MLP Coursework 1

Name: Wenzhe Liu Student Number: s1631854

Part 1: Learning rate schedules

1. description of the methods used and algorithms implemented

   a) I choose to implement reciprocal algorithms. So I implement the algorithm as the formula shows: $\eta(t) = \eta_0(1 + t/r)^{-1}$.

   b) I create a new class named ReciprocalLearningRateScheduler which is inherited from class ConstantLearningRateScheduler in mlp.schedulers.py module.

   c) I define a __init__ function to initialize the parameters. I call the super function to complete my learning rate parameter initialization and add one more line to initialize my new parameter called decay_rate(r).

   d) I also define a updata_learning_rule function to update learning rule. Just as the formula shows, learning_rule.learning_rate = self.learning_rate / float(1 + epoch_number/self.decay_rate).

   e) Using the module just implemented and new an array of schedulers and make use of GradientDescentLearningRule. Finally, show the plots via train_model_and_plot_stats function.

2. results for the experiments you carried out including relevant graphs

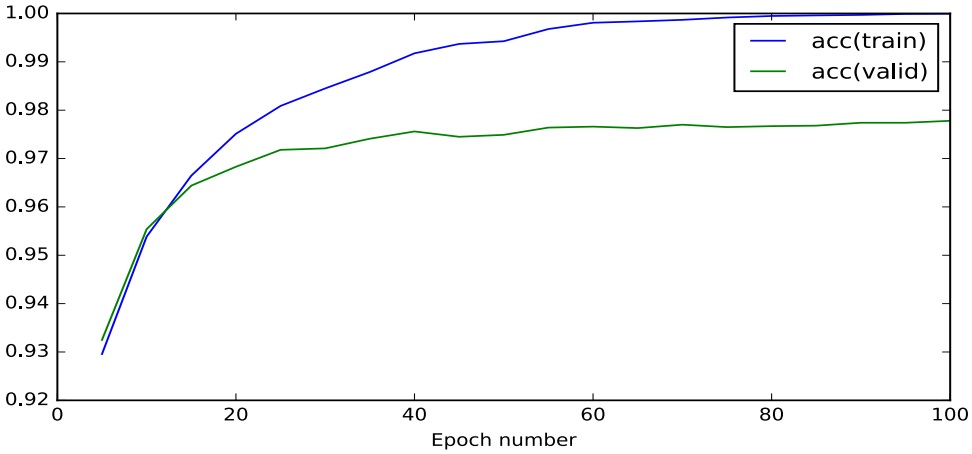


