Machine Learning Practical

Coursework 1 report

Jian Hou - 2016年10月27日

Part 1: Learning rate schedules

Part 2: Momentum learning rule

Part 3: Adaptive learning rules

Preparation

Firstly, in order to make the result of my experiments more easily comparable. I used the **same model architecture**, particularly using **three affine transformations** interleaved with **logistic sigmoid nonlinearities**, and a **softmax output layer**. And I used the **same parameter initialisation method**, particularly initialised the biases to **zero** and used **GlorotUniformInit** to initialise the weight. I used a **batch size of 50** and train for a total of **100 epochs** for all reported runs.

Then I defined a function which is used to train model and plot stats. This function can plot different lines which using the different parameters of learning rules on one figure to show the error and accuracy values for training and validation sets.

Finally, I set up the data providers, random number generator, logger objects needed for training runs.

Part 1: Learning rate schedules

In the first part of the assignment I investigate how using a time-dependent learning rate schedule influences training.

Method and algorithms

I choose **reciprocal** to be the **time-dependent learning rate schedules** and the calculation formula is shown below:

$$\eta(t) = \eta_0 (1 + t / r)^{-1}$$

where η_0 is the **initial learning rate** (In the codeI declare it as a variable **learning rate** and assign the learning rate value to it), **t** the **epoch number**, **η(t)** the learning rate at epoch t and **r** a free parameter governing how quickly the learning rate **decays**.(In the code I declare it as **decaying_ rate**)

I implemented the schedule by creating a new scheduler class in the *mlp.schedulers.py* module by defining a *update_learning_rule* method which sets the *learning_rate* attribute of a learning rule object based on the formula above and defining *__init__* method to initialise *learning_rate* and *decaying_rate*.

Experiment 1.1

Aims:
Compare the performance of **time-dependent learning rate schedule** and **constant learning rate baseline** when training the standard model on the MINIST digit classification task.

Quantitative results:

Here is the case where I use the time-dependent learning rate schedule and the constant learning rate baseline and they are used at the **same decaying rate***(decaying_rate = 40)* and **same learning rate** for **five different learning rates** *(learning rate = [0.001, 0.05, 0.1, 0.5, 0.9]*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

