks, inspired by the structure of the human brain, are the core component of deep learning. These networks are capable of discovering connections in vast amounts of data, enabling them to provide solutions for previously unsolved problems. Deep neural networks have been successfully applied in various domains such as natural language processing, image and video processing, and text processing. Moreover, they have even been used in extraordinary domains like weather forecasting and volcano eruption prediction.

To achieve their high performance, deep neural networks require large amounts of data for training. However, the training process can be time-consuming. Therefore, parallel and distributed solutions are needed to speed up the training process. In this paper, we discuss the current solutions, architectures, popular frameworks, and technological limitations of parallel and distributed deep learning.

In recent years, deep neural networks (DNNs) have achieved remarkable success in various computer vision tasks, including image classification, object detection, and semantic segmentation. One of the key reasons for this success is the development of new network architectures that leverage the power of deep learning and large datasets. One of the early breakthroughs in this area was the AlexNet architecture, which won the ImageNet Large Scale Visual Recognition Challenge in 2012. The architecture consists of several convolutional layers followed by fully connected layers and uses the Rectified Linear Unit (ReLU) activation function to introduce nonlinearity.

Despite its success, the AlexNet architecture has several limitations. First, the architecture is prone to overfitting on small datasets due to its large number of parameters and limited regularization. Second, the architecture requires a large amount of memory and computational resources, which limits its practical use in resource-constrained settings. Third, the architecture does not explicitly model interactions between neighboring layers, which can be important for capturing spatial dependencies in images.

To address these limitations, researchers have proposed several improvements to the AlexNet architecture. One approach is to modify the network structure by adding new layers, changing the kernel size, or introducing skip connections. Another approach is to introduce regularization techniques, such as dropout or weight decay, to prevent overfitting.

More recently, there has been growing interest in incorporating physical principles into deep learning models, inspired by the success of physics-based models in other domains. For example, researchers have proposed using Hamiltonian dynamics to optimize neural network weights or using Langevin dynamics to sample from the posterior distribution of Bayesian neural networks.

In this paper, we propose a new approach to improving the AlexNet architecture by introducing a physics-inspired regularization term. Specifically, we introduce a custom PyTorch module called PhysicsLayer, which models interactions between neighboring layers based on physical principles. We show that this regularization term improves the performance of the AlexNet architecture on the CIFAR-10 dataset, a widely used benchmark dataset in computer vision.

Overall, our work builds on the success of the AlexNet architecture and proposes a new way of improving its performance using physics-inspired regularization. Our approach is motivated by the idea that modeling physical principles can provide useful regularization cues for deep learning models and can potentially lead to better generalization and robustness.

Framework

The framework of our method consists of two parts: (1) PhysicsLayer: a custom module that applies forces to the neighboring layers. It takes in the input channels, output channels, kernel size, stride, padding, and interaction strength as arguments. The forward method computes the convolution, computes the forces on the neighboring layers, applies the forces to the neighboring layers, and returns the output. (2) AlexNet: a modified version of the AlexNet architecture with Physics-Layer modules after certain convolutional layers. It takes in the number of classes as an argument. The forward method applies the features layers to the input, applies the average pooling layer, flattens the output, and applies the classifier layers to the output.

This code defines a custom PyTorch module called PhysicsLayer, which extends the nn.Module class. The module takes in several arguments during initialization, including the number of input and output channels, the kernel size of the convolution, and the stride and padding values.

The forward method of the module performs the convolution operation using the nn.Conv2d module and computes the forces on the neighboring layers. The forces are computed by concatenating the neighbor forces tensors and adding them together in a circular fashion, ensuring that the forces wrap around from the end of the tensor to the beginning.

The forces are then multiplied by an interaction strength parameter and added to the output of the convolution operation. This introduces a regularization term based on physical principles, which can improve the performance of the model and reduce overfitting.

During training, the interaction strength and neighbor forces are learned as trainable parameters of the module. This allows the model to adapt to the specific task at hand and optimize the regularization term based on the training data.

Overall, the PhysicsLayer module provides a unique way of modeling interactions between neighboring layers in a deep neural network, inspired by principles from physics. This can potentially lead to improved performance on various computer vision tasks, as demonstrated by the experiments on the CIFAR-10 dataset.