fig 1-1 ConstantLearningRate, learning rate = 0.1, accuracy

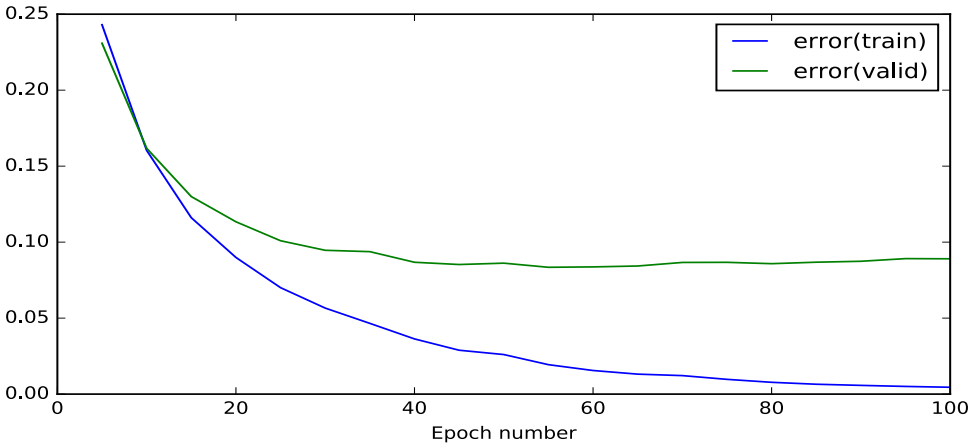


fig 1-2 ConstantLearningRate, learning rate = 0.1, error

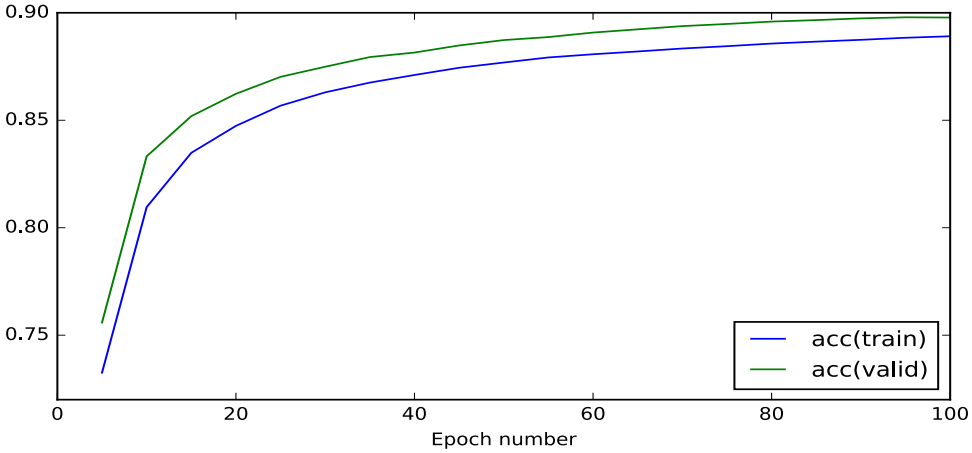


fig 1-3 ReciprocalLearningRate, learning rate = 0.01, decay rate = 5, accuracy

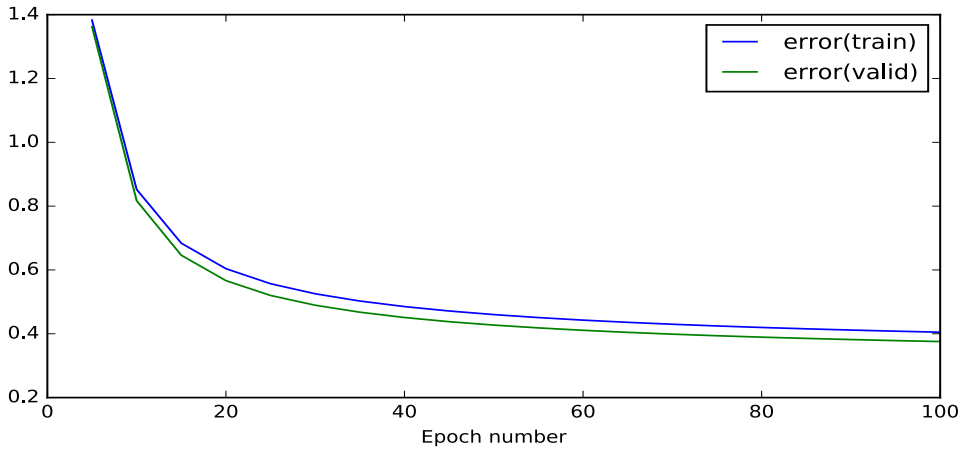


fig 1-4 ReciprocalLearningRate, learning rate = 0.01, decay rate = 5, error

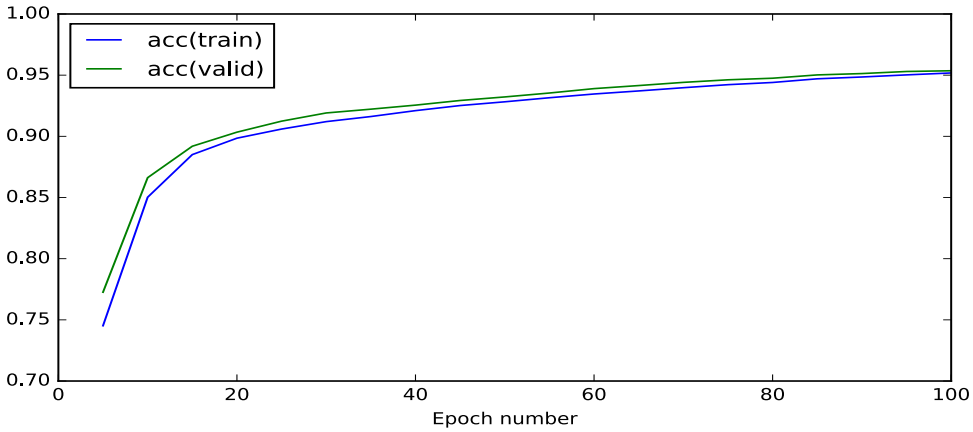


fig 1-5 ConstantLearningRate, learning rate = 0.01, accuracy

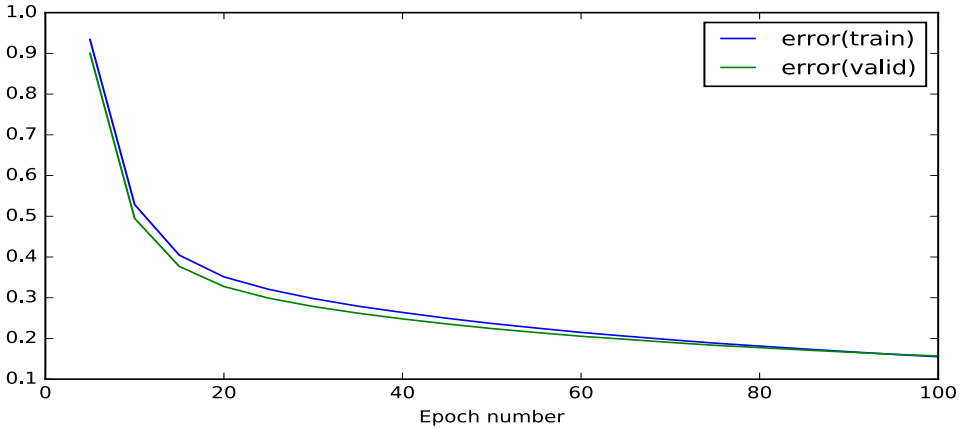


fig 1-6 ConstantLearningRate, learning rate = 0.01, error

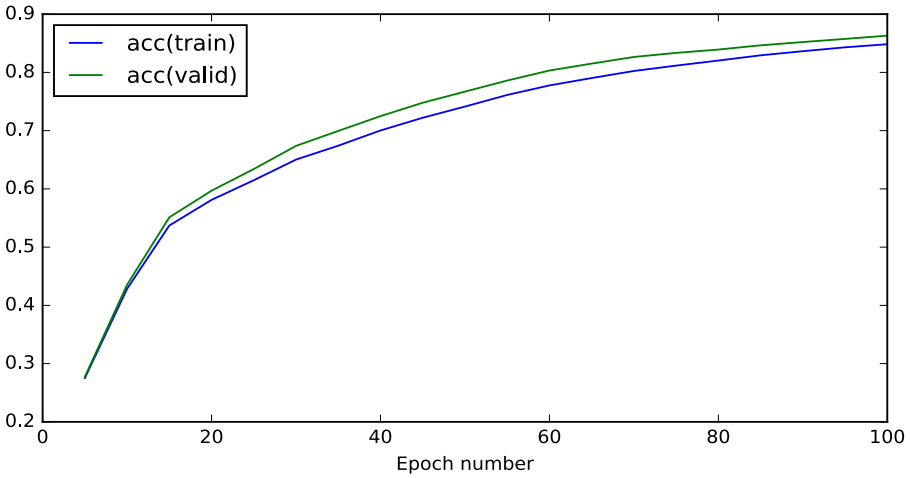


fig 1-7 ConstantLearningRater, learning rate = 0.001, accuracy

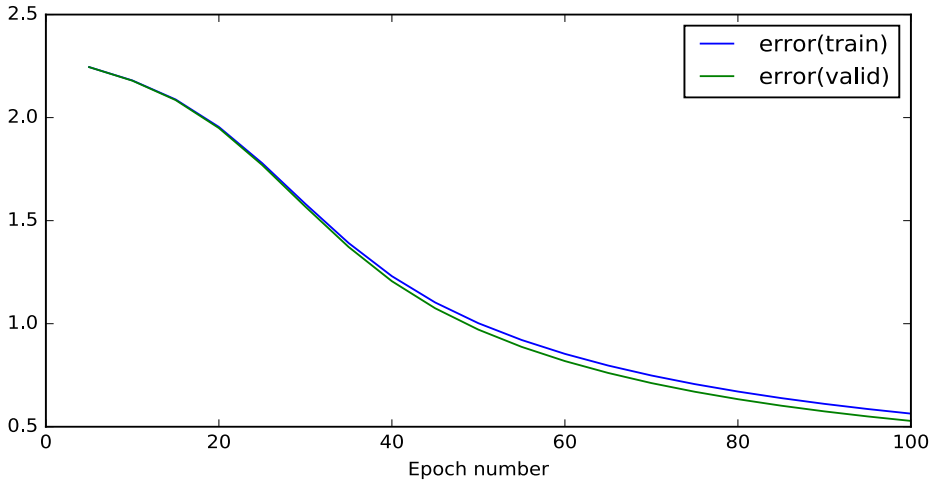


fig 1-8 ConstantLearningRater, learning rate = 0.001, error

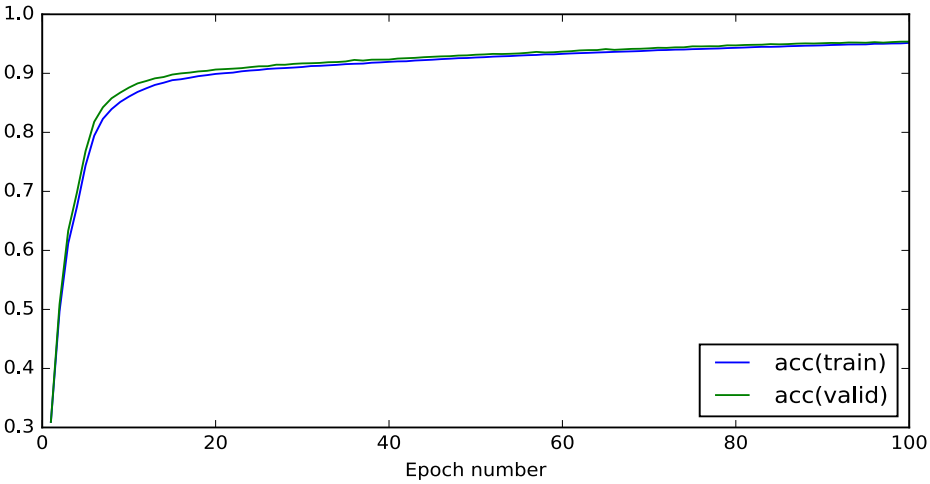


fig 1-9 ReciprocalLearningRate, learning rate = 0.01, decay rate = 500000, accuracy

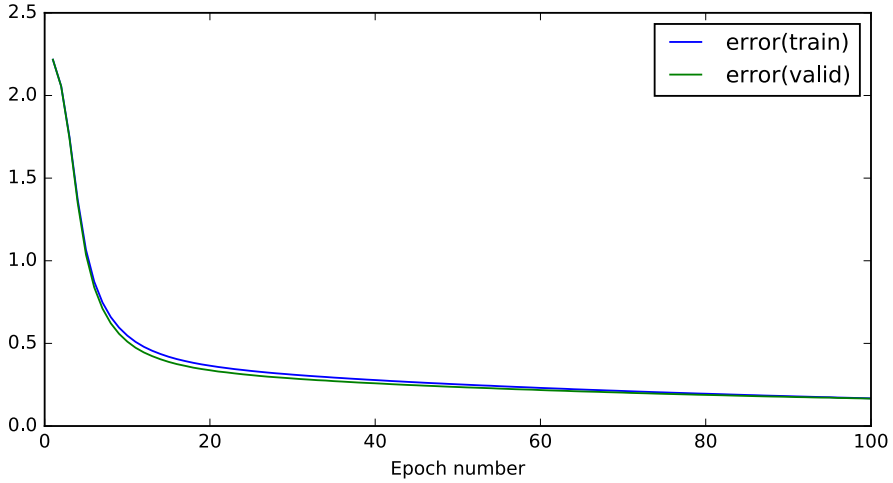


fig 1-10 ReciprocalLearningRate, learning rate = 0.01, decay rate = 500000, error

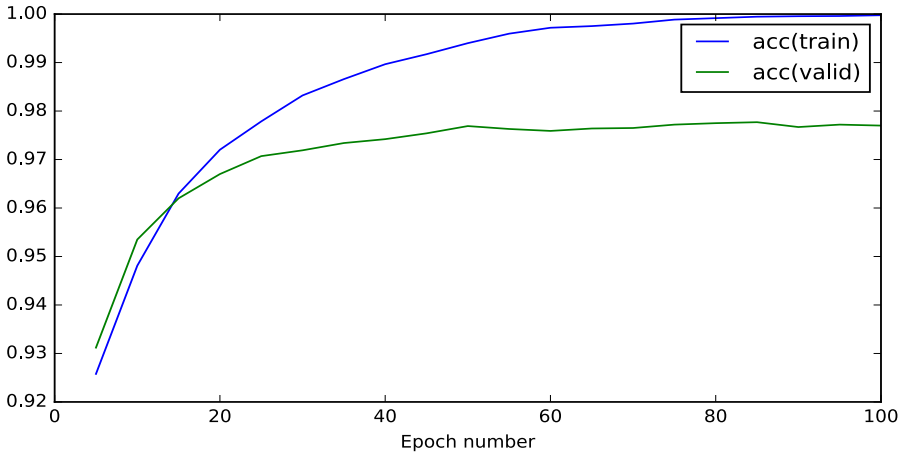


fig 1-11 ReciprocalLearningRate, learning rate = 0.1, decay rate = 500000, accuracy

