Project-1 of “Neural Network and Deep Learning”

Student ID: 16307110435   Student Name: JiwenZhang

Before Modification : Exploration

Before we start the project, we want to see the effect of our primary network and explore how hyper-parameters influence its performance. So we draw the following form:

Fixed: `stepSize = 1e-3` `batch_size = 1`	nHidden	Num_Epochs	Test Error	Running Time
	[10]	20	0.4020	15.659 s
	[32]	20	0.3080	39.236 s
	[64]	20	0.2410	60.541 s
	[128]	**20**	**0.1910**	**86.708 s**
	[256]	**20**	**0.1980**	**171.939 s**

* `maxIter = 5000 * num_epochs`
* random seed is fiexed by `rng(2020);`

From above we can see that the test error decreases along with the increase of hidden layer dimension. So a naïve conclusion is: more trainable parameters in networks maybe helpful to cut the test error.

Modifications

3. Make the compute of loss and gradient be matrix friendly.

Code Change:

```
%% Compute Output
ip{1} = X * inputWeights;
fp{1} = tanh(ip{1});
for h = 2:length(nHidden)
    ip{h} = fp{h-1} * hiddenWeights{h-1};
    fp{h} = tanh(ip{h});
end
yhat = fp{end} * outputWeights;
```

Similar changes have been made on **Back-Propagate Error** process.

First do this in order to accelerate the computation of loss. Otherwise the cpu will waste a lot of time in loops. After doing this, we can see that under the same parameter setting, the running time of this program reduced **from 86s to 39s.** More than 100% change!

Fixed: `stepSize = 1e-3` `batch_size = 1`	nHidden	Num_Epochs	Test Error	Running Time
	[128]	20	0.1910	**39.330 s**

1. Hidden layer change

Now let's try to find a good combination of hidden layer ! Since we have make the computation of loss become matrix friendly, we can set the batch_size >= 1. This can also save our time greatly!

```
%% Choose network structur
nHidden = [512, 256];
```

	nHidden	Num_Epochs	Test Error	Running Time
Fixed: `batch_size = 20` `stepSize = 1e-3`	**[128]**	**20**	**0.2150**	**4.703 s**
	[257]	**20**	**0.2090**	**9.2123 s**
	[64, 10]	20	0.6340	7.672 s
	[64, 32]	20	0.4700	7.694 s
	[64, 64]	20	0.4250	7.7574 s
	[128, 64]	20	0.3770	9.6898 s
	[128, 128]	20	0.3330	11.28 s
	[257, 128]	**20**	**0.2970**	**19.0264 s**
	[512, 256]	**20**	**0.2710**	**39.450 s**
	[512, 128]	20	0.3150	27.36 s
	[64, 32, 16]	40	0.6750	16.0709 s
	[128, 32, 16]	40	0.6590	20.876 s
	[128, 128, 64]	40	0.5100	23.523s
	[256, 128, 128]	40	0.4360	40.624 s
	[257, 128, 64]	40	0.5370	37.622 s
	[257, 256, 128]	40	0.3970	50.847 s

From above from we can see that one hidden layer or two hidden layer maybe a good choice for our program. So we select four candidate that will be tested in the following steps:

$$nHidden = [128]$$
$$nHidden = [257]$$
$$nHidden = [257, 128]$$
$$nHidden = [512, 256]$$

2. Set the stepsize decay exponentially, and replace the stochastic gradient descent with momentum gradient descent.

- Stepsize is updated after loop over an epoch.

$$Stepsize \leftarrow Stepsize * 0.96$$

- Momentun Gradient Descent (since we have added L2 regularization)

$$w \leftarrow w - Stepsize * g + beta * (w - pw)$$

Since the common selection of beta is 0.9, we will follow this routine in this program.

$$w \leftarrow w - Stepsize * g + 0.9 * (w - pw)$$

where $stepsize$ is free.