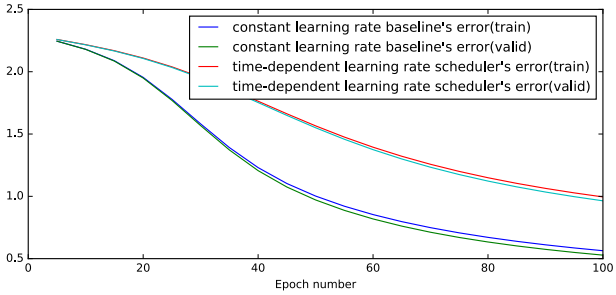

Figure 1.1.1.1, learning rate = 0.001

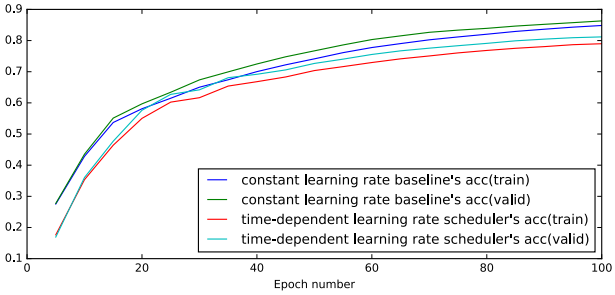

Figure 1.1.1.2, learning rate = 0.001

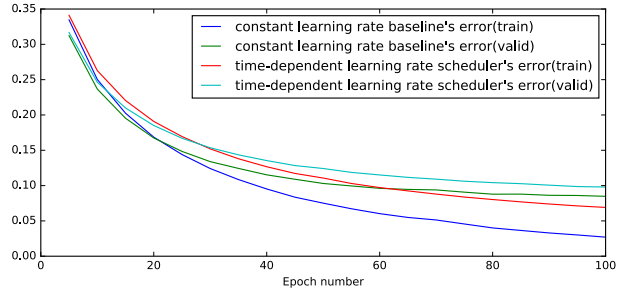

Figure 1.1.2.1, learning rate = 0.05

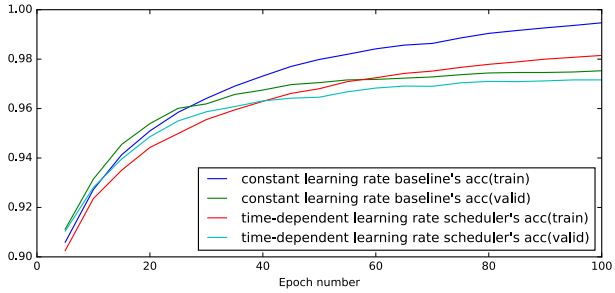

Figure 1.1.2.2, learning rate = 0.05

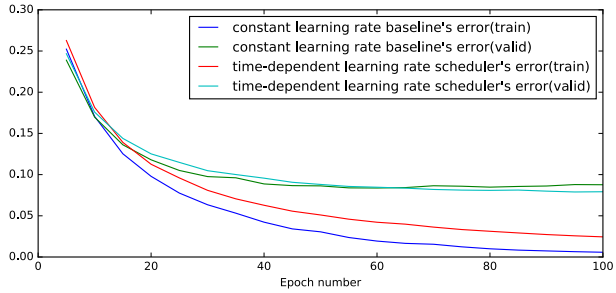

Figure 1.1.3.1, learning rate = 0.1

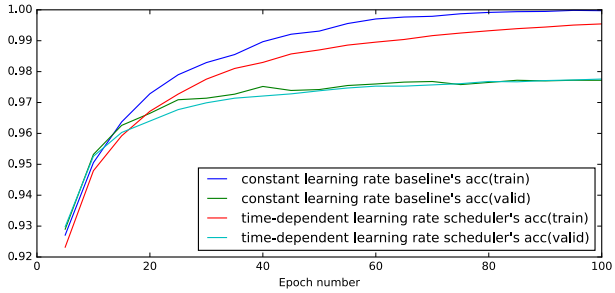

Figure 1.1.3.2, learning rate = 0.1

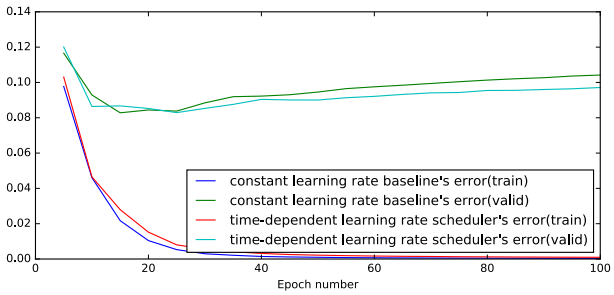

Figure 1.1.4.1, learning rate = 0.5

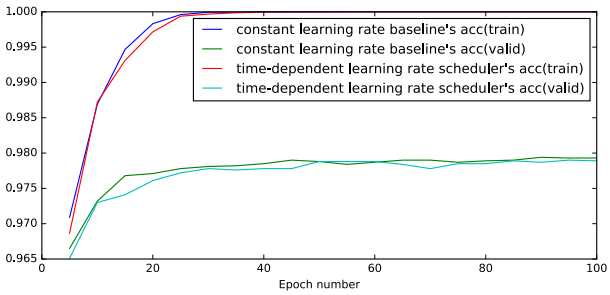

Figure 1.1.4.2, learning rate = 0.5

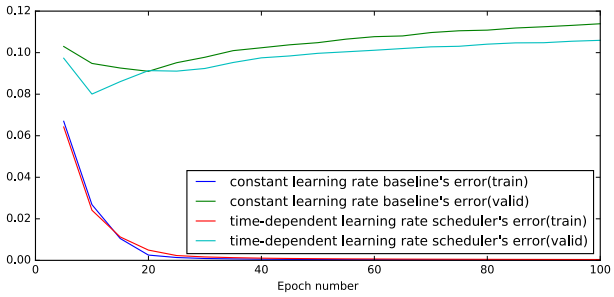

Figure 1.1.5.1, learning rate = 0.9

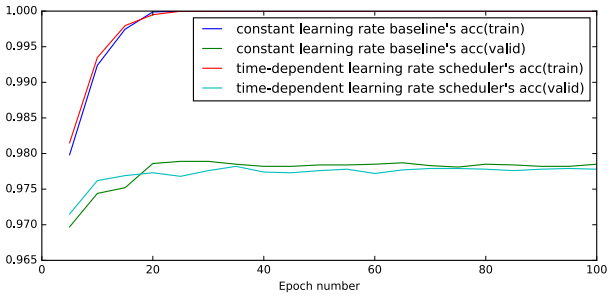

Figure 1.1.5.2, learning rate = 0.9

When learning rate = 0.001:

	Constant learning rate baseline	time-dependent learning rate schedule
final error(train)	5.64E-01	9.95E-01
final error(valid)	5.28E-01	9.64E-01
final acc(train)	8.48E-01	7.9E-01
final acc(valid)	8.63E-01	8.12E-01

When learning rate = 0.05:

	Constant learning rate baseline	time-dependent learning rate schedule
final error(train)	5.64E-01	9.95E-01
final error(valid)	5.28E-01	9.64E-01
final acc(train)	8.48E-01	7.9E-01
final acc(valid)	8.63E-01	8.12E-01

When learning rate = 0.1:

	Constant learning rate baseline	time-dependent learning rate schedule
final error(train)	5.64E-01	9.95E-01
final error(valid)	5.28E-01	9.64E-01
final acc(train)	8.48E-01	7.9E-01
final acc(valid)	8.63E-01	8.12E-01

When learning rate = 0.5:

	Constant learning rate baseline	time-dependent learning rate schedule
final error(train)	5.64E-01	9.95E-01
final error(valid)	5.28E-01	9.64E-01
final acc(train)	8.48E-01	7.9E-01
final acc(valid)	8.63E-01	8.12E-01

When learning rate = 0.9:

	Constant learning rate baseline	time-dependent learning rate schedule
final error(train)	5.64E-01	9.95E-01
final error(valid)	5.28E-01	9.64E-01
final acc(train)	8.48E-01	7.9E-01
final acc(valid)	8.63E-01	8.12E-01

Discussion and conclusion:

From the figures of 1.1.1.1 - 1.1.5.2 we can see that when the learning rate is less than 0.5, the accuracy (valid) value with time-dependent learning rate schedule method is lower than one without the schedule method, and the error (valid) value is higher. But when the learning rate is more than 0.5, these two values with and without schedule method become more similar.

The difference of the accuracy on training set and validation set with and without scheduler method is growing with the learning rate increases. As well as we can see that when learning rate is more than 0.5, there is an overfitting. The accuracy on validation set decrease.

So the performance decreased rapidly when learning rate is more than 0.5. And the performance of using scheduler method and not using is similar.

Experiment 1.2

Aims:

I**ndicate** how the different **η0(learning rate)** affects the evolution of the training.

