This repository Search Pull requests Issues Gist

danhtuan / **deep_learning** Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 1 Wiki Pulse Graphs Settings

Branch: master deep_learning / prj2 / **README.md** Find file Copy path

danhtuan Update README.md 6da05f2 a minute ago

1 contributor

```
265 lines (213 sloc)   7.74 KB
```

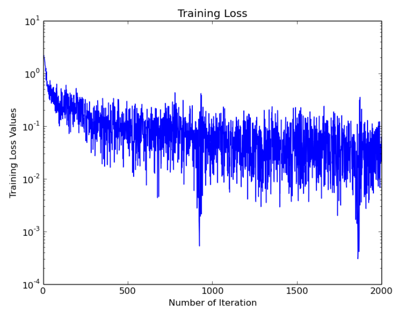
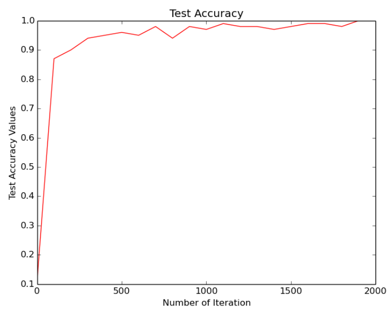
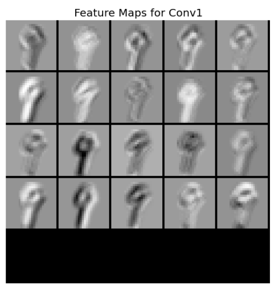
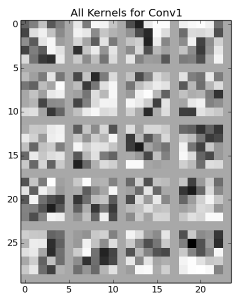
Raw Blame History

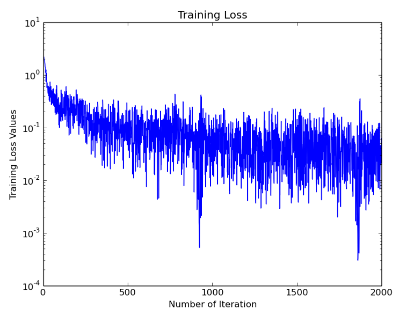
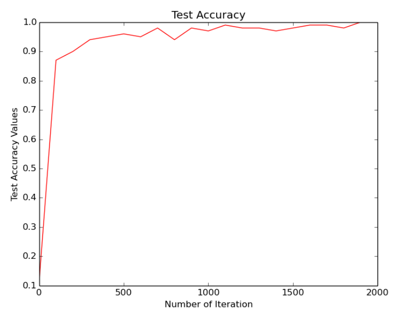
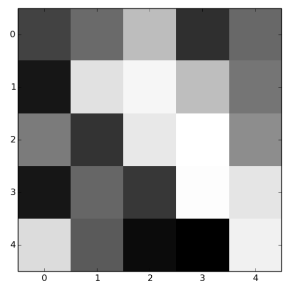
Miniproject 2: Caffe and MNIST dataset

- Tuan Nguyen
- Spring 17
- Deep learning - Dr. Martin Hagan

1. Setting up

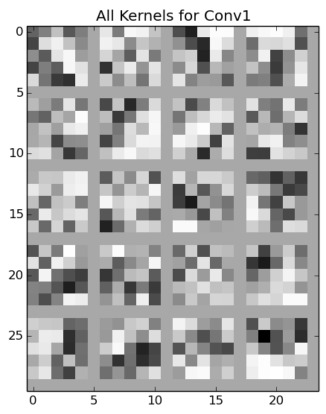
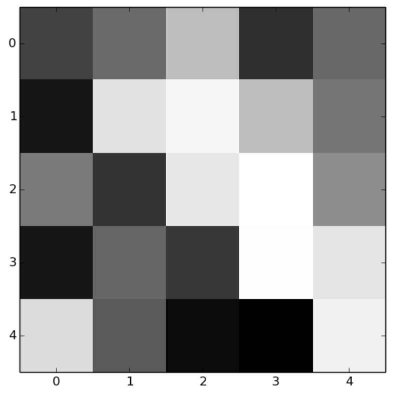
- Download 4 data files and unzip
- Modify create_mnist.sh
- Modify prototxt files
- Run `train_mnist.py` file and observe the outputs as following:

2. Investigate the kernels

Basically, each kernel is used to explor a specific `feature` in the input data. The provided CVN uses two convolution layers, the first convolution layer has 20 kernels and the second one has 50 kernels. Below is all 20 kernels and one specific kernel respectively for the first layer.

