华南理工大学

South China University of Technology

The Experiment Report of Deep Learning

School: School of Software Engineering

Subject: Software Engineering

Author:

Aiqing Wen,

Xiaoli Tang

Student ID:

201720145037

201720145044

Supervisor:

Mingkui Tan

Grade:

Post Graduate

December 19, 2017

Handwritten Digit Recognition Based on Shallow Neural Network

***Abstract*—Convolutional Neural Network is a special kind of multi-layer neural networks. Like almost every other neural networks they are trained with a version of the back-propagation algorithm. Where they differ is in the architecture. In this experiment, we use LeNet-5 to do handwritten and machine-printed character recognition with MINST dataset.**

i. INTRODUCTION

Convolutional Neural Networks are designed to recognize visual patterns directly from pixel images with minimal preprocessing. They can recognize patterns with extreme variability (such as handwritten characters), and with robustness to distortions and simple geometric transformations.

The LeNet architecture is an excellent “first architecture” for Convolutional Neural Networks (especially when trained on the MNIST dataset, an image dataset for handwritten digit recognition). LeNet is small and easy to understand — yet large enough to provide interesting results. Furthermore, the combination of LeNet + MNIST is able to run on the CPU, making it easy for beginners to take their first step in Deep Learning and Convolutional Neural Networks.In many ways, LeNet + MNIST is the “Hello, World” equivalent of Deep Learning for image classification.

This experiment aims learning pytorch with the handwritten and machine-printed character recognition problem using LeNet. The first part comes to the introduction of pytorch, MINST and LeNet-5. Secondly, we will briefly show and analysis the result of this experiment. Finally, we will give a conclusion to summarize this experiment.

ii. METHODS AND THEORY

A. Pytorch

PyTorch is an open source machine learning library for Python, used for applications such as natural language processing. It is primarily developed by Facebook's artificial-intelligence research group, and Uber's "Pyro" software for probabilistic programming is built upon it.

PyTorch is a python package that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration.
- Deep Neural Networks built on a tape-based autograd system.

a) Ramp-up Time

PyTorch is essentially a GPU enabled drop-in replacement for NumPy equipped with higher-level functionality for building and training deep neural networks. This makes PyTorch especially easy to learn if you are familiar with NumPy, Python and the usual deep learning abstractions (convolutional layers, recurrent layers, SGD, etc.).

b) Graph Creation and Debugging

Creating and running the computation graph is perhaps where the two frameworks differ the most. In PyTorch the graph construction is dynamic, meaning the graph is built at run-time.

The simple graph construction in PyTorch is easier to reason about, but perhaps even more importantly, it's easier to debug. Debugging PyTorch code is just like debugging Python code. You can use pdb and set a break point anywhere.

c) Data Loading

The APIs for data loading are well designed in PyTorch. The interfaces are specified in a dataset, a sampler, and a data loader. A data loader takes a dataset and a sampler and produces an iterator over the dataset according to the sampler's schedule. Parallelizing data loading is as simple as passing a num_workers argument to the data loader.

B. MINST

The MNIST dataset is arguably the most well-studied, most understood dataset in the computer vision and machine learning literature, making it an excellent "first dataset" to use on your deep learning journey.

The goal of this dataset is to classify the handwritten digits 0-9. We're given a total of 70,000 images, with (normally) 60,000 images used for training and 10,000 used for evaluation; however, we're free to split this data as we see fit. Common splits include the standard 60,000/10,000, 75%/25%, and 66.6%/33.3%. I'll be using 2/3 of the data for training and 1/3 of the data for testing later in the blog post.

Each digit is represented as a 28 x 28 grayscale image (examples from the MNIST dataset can be seen in the figure above). These grayscale pixel intensities are unsigned integers, with the values of the pixels falling in the range [0, 255]. All digits are placed on a black background with a light foreground (i.e., the digit itself) being white and various shades of gray.

It's worth noting that many libraries (such as scikit-learn) have built-in helper methods to download the MNIST dataset, cache it locally to disk, and then load it. These helper methods normally represent each image as a 784-d vector.

C. LeNet

This section describes in more detail the architecture of LeNet-5, the CNN used in the experiments. Convolutional layers and max-pooling are at the heart of the LeNet family of models. While the exact details of the model will vary greatly, the figure below shows a graphical depiction of a LeNet model.

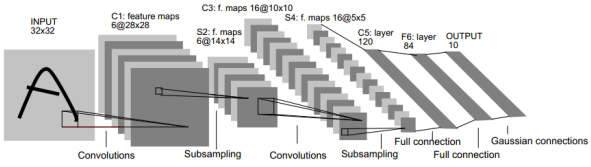

Fig.1. LeNet-5

The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all features maps at the layer below. The model we used is explained in details in following contents:

The C1 layer is a conv layer with 6 5*5 kernels, the output of this layer is a 6*24*24 image(the input image is 1*28*28).

The S2 is a subsampling using max-pooling, with kernel 2*2, after that the image would be 6*12*12.