Quantitative results:

Here is the case where I use the time-dependent learning rate schedule and it is used at the **same decaying rate**(*decaying rate = 40*) and the **same learning rate** for **five different learning rates** *(learning rate = [0.001, 0.05, 0.1, 0.5, 0.9]*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

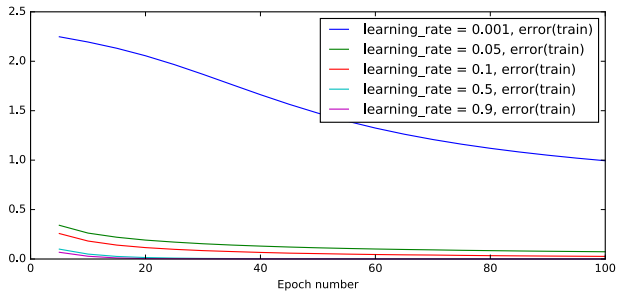

Figure 1.2.1, error(train)

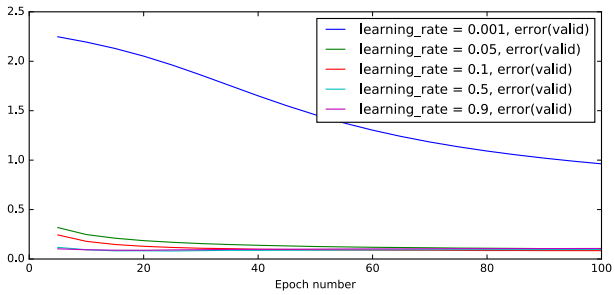

Figure 1.2.2, error(valid)

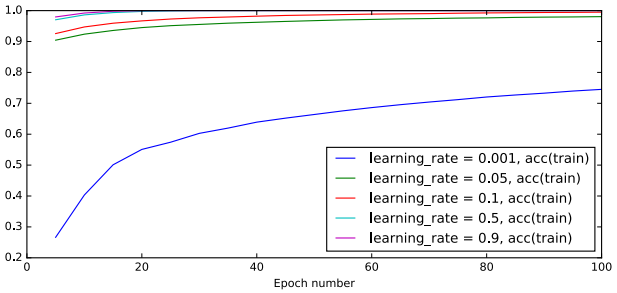

Figure 1.2.3, acc(train)

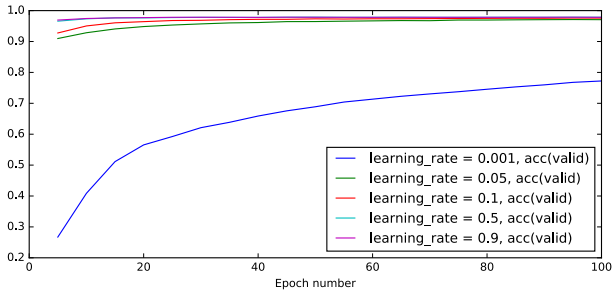

Figure 1.2.4, acc(valid)

When decaying rate = 40 and learning rate = [0.001, 0.05, 0.1, 0.5, 0.9]:

	0.001	0.05	0.1	0.5	0.9
final error(train)	9.94E-01	7.33E-02	2.65E-02	9.69E-04	4.09E-04
final error(valid)	9.63E-01	1.03E-01	8.43E-02	9.63E-02	1.06E-01
final acc(train)	7.45E-01	9.81E-01	9.95E-01	1.00E+00	1.00E+00
final acc(valid)	7.72E-01	9.71E-01	9.76E-01	9.79E-01	9.78E-01

Discussion and conclusion:

From the figures of 1.2.1 - 1.2.4 we can see that when the learning rate is in the range of 0.001 to 0.9, the accuracy and error values are very close to each other except for learning rate equal 0.001 where accuracy value is much lower than the other cases. However, we found that when the epoch number is less than 20, the accuracy is very close to the 1.

So that the learning rate has little effect in the case of the train using the scheduler method except for a very little value of learning rate, but in this case training will be early-stopping in advance, the performance is not good

Experiment 1.3

Aims:

Indicate how the different **r(decaying rate)** affects the evolution of the training.

Quantitative results:

Here is the case where I use the time-dependent learning rate schedule and it is used at the **same learning rate**(*learning rate = 0.1*) and the **same decorate** for **five different decaying rates** (*decaying rate = [1, 5, 10, 50, 100]*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

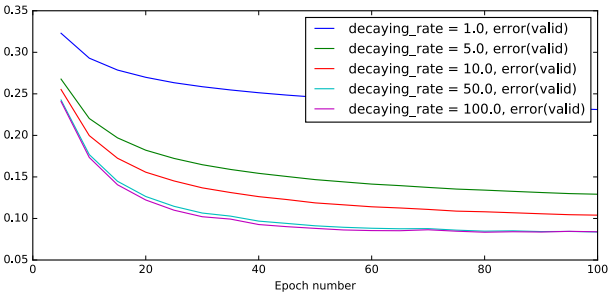

Figure 1.3.1, error(train)

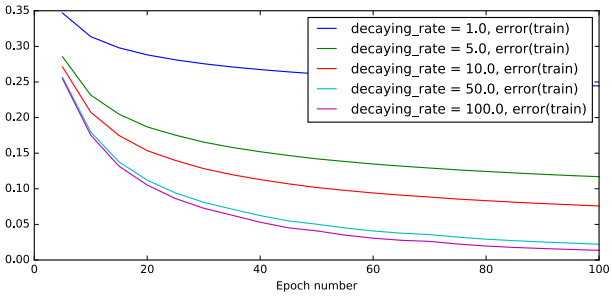

Figure 1.3.2, error(valid)

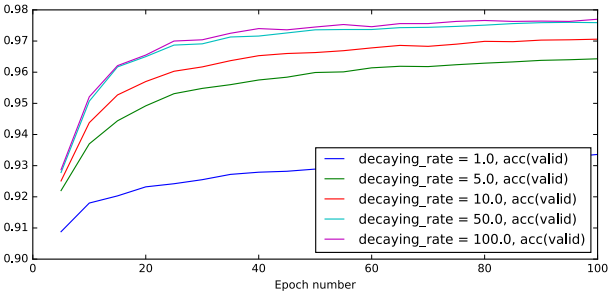

Figure 1.3.3, acc(train)

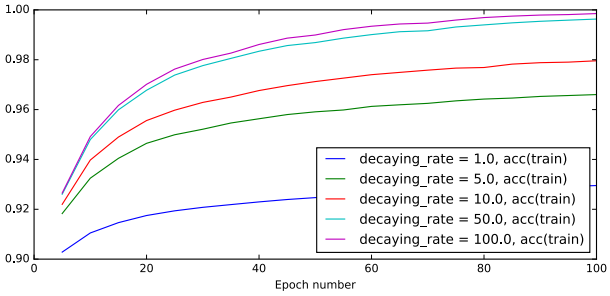

Figure 1.3.4, acc(valid)

