CSE 4554
Machine Learning Lab

Experiment No: 5
Name of the experiment: Neural Networks and Convolutional Neural Networks

Dr. Hasan Mahmud
Associate Professor, Department of CSE
Md. Shihab Shahriar
Lecturer, Department of CSE

November 16, 2023

Contents

1 Objectives

- To understand how neural networks work
- To implement simple Neural Networks using PyTorch
- To set up a deep learning-based image classification pipeline using only Neural Networks
- To implement Convolutional Neural Networks using PyTorch
- To set up a deep learning-based image classification pipeline using CNN's
- To compare CNN-based image classification pipeline with the pipeline using only Neural Networks
- To compare CNN-based image classification accuracies after data augmentation using different strategies

2 What is PyTorch?

PyTorch is a Python-based scientific computing package serving two broad purposes:

1. A replacement for NumPy to use the power of GPUs and other accelerators.
2. An automatic differentiation library that is useful to implement neural networks.

Some key features of PyTorch:

1. PyTorch provides tensors and dynamic neural networks with strong GPU acceleration. Tensors are similar to NumPy arrays and can run on GPUs.
2. It has an autograd system that automatically calculates gradients required for backpropagation in neural networks. This makes training models easier and more intuitive.
3. PyTorch has modular, readable and flexible code making it easy to use and integrate into existing Python programs.
4. It supports rapid prototyping through Python libraries and tools like IPython/Jupyter notebooks.
5. PyTorch can convert models into production-ready models in C++ and Python using TorchScript.
6. It has distributed training capabilities making it easy to train models across multiple GPUs/nodes.

PyTorch is optimized for fast, flexible deep learning research and development. It is gaining popularity due to its ease of use and focus on Python programming. Key advantages include GPU support, automated gradients, modular design and Python focus.

3 Simple Implementation of a Neural Network using Pytorch

A PyTorch implementation of a neural network looks exactly like a NumPy implementation. The goal of this section is to showcase the equivalent nature of PyTorch and NumPy. For this purpose, let's create a simple three-layered network having 5 nodes in the input layer, 3 in the hidden layer, and 1 in the output layer. We will use only one training example with one row which has five features and one target.

```
import torch
n_input, n_hidden, n_output = 5, 3, 1
```

A simple neural network can be defined and trained in five key steps:

1. parameter initialization
2. Forward Propagation
3. Loss computation
4. Backpropagation
5. Updating the parameters

Let's see each of these steps in a bit more detail.

3.1 Parameter Initialization:

The first step is to do parameter initialization. Here, the weights and bias parameters for each layer are initialized as the tensor variables. Tensors are the base data structures of PyTorch which are used for building different types of neural networks. They can be considered as the generalization of arrays and matrices; in other words, tensors are N-dimensional matrices.

```
## Initialize tensor for inputs and outputs
x = torch.randn((1, n_input))
y = torch.randn((1, n_output))
## Initialize tensor variables for weights
w1 = torch.randn(n_input, n_hidden) # weight for hidden layer
w2 = torch.randn(n_hidden, n_output) # weight for output layer
## initialize tensor variables for bias terms
b1 = torch.randn((1, n_hidden)) # bias for hidden layer
b2 = torch.randn((1, n_output)) # bias for output layer
```

3.2 Forward Propagation:

In this step, activations are calculated at every layer using the two steps shown below. These activations flow in the forward direction from the input layer to the output layer in order to generate the final output.

```
z = weight * input + bias
a = activation_function (z)
## sigmoid activation function using pytorch
def sigmoid_activation(z):
return 1 / (1 + torch.exp(-z))
## activation of hidden layer
z1 = torch.mm(x, w1) + b1
```

```
a1 = sigmoid_activation(z1)
## activation (output) of final layer
z2 = torch.mm(a1, w2) + b2
output = sigmoid_activation(z2)
```

The code blocks show how we can write these steps in PyTorch. Notice that most of the functions, such as exponential and matrix multiplication, are similar to the ones in NumPy.

3.3 Loss Computation:

In this step, the error (also called loss) is calculated in the output layer. A simple loss function can tell the difference between the actual value and the predicted value. Later, we will look at different loss functions available in PyTorch.

```
loss = y - output
```

3.4 Backpropagation:

The aim of this step is to minimize the error in the output layer by making marginal changes in the bias and the weights. These marginal changes are computed using the derivatives of the error term. Based on the Calculus principle of the Chain rule, the delta changes are back passed to hidden layers where corresponding changes in their weights and bias are made. This leads to an adjustment in the weights and bias until the error is minimized.

```
## function to calculate the derivative of activation
def sigmoid_delta(x):
return x * (1 - x)
## compute derivative of error terms
delta_output = sigmoid_delta(output)
delta_hidden = sigmoid_delta(a1)
## backpass the changes to previous layers
d_outp = loss * delta_output
loss_h = torch.mm(d_outp, w2.t())
d_hidn = loss_h * delta_hidden
```

3.5 Updating the Parameters:

Finally, the weights and biases are updated using the delta changes received from the above back-propagation step.

```
learning_rate = 0.1
w2 += torch.mm(a1.t(), d_outp) * learning_rate
w1 += torch.mm(x.t(), d_hidn) * learning_rate
b2 += d_outp.sum() * learning_rate
b1 += d_hidn.sum() * learning_rate
```

Finally, when these steps are executed for a number of epochs with a large number of training examples, the loss is reduced to a minimum value. The final weight and bias values are obtained which can then be used to make predictions on the unseen data.