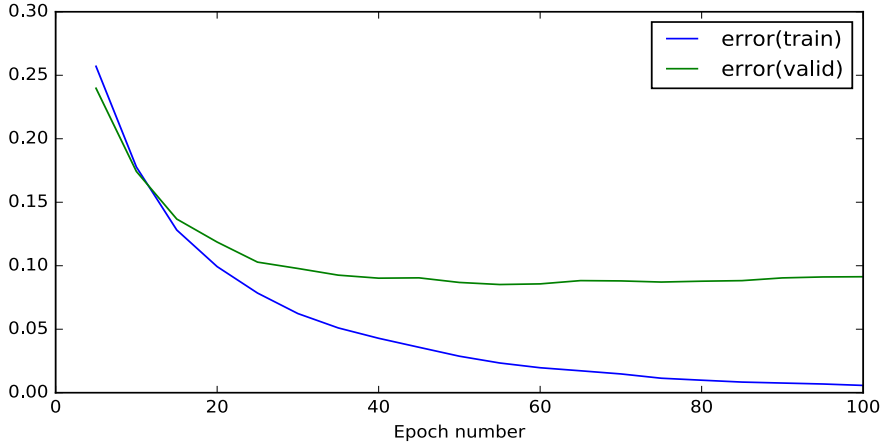


fig 1-12 ReciprocalLearningRate, learning rate = 0.1, decay rate = 500000, error

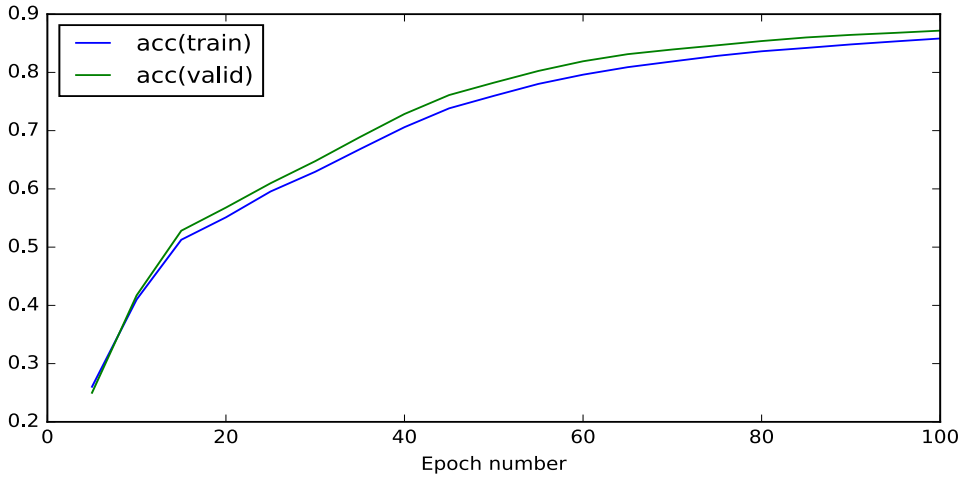


fig 1-13 ReciprocalLearningRate, learning rate = 0.001, decay rate = 500000, accuracy

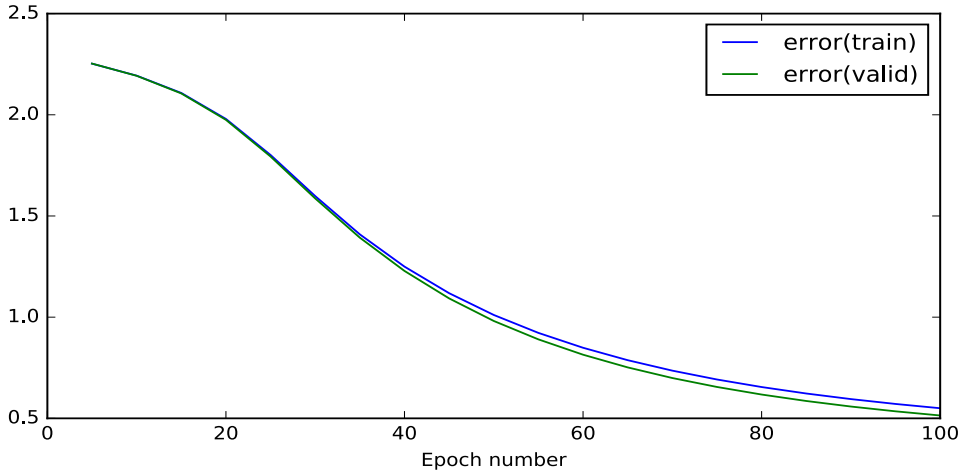


fig 1-14 ReciprocalLearningRate, learning rate = 0.001, decay rate = 500000, error

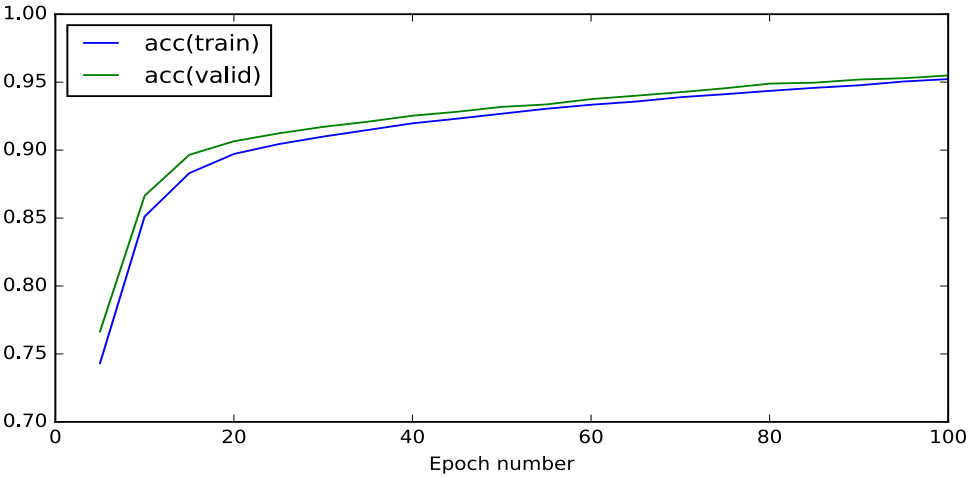


fig 1-15 ReciprocalLearningRate, learning rate = 0.01, decay rate = 5000, accuracy

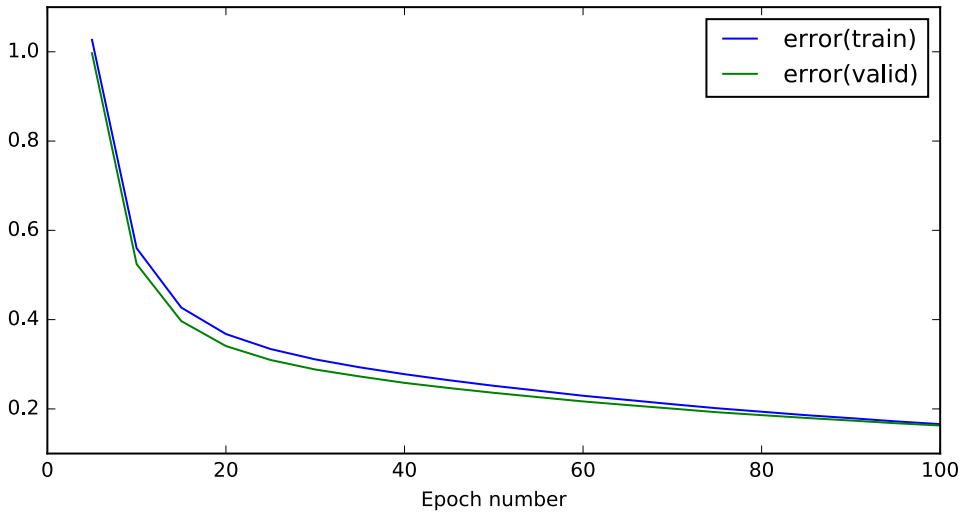


fig 1-16 ReciprocalLearningRate, learning rate = 0.01, decay rate = 5000, error

3. discussion of the results of your experiments and any conclusions you have drawn

a) Comparing the fig 1-5&1-6 with fig 1-9&1-10 and table 1-1, I conclude that Reciprocal Learning Rate Schedule can speed up convergence than Constant Learning Rate Schedule. And because the learning rate of Reciprocal is changed step by step, so the curve is more smooth than Constant one. From the table 1-1, we can know Reciprocal Learning Rate Schedule cannot change the performance a lot.

table 1-1

Learning rate = 0.01	Constant learning rate		Reciprocal learning rate	
	Train set	Valid set	Train set	Valid set
accuracy	9.52e-01	9.54e-01	9.51e-01	9.54e-01
error	1.68e-01	1.67e-01	1.67e-01	1.65e-01

b) I think the suitable constant learning rate value should be in the range of 0.1 to 0.001. Because fig 1-1&1-2 show the model is overfitting, and with the value going down to 0.01, fig 1-5&1-6 show it fits best. But with the decreasing of

learning rate from 0.01 to 0.001, the accuracy is falling and the error is going up according to fig 1-7&1-8. The table 1-2 provides information in detail.

table 1-2

	LR = 0.1		LR =0.01		LR= 0.001	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	1.00e+00	9.79e-01	9.52e-01	9.54e-01	8.55e-01	8.70e-01
error	5.29e-03	8.46e-02	1.68e-01	1.67e-01	5.63e-01	5.26e-01

c) I think the suitable reciprocal learning rate value should be in the range of 0.1 to 0.001 because fig 1-11&1-12 show the model is overfitting and fig1-13&1-14 show the decreasing of accuracy and increasing of error and fig 1-9&1-10 show it fits best. The table 1-3 provides information in detail.