When learning rate = 0.1 and decaying rate = [1, 5, 10, 50, 100]:

	1	5	10	50	100
final error(train)	2.44E-01	1.17E-01	7.59E-02	2.22E-02	1.36E-02
final error(valid)	2.31E-01	1.29E-01	1.04E-01	8.38E-02	8.40E-02
final acc(train)	9.3E-01	9.66E-01	9.8E-01	9.96E-01	9.99E-01
final acc(valid)	9.34E-01	9.64E-01	9.71E-01	9.76E-01	9.77E-01

Discussion and conclusion:

From the figures of 1.3.1 - 1.3.4 we can see that When the decaying rate from 1 began to increase, accuracy is gradually becoming larger, error is gradually reduced, and according to the table we also see the accuracy on validation set and on training set is very close.

At the same time can be seen when the decaying rate to greater than 50, the accuracy curve and error curve is very close.

Summary, when the decay rate increases, the performance is getting better. When the decay rate is greater than or equal to 50, although the performance is getting better, but the performance increase has become very small.

Part 2: Momentum learning rule

In this part of assignment I investigate suing a gradient descent learning rule with momentum.

Method and algorithms

This extends the basic gradient learning rule by introducing extra momentum state variables for the parameters. This way is called **momentum learning rule**. In particular one possible schedule is shown below:

$$\alpha(t) = \alpha_{\infty}(1 - \frac{\gamma}{t+\tau})$$

where $\alpha_{\infty} \in [0,1]$ determines the asymptotic **momentum coefficient**(In the code I declare it as **mom_coeffs**) and **τ>= 1** and **0<=γ<=τ** determine the initial momentum coefficient and how quickly the coefficient tends to α(In the code I declare them as **initial_mom** and **coeff_tend_speed**).

I implement the momentum learning rule by using a *MomentumLearningRule* class in the *mlp.learning_rules.py module* and create an *MomentLearningRateScheduler* class which is similar to the *TimedependentLearningRateScheduler.*

Experiment 2.1

Aims:
Compare the performance of **basic gradient descent learning rule** and **momentum learning rule** for several values of the **momentum coefficient** when training the standard model on the MINIST digit classification task

Quantitative results:
Here is the case where I use the **basic gradient descent learning rule** and **momentum learning rule** and they are used at the **same learning rate**(*learning rate = 0.1*) and the **same momentum coefficient** for **five different momentum coefficient** (*momentum coefficient = [0.1, 0.2, 0.5, 0.7, 0.9]*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

