Softmax Regression Implementation

– Classification on Handwritten Digit Data Set

Abstract

The task of this experiment is to implement SoftMax regression for classification on handwritten digits. The data is from the MNIST data set[1].

After filling in softmax, cross entropy loss function and gradient calculation to the provided framework, we tune parameters: batch size, max epoch, learning rate and momentum, one by one, and choose the optimal solution among our tests based on the convergence, performances and time cost.

The optimal model we choose has the parameters as following: learning rate 0.53, momentum 0.9, batch size 5000 and max epochs 128. The accuracy on the test set reaches to 0.9253 and has average loss at 0.2663.

For extended experiments, we tested step decay and exponential decay on the learning rate, and tried to observe the differences between different gradient descent methods. Decaying can lead to faster convergence and smoother loss and accuracy, especially for exponential decay. SGD only becomes better until the batch size becomes huge. Since parameter tuning does not involve in this compare and contrast stage, we still could not come to a conclusion on which one is better for our data and task.

Adam(Adaptive Movement Estimation algorithm) is implemented. The best model among the ones we test is: batch size 5000, decay for mean 0.9, decay for variance 0.99, step size 0.01, and the model will stop learning if updates on biases and weights are lower than a small threshold for 600 continuous steps. The mean accuracy and mean loss for this model on the test set prediction are 0.9264 and 0.2643. Overall, Adam can give reliable and statistical significance good learning results, with smooth loss and fast convergence, stable accuracy and less learning cost.

1 Introduction

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image[1].

The task is to use Softmax regression to classify these digits. Parameter tuning and analysis happen on the train set and the validation set, and the selected optimal model runs predictions on the test set for experiment results. Loss, train time and accuracy scores are used to measure the performance of a model.

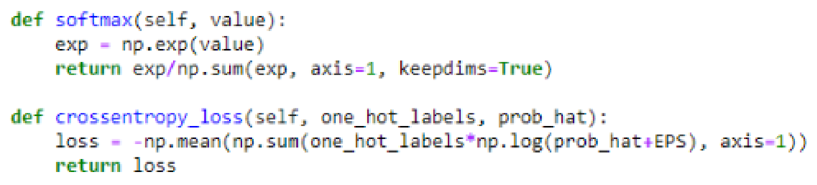
A code frame has been provided. One of the challenges is to fill the Softmax function, cross entropy function and gradient update into this framework. Since the pivot of this assignment is to implement algorithms and analyze the model behavior, the data fed into the model does not go through any pre-processing, except regularize the pixel color info to 0-1 range..

2 Programming Modules

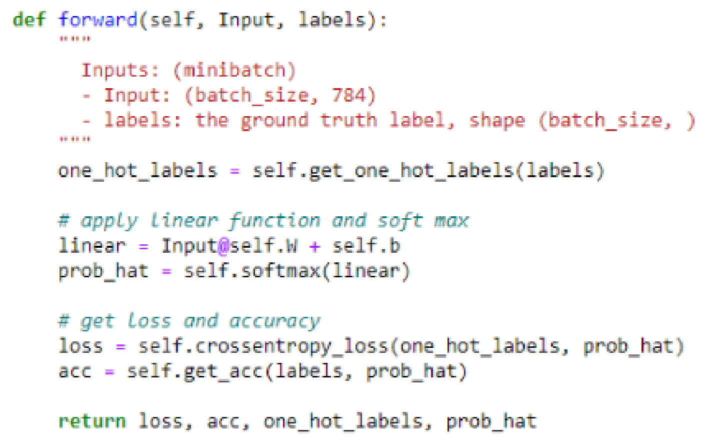
Numpy is invoked for better math calculation. The ‘struct’ module is used to read and unpack the ‘ubyte’ format data. To measure the performance of a model, we also use the ‘time’ module and matplotlib to track the learning cost and to visualize learning results.

3 Parameter Tuning

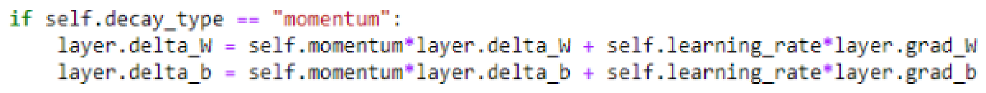
We try to find the best configurations for a model that produces the lowest loss and highest accuracy with a limited time budget. Cross-entropy is used to calculate the loss and the gradient to update both the weights and bias in our regression function. Below are some code snippets.



