Lirane Bitton

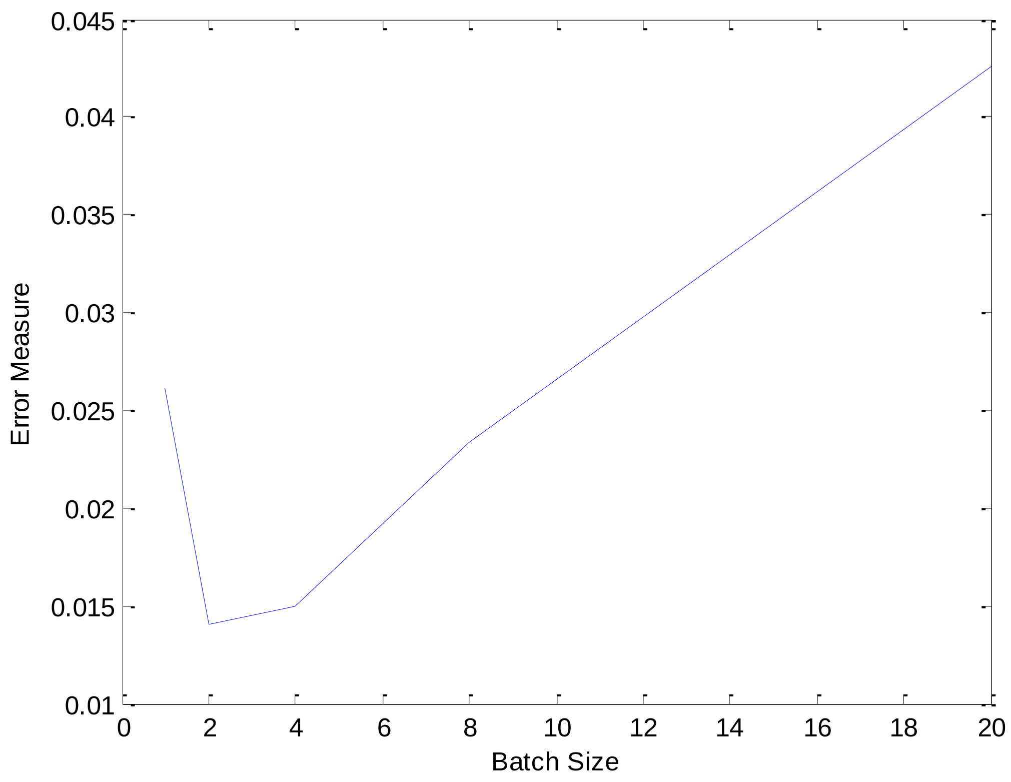
200024677

MATLAB Assignment 4:

Convolutional Neural Networks

Part 1:

Produce a plot of these train errors as a function of the batch size and attach it to your report. What is the optimal batch size? Explain the tradeoff that comes into play here. How do you expect the complexity of the dataset (image size, number of classes, intra-class appearance variability etc.) to affect the optimal batch size?



Batch size	1	2	4	8	20
Best result	0.0262	0.0141	0.0150	0.0234	0.0426

According to the above results, the optimal batch size is 2. The tradeoff that came into play here is between the batch sizes to the number of SGD iterations. Higher the batch size is smaller the number of iteration for learning is. Obviously, a low number of iteration doesn't allow us to reach to a convergence therefore we got bad results for higher batch sizes.

If we are increasing the number of classes or image size, we would expect to get a higher number of connection in the network. Thus leading to an increased dimension distribution resulting in a more complex gradient direction requiring the use of larger batches. And inversely for smaller batches.

Considering the intra-class appearance, if we have trained on a set of MINST containing only two of the 10 numbers, the network would converge gradually into weights that maximize these classes, minimizing the error and allowing the use of smaller batches.

Part 2:

Add to your report the train error and best learning rate for each momentum value. Can you explain why higher momentum values require lower learning rates?