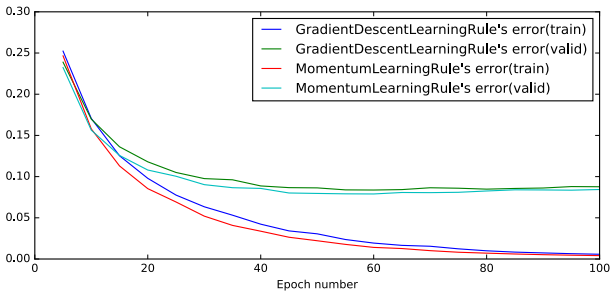

Figure 2.1.1.1, *momentum coefficient* = 0.1

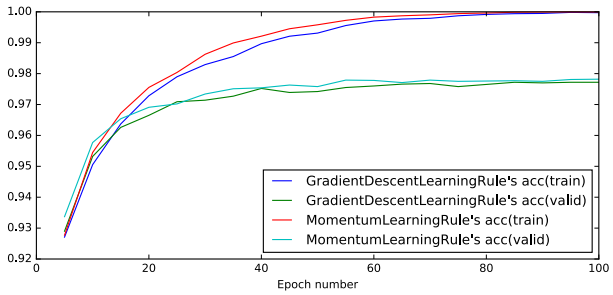

Figure 2.1.1.2, *momentum coefficient* = 0.1

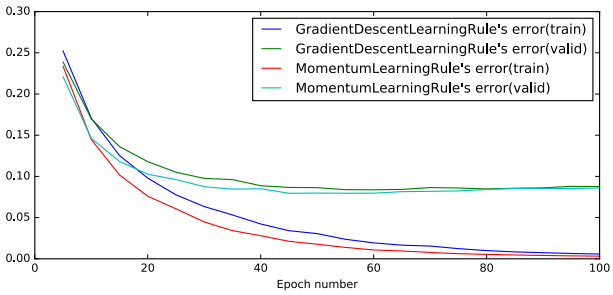

Figure 2.1.2.1, *momentum coefficient* = 0.2

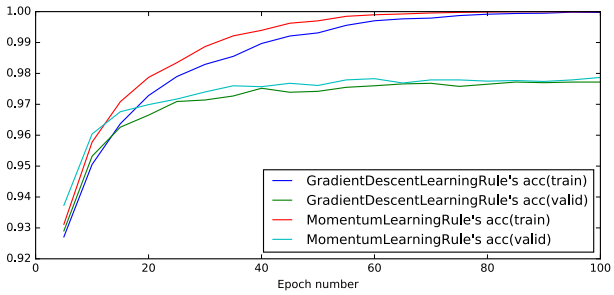

Figure 2.1.2.2, *momentum coefficient* = 0.2

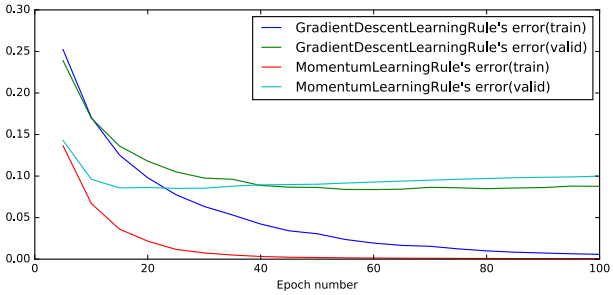

Figure 2.1.3.1, *momentum coefficient* = 0.5

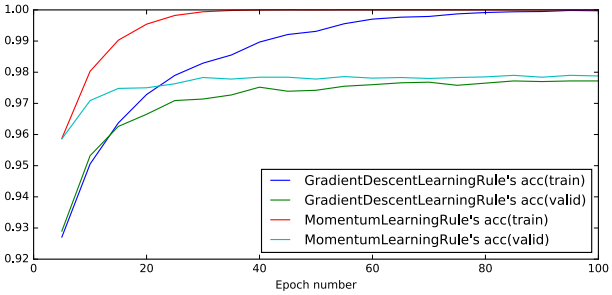

Figure 2.1.3.2, *momentum coefficient* = 0.5

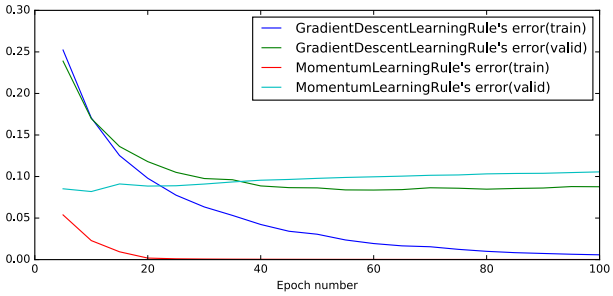

Figure 2.1.4.1, *momentum coefficient* = 0.7

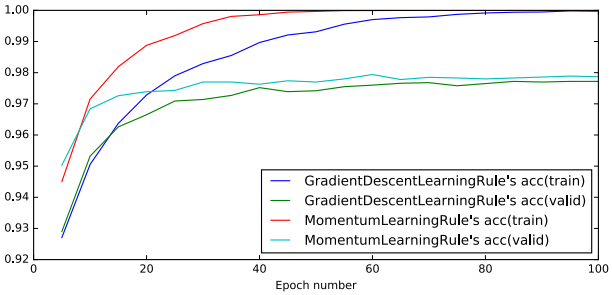

Figure 2.1.4.2, *momentum coefficient* = 0.7

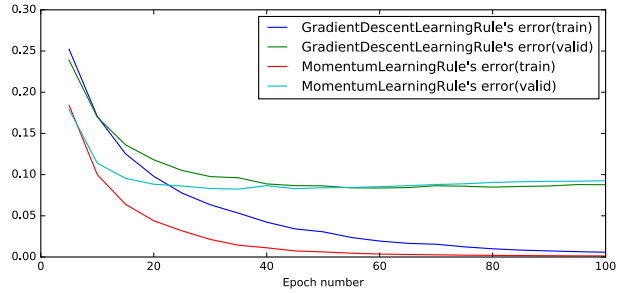

Figure 2.1.5.1, *momentum coefficient* = 0.9

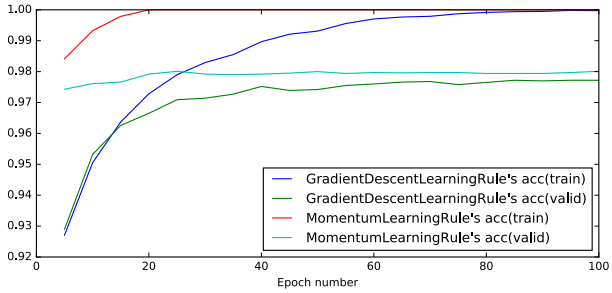

Figure 2.1.5.2, *momentum coefficient* = 0.9

Discussion and conclusion:

From the figure 2.1.1.1 - 2.1.5.2, we can see that using the momentumlearningrule and using gradientdescentlearningrule their accuracy and error curves are consistent, and when the momentum coefficient becomes larger, the accuracy increase to 1 which the required epoch number becomes less.

So using the momentumlearningrule and using gradientdescentlearningrule their performance is similar. But when momentum coefficient is greater than 0.5, there will be situation of early-stopping and performance began to deteriorate.

Experiment 2.2

Aims:

Indicate how the different **momentum coefficient** affects the evolution of the training.

Quantitative results:

Here is the case where I use the **momentum learning rule** and it is used at the **same learning rate**(*learning rate = 0.1*) and the **same momentum coefficient** for **five different momentum coefficient** (*momentum coefficient = [0.1, 0.2, 0.5, 0.7, 0.9]*). Also set the *initial_mom* = 10 and *coeff_tend_speed* = 5. And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

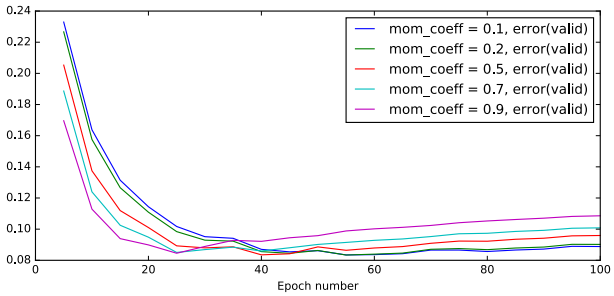

Figure 2.2.1, error(train)

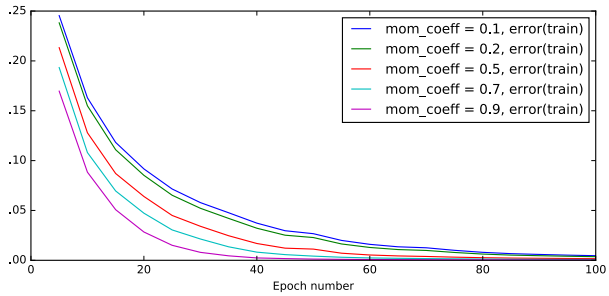

Figure 2.2.2, error(valid)