table 1-3

	LR = 0.1		LR =0.01		LR= 0.001	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	1.00e+00	9.78e-01	9.51e-01	9.54e-01	8.56e-01	8.70e-01
error	5.50e-03	8.46e-02	1.67e-01	1.65e-01	5.58e-01	5.18e-01

d) I think, at first, with the increasing of the value of r, the performance of Reciprocal Learning Rate Schedule is getting better and better according to fig 1-3&1-4(r=5) and fig 1-15&1-16(r=5000) because at first the learning rate changes too large and with a little increase of r, the learning rate changes gradually. But when the value is going up to large number, the change of the performance is not obvious because the learning rate is almost equal to constant learning rate.

table 1-4

LR=0.01	r=5		r=5000		r=500000	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	8.94e-01	9.02e-01	9.52e-01	9.54e-01	9.51e-01	9.54e-01
error	3.84e-01	3.56e-01	1.65e-01	1.64e-01	1.67e-01	1.65e-01

Part 2: Momentum learning rule

1. description of the methods used and algorithms implemented

   a) I implement the algorithm as the formula shows: $\alpha(t) = \alpha_{\propto}(1 - r/(t + \tau))$.

   b) I create a new class named MomentumLearningRateScheduler which is inherited from class ConstantLearningRateScheduler in mlp.schedulers.py module.

   c) I define a __init__ function to initialize the parameters. I call the super function to complete my learning rate parameter initialization and add three more line to initialize my new parameter called mom_coeff($\alpha_{\propto}$), Tau(τ) and Gamma(r).

   d) I also define a updata_learning_rule function to update learning rule. Just as the formula shows, learning_rule.mom_coeff = self.mom_coeff - ((self.mom_coeff*self.Gamma) / float(epoch_number+self.Tau)).

   e) Using the module just implemented and new an array of schedulers and make use of GradientDescentLearningRule. Finally, show the plots via train_model_and_plot_stats function.

2. results for the experiments you carried out including relevant graphs

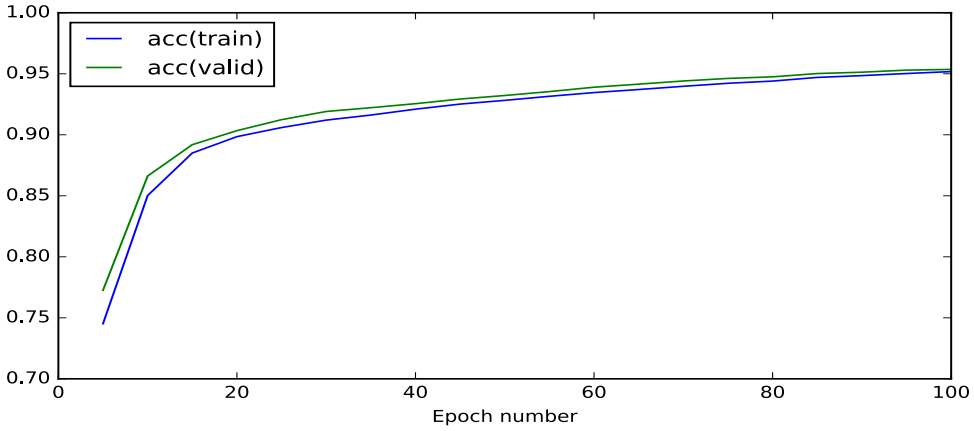


fig 2-1 Momentum learning rule, learning rate = 0.01, $\alpha = 0$, accuracy

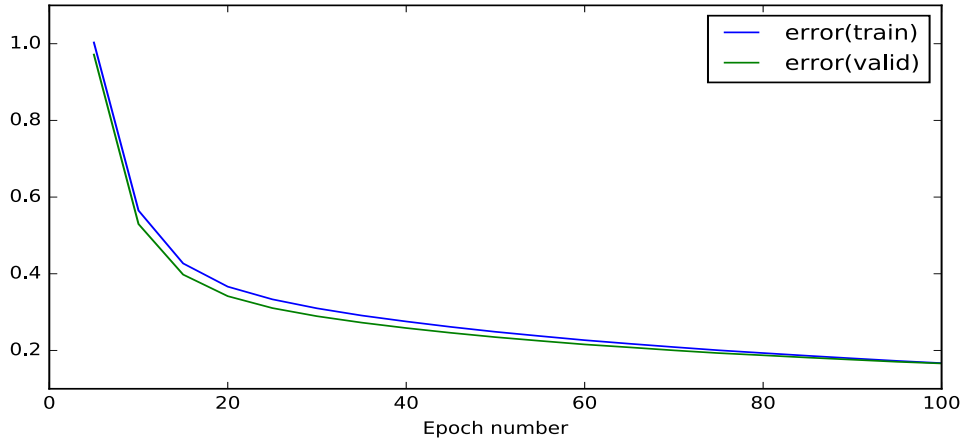
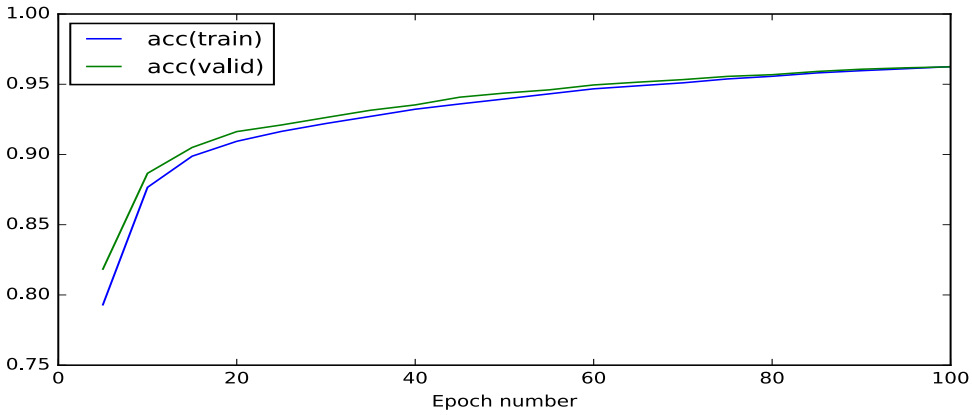