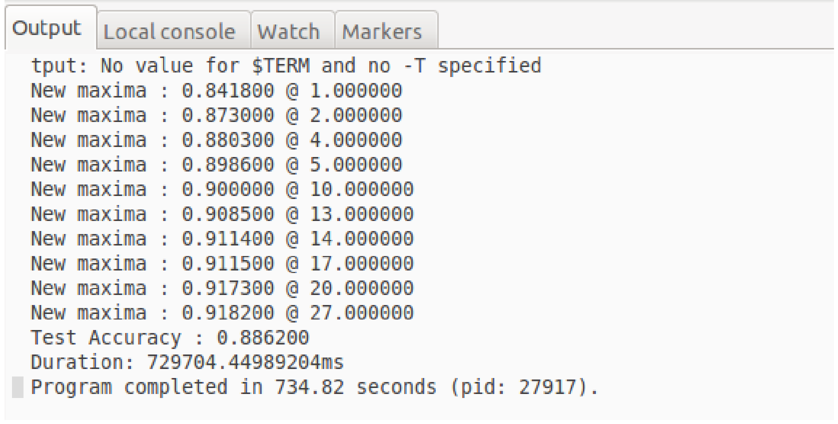
All kernels for Conv1	One kernel for Conv1
	

Investigating into above kernels, we can see that different kernels try to explor different `features` , or in this case, different `textures/edges` in the images of numerals. The given specific kernel, for example, is helpful for numerals that have the diagonal edges/curve (back slash/curve) such as `6, 8, 9` .

3. Performance Convolution Network (CVN) vs. Multilayer Network (MLP)

3.1 Accuracy

Here is the output when runing MLP:



```
tput: No value for $TERM and no -T specified
New maxima : 0.841800 @ 1.000000
New maxima : 0.873000 @ 2.000000
New maxima : 0.880300 @ 4.000000
New maxima : 0.898600 @ 5.000000
New maxima : 0.900000 @ 10.000000
New maxima : 0.908500 @ 13.000000
New maxima : 0.911400 @ 14.000000
New maxima : 0.911500 @ 17.000000
New maxima : 0.917300 @ 20.000000
New maxima : 0.918200 @ 27.000000
Test Accuracy : 0.886200
Duration: 729704.44989204ms
Program completed in 734.82 seconds (pid: 27917).
```

And here is the output when runing CVN:

```
Iteration 1900 testing... accuracy: 0.990000009537
data    (128, 1, 28, 28)
label   (128,)
conv1   (128, 20, 24, 24)
pool1   (128, 20, 12, 12)
conv2   (128, 50, 8, 8)
pool2   (128, 50, 4, 4)
ip1 (128, 500)
ip2 (128, 10)
loss    ()
conv1   (20, 1, 5, 5) (20,)
conv2   (50, 20, 5, 5) (50,)
ip1 (500, 800) (500,)
ip2 (10, 500) (10,)

IPython CPU timings (estimated):
  User   :      84.52 s.
  System :      14.62 s.
Wall time:     128.77 s.
```

As shown, the CVN is better in term of accuracy in comparison with MLP. Based on LeCun's website, though:

- Convolutional net LeNet-5, [no distortions] -- > 0.95 %lost
- 3-layer NN, 500+150 hidden units --> 2.95 %lost

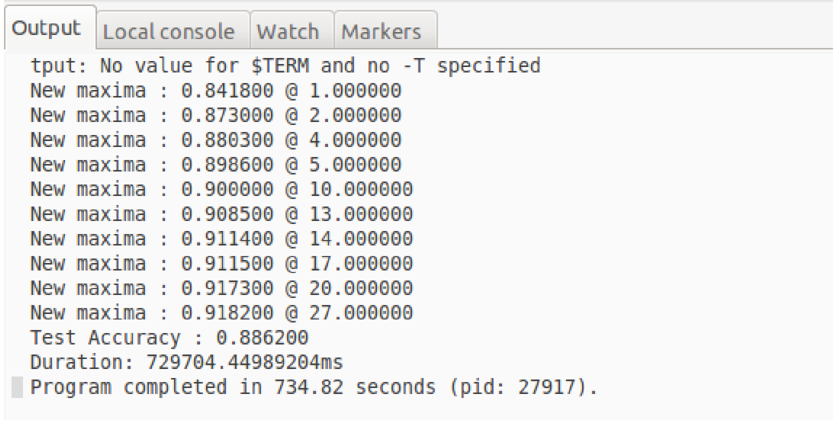
So it is reasonable to conclude that CVN is better than MLP in this criteria.