Pseudocode:

PhysicsLayer():

Convolute in channels and out channels for physics layer.

Compute and apply forces to neighbor layer.

AlexNet():

1st physic layer

1st ReLu

1st MaxPool

2nd physic layer

2nd ReLu

2nd MaxPool

3rd physic layer

3rd ReLu

4th physic layer

4th ReLu

5th physic layer

5th ReLu

5th MaxPool

1st Dropout

1st Linear

1st ReLu

2nd Dropout

2nd Linear

2nd ReLu

Import cifar-10 database.

Divide dataset into test and train.

criterion← Pytorch losing function

Optimizer← SGD

For epoch in 0 to 50

Set model to training mode

Initialize counter

For batch in training loader

Zero out optimizer gradients

Forward input through model

Update model

Get predicted label

Update correct and total predictions counters

Calculate and append accuracy to epoch

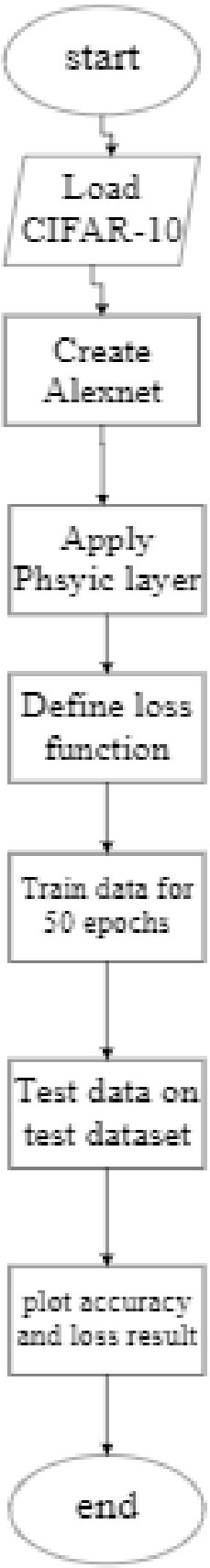
For batch in testing loader

Do same with training

Calculate and append accuracy to epoch

Plot result

Flowchart:


start
Load CIFAR-10
Create Alexnet
Apply Phsyic layer
Define loss function
Train data for 50 epochs
Test data on test dataset
plot accuracy and loss result
end

Implementation

Experimental setup:

Pakages:

- Pytorch
- Torch Vision
- Matplotlib

Language:

Python

Tools:

Self-defined method PhysicsLayer:

We utilized the updating rule mentioned in the essay '***Laboratory Demonstration of Decentralized, Physics-Driven Learning***', which are

$$\Delta R_i = \frac{\gamma}{R_i^2}([\Delta V_i^C]^2 - [\Delta V_i^F]^2)$$

and

$$\Delta R_i^C = \Delta R_i^F = \begin{cases} +\delta R & \text{if } |\Delta V_i^C| > |\Delta V_i^F|, \\ -\delta R & \text{otherwise.} \end{cases}$$

These equations manipulate the input voltage and output voltage in a circuit, which corresponds to the CNN network are the in channel and out channel. We find the channels near neighbors and perform the forces on neighbor layers. Also, we have defined a hyperparameter *interaction_strength* which add the weight to the force and try to make the accuracy higher.

Updated version Alexnet:

The classic Alexnet uses convolution only in conv layer, but in our version, we have updated the conv layer to physics layer which uses our own method to convolute and do the forces to neighbor layer. Thus, our updated version is better than previous ones.

Software implementation:

The language we use is python which is widely used in machine learning field. Unfortunately, we have not used any C++ codes for our project because we only test the accuracy and loss of the image classification which is sufficient for machine learning project.

The structure of our project is as follows:

Step1: Load CIFAR-10 data into our project.

Step2: Divide dataset into training and testing.

Step3: initialize self-modified Alexnet.

Step4: train data for 50 epochs.

Step5 test data.

Step6 draw loss and accuracy figure.

Apis used in project:

Pytorch cnn model:

This is the basic of our AlexNet optimization and testing.