fig 2-2 Momentum learning rule, learning rate = 0.01, $\alpha = 0$, error



- fig 2-3 Momentum learning rule, learning rate = 0.01, $\alpha = 0.3$, accuracy

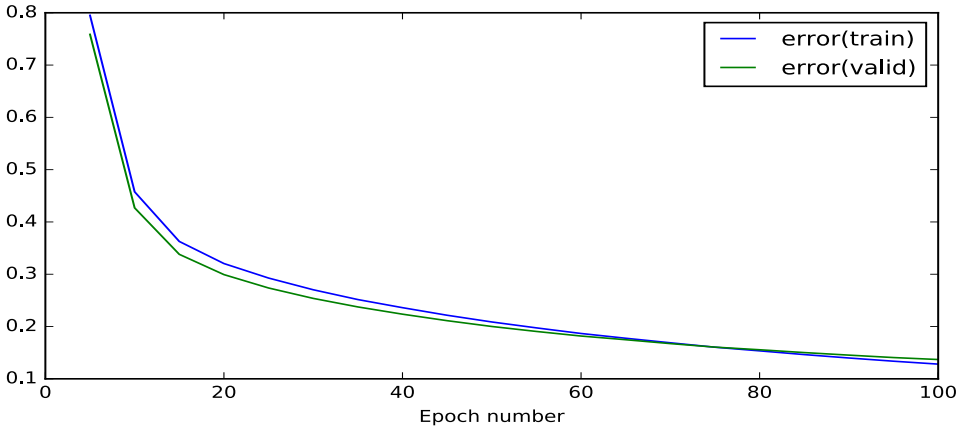


fig 2-4 Momentum learning rule, learning rate = 0.01, $\alpha = 0.3$, error

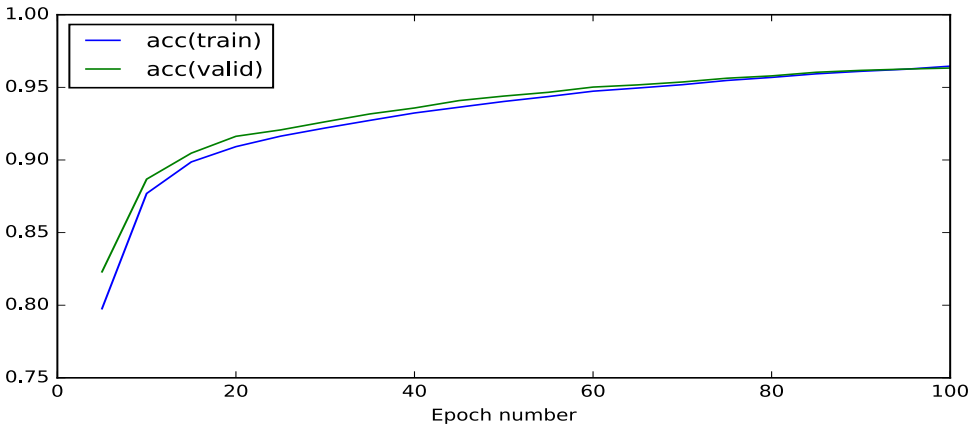


fig 2-5 Momentum learning rule, learning rate = 0.01, $\alpha = 0.5$, accurac

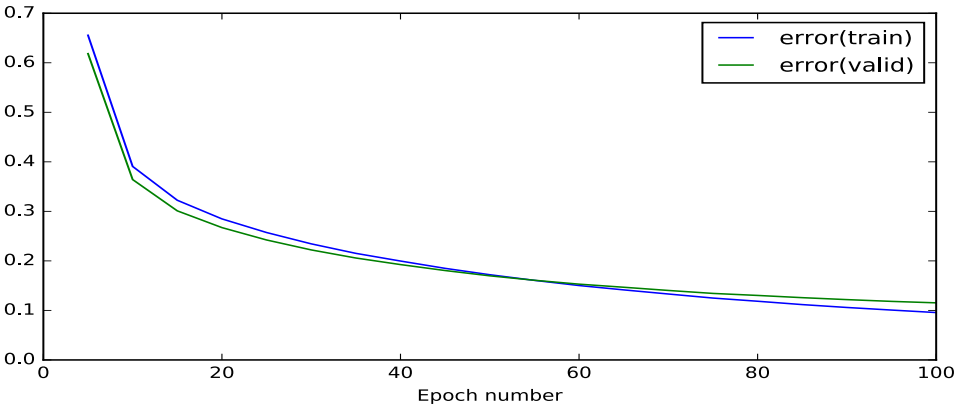


fig 2-6 Momentum learning rule, learning rate = 0.01, $\alpha = 0.5$, error

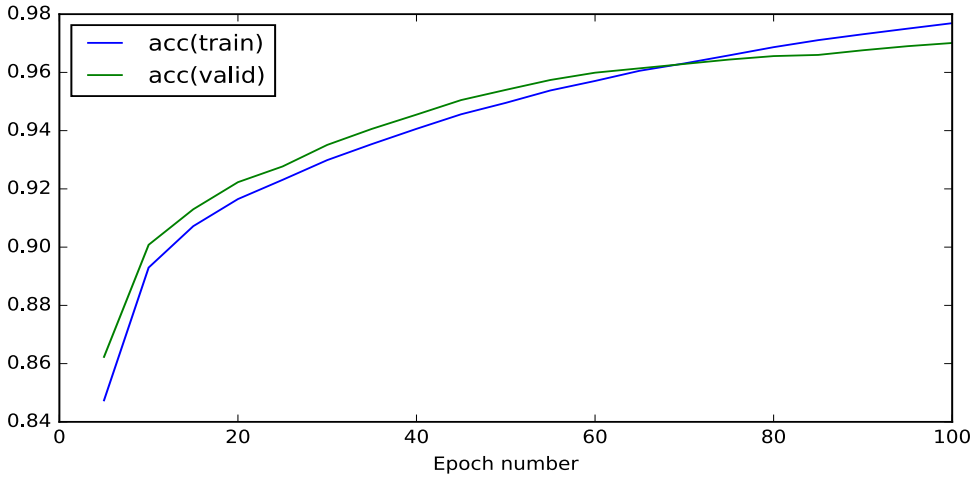


fig 2-7 Momentum learning rule, learning rate = 0.01, $\alpha = 0.8$, accuracy

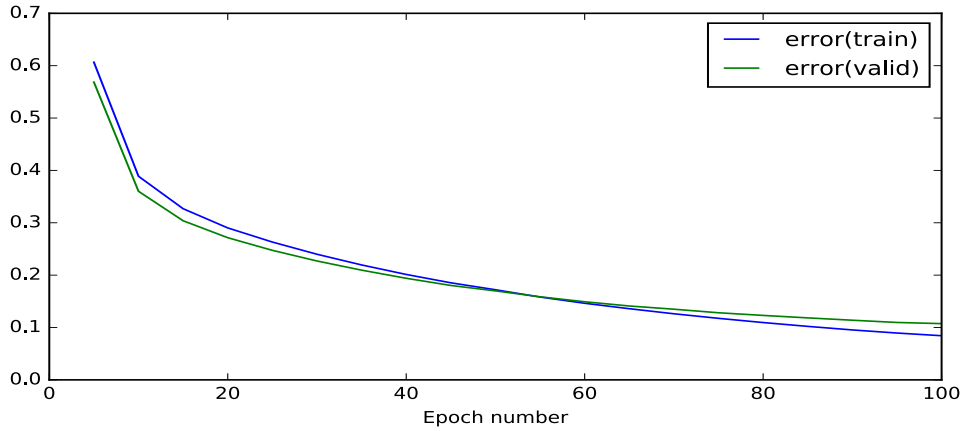


fig 2-8 Momentum learning rule, learning rate = 0.01, $\alpha = 0.8$, error

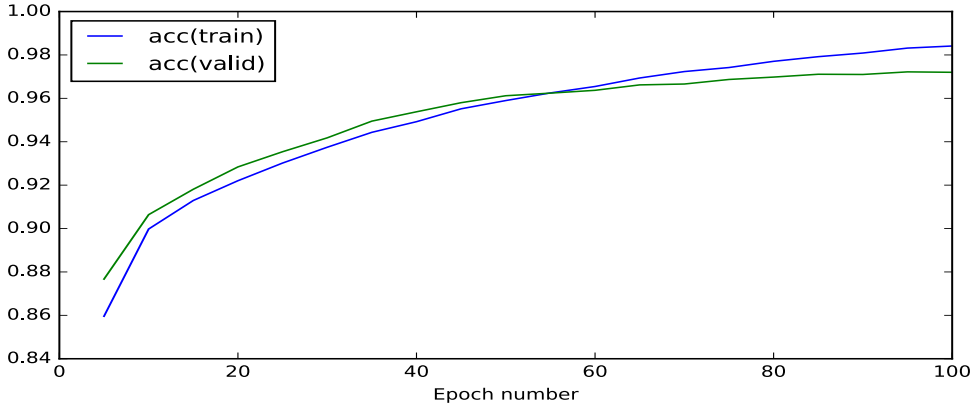


fig 2-9 Momentum learning rule, learning rate = 0.01, $\alpha = 1$, accuracy

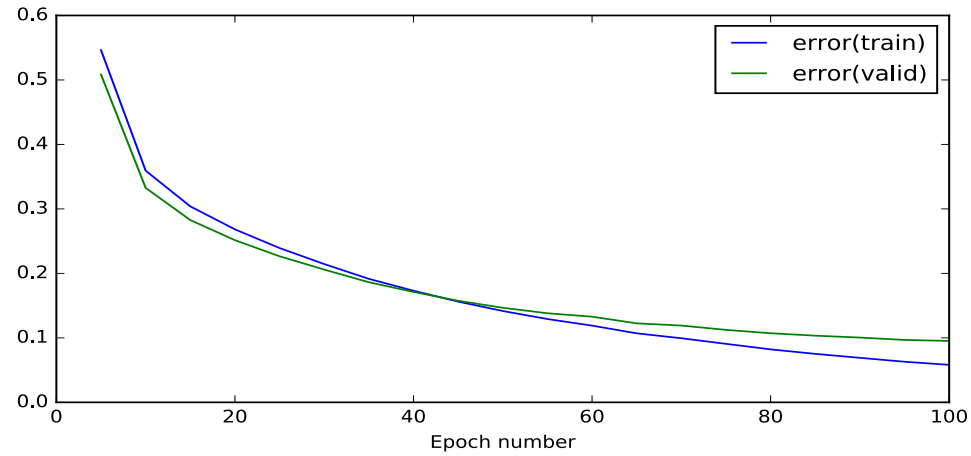


fig 2-10 Momentum learning rule, learning rate = 0.01, $\alpha = 1$, error

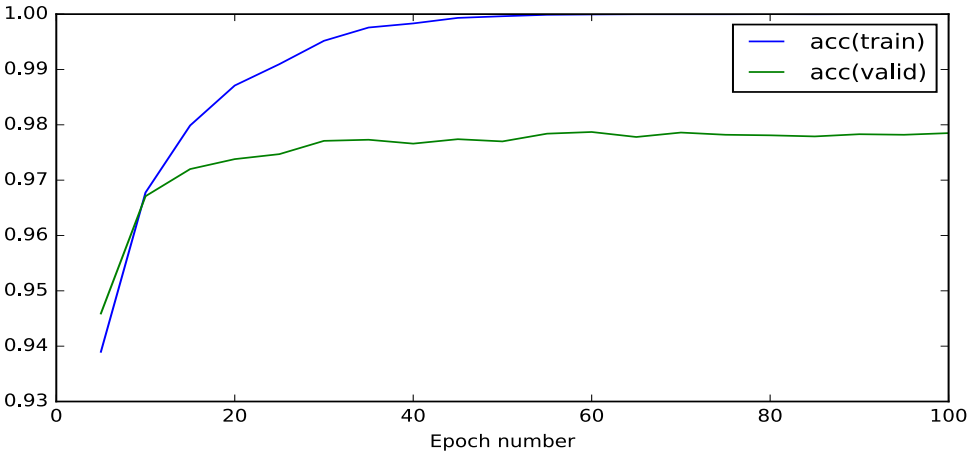


fig 2-11 Momentum learning rule, learning rate = 0.1, $\alpha = 0.1$, accuracy

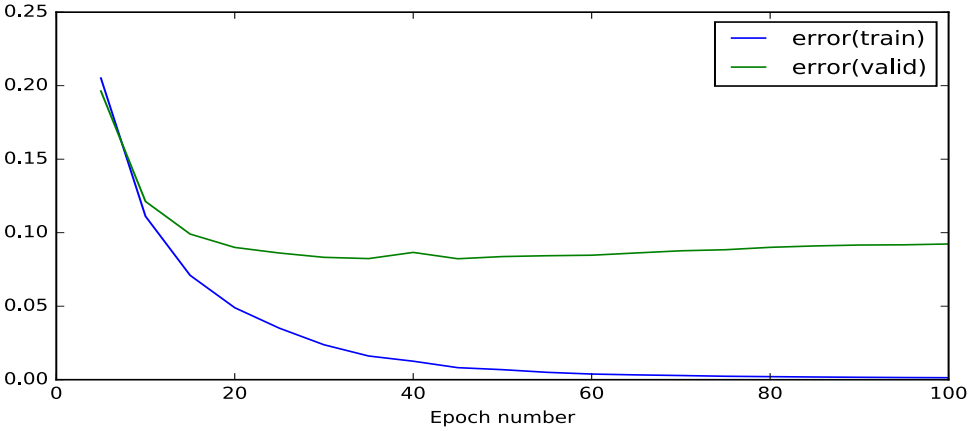


fig 2-12 Momentum learning rule, learning rate = 0.01, $\alpha = 0.1$, error

3. discussion of the results of your experiments and any conclusions you have drawn

a) Comparing figures from fig 2-1 to fig 2-10 and based on table 2-1, I conclude that when the value of α is going up, the performance of Momentum Learning Rule is getting better and better.

table 2-1

Lr=0.01	$\alpha = 0$		$\alpha = 0.3$		$\alpha = 0.5$		$\alpha = 0.8$		$\alpha = 1$	
	Train	Valid	Train	Valid	Train	Valid	Train	Valid	Train	valid
accuracy	9.52e-1	9.54e-1	9.59e-1	9.59e-1	9.65e-1	9.63e-1	9.77e-1	9.70e-1	9.84e-1	9.72e-1
error	1.67e-1	1.66e-1	1.42e-1	1.47e-1	1.23e-1	1.33e-1	8.43e-2	1.07e-1	5.82e-2	9.52e-2

b) Comparing fig 1-1&1-2 with fig 2-11&2-12, I find that Momentum Learning Rule can accelerate the speed to converge. For example, the train accuracy curve of Momentum starts convergence at around epoch 50, and the same one of Constant starts to converge at about epoch 80. But Momentum Learning Rule cannot improve the performance a lot.

table 2-2

Learning rate = 0.1	Constant learning rate		Momentum learning rate	
	Train set	Valid set	Train set	Valid set
accuracy	1.00e+00	9.77e-01	1.00e+00	9.78e-01
error	5.77e-03	8.77e-02	1.35e-03	9.23e-02

c) I try different values for τ, I find there is no obvious change no matter from the figures or final result.