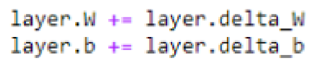
```
def softmax(self, value):
    exp = np.exp(value)
    return exp/np.sum(exp, axis=1, keepdims=True)

def crossentropy_loss(self, one_hot_labels, prob_hat):
    loss = -np.mean(np.sum(one_hot_labels*np.log(prob_hat+EPS), axis=1))
    return loss
```



```
def forward(self, Input, labels):
    """
      Inputs: (minibatch)
      - Input: (batch_size, 784)
      - labels: the ground truth label, shape (batch_size, )
    """
    one_hot_labels = self.get_one_hot_labels(labels)

    # apply linear function and soft max
    linear = Input@self.W + self.b
    prob_hat = self.softmax(linear)

    # get loss and accuracy
    loss = self.crossentropy_loss(one_hot_labels, prob_hat)
    acc = self.get_acc(labels, prob_hat)

    return loss, acc, one_hot_labels, prob_hat
```



```
if self.decay_type == "momentum":
    layer.delta_W = self.momentum*layer.delta_W + self.learning_rate*layer.grad_W
    layer.delta_b = self.momentum*layer.delta_b + self.learning_rate*layer.grad_b
```



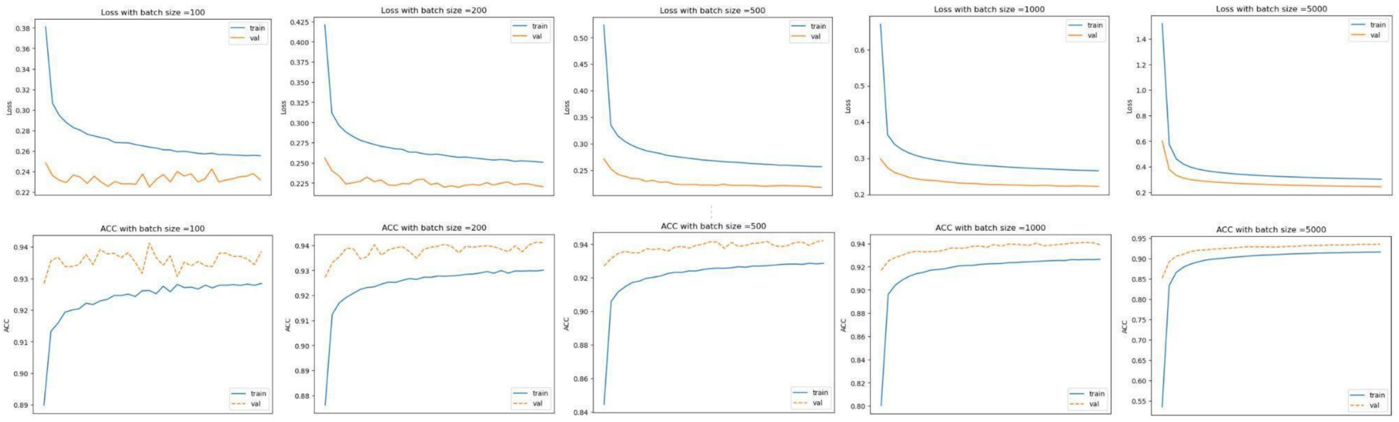
```
layer.W += layer.delta_W
layer.b += layer.delta_b
```

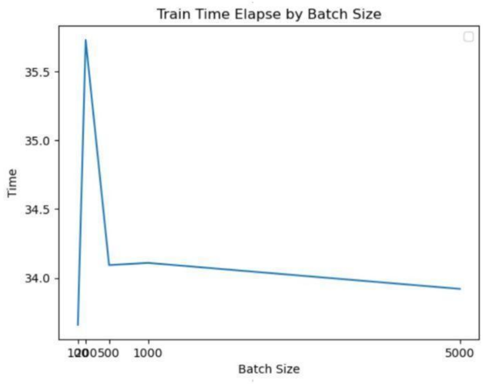
3.1 Batch Size

The smaller the batch size is, the more random the gradient can be. Large batch sizes usually suggest more stable gradients.

When using a small batch size, due to the outliers or differences between samples, the weights and biases for the model can have large changes with gradients that are highly random. This increases the time lapse and also makes the model hard to converge. Although large batch sizes commit to more stable commits they can also decrease the generalization ability of the model[2].

Generally, the default value for epoch is 32, 0.1 for learning rate and 0.9 for momentum. We will use this value for testing batch size. Batch size for testing are: 100, 200, 500, 1000, 5000.





PS. For better resolution on all diagrams used in this report, please refer to the /graph/ directory under the same path.