4 Autograd Feature of PyTorch

PyTorch's Autograd feature is part of what makes PyTorch flexible and fast for building machine learning projects. It allows for the rapid and easy computation of multiple partial derivatives (also

referred to as gradients) over a complex computation. This operation is central to backpropagation-based neural network learning.

The power of Autograd comes from the fact that it traces your computation dynamically at runtime, meaning that if your model has decision branches or loops whose lengths are not known until runtime, the computation will still be traced correctly, and you'll get correct gradients to drive learning. This, combined with the fact that your models are built in Python, offers far more flexibility than frameworks that rely on static analysis of a more rigidly structured model for computing gradients.

For more information refer to this YouTube video:
https://www.youtube.com/watch?v=M0fX15_-xrY&t=9s&ab_channel=PyTorch.

So, the good news is you do not need to compute derivatives when using Pytorch. Pytorch will handle that for you.

5 Building Models with PyTorch (The PyTorch Way)

The simple implementation that we learned above is not suitable for building neural network models efficiently and with ease. PyTorch has a lot of features that let you create and train various types of neural networks with ease. Learn about the different tools PyTorch makes available to create different types of neural networks from this notebook:
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/fe726e041160526cf828806536922cf6/modelsyt_tutorial.ipynb

You may also refer to this YouTube video:
https://www.youtube.com/watch?v=OSqIP-mOWOI&list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN&index=4&ab_channel=PyTorch

6 PyTorch Tensorboard Support

TensorBoard is a well-known tool for visualizing the activity of machine learning models and training progress. PyTorch provides support for TensorBoard use, with no dependency on other ML frameworks. This video here
https://www.youtube.com/watch?v=6CEld3hZgqc&list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN&index=5&ab_channel=PyTorch
will show you how to visualize graphs of training progress, input images, model structure, and dataset embeddings with TensorBoard and PyTorch. An example is shown in this notebook
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/e2e556f6b4693c2cef716dd7f40caaf6/tensorboardyt_tutorial.ipynb

The notebook above shows you how to use Tensorboard in a new browser tab. If you run the notebook in Google Colab you just need to run the following code to start Tensorboard.

```
%load_ext tensorboard
%tensorboard --logdir runs
```

7 Traning a Convolutional Neural Network Model with Pytorch

You can use PyTorch to build a CNN model to classify images. An example is shown in this notebook
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/

770632dd3941d2a51b831c52ded57aa2/trainingyt.ipynb
where you train a CNN-based model on the Fashion-MNIST dataset which has the following ten classes - 'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', and 'Ankle Boot'.

For more understanding, you can refer to this video:
https://www.youtube.com/watch?v=jF43_wj_DCQ&list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN&index=6&ab_channel=PyTorch

It is highly suggested to go through this Youtube tutorial
https://www.youtube.com/playlist?list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN
playlist if you want to better understand how to use PyTorch.

8 Tasks

8.1 Load Dataset:

Load a dataset based on your student ID (if ID%3=0 then 'CIFER10' else if ID%3=1 then 'Fashin-MNIST' else if ID%3=2 then 'MNIST')

8.2 Downsample Dataset

Sample data for only 5 classes in train, test and validation sets

8.3 Train a Simple Neural Network Model

Flatten the images and train a simple neural network model with 10 epochs and a suitable learning rate. Get the accuracy on the test set. There should be 5 Linear layers in your model with the number of neurons in the 3 hidden layers being set according to your student ID. (ID%1000 number of neurons)

8.4 Train a Convolutional Neural Network Model

Train a CNN model with 3 conv2d layers and 3 Linear layers in the end. Train for 10 epochs and a suitable learning rate. Get the accuracy on the test set.

8.5 Use Pooling layers

Use a maxpool2d layer after each of the conv2d layers. Train the model again and check the accuracy.

8.6 Use Droput layers

This time use a dropout layer after each of the maxpool2d layers. Train the model again and check the accuracy.

8.7 Use Augmentation Techniques

Use data augmentation techniques to augment your training data. Train the model again and check the accuracy. You may use any two of the data augmentation strategies below:

1. v2.AutoAugment

2. v2.ColorJitter
3. v2.functional.horizontal_flip
4. v2.RandomRotation

Train for 20 epochs and a suitable learning rate. More about these data augmentation techniques can be found here (`https://pytorch.org/vision/stable/transforms.html`.)

8.8 Use a Tensorboard to Visalize

Use a tensorboard to show the loss curve of your train and validation sets in the last training task.