table 2-3

LR=0.01	τ =1		τ =100		τ =100	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	9.74e-01	9.69e-01	9.74e-01	9.68e-01	9.51e-01	9.54e-01
error	9.18e-02	1.12e-01	9.39e-02	1.15e-01	1.67e-01	1.65e-01

d) I try different values for γ, I find there is no obvious change(τ=1000) no matter from the figures or final result.

table 2-4

LR=0.01	γ=5		γ =50		γ =500	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	9.74e-01	9.69e-01	9.73e-01	9.69e-01	9.62e-01	9.60e-01
error	9.11e-02	1.12e-01	9.44e-02	1.10e-01	1.33e-01	1.39e-01

Part 3: Adaptive learning rule

1. description of the methods used and algorithms implemented

a) I implement the algorithm as formulas show.

b) I define a new class named AdaGradLearningRule inheriting from GradientDescendLearningRule in the mlp/learning_rules.py module.

c) I define a __init__ function to initialize the parameters. I call the super function to complete my learning rate parameter initialization and add one more line to initialize my new parameter called epsilon.

d) I also define a initialize function, by using the for loop to initialize Sum_quared_grads array.

e) I also define an updata_learning_rule function to update learning rule. The first step is to accumulate sum square gradient by using += method to add square gradient. The next step is to use AdaGrad formula to calculate the AdaGrad value. Finally, add a new AdaGrad value to parameter so all parameters will be updated via using in-place operations.

f) The implementation of RMSProp is almost the same. The only different area is updata_params function. We should use *= method to plus decay_rate and use += method to sum up the result of square gradient which should plus (1-decay_rate). Then we can use the result of Average_square_grad to calculate RMSProp value. Finally, add a new RMSProp value to parameter so all parameters will be updated via using in-place operations.

g) Using the module just implemented and new an array of schedulers and make use of AdaGradLearningRule and RMSPropLearningRule separately. Eventually, show the plots via train_model_and_plot_stats function.

2. results for the experiments you carried out including relevant graphs

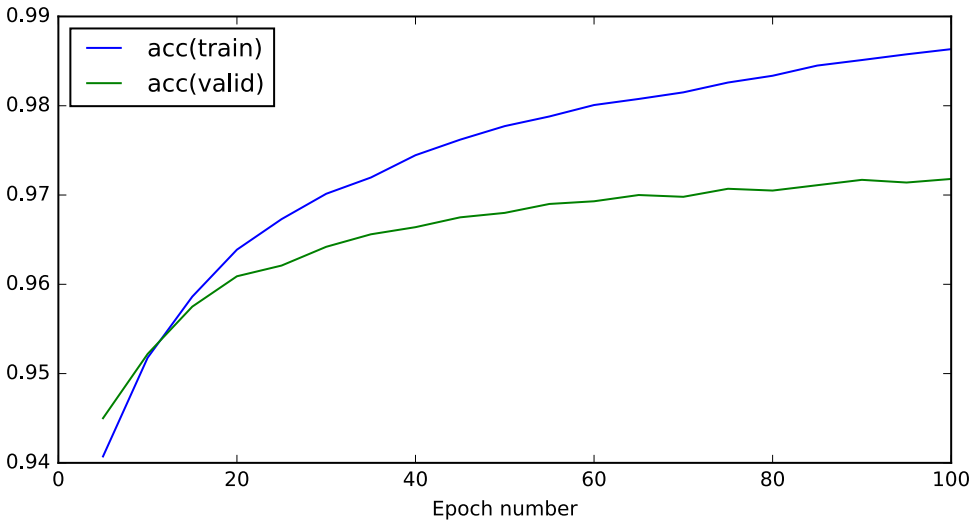


fig 3-1 AdaGrad learning rule, learning rate = 0.01, accuracy

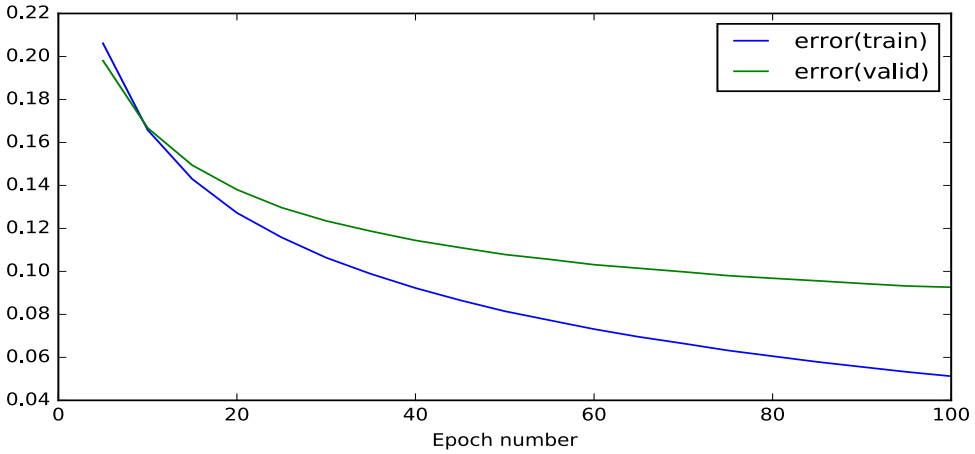
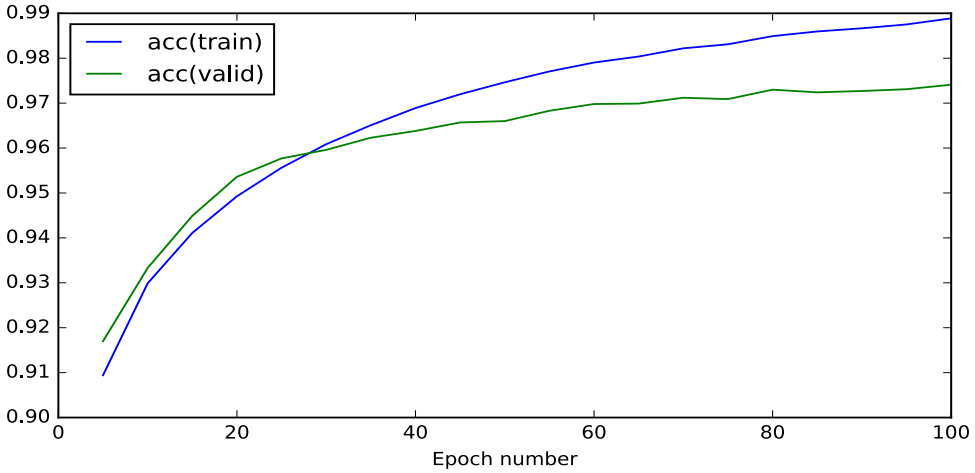
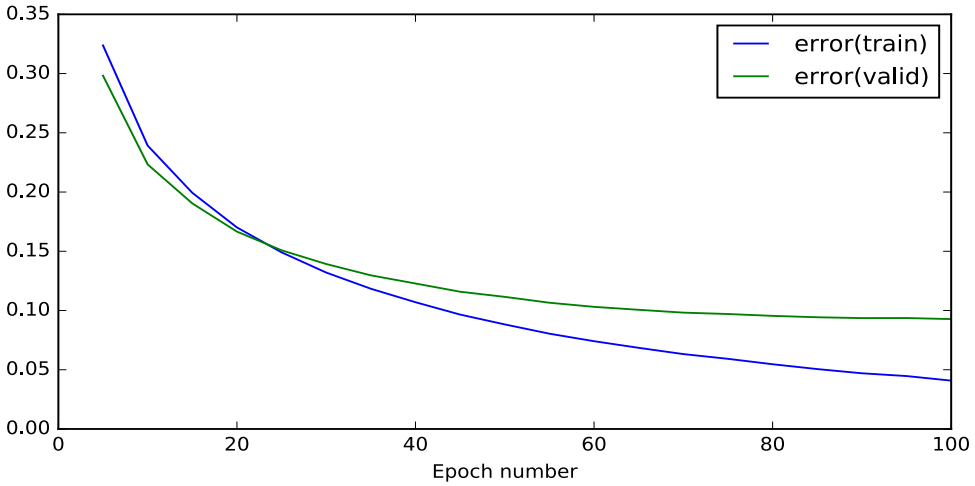