Momentum	0.9	0.95	0.99
Learning rate	1e-3	5e-4	1e-4
Train error	0.0241	0.0167	0.0318

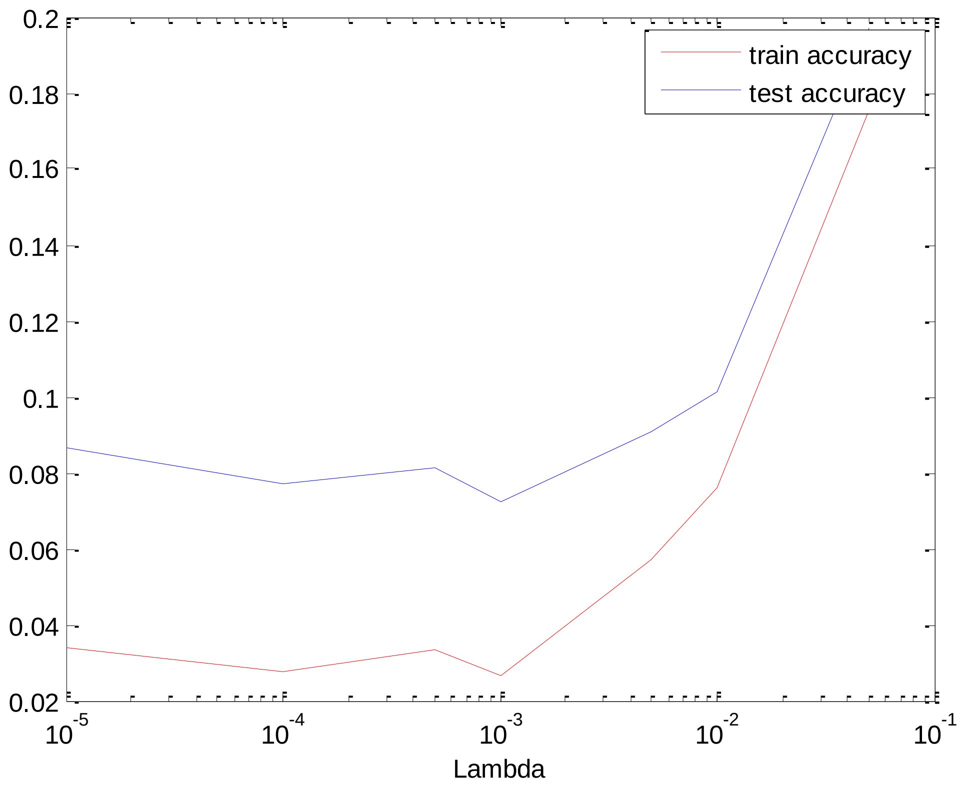
Let remind the Nesterov acceleration:

$$w^{t+1} = w^t + v^{t+1}$$
$$v^{t+1} = \mu \cdot v^t - \eta \nabla f(w^t)\ s.t. \mu \in [0,1]$$

We can see that the momentum is actually remembering the previous value of the update, whereas the learning rate is connected to the gradient of the new w. Considering that f is smooth in close points w, we get that its gradient in those points is similar and need a small correction. This correction is given by the learning rate. In other word more important the previous step (momentum) is, lower is the correction we need to fit the current gradient (learning rate).

Part 3:

Plot the resulting train and test errors as a function of λ. How does this plot compare to those you are used to seeing in classical machine learning?



regularization_param	0(1e-5)	1e-4	5e-4	1e-3	5e-3	1e-2	5e-2
avg_train_errors(of 5 runs)	0.0340	0.0275	0.0333	0.0263	0.0572	0.0762	0.1758
avg_test_errors (of 5 runs)	0.0868	0.0772	0.0812	0.0722	0.0908	0.1014	0.1969

In classical machine learning, we've seen that the regularization factor λ prevents from overfitting by regulating w and therefore prevents the creation of a large difference in the error of classifying train and test sets. Here in our network, the regularization also improved the training error thus also the optimization process.