We only changed the weight update code:

```
tmp = w;
if train_mode == "momentum"
    w = w - stepSize * g + beta*(w - pw); % momentum
elseif train_mode == "l2reg-momentum"
    w = w - stepSize * (g + lambda*w.*bias_mask) + beta*(w - pw);
elseif train_mode == "sgd"
    w = w - stepSize * g; % pure sgd
end
pw = tmp;
```

And add one line in the train loop:

```
stepSize = stepSize * decay_factor; % decay factor = 0.96
```

Fixed: `batch_size = 20`

nHidden	stepSize	Num_epochs	Test Error	Running Time
[128]	1e-3	20	0.9240	5.599 s
[128]	1e-4	20	0.2260	5.279 s
[128]	**1e-5**	**20**	**0.1860**	**5.53 s**
[128]	1e-6	20	0.9030	5.532 s
[257]	1e-4	30	0.2210	14.388 s
[257]	**1e-5**	**30**	**0.1370**	**15.089 s**
[257, 128]	1e-4	40	0.3090	35.982 s
[257, 128]	1e-5	40	0.2870	33.284 s
[512, 256]	1e-4	40	0.2920	89.769 s
[512, 256]	**1e-5**	**40**	**0.2150**	**96.4538 s**

Generally speaking, $\boldsymbol{stepSize}$ should be chosen from $[\mathbf{1e-4}, \mathbf{1e-5}]$ to get better outcome. To figure out which is better under exponential decay, we tried:

nHidden	stepSize	Num_epochs	Test Error	Running Time
[257]	1e-4	30	0.1480	13.542 s
[257]	1e-5	30	0.3460	12.730 s

So we may better choice $\boldsymbol{stepSize} = \mathbf{1e-4}$

4. Add $l2$ regulation.

$$Loss_{total} = Loss + Loss_{reg}$$

where

$$Loss_{reg} = \frac{\lambda}{2} \| w \|_2^2$$

so the update formula for momentum gradient is

$$w \leftarrow w - Stepsize * (g + \lambda * w) + 0.9 * (w - pw)$$

The traditional setting of l2 regulation is cncentrated on weight parameter. So we need a mask vector that can set the bias == 0.

$$w \leftarrow w - Stepsize * (g + \lambda * w * bias\_mask) + 0.9 * (w - pw)$$

Fixed: `batch_size = 20, stepSize = 1e-4(exp decay)`

Lambda	nHidden	Num_epochs	Test Error	Running Time
0.01	[257]	30	0.1480	16.283 s
0.1	**[257]**	**30**	**0.1480**	**12.755 s**
0.5	**[257]**	**30**	**0.1430**	**12.753 s**
1	**[257]**	**30**	**0.1230**	**13.487 s**
2	[257]	30	0.1900	13.14 s
0.1	[512, 256]	40	0.1930	84.672 s
0.5	[512, 256]	40	0.1910	74.683 s
1	**[512, 256]**	**40**	**0.0950**	**72.256 s**
2	[512, 256]	20	0.1530	39.655 s

From above form we can see that λ can be seleected from $[0.1, 0.5, 1]$, whereas $\lambda = 1$ have the best outcome.

5. Add a softmax layer and change the loss function as negative log-likelihood.

$$z = h\{end\} * outWeight \ \in \mathbb{R}^{n \times K}$$
$$\boldsymbol{y} = softmax(z)$$

Consider one example case,

$$Loss = -\sum_{k=1}^{K} t_k * \log(y_k), \qquad \boldsymbol{t} = true\ label$$

$$\frac{\partial Loss}{\partial z_i} = \sum_{k=1}^{K} \frac{\partial L}{\partial y_k} * \frac{\partial y_k}{\partial z_i} = -\sum_{k=1}^{K} \frac{t_k}{y_k} * \frac{\partial y_k}{\partial z_i} = -\frac{t_i}{y_i} * \frac{\partial y_i}{\partial z_i} - \sum_{\substack{k=1 \\ k \neq i}}^{K} \frac{t_k}{y_k} * \frac{\partial y_k}{\partial z_i}$$

$$= -\frac{t_i}{y_i} * y_i(1 - y_i) + \sum_{\substack{k=1 \\ k \neq i}}^{K} \frac{t_k}{y_k} * y_k y_i = -t_i(1 - y_i) + \sum_{\substack{k=1 \\ k \neq i}}^{K} t_k y_i$$

$$= -t_i + \sum_{k=1}^{K} t_k y_i = -t_i + y_i \sum_{k=1}^{K} t_k = y_i - t_i$$

$$\frac{\partial Loss}{\partial \boldsymbol{z}} = \boldsymbol{y} - \boldsymbol{t}$$

