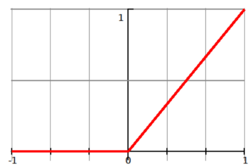
EE 569 Homework #5
Name: YAN JIAO
USC-ID: 6419057887

## (a) CNN Architecture and Training

| CNN components | description | functions |
|---|---|---|
| Fully connected layer | ▪ This layer takes an input volume, for example x*y*z;<br>▪ And it uses n filters each also with size of x*y*z;<br>▪ For each filter, we need to calculate the sum of elementwise product of the input vector and the filter vector to form 1 neuron (1 number);<br>▪ So the output is a vector with n neuron (1*1*n). | ▪ This layer usually used at the end of the CNN process, because several times repeating of the CONV, RELU and POOL layers has already reduced the data size into a much smaller size which can apply FC layer easily;<br>▪ The output will be the class scores;<br>▪ For example, if the output is [0 .1 .1 .75 0 0 0 0 0 .05], then this represents a 10% probability that the image is a 1, a 10% probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9. |
| Convolutional layer | ▪ This layer uses n filters, and each filter only has the sum of the dot product of the local regions of the input volume, and form one neuron at the center pixel location;<br>▪ The filter should be the same size of the local region, for example, 5*5*3;<br>▪ Usually the spatial size of the local region should have a odd side length, so the neuron value can go the center pixel;<br>▪ If the input image has the volume of 32*32*3, and the stride is 1, each of the n filters should give one layer of neurons (size 28*28*1); since we have n such filters, so finally we will have a tensor with the size 28*28*n. | ▪ Use CONV layer rather than fully connected layer can largely reduce the computational cost;<br>▪ Each filter can extract some important information about one or several features of the image. |
| Max pooling layer | ▪ This process is a down-sampling process, for each;<br>▪ For each depth layer, we usually divide the image into several 2*2 square area, then for each little square, we only keep the max value of the 4 pixels; | ▪ To make the data volume smaller, speed up the computing process. |

| | | |
|---|---|---|
| | ▪ So, if the input volume has size of x*y*n, then the output should be (x/2)*(y/2)*n;<br>▪ If the x or y is odd, then we should do some padding process to let the side length to be even. | |
| Activation function | ▪ This function is biologically inspired by activity in our brains, where different neurons are activated by different stimuli;<br>▪ The most commonly used activation function is RELU function:<br><br>ReLU<br>▪ When the neuron value is above 0, then we can say that this neuron is activated, otherwise, this neuron is inactivated. | ▪ Let the neuron network know which neurons are activated;<br>▪ These activated neurons can help us know the features of the images. |
| Softmax function | ▪ The softmax function squashes the outputs of each unit to be between 0 and 1;<br>▪ But it also divides each output such that the total sum of the outputs is equal to 1;<br>▪ The output of the softmax function is equivalent to a categorical probability distribution, it tells you the probability that any of the classes are true. | ▪ It can help us know the probability of the classification clearly. |

over-fitting: happens when your model fits too well to your training set. So, when comes to the test set, during the testing loop, if the image is not in the training set, it will be very difficult for the model to classify it very accurately. It is like that the model can recognize specific images in the training set but not the image with general patterns.

If visualize the over-fitting process, it will be like Figure 1. If we change the model form a very simple one to a very complex one (model more complex means that there are more parameters that we can adjust), at the beginning (model complexity <= 3 in Figure 1), the more complex the model is, the lower classify error rate is; along with the model complexity keeps going up, the classify error rate inside the training set still keeps going down, but for images outside the training set, the classify error rate will go up very fast. This is because when the complexity is too high, the model parameters can describe the classes' features  too well (like can pay more attention to details), so when an image is not in the training data set, the CNN model may think too much about the details and make a wrong classification.

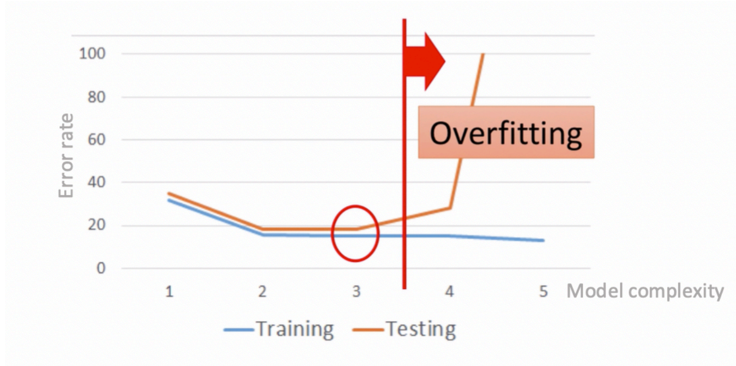



Figure 1: over-fitting

Ways to avoid over-fitting in CNN:

1. Add more data: if we cannot change the model's parameter amount, we can do some transformations on the input data set, like geometric transformation, to generate more input data.
2. Use data augmentation
3. Use architectures that generalize well
4. Reduce architecture complexity: processes like pooling layers

Some other ways to avoid over-fitting in CNN: add regularization (mostly dropout, L1/L2 regularization are also possible)

Compare to traditional image processing method:

| CNN | tradition |
|---|---|
| Can handle a huge data set | Similar (even a slightly better) performance compared to CNN when data set is relatively small |
| pass the data directly to the network and usually achieve good performance very fast | Need complex feature engineering (deep exploratory data, dimensionality reduction, select best features pass to the algorithm) |

Loss function:

To find the difference between the prediction and the true value.
Usually, we use cross entropy loss to be the loss function, rather than MSE.

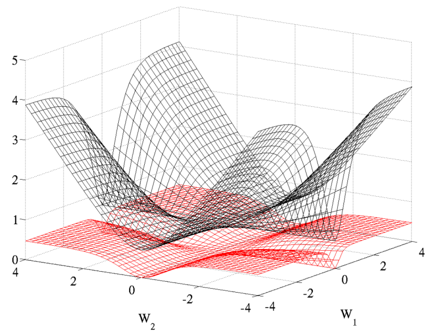



Figure 2: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W1 respectively on the first layer and W2 on the second, output layer. (plot from: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)

MSE will update the parameter in a small range, so cross entropy loss is better.

### Backpropagation:

Use chain rule to efficiently calculate the gradient. From the output layer to the input layer, calculate each layer's partial deviation between input and output parameter, and then chain these partial deviations to get interested gradient. The key formula is:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

(b) Train LeNet-5 on MINST Dataset

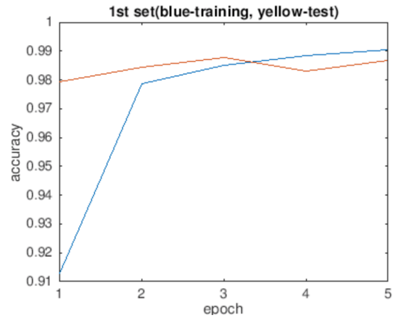
## 1st parameter set

<table>
<tr><td>Reading data parameter setting</td><td><pre>transform = transforms.Compose( [transforms.ToTensor(), transforms.Normalize((0.5,), (1.0,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)</pre></td></tr>
<tr><td>Model parameter setting</td><td><pre>def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(1, 6, 5,1,2)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x</pre></td></tr>
<tr><td>Loss function parameter setting</td><td><pre>criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)</pre></td></tr>
<tr><td>Epoch#</td><td><pre>5</pre></td></tr>
</table>

Output:

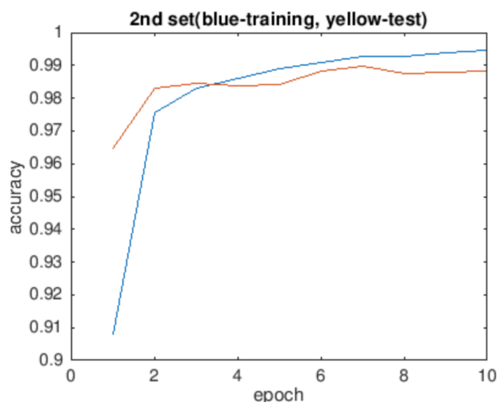



```
([train-accuracy, test-accuracy] order)
[[0.912766695022583, 0.9794999957084656], [0.978600025177002, 0.984499990940094], [0.9849666953086853,
0.9876999855041504], [0.9883833527565002, 0.983199954032898], [0.9905333518981934, 0.9866999983787537]]
```
**Time: 425861.3125**

## 2nd parameter set (differ from the 1st)

| | |
|---|---|
| Epoch# | **10** |

Output:



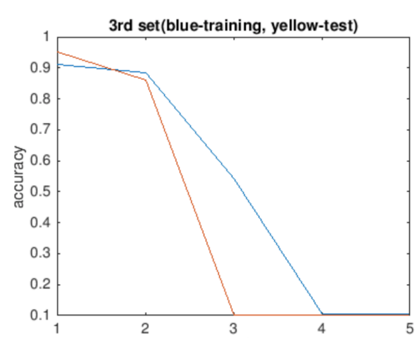

```
[[0.9078666567802429, 0.9648999571800232], [0.975433349609375, 0.983199954032898], [0.9829000234603882,
0.9845999479293823], [0.9860000014305115, 0.9835999608039856], [0.9890833497047424, 0.9839999675750732],
[0.9907333254814148, 0.988099992275238], [0.9926833510398865, 0.9896999597549438], [0.9928333163261414,
0.9874999523162842], [0.9937999844551086, 0.9878000020980835], [0.9947500228881836, 0.9881999492645264]]
```

The 2nd parameter set performants the best among the 5 sets. Compare to 1st set, the epoch# is bigger in a reasonable range, then the model will update its parameters several times more to get a better performance.

## 3rd parameter set (differ from the 1st)

| | |
|---|---|
| Learning rate | `optimizer = optim.SGD(net.parameters(), **lr=0.01**, momentum=0.9)` |

Output:



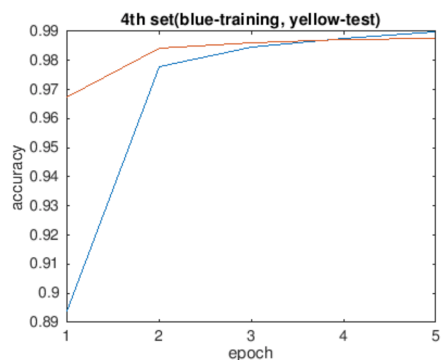
```
[[0.9123333692550659, 0.9519000053405762], [0.8834500312805176, 0.8606999516487122], [0.5423499941825867, 0.10099999606609344], [0.10463333129882812, 0.10279999673366547], [0.10590000450611115, 0.10089999437332153]]
```

This set is unsuccessful. If we set the learning rate too big, when we update our parameters to get good performance, we may step over the best parameters set, and never get the good performance since the learning is too big.

### 4rd parameter set (differ from the 1st)

| | |
|---|---|
| Layer parameter | self.fc1 = nn.Linear(16 * 5 * 5, **80**)<br>self.fc2 = nn.Linear(**80**, 84) |

Output:



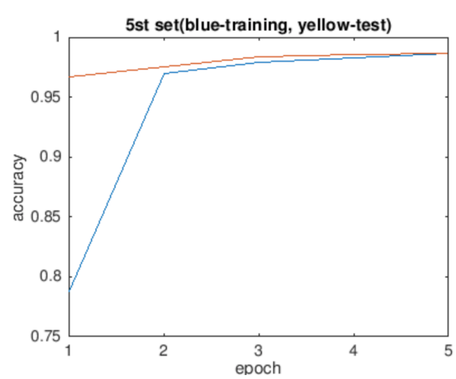

```
[0.8938000202178955, 0.9674999713897705], [0.9778333306312561, 0.9840999841690063], [0.9846667051315308, 0.9861999750137329], [0.9876000285148621, 0.9872999787330627], [0.9898333549499512, 0.9876999855041504]]
```

If we change the FC layer filter# (from 120 to 80), the final accuracy is a little smaller than the original one. This may because the less filters#, the less information for the network to adjust the model.

### 5rd parameter set (differ from the 1st)

| | |
|---|---|
| Batch size | trainloader = torch.utils.data.DataLoader(trainset, batch_size=**10**, shuffle=True, num_workers=2)<br>testloader = torch.utils.data.DataLoader(testset, batch_size=**10**, shuffle=False, num_workers=2) |

Output:




```
[[0.7875333428382874, 0.9672999978065491], [0.9697999954223633, 0.9750999808311462], [0.9791333675384521, 0.9836999773979187], [0.9831666946411133, 0.9860000014305115], [0.9862333536148071, 0.986499955368042]]
```

**Time: 193446.84375**

If we set the batch_size a little bigger, although the accuracy is not higher than the 1st set, the training and testing time is shorter. This is because that bigger batch_size will reduce the loop times, and will speed up the parameter adjustment.

Summary:

| | 1st | 2nd (best) | 3rd | 4th | 5th | mean | variance |
|---|---|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| accuracy | 98.67% | 98.82% | 10.09% | 98.77% | 98.65% | 81.00% | 15.71% |

(c) Apply trained network to negative images

1)

Use the 2nd parameter set.