Remark on the graph: the 0 lambda is set to 1e-5 since I chose to represent a logarithmic scale.

Part 4:

Try replacing 'Xavier' initialization of affine weights in *template_script.m* with 'Zeros'. What happened? Does this phenomenon occur in classical machine learning as well?

Initialization	Train error	Test error
'Xavier'	0.0204	0.0717
'Zeros'	0.8892	0.8865

We see that both the train and test error are much better when initializing with 'Xavier'.

We can explain this phenomenon with two theories. First we reached a minimum but the function is not convex thus this minimum is local and the zero initialization false our training stage. The other assumption can be that we didn't reach a minimum the weak convex property pointed us far from optimization and the iteration number was not enough to reach a better result.

In classical machine learning, the initialization has less effect than here since the algorithms guarantee to converge to an optimal solution.

Part 5:

How does this affect the training error? Can you provide an intuitive explanation for this?

The best train error I got was 0.0096. We can see that it's lower than the results got above meaning that the improved η was better. I suppose that the after 75% of the iteration (7500 of 10000) we are very close to the minimum point and in order to reach it we don't need any more large steps but smaller. An order of magnitude less allowed to affine the steps and reach a more accurate minimum.

Part 6:

Add to your report the selected configuration for these hyper-parameters, as well as the resulting test error and its corresponding train error.

```
batch_size = 1;
T      = 2e4;
mu     = single(0.97);
lambda = single(5e-4);
eta    = @(t)((t > 7500)*1e-4 + (t <= 7500)*(5e-4));
```

The results of 5 runs:

Train error	0.0040	0.0064	0.0064	0.0068	0.0048
Test error	0.0460	0.0558	0.0520	0.0472	0.0484

Part 7:

What happened to the train and test errors? How does this relate to the well-known bias-variance tradeoff of classical machine learning?

Train accuracy small network: 0

Test accuracy small network: 0.05

Train accuracy big network: 0

Test accuracy big network: 0.0311

In our case the test error got smaller after training on the large network while the training error stay the same. Obviously the new network is more complex since it contains an additional conv->ReLU->pool layer. The tradeoff here is creating a network that can satisfy the training without being too much specific in order to classify new data. Here we can see that our model is a bit more complex than we can express with a small convex thus we can improve the test error by creating more complex network and by allowing greater variance leading to a better classification. Usually greater variance lead to overfitting but here we manage it in order to stay in the best classification area.

Part 8:

```
batch_size = 2;part one
T      = 1e4;
mu     = single(0.9);
lambda = single(5e-4);
eta    = @(t)(5e-3 * (t <=  (0.84*T))  + (t > (0.84*T)) * 5e-
4);part 5
pool_kernel=3;
conv_channels_second=50;(second convolution)
```

results =

```
   0.0004   0.0016       0        0.0004   0.0020

   0.0336   0.0298   0.00359   0.0274   0.0271
```

We can see that the best train error is 0 and the corresponding test error is 0.0359

Bonus Part:

In order to implement the bonus I added a dropout case to the constructor in addition to the forward pass and backward pass functions in ConvNet.m. The constructor receive the dropout probability parameter p. During the learning step, I used a train flag to differentiate between learning from simple use of the network, insuring proper use of the probability parameter.

The forward pass function the neuron is dropped at probability p in learning and multiplied by 1-p in testing. In the backward pass the neuron is multiplied by 1-p if the neuron isn't already dropped.

To test the new case I used the same parameters as in part 8 and tuned the dropout probability to be 0.3 (in the publication the probability of retaining was between 0.5 to 0.8 so I decided to use 0.3 to drop)

Results =

```
0.0008   0.0020   0.0024   0.0008       0

0.0279   0.0300   0.0318   0.0299   0.0272
```

We can see that the best train error is 0 and the corresponding test error is 0.0272. As expected with the same parameters we got a better test error.