Fixed: `batch_size = 20, stepSize with exp decay(0.96)`

nHidden	Num epochs	stepSize	Lambda	Test Error	Running Time
[257]	20	1e-4	1	0.7330	11.78 s
[512, 256]	40	1e-4	1	0.1650	70.4762 s

It seems that using softmax layer will lead to **inferior outcome** under the same parameter setting of squard loss, since the gradient of loss is numerically smaller. So we need to fine-tuning on hyper-parameter $stepSize$ and $Lambda$.

nHidden	Num epochs	stepSize	Lambda	Test Error	Running Time
[128]	40	1e-3	0.1	0.1250	9.041 s
[257]	40	1e-3	0.1	0.1090	16.295 s
[257, 128]	50	1e-3	0.1	0.0800	42.288 s
[512, 256]	**50**	**1e-3**	**0.1**	**0.0730**	**88.542 s**

We can easily find that smaller learning rate (stepSize here) and smaller regulation term will promote our model performance, which is consistent with our previous observations.

6. Add bias to each layer.

We need to do three modifications this time. The first is in ***pure_neuralNetwork.m***:

```
nParams = d*nHidden(1);
for h = 2:length(nHidden)
    nParams = nParams+(nHidden(h-1)+1)*nHidden(h); % bias
end
nParams = nParams+(nHidden(end)+1)*nLabels; % bias
w = randn(nParams,1);
```

The second place is in ***MLPclassificationLoss.m***:

```
%% Form Weights and Drops
inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
offset = nVars*nHidden(1);
for h = 2:length(nHidden)
  hiddenWeights{h-1} = reshape(...
      w(offset+1:offset+(nHidden(h-1)+1)*nHidden(h)),...
      nHidden(h-1)+1, nHidden(h)); % bias
  offset = offset+(nHidden(h-1)+1)*nHidden(h); % bias
end
outputWeights = w(offset+1:offset+(nHidden(end)+1)*nLabels); % bias
outputWeights = reshape(outputWeights,nHidden(end)+1,nLabels); % bias
```

The third place is in ***MLPclassificationLoss.m***:

```
backprop = sech(ip{end}).^2 .* (err * outputWeights(2:end,:)'); % bias
```

After that, let's see what happens.

nHidden	Num epochs	stepSize	Lambda	Test Error	Running Time
[128]	40	1e-3	0.1	0.1870	9.622 s
[257]	40	1e-3	0.1	0.1440	18.075 s
[257, 128]	**50**	**1e-3**	**0.1**	**0.0830**	**43.121 s**
[512, 256]	**50**	**1e-3**	**0.1**	**0.0650**	**92.077 s**

For one hidden layer network, the error will grow up if bias is added, since the network is shallow and it may not need so much parameters. While for network with two layers, bias will promote the performance of network as a bis layer can learn more from data. Ingeneral, the best network structure we have ever find is

$$nHidden = [512, 256], \qquad stepSize = 1e-3, \qquad \lambda = 0.1$$

7. Add dropout in training process.

By constructing a Dropout matrix which has the same size as hiddenWeight matrix, we can apply drop out by

$$ip\{h\} = fp\{h-1\} * \frac{hiddenWeight\{h-1\} .* hiddenDrop\{h-1\}}{dropoutRate}$$

And the code for generating Dropout matrix is

```
hiddenDrop{h-1} = double(rand(nHidden(h-1)+1, nHidden(h)) < p) / p;
```

Just do it and see what happens.

nHidden	Num epochs	stepSize	Lambda	Test Error	Running Time
[257, 128]	50	1e-3	0.1	0.0720	47.552 s
[512, 256]	50	1e-3	0.1	0.0650	118.048 s

We can see that dropout may or may not enhance the network performance. Maybe this due to the random noise introduced by mini-batch. But generally speaking, we say that dropout will not degrade the network performance.