fig 3-2 AdaGrad learning rule, learning rate = 0.01, error



- fig 3-3 RMSProp learning rule, learning rate = 0.0001, accuracy



- fig 3-4 RMSProp learning rule, learning rate = 0.0001, error

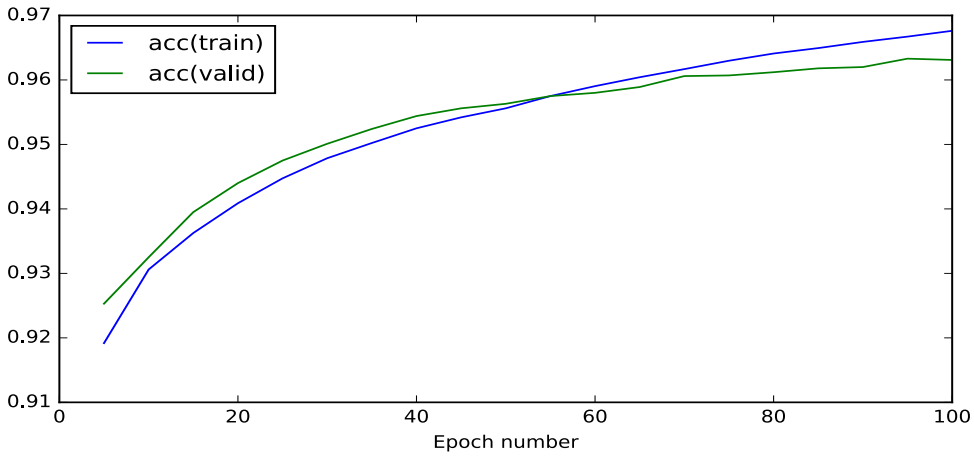


fig 3-5 AdaGrad learning rule, learning rate = 0.005, accuracy

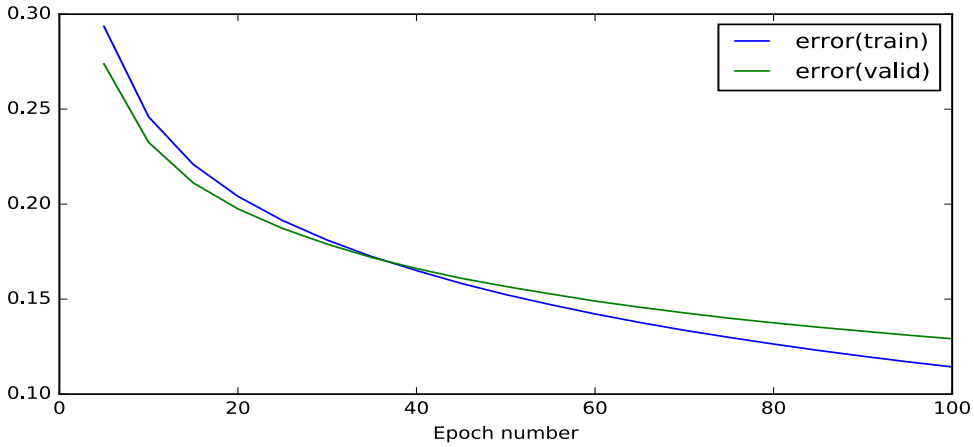


fig 3-6 AdaGrad learning rule, learning rate = 0.005, error

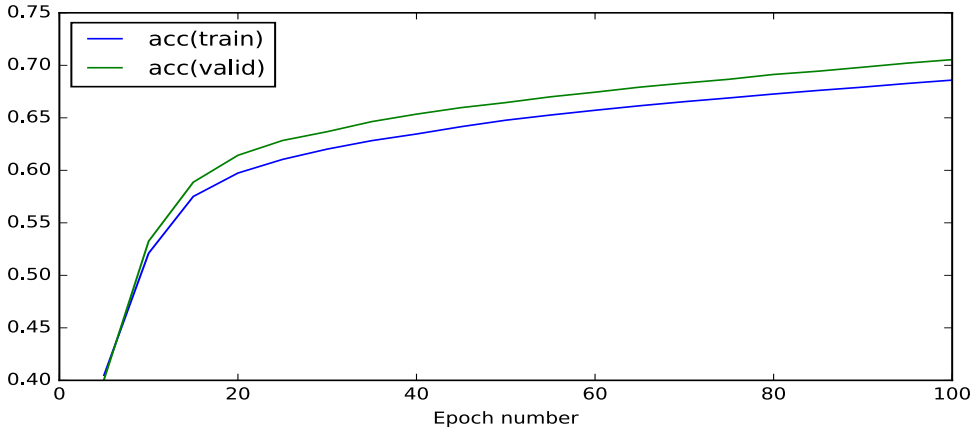


fig 3-7 AdaGrad learning rule, learning rate = 0.0001, accuracy

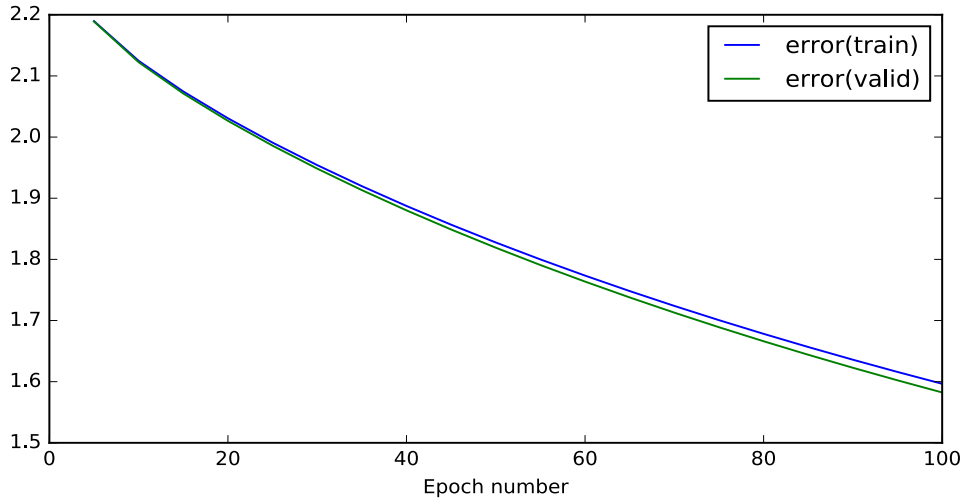


fig 3-8 AdaGrad learning rule, learning rate = 0.0001, error

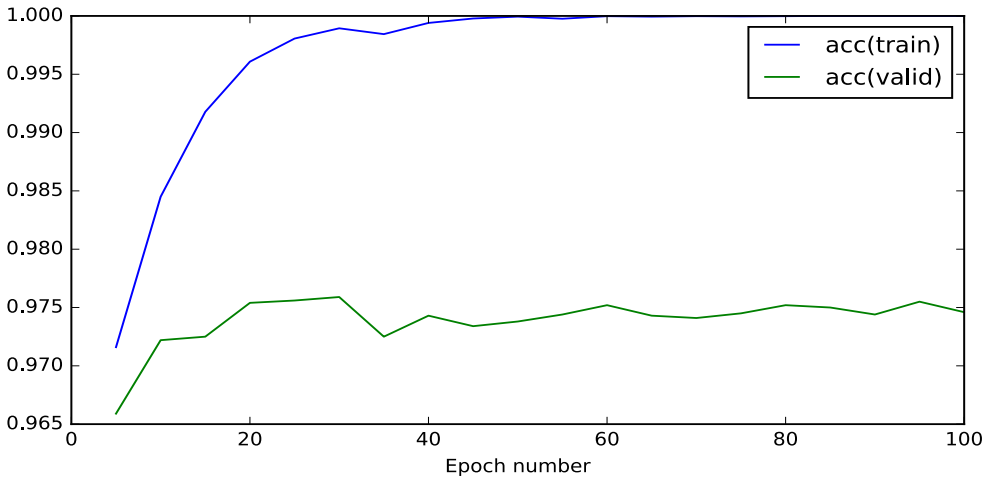


fig 3-9 RMSProp learning rule, learning rate = 0.001, accuracy

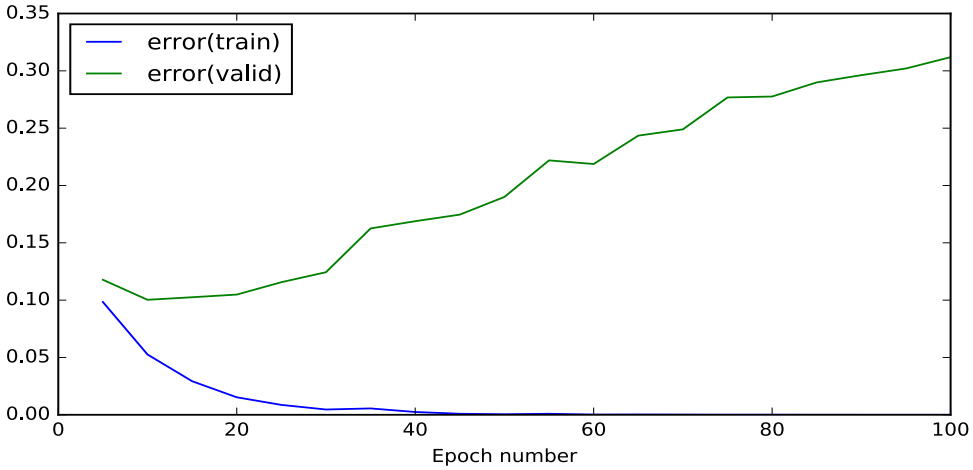


fig 3-10 RMSProp learning rule, learning rate = 0.001, error

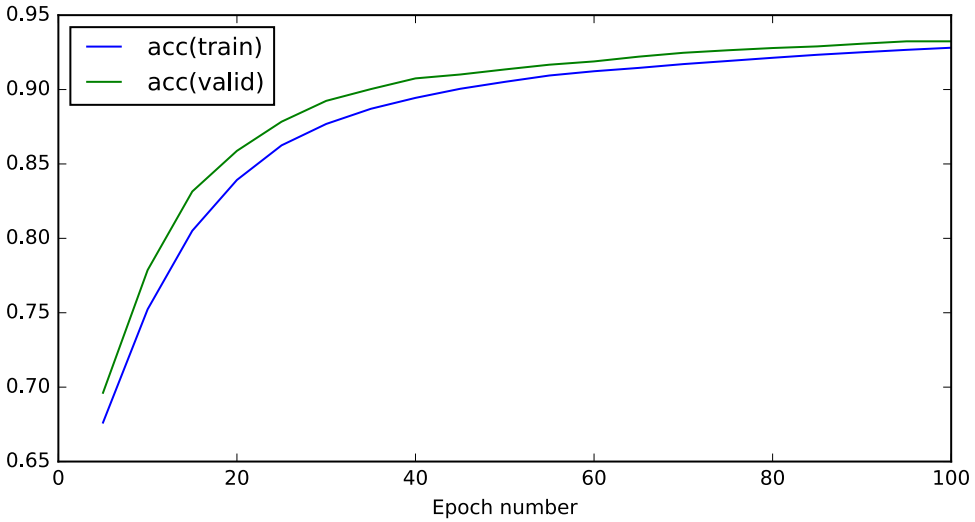


fig 3-11 RMSProp learning rule, learning rate = 0.00001, accuracy

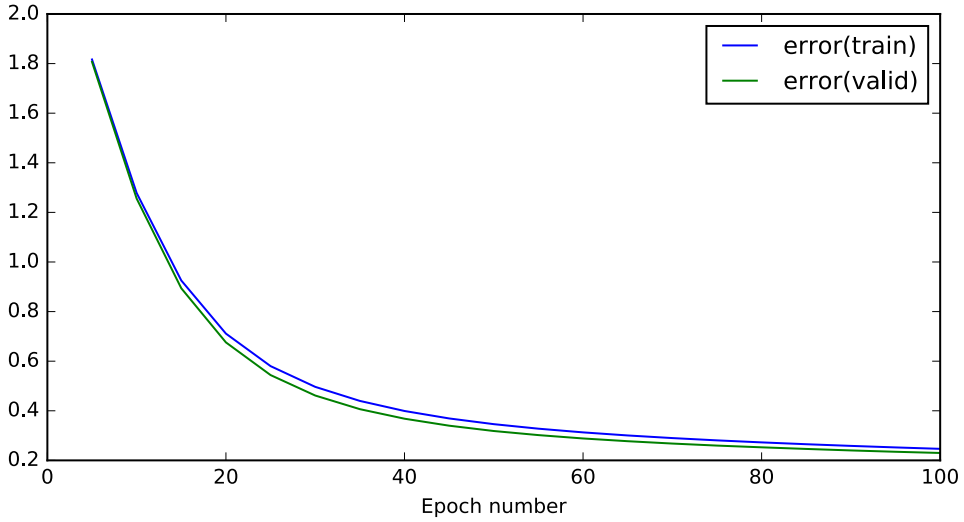


fig 3-12 RMSProp learning rule, learning rate = 0.00001, error

3. discussion of the results of your experiments and any conclusions you have drawn

a) Comparing fig 2-5&2-6 with fig 3-1&3-2, I conclude that, for this dataset and learning rate (0.01), Momentum Learning Rule performs better than AdaGrad Learning Rule. Firstly, the speed of convergence of the first one is faster than the latter one. Secondly, even if the classification accuracy of training set of AdaGrad Learning Rule is higher and final error is lower, there exit some overfitting and the curve of train and valid set of Momentum Learning Rule fit each other better. Finally, as for the training time, Momentum Learning rule takes less than 2 seconds per epoch but AdaGrad learning rules takes more than 2 seconds per epoch.

table 3-1

Learning rate = 0.01	Momentum learning rate		AdaGrad learning rate	
	Train set	Valid set	Train set	Valid set
accuracy	9.61e-01	9.61e-01	9.86e-01	9.72e-01
error	1.33e-01	1.40e-01	5.12e-02	9.26e-02