Observation:

1. The time cost for different batch sizes are similar, all around 34-36 seconds. Training time increases massively when batch size increases from 100 to 200, then rapidly drops when batch size gets to 500, after which the curve follows a gentle downward trend.
2. The accuracy scores keep increasing and still have an upward trend at the last epoch. Therefore, we could use larger epoch values for later testing if they can provide better learning results.
3. When the batch size is smaller than 500, fluctuation happens on both loss and accuracy, especially on the validation set. And larger batch sizes have smoother loss curves and accuracy curves. Smaller batch sizes give noisy gradients but they converge faster because per epoch you have more updates. If your batch size is 1 you will have N updates per epoch. If it is N, you will only have 1 update per epoch. On the other hand, larger batch sizes give a more informative gradient but they converge slower and increase computational complexity[3]. Our train dataset has 5.5K samples, with a comparatively small batch size, it could lead to underfitting easily.

4. We will use batch size 100, 500 and 5000 for tuning epoch values. These three are: the worst, start to converge better and the best performances among our tests.

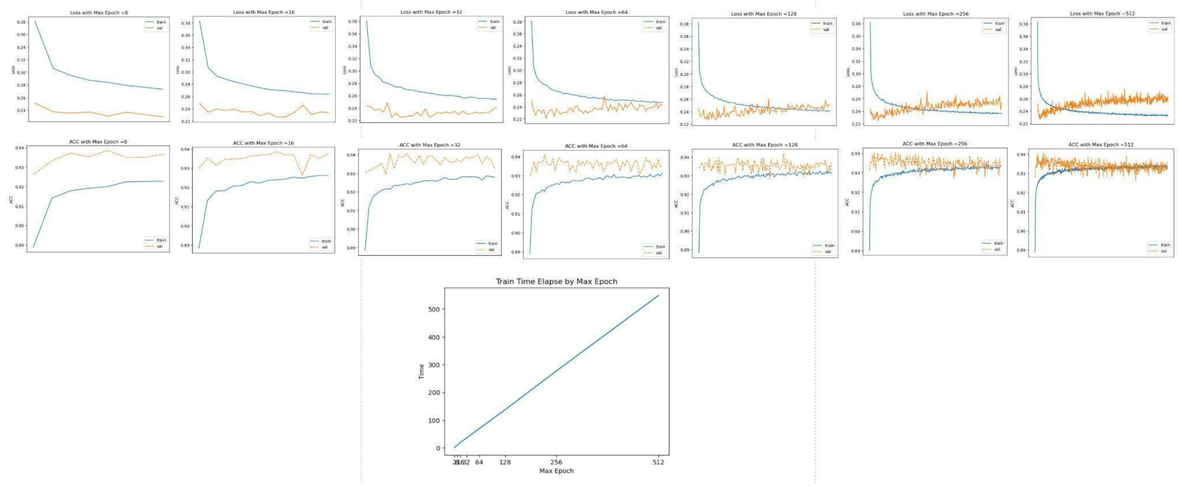
3.2 Max Epoch

An epoch is defined as the total number of iterations of all the training data in one cycle for training the machine learning model. A cycle refers to when all the training data is used across several mini batches.

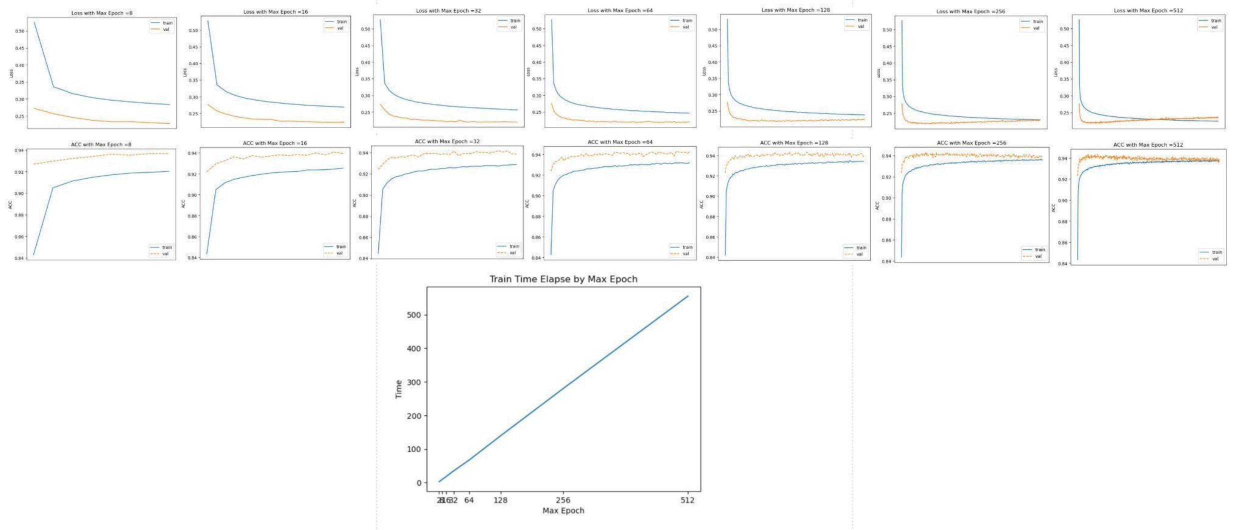
In one iteration alone you can not guarantee that the gradient descent algorithm would converge to a local minima with the specified learning rate. That is the reason to invoke epochs for the gradient descent to converge better[4].