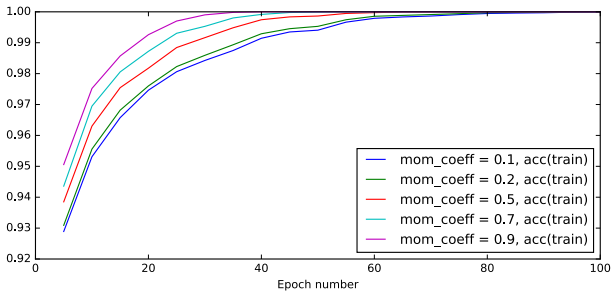

Figure 2.2.3, acc(train)

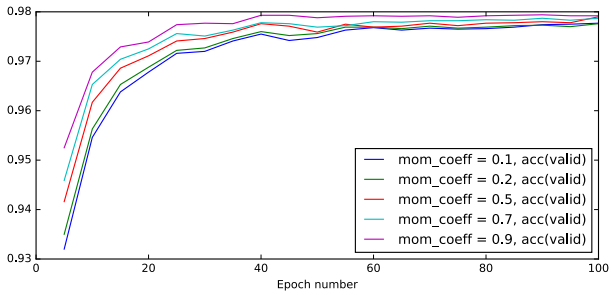

Figure 2.2.4, acc(valid)

Discussion and conclusion:
From the figures above, we can see that when mom_coeff becomes larger, accracy becomes larger, error decreases.

So when mom_coeff becomes larger, the performance improves.

Experiment 2.3

Aims:
Indicate how the different **initial momentum coefficient** affects the evolution of the training.

Quantitative results:
Here is the case where I use the **momentum learning rule** and it is used at the **same learning rate**(*learning rate = 0.1*) and the **same initial momentum coefficient** for **five different initial momentum coefficient** (*initial_mom = [0.1, 0.2, 0.5, 0.7, 0.9]*). Also set the *mom_coeff* = 0.2 and *coeff_tend_speed* = 2. And show the evolution of the **error** and

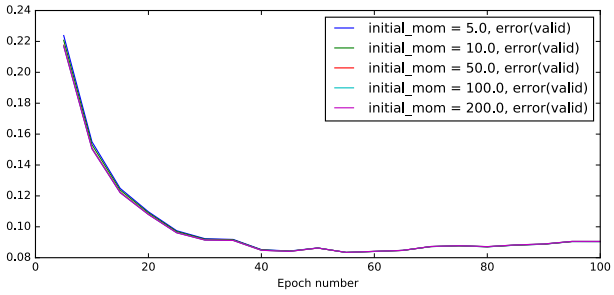

Figure 2.3.1, error(train)

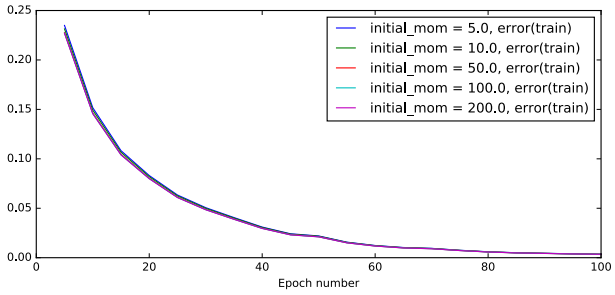

Figure 2.3.2, error(valid)

accuracy for both the **training** and **validation** sets.

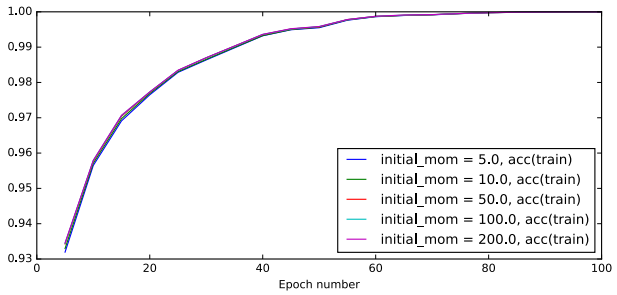

Figure 2.3.3, acc(train)

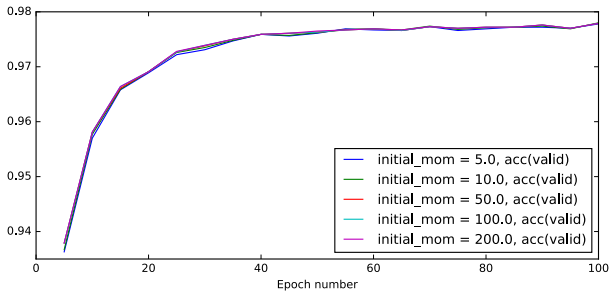

Figure 2.3.4, acc(valid)

Discussion and conclusion:
From the figures above, we can see that when the initial_mom changes, we can see that the curve is almost unchanged.

So we conclude that initial_mom has no effect on performance

Experiment 2.4

Aims:
Indicate how the different **coefficient tends speed** affects the evolution of the training.

Quantitative results:
Here is the case where I use the **momentum learning rule** and it is used at the **same learning rate**(*learning rate = 0.1*) and the **same coefficient tend speed** for **five different initial coefficient tend speed** (*coeff_tend_speed = [1, 5, 10, 20, 50]*). Also set the *mom_coeff* = 0.2 and *initial_mom* = 50. And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

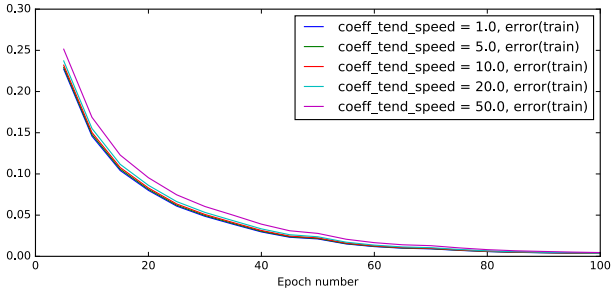

Figure 2.4.1, error(train)