3.2 Performance

For CVN timings:

```
IPython CPU timings (estimated):
  User   :      84.52 s.
  System :      14.62 s.
Wall time:     128.77 s.
```

For MLP timings:



```
tput: No value for $TERM and no -T specified
New maxima : 0.841800 @ 1.000000
New maxima : 0.873000 @ 2.000000
New maxima : 0.880300 @ 4.000000
New maxima : 0.898600 @ 5.000000
New maxima : 0.900000 @ 10.000000
New maxima : 0.908500 @ 13.000000
New maxima : 0.911400 @ 14.000000
New maxima : 0.911500 @ 17.000000
New maxima : 0.917300 @ 20.000000
New maxima : 0.918200 @ 27.000000
Test Accuracy : 0.886200
Duration: 729704.44989204ms
Program completed in 734.82 seconds (pid: 27917).
```

Intuitively, CVN is slower than MLP, partly because the convolution is normally slower than matrix multiplication. However, it is shown above that MLP is much slower in total time. It is reasonable because CVN might converge faster than MLP. However, because Miniproject 1 uses Torch and Miniprojet 2 uses Caffe, the comparison is not fair and may be used as reference only.

4. Minibatches Size (BS)

4.1 Experiment

- BS = 64 (default)

```
IPython CPU timings (estimated):
  User   :      54.06 s.
  System :      10.18 s.
Wall time:     155.99 s.
```

- BS = 128 (double of default)

```
IPython CPU timings (estimated):
  User   :      84.52 s.
  System :      14.62 s.
Wall time:     128.77 s.
```

- BS = 32 (half of default)

```
IPython CPU timings (estimated):
  User   :      37.98 s.
  System :       6.54 s.
Wall time:      60.55 s.
```

- BS = 8 (1/8 of default) --> Better speed but Accuracy is not good

```
IPython CPU timings (estimated):
  User   :      27.39 s.
  System :       4.52 s.
Wall time:      82.52 s.
```

- BS = 1 (1/64 of default) --> Accuracy is really bad (0.07) and Error observed

```
Iteration 1900 testing... accuracy: 0.070000000298
/usr/local/lib/python2.7/dist-packages/numpy/ma/core.py:4144: UserWarning: Warning: converting a masked ele
  warnings.warn("Warning: converting a masked element to nan.")
```

4.2 Comparison

Bigger Batch Size	Smaller Batch Size
more accurate	less accurate
slower training	faster training

Based on observation, BS = 32 is the best in speed given the constraint about Accuracy.

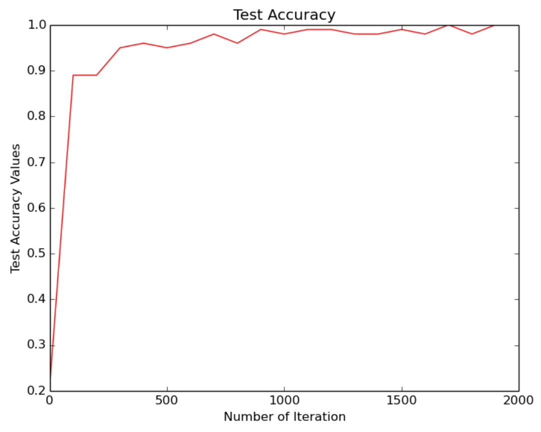
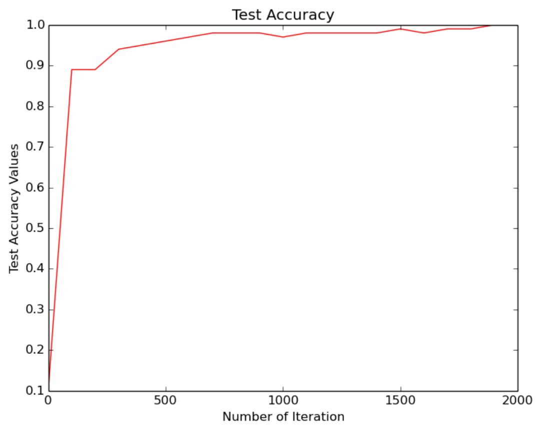
5. Dropout Layer

Dropout layer is claimed a simple way to reduce/prevent overfitting in training NN. Here I added a dropout layer at fully-connected layer (fc or ip) as following:

```
layer{
    name: "dropout1"
    type: "Dropout"
    bottom: "ip1"
    top: "ip1"
    dropout_param {
        dropout_ratio: 0.5
    }
}
```

Here is the running time and test accuracy:

```
IPython CPU timings (estimated):
  User   :      52.60 s.
  System :      10.15 s.
Wall time:     131.81 s.
```

Test Accuracy (No Dropout)	Test Accuracy (Dropout)
	