8. Fine-Tuning the last layer

Denote the output of last hidden layer as X. Since the parameters of previous layers are all fixed, we can regard fine-tuning the last layer as a convex optimization problem with an objective function

$$SELoss = \| yTrue - yhat \|_2^2$$

where

$$yhat = X * weight + bias = [\mathbf{1} \quad X] * \begin{bmatrix} bias \\ weight \end{bmatrix} = \tilde{X} * outWeight$$

For simplicity, take $\beta = outWeight$, we can compute the first derivative:

$$\frac{\partial SE}{\partial \beta} = \frac{\partial}{\partial \beta} \| \boldsymbol{y} - \tilde{X}\beta \|_2^2 = 2 * \tilde{X}^T(y - \tilde{X}\beta)$$

Therefore the closed form solution is:

$$\boldsymbol{\beta} = \left(\tilde{X}^T\tilde{X}\right)^{-1}\tilde{X}\boldsymbol{y}$$

Simply assign this value to the $\boldsymbol{\beta} = OutWeight$ as a result of our fine-tuning.

The code for this part is written as a file named “***MLPclassificationFineTuning.m***”, so we will omit code this time since it will take up too much space.

nHidden	Hyper-Parameter	Before - Test Error	After – Test Error	Change	Running Time
[257, 128]	stepSize = 1e-3(0.96) lambda = 0.1	0.0720	0.0890	↑	49.152 s
	stepSize = 2e-3(0.96) lambda = 0.06	0.0640	0.0660	↑	50.011 s
[512, 256]	stepSize = 1e-3(0.96) lambda = 0.1	0.0650	0.0780	↑	121.328 s
	stepSize = 2e-3(0.96) lambda = 0.06	0.0600	0.0610	↑	118.395 s
	stepSize = 2e-3(0.96) lambda = 0.08	**0.0800**	**0.0540**	↓	**108.716 s**
[512, 256, 128]	stepSize = 2e-3(0.96) lambda = 0.06	0.0720	0.066	↓	144.217 s

* beta = 0.9; num_epoch = 50; batch_size=20;

From the experiment outcome, it not difficult to find that : before fine-tuning, if there are still relatively large error, fine-tuning can help to decrease it. However, if before fine-tuning the model is already properly fitted in train data, then fine-tuning may cause the error to incerase since it causes overfitting.

9. Data Argumentation

There are many data argumentation methods. We select 3 methods to implement.

(1) Scale

```
tmp = img * 0.6;
```

(2) Rotation

```
tmp = rot90(img, 1);
```

(3) Translation

```
se = translate(strel(1), [1, 1]);
tmp = imdilate(img, se);
tmp(tmp == -Inf) = 0;
```

The entire code for this part is written as a file named “***dataAugmentation.m***”, so we will omit code this time since it will take up too much space.

So let’s see that happens:

nHidden	Argumentation	stepSize	Lambda	Test Error
[512, 256]	scale	2e-3	0.08	0.0860
	rotation			0.2190
	translation			0.0950

Fixed: `num_batch=50; batch_size = 20; stepSize with exp decay(0.96)`

Compared with the previous result,

nHidden	Hyper-Parameter	Test Error
[512, 256]	**stepSize = 2e-3(0.96)** **lambda = 0.08**	**0.0800**

We can see that data argumentation can not enhance the network performance in this data set. This could probably result from the low pixels of the images. Or maybe too much data cause our model to overfit in the training data.

10. Replace the first layer of the network with a 2D convolutional layer

- The input X has shape $[batch\_size, 16, 16]$ and y has shape $[batch\_size, nLabels]$. Given $kernel_{size} = k,$ for each example

$$Z^{(1)}(i,:,:) = W^{(1)} \otimes X(i,:,:) + bias$$

where

$$W^{(1)} \in \mathbb{R}^{k*k}, \qquad bias \in \mathbb{R}$$

The output of this layer has shape

$$Z^{(1)} \in \mathbb{R}^{[batch\_size, 16-k+1, 16-k+1]}$$

- Then we need a **FULL CONNECTION** layer to make data can be bumped into our previously defined MLP model. That is,

$$Z^{(2)}(i,j) = W^{(2,j)} \otimes X^{(1)}(i,:,:) \ + bias(j)$$

Where

$$W^{(2,j)} \in \mathbb{R}^{[16-k+1, 16-k+1]}, \qquad bias(j) \in \mathbb{R}$$

The output of this layer has shape

$$Z^{(2)} \in \mathbb{R}^{[batch\_size,\ \ dim]}$$

- To Update the parameters, we need to compute the gradient, which is not easy. To clarify our gradient computation, we look one more downwards layer

$$Z^{(3)} = X^{(2)} * W^{(3)} + \boldsymbol{b} = [\mathbf{1} \quad X^{(2)}] \begin{bmatrix} \boldsymbol{b} \\ W \end{bmatrix}$$