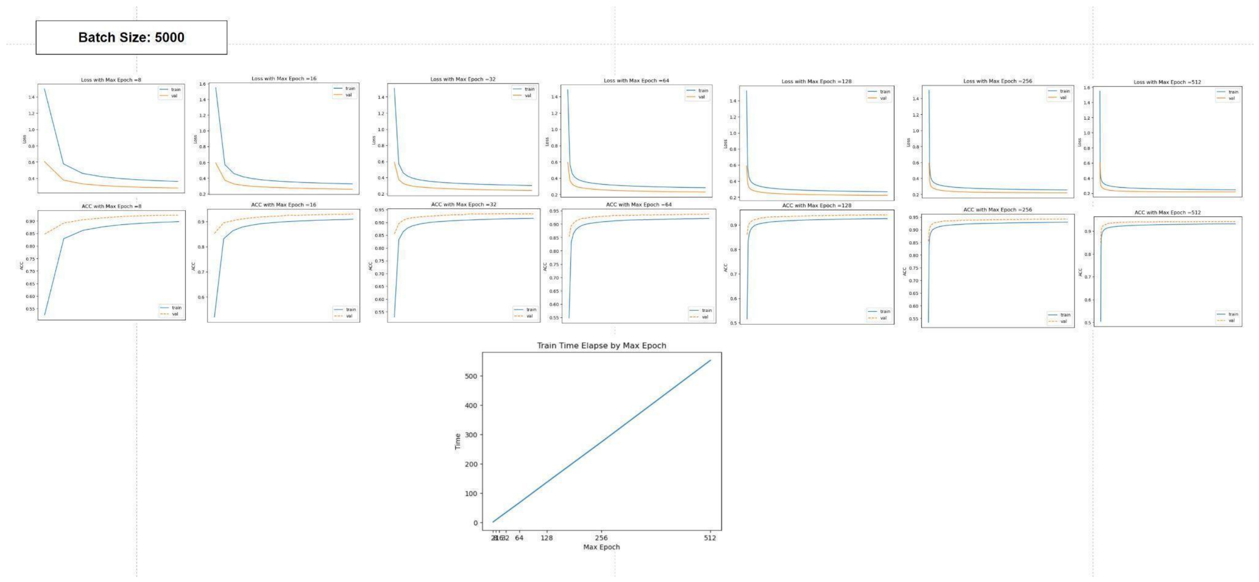
Small batch and large batch each have advantages and disadvantages. We need to jointly control the model with epochs. One example is epoch usually increases when the batch size grows. Epoch tested are: 8, 16, 32, 64, 128, 256, 512.

Batch Size: 100



Batch Size: 500





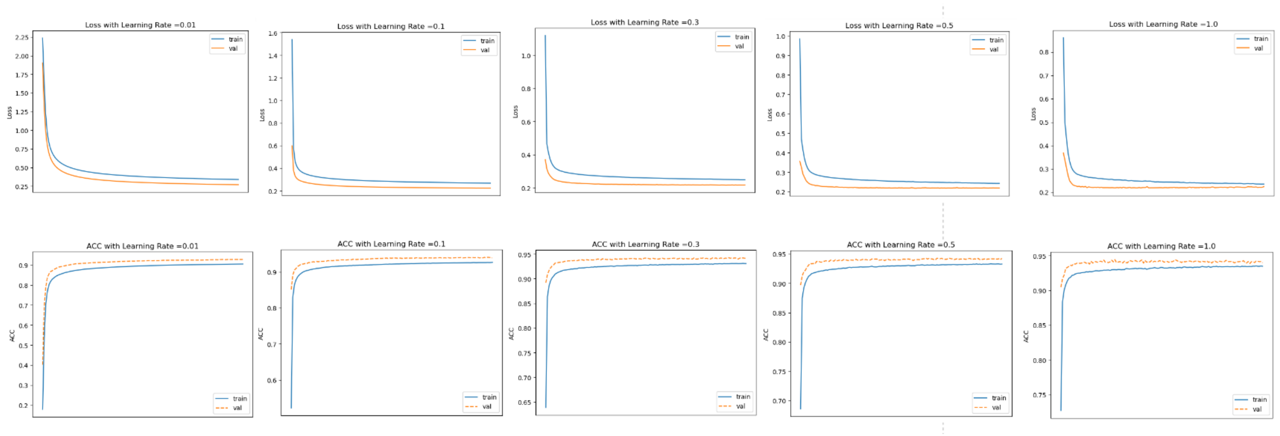
Observation:

1. The balance between batch sizes and epochs has sufficient impact on how the model converges. Small batch size with large epochs can cause frequent local oscillations[5]. This can be seen from the diagram when the batch size is set to 100. Under this parameter setting, we can see that both the validation loss and accuracy fluctuation gets greater. The loss on the train set still shows a downward trend at the last epoch. This implies the model is still learning, usually due to underfitting.
2. With batch size set to 100, when the epoch gets above 16, the loss curve on the train set follows a downward trend whilst follows an upward trend on the validation set. The occasion happens when the structure design of the network or the hyperparameter are completely unreasonable.
3. The phenomenon in 2 also happens when the batch size is set to 500, but with less fluctuations. It is fair to make a conclusion that a model with a small batch size should not have a large epoch. The model can be too sensitive to noises and outliers.
4. Larger batch sizes imply smoother loss curves and more stable performance. Larger batch sizes also converge much faster than small batch sizes.
5. When the batch size is 5000 and the epoch is 128, the loss on the train set and the validation set gets to the lowest points. When the epoch gets greater than 128, the loss bounces back at a minor rate. Our model converges to a local optimal solution with this parameter combination.
6. For all batch sizes, the train time linearly increases with the epoch values in positive directions.
7. We will use batch size: 5000 and epoch 128 for tuning momentum and learning rate, since it has the best convergence and accuracy. The loss on the validation and train sets both decrease towards stability.

3.3 Learning Rate

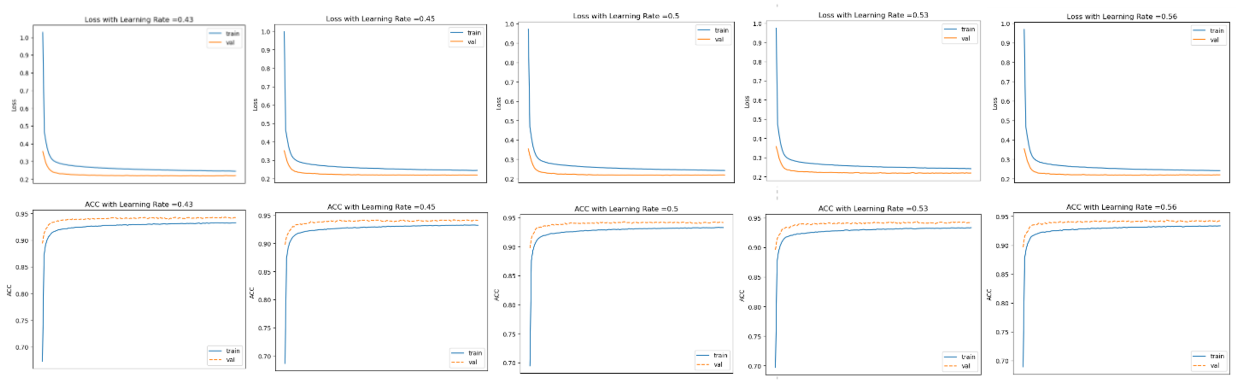
The amount that the weights are updated during training is referred to as the step size or the learning rate, often in the range between 0.0 and 1.0[6].

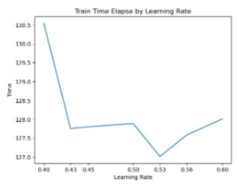
A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck. The challenge of training deep learning neural networks involves carefully selecting the learning rate.