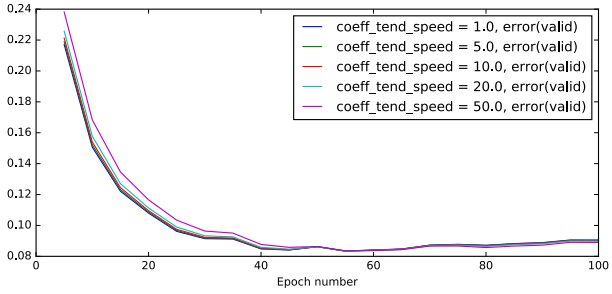

Figure 2.4.2, error(valid)

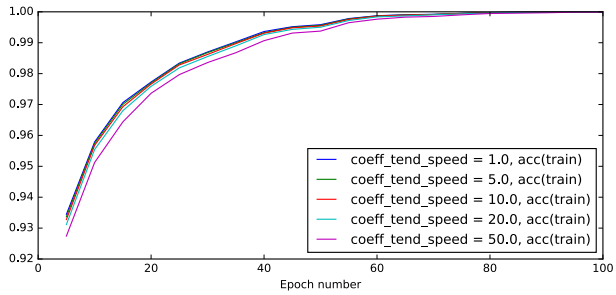

Figure 2.4.3, acc(train)

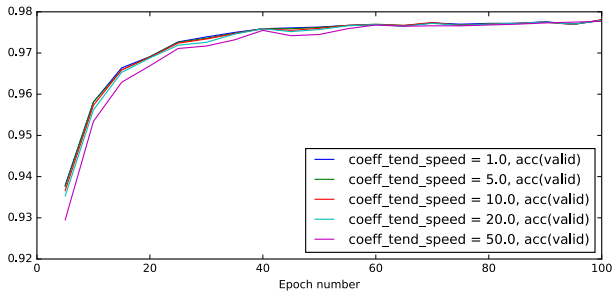

Figure 2.4.4, acc(valid)

Discussion and conclusion:
From the figures above, when the initial_mom changes, we can see that the curve is almost unchanged.

So we conclude that initial_mom has no effect on performance

Part 3: Adaptive learning rule

In the final part of the assignment I investigate adaptive learning rules which attempt to automatically tune the scale of updates in a parameter-dependent fashion.

Method and algorithms

I use two adaptive learning rules called **AdaGrad** and **RMSProp**. In particular there are two calculation given below:
AdaGrad:
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)

Notice that the variable cache has size equal to the size of the gradient and eps usually set somewhere in range from 1e-4 to 1e-8

RMSProp:
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)

Decay_rate is a hyperparameter and typical values are [0.9, 0.99, 0.999].

I implement the learning rule by creating *AdaGradLearningRule* class and *RMSPropLearningRule* classin the *mlp.learning_rules.py module which I refer to the MomentumLearningRule class. They are use additional state variables to calculate the updates to the parameters.*

Experiment 3.1

Aims:
Compare the performance of **AdagradLearningRule** and **RMSpropLearningRule** when training the standard model on the MINIST digit classification task

Quantitative results:
Here is the case where I use the **AdagradLearningRule** and **RMSpropLearningRule. There are** used at the **same learning rate**(*learning rate = 0.1*) and the **same decay rate**(*decay_rate = 0.9*) as well as the **same eps**(*eps = 1e-8*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

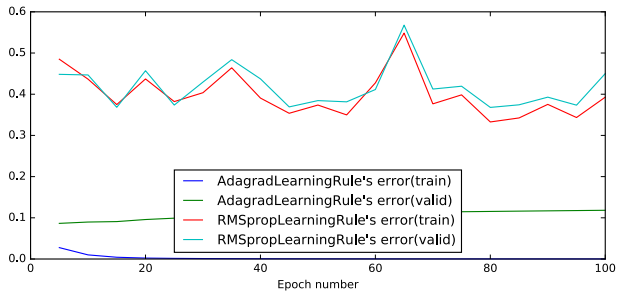

Figure 3.1.1, acc(train)

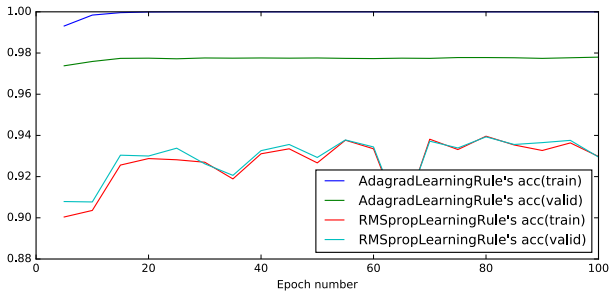

Figure 3.1.2, acc(valid)

Name	Final error(train)	Final error(valid)	accuracy(train)	accuracy(valid)	Run time per epoch
GradientDescent	5.77E-03	8.77E-02	1.00E+00	9.77E-01	0.98s
Momentum	4.11E-03	8.43E-02	1.00E+00	9.78E-01	0.98s
AdaGrad	1.85E-04	1.18E-01	1.00E+00	9.78E-01	1.29s
RMSProp	3.93E-01	4.5E-01	9.3E-01	9.29E-01	1.68s

Discussion and conclusion:

From the figure 3.1.1, 3.1.2, 2.1.1.1, 2.1.1.2 and form above, we can see that the fastest speed of convergence are gradientdescen and momentum one. And RMSProp did not reach the accuracy 1.0.

So I think RMSProp's performance is very bad for training, other learning rate rule performance is normal.