Tip: A useful formula – if $Y = W \otimes X$

$$\frac{\partial f(Y)}{\partial X} = \boldsymbol{rot180}\left(\frac{\partial f(Y)}{\partial Y}\right) \widetilde{\otimes}\, W = \boldsymbol{rot180}(W) \,\widetilde{\otimes}\, \frac{\partial f(Y)}{\partial Y}$$

- Now let's compute the gradient w.r.t. parameters. Consider one example case:

$$\frac{\partial L(Y,\hat{Y})}{\partial W^{(2,j)}} = \frac{\partial L(Y,\hat{Y})}{\partial Z^{(2,j)}} \otimes X^{(1)} = \delta^{(2,j)} \otimes X^{(1)}$$

where the error term

$$\delta^{(2,j)} = \frac{\partial L(Y,\hat{Y})}{\partial Z^{(2,j)}}$$

Similarly we can have

$$\frac{\partial L(Y,\hat{Y})}{\partial bias^{(2)}(j)} = \sum_{i,k} [\delta^{(2,j)}]_{i,k}$$

- The computation of the error term

$$\delta^{(2,j)} = \frac{\partial L(Y,\hat{Y})}{\partial Z^{(2,j)}} = \frac{\partial X^{(2,j)}}{\partial Z^{(2,j)}} * \frac{\partial Z^{(3)}}{\partial X^{(2,j)}} * \frac{\partial L(Y,\hat{Y})}{\partial Z^{(3)}}$$

$$= f_l'\left(Z^{(2,j)}\right) * W^{(3)}(j,:) * \delta^{(3)^T}$$

Similarly,

$$\delta^{(1)} = \frac{\partial L(Y,\hat{Y})}{\partial Z^{(1)}} = \frac{\partial X^{(1)}}{\partial Z^{(1)}} * \frac{\partial L(Y,\hat{Y})}{\partial X^{(1)}}$$

$$= f_l'(Z^{(1)}) \odot \sum_j \left(rot180\left(W^{(2,j)}\right) \tilde{\otimes}\, \delta^{(2,j)}\right)$$

After modifications, the outcome is as follows:

nHidden	Num epochs	stepSize	Lambda	Test Error	Running Time
[32]	20	3e-2	5e-3	0.2590	49.695 s
[64]	40	2e-2	5e-3	0.1710	120.232 s
[64 128]	**40**	**2e-2**	**5e-3**	**0.1490**	**166.757 s**

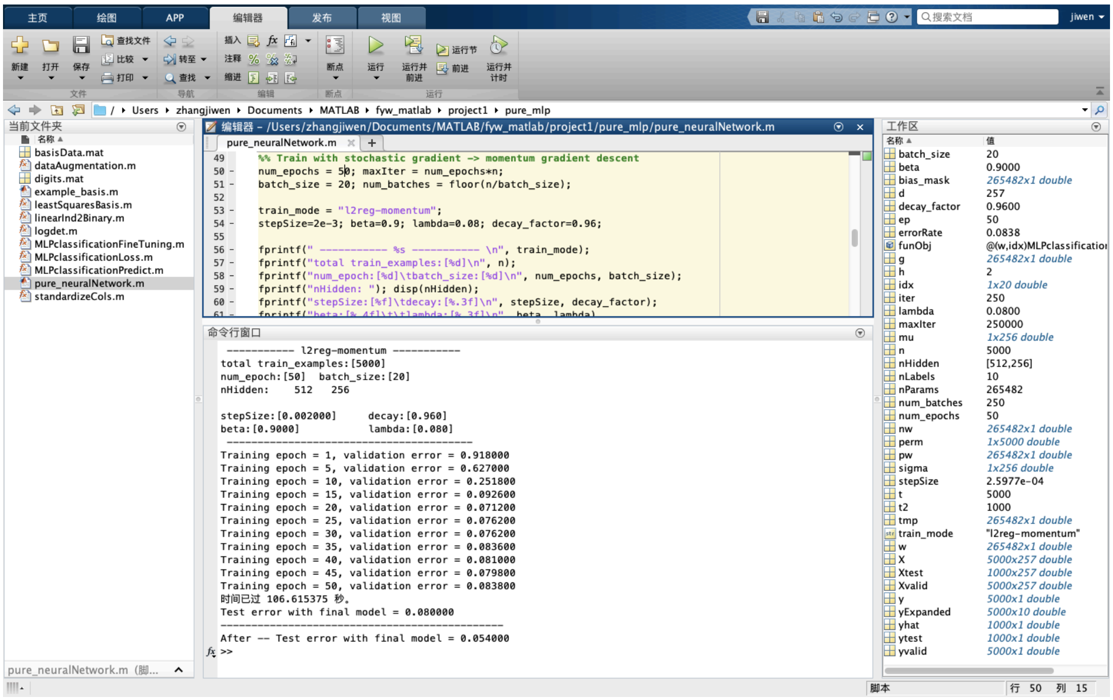
* kernel_size=5; batch_size=20; beta=0.9, decay_factor = 0.96;