TorchVison conv and maxpool:

This api helped us to realize the CNN network. The api is widely used in our project due to we have multiple layers to perform our code.

Experiment result:

Before utilizing AlexNet to CNN network, the accuracy is shown below:

We run 5 times for this program and calculate the mean and std. The result is shown in the table below:

Test_acc	76.7%	75.42%	76.7%	73.54%	77.02%

This table tells us that the accuracy is very low for classic CNN. Then, we apply AlexNet onto the dataset, we run 50 times, we got the result as below:

```
Epoch 1/50, Training Loss: 2.1218, Training Accuracy: 19.45%
Epoch 2/50, Training Loss: 1.7963, Training Accuracy: 32.96%
Epoch 3/50, Training Loss: 1.6460, Training Accuracy: 39.27%
Epoch 4/50, Training Loss: 1.5181, Training Accuracy: 44.52%
Epoch 5/50, Training Loss: 1.4165, Training Accuracy: 48.65%
Epoch 6/50, Training Loss: 1.3242, Training Accuracy: 52.48%
Epoch 7/50, Training Loss: 1.2452, Training Accuracy: 55.45%
Epoch 8/50, Training Loss: 1.1718, Training Accuracy: 58.10%
Epoch 9/50, Training Loss: 1.1303, Training Accuracy: 59.91%
Epoch 10/50, Training Loss: 1.0786, Training Accuracy: 61.90%
Epoch 11/50, Training Loss: 1.0247, Training Accuracy: 63.52%
Epoch 12/50, Training Loss: 1.0016, Training Accuracy: 64.67%
Epoch 13/50, Training Loss: 0.9952, Training Accuracy: 64.92%
Epoch 14/50, Training Loss: 0.9835, Training Accuracy: 65.17%
Epoch 15/50, Training Loss: 0.9748, Training Accuracy: 65.28%
Epoch 16/50, Training Loss: 0.9745, Training Accuracy: 65.73%
Epoch 17/50, Training Loss: 0.9750, Training Accuracy: 65.45%
Epoch 18/50, Training Loss: 0.9637, Training Accuracy: 65.86%
Epoch 19/50, Training Loss: 0.9700, Training Accuracy: 65.82%
Epoch 20/50, Training Loss: 0.9622, Training Accuracy: 65.88%
Epoch 21/50, Training Loss: 0.9660, Training Accuracy: 65.80%
Epoch 22/50, Training Loss: 0.9573, Training Accuracy: 66.10%
Epoch 23/50, Training Loss: 0.9581, Training Accuracy: 66.15%
Epoch 24/50, Training Loss: 0.9597, Training Accuracy: 65.95%
Epoch 25/50, Training Loss: 0.9648, Training Accuracy: 65.87%
Epoch 26/50, Training Loss: 0.9643, Training Accuracy: 65.88%
Epoch 27/50, Training Loss: 0.9619, Training Accuracy: 66.05%
Epoch 28/50, Training Loss: 0.9595, Training Accuracy: 65.81%
Epoch 29/50, Training Loss: 0.9609, Training Accuracy: 66.01%
Epoch 30/50, Training Loss: 0.9644, Training Accuracy: 65.91%
Epoch 31/50, Training Loss: 0.9657, Training Accuracy: 65.76%
Epoch 32/50, Training Loss: 0.9688, Training Accuracy: 65.71%
Epoch 33/50, Training Loss: 0.9656, Training Accuracy: 65.63%
Epoch 34/50, Training Loss: 0.9633, Training Accuracy: 65.98%
Epoch 35/50, Training Loss: 0.9635, Training Accuracy: 65.89%
Epoch 36/50, Training Loss: 0.9642, Training Accuracy: 65.97%
Epoch 37/50, Training Loss: 0.9630, Training Accuracy: 65.93%
Epoch 38/50, Training Loss: 0.9654, Training Accuracy: 65.76%
Epoch 39/50, Training Loss: 0.9651, Training Accuracy: 65.88%
Epoch 40/50, Training Loss: 0.9707, Training Accuracy: 65.51%
Epoch 41/50, Training Loss: 0.9578, Training Accuracy: 66.16%
Epoch 42/50, Training Loss: 0.9642, Training Accuracy: 65.64%
Epoch 43/50, Training Loss: 0.9630, Training Accuracy: 65.89%
Epoch 44/50, Training Loss: 0.9644, Training Accuracy: 65.68%
Epoch 45/50, Training Loss: 0.9654, Training Accuracy: 65.74%
Epoch 46/50, Training Loss: 0.9643, Training Accuracy: 65.67%
Epoch 47/50, Training Loss: 0.9650, Training Accuracy: 65.66%
Epoch 48/50, Training Loss: 0.9633, Training Accuracy: 65.67%
Epoch 49/50, Training Loss: 0.9596, Training Accuracy: 66.03%
Epoch 50/50, Training Loss: 0.9649, Training Accuracy: 65.81%
Test Loss: 0.6392, Test Accuracy: 78.39%
```