b) Comparing fig 2-11&2-12 with fig 3-3&3-4. For this two method, we cannot use the same learning rate to judge the performance because they cannot perform well at the same value. So I compare both the best performance of each other. For this dataset, I conclude that Momentum learning rate seems to perform better than RmsProp learning rate. Firstly, the speed of convergence of the first one is faster than the latter one. Secondly, accuracy of Momentum Learning Rule is higher and final error is lower than RmsProp. Finally, as for the training time, Momentum Learning rule takes less than 2 seconds per epoch but RmsProp learning rules takes more than 2 seconds per epoch. So Momentum Learning rule takes less training time.

table 3-2

	Momentum learning rate		RmsProp learning rate	
	Train set	Valid set	Train set	Valid set
accuracy	1.00e+00	9.78e-01	9.89e-01	9.75e-01
error	1.35e-03	9.23e-02	3.97e-02	9.23e-02

c) Comparing fig 3-3&3-4, fig 3-9 to 3-12 and based on table 3-3, I find that when the learning rate of RmsProp is too big, the figures show it is overfitting, but the accuracy and error is best. And when reducing the value to a proper number, the curve of train and valid fit each other better but the accuracy and error is getting a little bit worse. And when the learning rate is too small, even though the two curve fit each other, the accuracy and error performs worst.

table 3-3

RmsProp	LR = 0.001		LR = 0.0001		LR = 0.00001	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	1.0e+00	9.75e-01	9.89e-01	9.75e-01	9.28e-01	9.32e-01
error	1.18e-04	2.94e-01	3.97e-02	8.56e-02	2.47e-01	2.30e-01

d) Comparing fig 3-1&3-2, fig 3-5 to 3-8 and based on table 3-4, I find that the performance with the change of AdaGrad learning rate is the same as RmsProp.

table 3-4

AdaGrad	LR = 0.01		LR = 0.005		LR = 0.0001	
	Train set	Valid set	Train set	Valid set	Train set	Valid set
accuracy	9.86e-01	9.72e-01	9.68e-01	9.63e-01	7.30e-01	7.53e-01
error	5.12e-02	9.26e-02	1.14e-01	1.30e-01	1.67e+00	1.58e+00