Conclusion

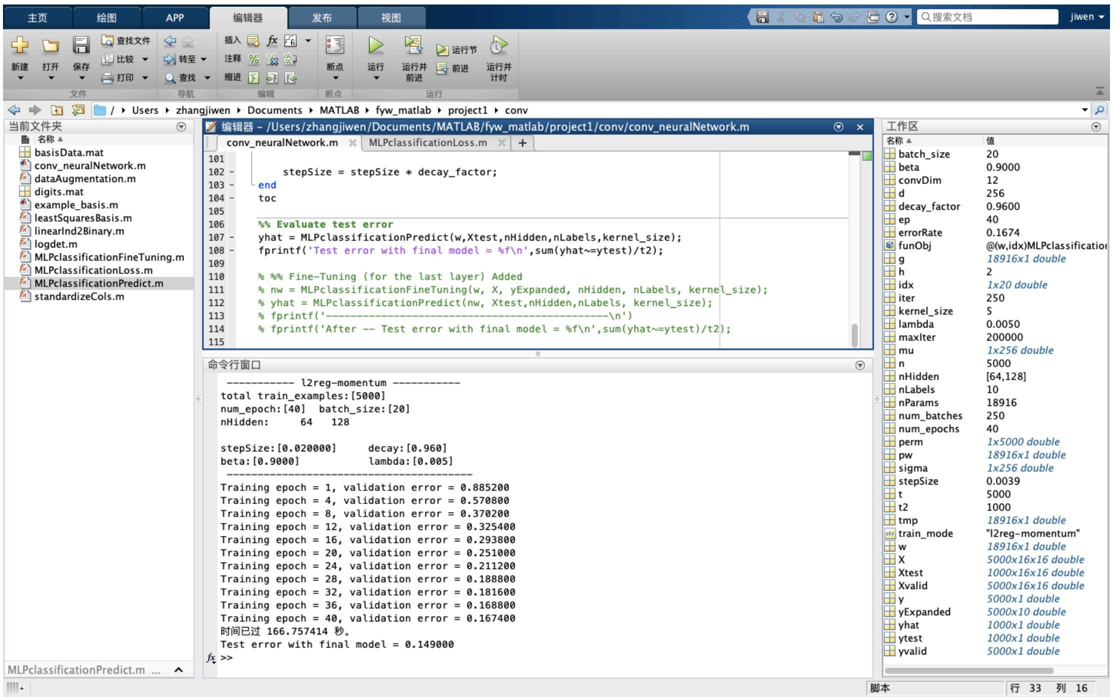
Final codes are packaged into a zip file, which contains two folder named “pure_mlp” and “conv_mlp”.

- Folder “**pure_mlp**” stores the modification 1~9 described in this report.
- Folder “**conv_mlp**” stores the modification 10 described in this report.

The best error generated by **pure_mlp** is **0.0540** after fine-tuning.



The best error generated by **conv_mlp** is **0.1490** bofore fine-tuning.



In short, the **best test error** we get is **0.0540**.