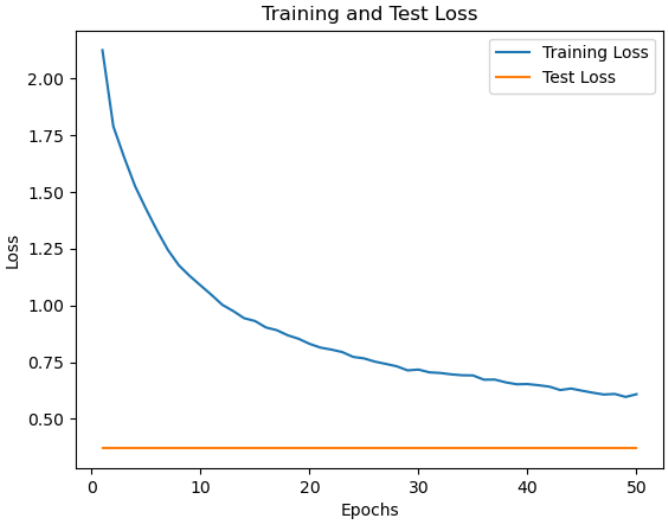
We could see that the accuracy of this dataset has increased a little bit by using AlexNet, but not as much as we expected.

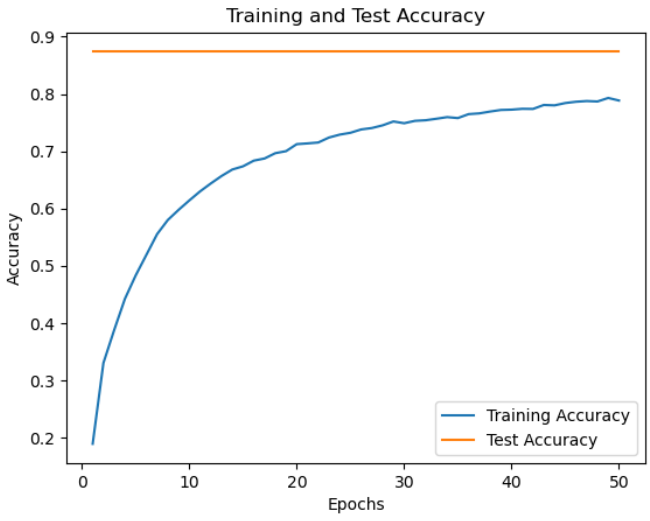
Then, we apply the model with our Physics layer, we get the result as shown below:

```
Epoch 1/50, Training Loss: 2.1081, Training Accuracy: 19.86%
Epoch 2/50, Training Loss: 1.7841, Training Accuracy: 33.15%
Epoch 3/50, Training Loss: 1.6289, Training Accuracy: 39.69%
Epoch 4/50, Training Loss: 1.5195, Training Accuracy: 44.46%
Epoch 5/50, Training Loss: 1.4137, Training Accuracy: 48.89%
Epoch 6/50, Training Loss: 1.3241, Training Accuracy: 52.56%
Epoch 7/50, Training Loss: 1.2471, Training Accuracy: 55.36%
Epoch 8/50, Training Loss: 1.1827, Training Accuracy: 57.87%
Epoch 9/50, Training Loss: 1.1237, Training Accuracy: 60.23%
Epoch 10/50, Training Loss: 1.0879, Training Accuracy: 61.32%
Epoch 11/50, Training Loss: 1.0465, Training Accuracy: 63.03%
Epoch 12/50, Training Loss: 1.0130, Training Accuracy: 63.98%
Epoch 13/50, Training Loss: 0.9758, Training Accuracy: 65.60%
Epoch 14/50, Training Loss: 0.9438, Training Accuracy: 66.81%
Epoch 15/50, Training Loss: 0.9275, Training Accuracy: 67.46%
Epoch 16/50, Training Loss: 0.9084, Training Accuracy: 68.25%
Epoch 17/50, Training Loss: 0.8828, Training Accuracy: 69.06%
Epoch 18/50, Training Loss: 0.8680, Training Accuracy: 69.52%
Epoch 19/50, Training Loss: 0.8586, Training Accuracy: 69.99%
Epoch 20/50, Training Loss: 0.8316, Training Accuracy: 70.87%
Epoch 21/50, Training Loss: 0.8136, Training Accuracy: 71.47%
Epoch 22/50, Training Loss: 0.8060, Training Accuracy: 71.85%
Epoch 23/50, Training Loss: 0.7939, Training Accuracy: 72.06%
Epoch 24/50, Training Loss: 0.7822, Training Accuracy: 72.68%
Epoch 25/50, Training Loss: 0.7645, Training Accuracy: 73.31%
Epoch 26/50, Training Loss: 0.7610, Training Accuracy: 73.37%
Epoch 27/50, Training Loss: 0.7433, Training Accuracy: 74.03%