The C3 layer is a conv layer with 16 kernels each has the size 5*5, so the output here is a 16*8*8. And after subsampling as before, the output would be 16*4*4.

After two convolutional layers, there are 3 fully-connected layers, the input and output of them is 16*4*4->120, 120->84 and 84->10.

The ReLu function is used as activation function in all the layers in the model, except for the output layer which uses soft-max function. Moreover, to get a better result and to avoid over-fitting in some ways, we add dropout in the second convolutional layer and the first fully-connected layer.

iii. EXPERIMENTS

A. Dataset

This experiment use handwritten digital data set - MNIST, which contains 60,000 hand-written digital images foe training and 10,000 hand-written digital images for validation. Each image is 28 × 28 pixels in size. periment uses a9a of LIBSVM Data, including 32561/16281(valuation) samples and each sample has 123/123 (valuation) features.

B. Implementation

Pytorch classes used in this experiment:

1. torch.nn.Module

Base class for all neural network modules. Your models should also subclass this class. Modules can also contain other Modules, allowing to nest them in a tree structure.

2. torch.nn.functional

a) torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)

   Applies a 2D convolution over an input image composed of several input planes.

b) torch.nn.functional.linear(input, weight, bias=None)

   Applies a linear transformation to the incoming data: y=xAT+b.

c) torch.nn.functional.max_pool2d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)

   Applies a 2D max pooling over an input signal composed of several input planes.

d) torch.nn.functional.dropout(input, p=0.5, training=False, inplace=False)

   Applies alpha dropout to the input.

e) torch.nn.functional.relu(input, threshold, value, inplace=False) → Variable

   Applies the rectified linear unit function element-wise.

f) torch.nn.functional.softmax(input, dim=None, _stacklevel=3)

   Applies a softmax function.It is applied to all slices along dim, and will rescale them

so that the elements lie in the range *(0, 1)* and sum to 1.

g) torch.nn.functional.log_softmax(input, dim=None, _stacklevel=3)

Applies a soft-max followed by a logarithm. While mathematically equivalent to log(softmax(x)), doing these two operations separately is slower, and numerically unstable. This function uses an alternative formulation to compute the output and gradient correctly.

h) torch.nn.functional.nll_loss(input, target, weight=None, size_average=True, ignore_index=-100, reduce=True)

The negative log likelihood loss.

3. torch.optim

Torch.optim is a package implementing various optimization algorithms. Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can be also easily integrated in the future.

a) torch.optim.SGD(params, lr=<object object>, momentum=0, dampening=0, weight_decay=0, nesterov=False)

Implements stochastic gradient descent (optionally with momentum. Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

4. torch.autograd

Torch.autograd provides classes and functions implementing automatic differentiation of arbitrary scalar valued functions. It requires minimal changes to the existing code - you only need to wrap all tensors in Variable objects.

a) torch.autograd.backward(variables, grad_variables=None, retain_graph=None, create_graph=None, retain_variables=None)

Computes the sum of gradients of given variables w.r.t. graph leaves.

The graph is differentiated using the chain rule. If any of variables are non-scalar (i.e. their data has more than one element) and require gradient, the function additionally requires specifying grad variables. It should be a sequence of matching length, that contains gradient of the differentiated function w.r.t. corresponding variables (None is an acceptable value for all variables that don't need gradient tensors).

This function accumulates gradients in the leaves - you might need to zero them before calling it.

5. torchvision

The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.

a) torchvision.datasets.MNIST(root, train=True, transform=None, target_transform=None, download=False)

C. Result and Analysis

The parameters using in the experiment are showed below:

- batch-size=64
- test-batch-size=1000
- epochs=10
- learning rate=0.01

- momentum=0.5

The loss and accuracy of the test set is showing in the Fig.2. It is showed that the loss of the model can be lower than 0.05 and the accuracy of the test set can be up to 98%(with the parameter and LeNet model setting as mentioned before). The result indicates that the LeNet-5 network can do a good job in handwritten digit recognition mission with a high efficiency.

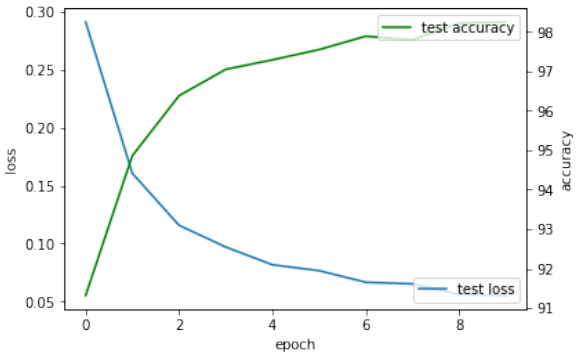

Fig.2. loss and accuracy of test set

iv. CONCLUSION

PyTorch is an open source machine learning library for Python which is worth learning. And LeNet is a shallow neural network model which is good for beginner to understand the basis of CNN. This experiment do benefit us greatly from the theory to practice.