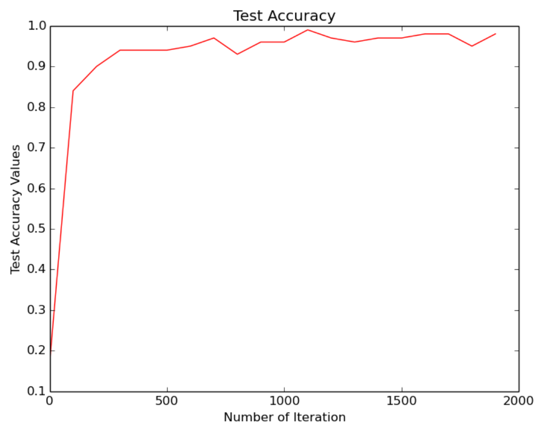
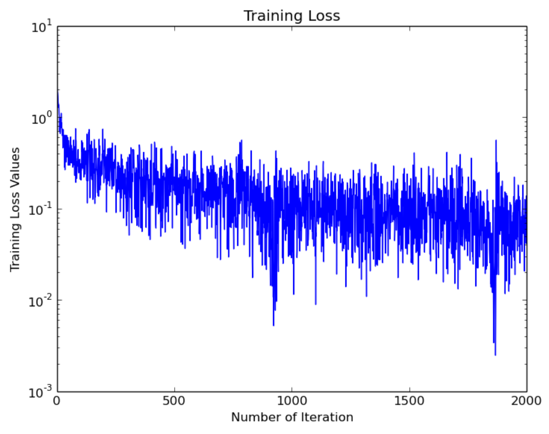
Based on above information, Dropout doesn't change too much running time while making the accuracy line smoother. So it is safe to say it helps improve the testing error.

6. Deep Learning vs. Narrow Wide Learning

The original lenet has 2 convolution layers and 2 pooling layers. To make it narrower but wider, I have modified it to make it having only 1 convolution layer and 1 pooling layer but increase the kernels/feature maps per layer from 20 to 70 (which equals to sum of two convolution layers). Here is the prototxt for the new network:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 70
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

Here is the test accuracy and training loss for the narrow wide network:

Test Accuracy	Training Loss
	

Here is the timing for the narrow wide network:

```
IPython CPU timings (estimated):
  User   :      32.57 s.
  System :       5.72 s.
Wall time:      63.28 s.
```

As shown above, the narrow wide network is a little bit faster than the deep network but it comes with less accuracy.

7. Alternative Training Function (AdaDelta)

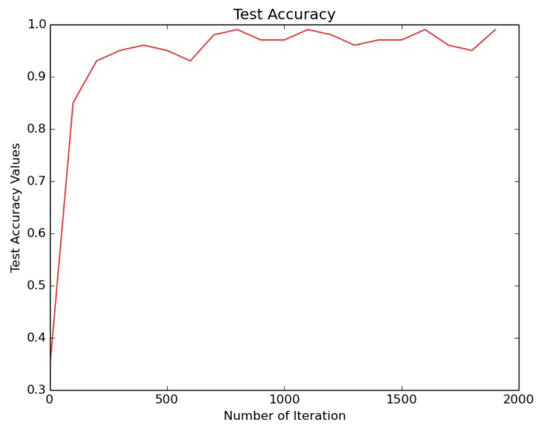
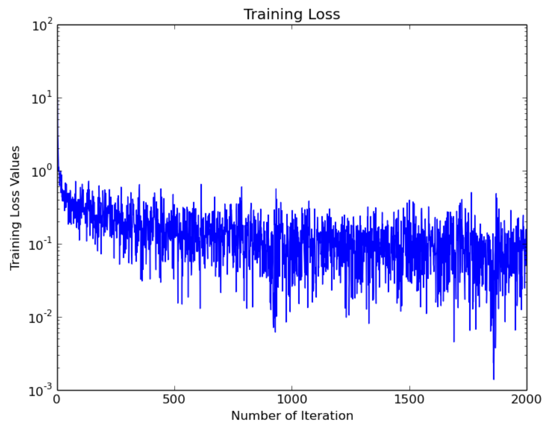
7.1 Solver prototxt

```
# The train/test net protocol buffer definition
net: "/home/martin/Desktop/tuandn/git_repo/deep_learning/prj2/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 1.0
lr_policy: "fixed"
momentum: 0.95
weight_decay: 0.0005
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
# snapshot: 500
#snapshot_prefix: "examples/mnist/lenet_adadelta"
# solver mode: CPU or GPU
solver_mode: GPU
type: "AdaDelta"
delta: 1e-6
```

7.2 Python

```
solver = caffe.AdaDeltaSolver('lenet_solver.prototxt')
```

7.3 Outputs

Test Accuracy	Loss
	

7.4 Comparison

The AdaDelta (type: "AdaDelta") method (M. Zeiler [1]) is a “robust learning rate method”. It is a gradient-based optimization method (like SGD). Because AdaDelta is also based on gradient, the performance is pretty the same as SGD