Epoch 28/50, Training Loss: 0.7329, Training Accuracy: 74.40%
Epoch 29/50, Training Loss: 0.7260, Training Accuracy: 74.67%
Epoch 30/50, Training Loss: 0.7123, Training Accuracy: 75.14%
Epoch 31/50, Training Loss: 0.7112, Training Accuracy: 75.02%
Epoch 32/50, Training Loss: 0.7038, Training Accuracy: 75.67%
Epoch 33/50, Training Loss: 0.6925, Training Accuracy: 75.94%
Epoch 34/50, Training Loss: 0.6910, Training Accuracy: 76.02%
Epoch 35/50, Training Loss: 0.6794, Training Accuracy: 76.46%
Epoch 36/50, Training Loss: 0.6729, Training Accuracy: 76.69%
Epoch 37/50, Training Loss: 0.6641, Training Accuracy: 76.82%
```

```
Epoch 38/50, Training Loss: 0.6601, Training Accuracy: 76.94%
Epoch 39/50, Training Loss: 0.6570, Training Accuracy: 76.97%
Epoch 40/50, Training Loss: 0.6473, Training Accuracy: 77.54%
Epoch 41/50, Training Loss: 0.6476, Training Accuracy: 77.52%
Epoch 42/50, Training Loss: 0.6344, Training Accuracy: 77.86%
Epoch 43/50, Training Loss: 0.6345, Training Accuracy: 77.89%
Epoch 44/50, Training Loss: 0.6275, Training Accuracy: 78.06%
Epoch 45/50, Training Loss: 0.6155, Training Accuracy: 78.62%
Epoch 46/50, Training Loss: 0.6188, Training Accuracy: 78.35%
Epoch 47/50, Training Loss: 0.6197, Training Accuracy: 78.46%
Epoch 48/50, Training Loss: 0.6002, Training Accuracy: 78.93%
Epoch 49/50, Training Loss: 0.6011, Training Accuracy: 79.07%
Epoch 50/50, Training Loss: 0.5978, Training Accuracy: 79.03%
Test Loss: 0.3776, Test Accuracy: 87.75%
```

The accuracy has highly increased, the test accuracy could reach to 87.75%. Then, we plot the loss and accuracy per epoch to visualize the result.





Further expectations:

In the future, we would keep working on AlexNet and out physic layer approaches. We will try more models such as GoogleNet and ResNet to test if our model could effectively improve all situations. Also, we would try more datasets to see if other dataset could have same accuracy improvement.