Experiment 3.2

Aims:

Indicate how the different **eps** affects the evolution of the training by using **AdagradLearningRule**

Quantitative results:
Here is the case where I use the **AdagradLearningRule** and it is used at the **same learning rate**(*learning rate = 0.1*) and the **same decay rate**(decay_rate = *0.1*) as well as the **same eps** for **five different epses** *(eps= [1e-4, 1e-5, 1e-6, 1e-7, 1e-8]*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

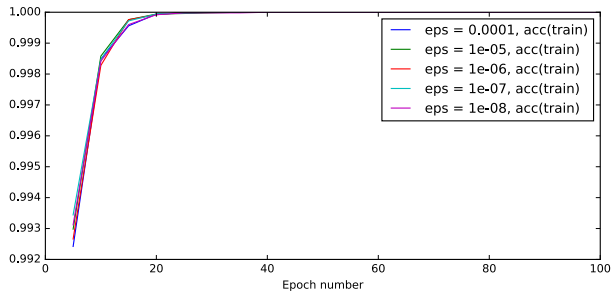

Figure 3.2.1, error(train)

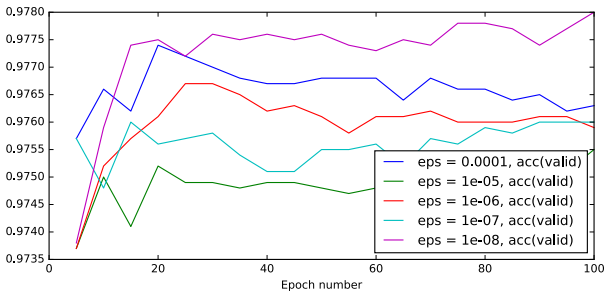

Figure 3.2.2, error(valid)

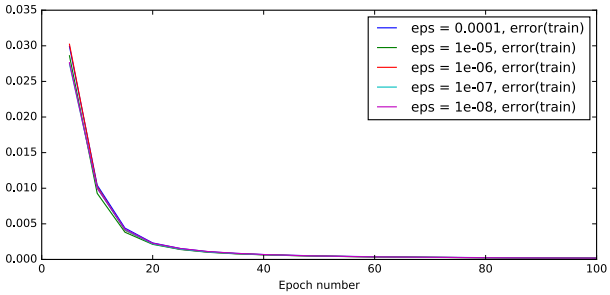

Figure 3.2.3, acc(train)

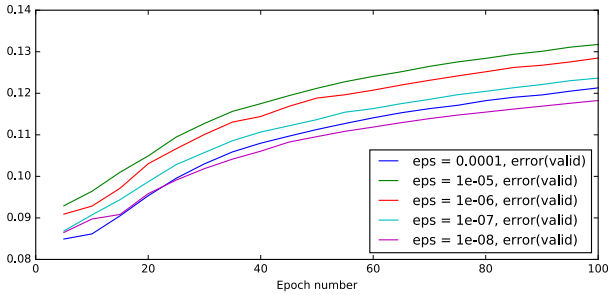

Figure 3.2.4, acc(valid)

Discussion and conclusion:
From the figures above, we can see that when eps changes, the values of accuracy and error are similar and the curve is almost unchanged.

So wen conclude that eps has no effect on performance.

Experiment 3.3

Aims:
Indicate how the different **decay rate** affects the evolution of the training by using **RMSpropLearningRule**.

Quantitative results:
Here is the case where I use the **RMSpropLearningRule** and it is used at the **same learning rate**(*learning rate = 0.1*) and the **same eps**(eps = *1e-8*) as well as the **same decay rate** for **five different decay rates**(*decay_rate= [0.2, 0.5, 0.9]*). And show the evolution of the **error** and **accuracy** for both the **training** and **validation** sets.

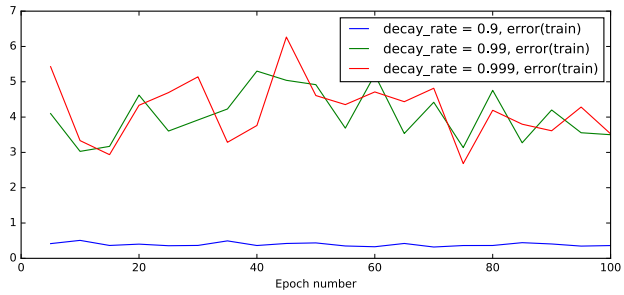

Figure 3.3.1, error(train)

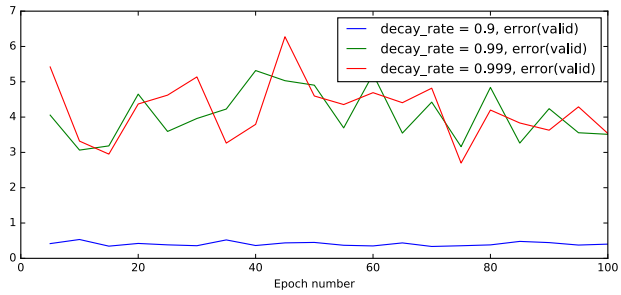

Figure 3.3.2, error(valid)

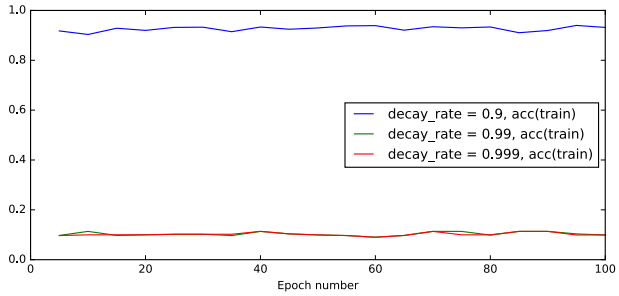

Figure 3.3.3, acc(train)

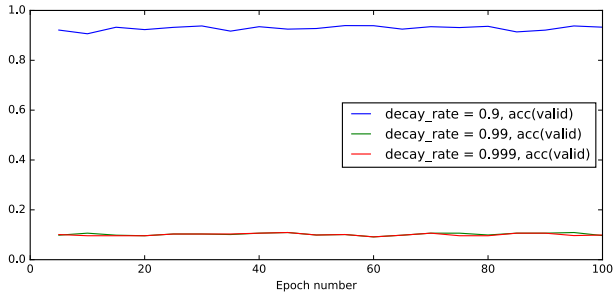

Figure 3.3.4, acc(valid)

Discussion and conclusion:

From the figures above, we can see that when decay rate is equal to 0.99 and 0.999 ,the accuracy is very small, error is great. But when decay rate equal to 0.9, these values are normal.

So when the decay rate is equal to 0.9 when the performance is normal