If change all the test images into negative images, the performance will be very bad, only 52.02%.

The output is following:

```
Accuracy of the network on the 10000 test images:
23.510000 %
Accuracy of the network on the 10000 test images:
35.880000 %
Accuracy of the network on the 10000 test images:
44.400000 %
[4,  2000] loss: 0.0376
[4,  4000] loss: 0.0336
[4,  6000] loss: 0.0335
[4,  8000] loss: 0.0360
[4, 10000] loss: 0.0314
[4, 12000] loss: 0.0340
[4, 14000] loss: 0.0321
Accuracy of the network on the 10000 test images:
44.730000 %
[5,  2000] loss: 0.0324
[5,  4000] loss: 0.0218
[5,  6000] loss: 0.0264
[5,  8000] loss: 0.0308
[5, 10000] loss: 0.0239
[5, 12000] loss: 0.0268
[5, 14000] loss: 0.0286
Accuracy of the network on the 10000 test images:
46.140000 %
[6,  2000] loss: 0.0232
[6,  4000] loss: 0.0192
[6,  6000] loss: 0.0225
[6,  8000] loss: 0.0238
[6, 10000] loss: 0.0233
[6, 12000] loss: 0.0215
[6, 14000] loss: 0.0246
Accuracy of the network on the 10000 test images:
46.670000 %
[7,  2000] loss: 0.0156
[7,  4000] loss: 0.0211
[7,  6000] loss: 0.0158
[7,  8000] loss: 0.0164
[7, 10000] loss: 0.0151
[7, 12000] loss: 0.0254
[7, 14000] loss: 0.0195
Accuracy of the network on the 10000 test images:
46.340000 %
[8,  2000] loss: 0.0149
[8,  4000] loss: 0.0143
[8,  6000] loss: 0.0112
[8,  8000] loss: 0.0143
[8, 10000] loss: 0.0243
[8, 12000] loss: 0.0157
[8, 14000] loss: 0.0140
Accuracy of the network on the 10000 test images:
43.390000 %
[9,  2000] loss: 0.0128
[9,  4000] loss: 0.0126
[9,  6000] loss: 0.0081
[9,  8000] loss: 0.0150
[9, 10000] loss: 0.0110
[9, 12000] loss: 0.0146
[9, 14000] loss: 0.0170
Accuracy of the network on the 10000 test images:
47.780000 %
[10,  2000] loss: 0.0107
[10,  4000] loss: 0.0084
[10,  6000] loss: 0.0073
[10,  8000] loss: 0.0119
[10, 10000] loss: 0.0093
[10, 12000] loss: 0.0194
[10, 14000] loss: 0.0152
Accuracy of the network on the 10000 test images:
52.020000 %
time: 1023257.625
```

The low accuracy is because that the feature in the negative image is very different from the positive image, even the edge location is the same.

2)

To improve the accuracy, we should retrain the model, take the negative image into the training process. The final accuracy is 98.5%.

The training process takes much more time, but the performance improves a lot.

Output:

```
Accuracy of the network on the 10000 test images:
98.490000 %
Accuracy of the network on the 10000 test images:
98.890000 %
Accuracy of the network on the 10000 test images:
98.700000 %
Accuracy of the network on the 10000 test images:
98.890000 %
Accuracy of the network on the 10000 test images:
98.890000 %
Accuracy of the network on the 10000 test images:
98.940000 %
Accuracy of the network on the 10000 test images:
99.130000 %
Accuracy of the network on the 10000 test images:
99.000000 %
Accuracy of the network on the 10000 test images:
98.780000 %
Accuracy of the network on the 10000 test images:
98.500000 %
time：2034769.0
```

If we want to get a similar accuracy with less time, we can select part of the positive images and part of the negative images to train the model.

Output:

```
Accuracy of the network on the 10000 test images: 98.090000 %
Accuracy of the network on the 10000 test images: 98.430000 %
Accuracy of the network on the 10000 test images: 97.950000 %
Accuracy of the network on the 10000 test images: 98.110000 %
Accuracy of the network on the 10000 test images: 98.240000 %
Accuracy of the network on the 10000 test images: 98.910000 %
Accuracy of the network on the 10000 test images: 98.950000 %
Accuracy of the network on the 10000 test images: 99.100000 %
Accuracy of the network on the 10000 test images: 99.040000 %
Accuracy of the network on the 10000 test images: 99.130000 %
```

time: 1047489.0