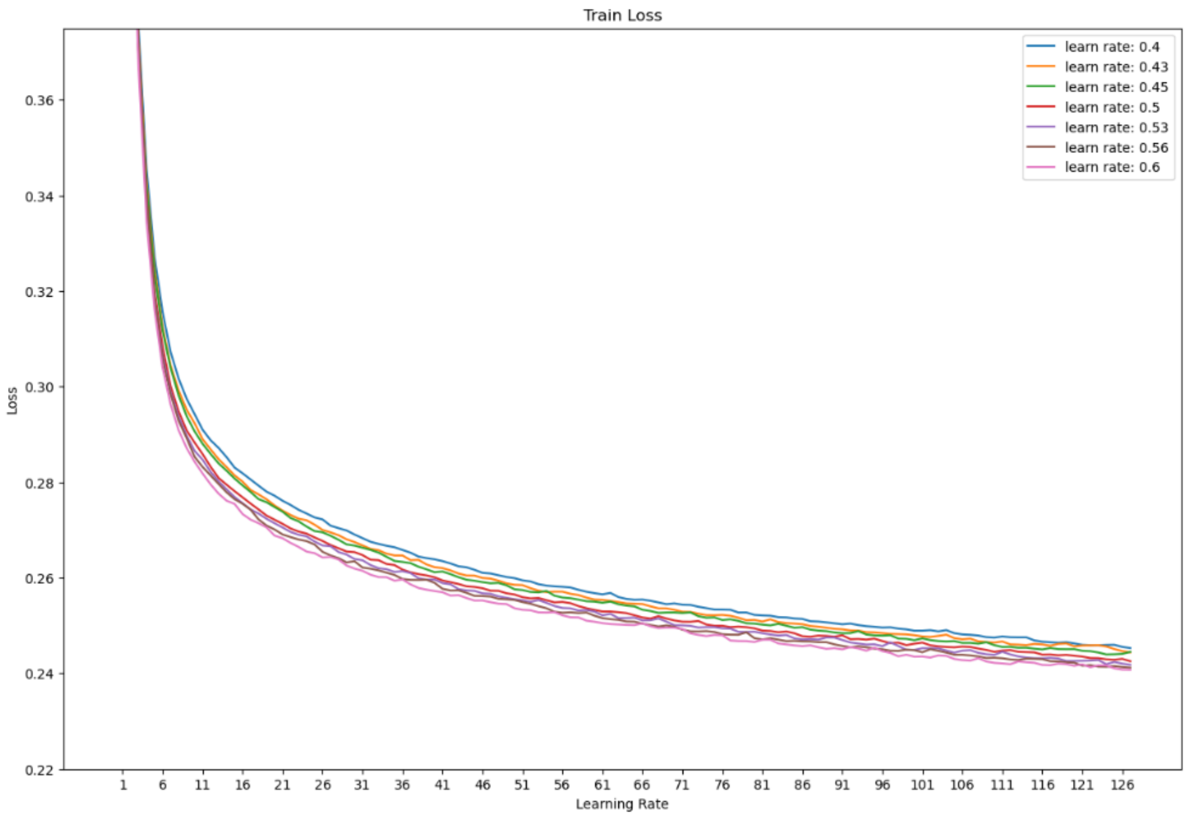


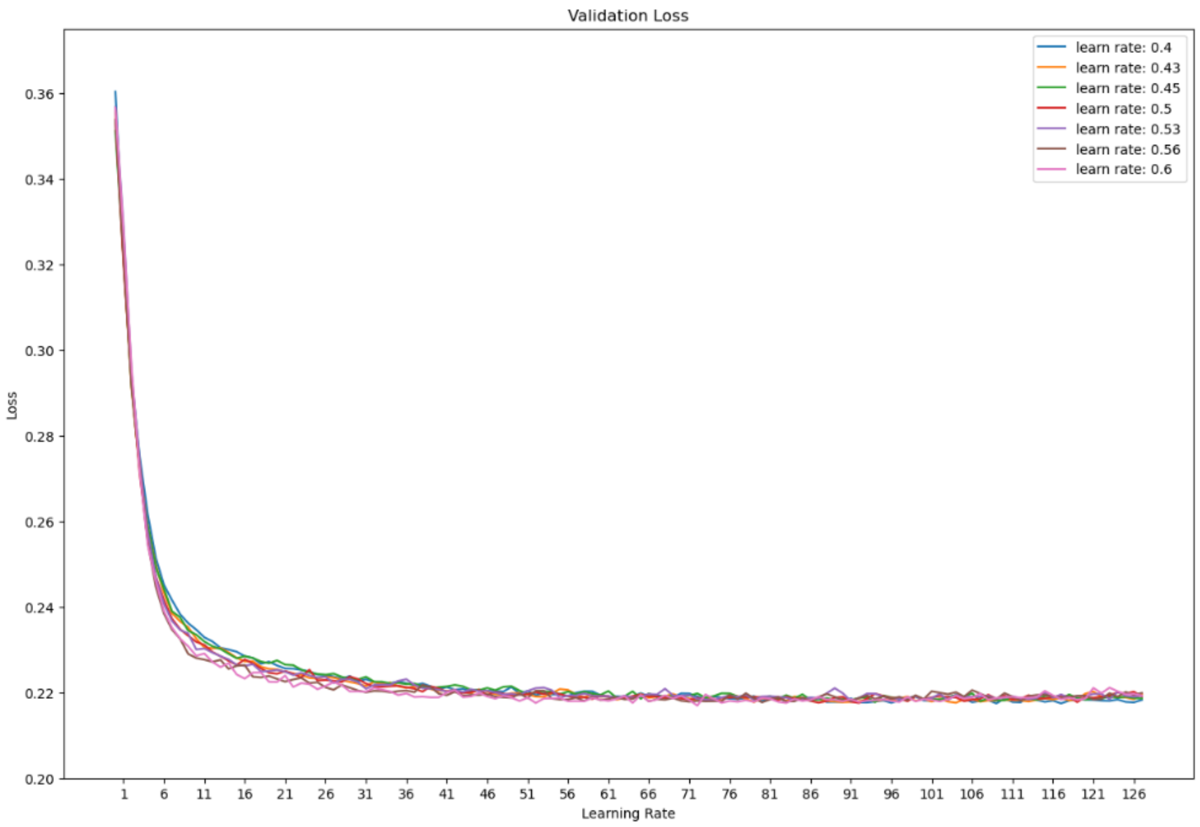
Observation:

1. Most of the batch average loss reaches to 0.2 and accuracy scores reach to 0.94-0.95.
2. Both low and high learning rates result in wasted time and resources. A lower learning rate means more training time to converge, a learning rate too high can cause the model to oscillate back and forth to the optimal solution[7].
3. The model convergence speed gets significantly improved when the learning rate switches from 0.01 to 0.1. Learning rate set to 0.01 also has the lowest loss among the tests we run. However, the time cost for training this model is the heaviest.
4. Our experiments do not show too much oscillation.
5. A desirable learning rate is one that's low enough so that the network converges to something useful but high enough so that it can be trained in a reasonable amount of time. Learning rates at 0.5 follows this rule. Therefore, we will try to find the optimal learning rate around this 0.5 value. The learning rates tested are: 0.4, 0.43, 0.45, 0.5, 0.53, 0.56, 0.6.






Train Loss
Loss
Learning Rate
learn rate: 0.4
learn rate: 0.43
learn rate: 0.45
learn rate: 0.5
learn rate: 0.53
learn rate: 0.56
learn rate: 0.6
0.36
0.34
0.32
0.30
0.28
0.26
0.24
0.22
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 106 111 116 121 126


Validation Loss
Loss
Learning Rate
learn rate: 0.4
learn rate: 0.43
learn rate: 0.45
learn rate: 0.5
learn rate: 0.53
learn rate: 0.56
learn rate: 0.6
0.36
0.34
0.32
0.30
0.28
0.26
0.24
0.22
0.20
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 106 111 116 121 126

	Train Time	Train Loss Average	Val Loss Average	Train Loss Variance	Val Loss Variance	Train ACC Average	Val ACC Average	Train ACC Variance	Val ACC Variance
Learn Rate: 0.43	127.75	0.2694	0.2242	0.00540	0.000317	0.9249	0.9394	0.000561	3.92089
Learn Rate: 0.53	127.01	0.2660	0.2245	0.00485	0.000305	0.9263	0.9398	0.000469	3.53123

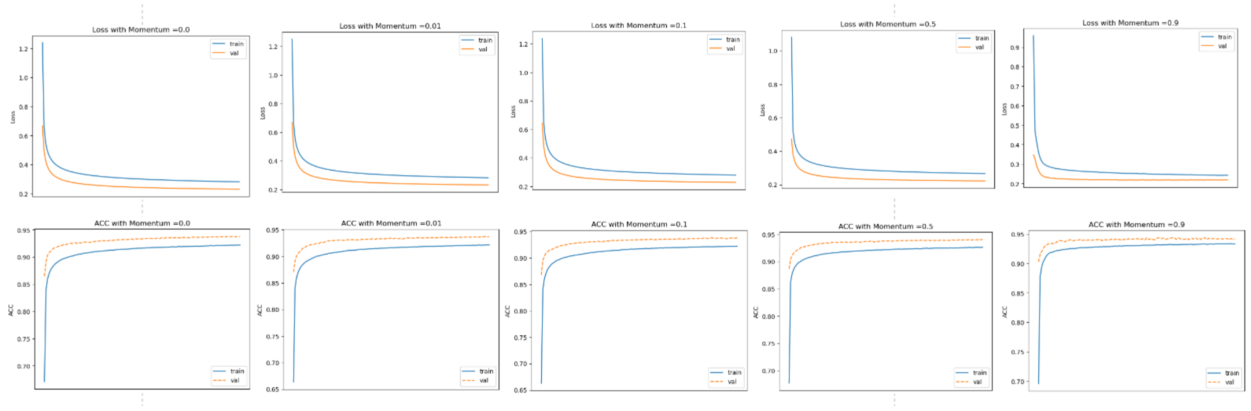
Observation:

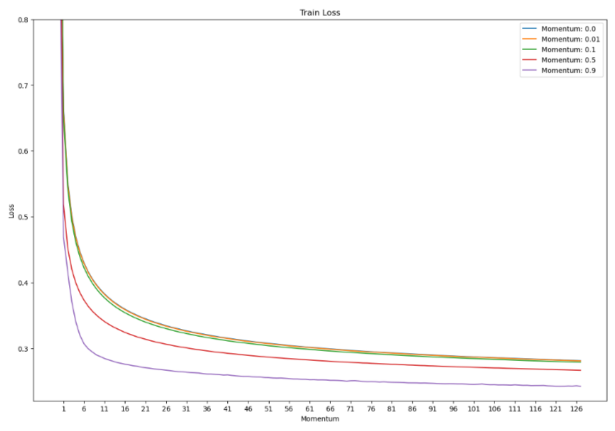
1. The lowest train time is around 127 seconds with the learn rate set to 0.53. The second lowest is also around 127 seconds with the learn rate at 0.43.
2. The lowest loss from above diagrams is when the learning rate is 0.43.
3. On the train set, the convergence happens earlier when the learning rate is greater.
4. Loss on the validation set drops down significantly, then follows a gentle downward trend and finally stabilizes at around 0.22.
5. So far, the optimal solution should be between learning rate 0.43 and 0.53. Above is the diagram for their performance with average scores and variances. We could see from the diagram, the learning rate 0.53 has better accuracy and loss on both the validation set and the train set. Variances between epochs for learning rate 0.53 are also lower on accuracy and loss. This could potentially suggest more stability and better performance from a statistical point of view. Therefore, we will use this learning rate for studying the impact of the momentum.

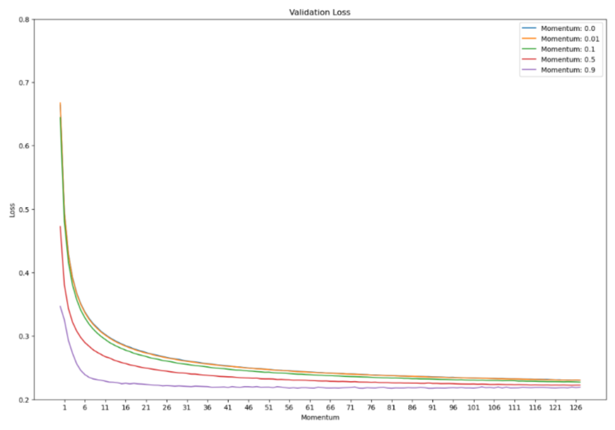
3.4 Momentum

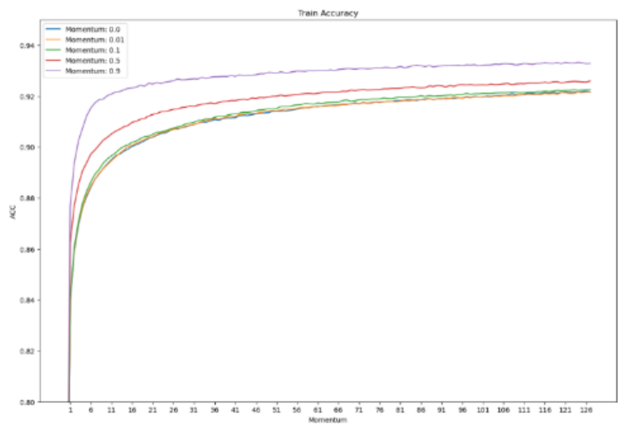
Momentum or SGD with momentum is a method which helps accelerate gradient vectors in the right directions, thus leading to faster convergence. Momentum accumulates the gradient of the past steps to determine the direction to go[8]. Since experiments before all initialize models with a momentum set to 0.9, an assumption is made that when we test smaller momentum, the convergence will be slower.

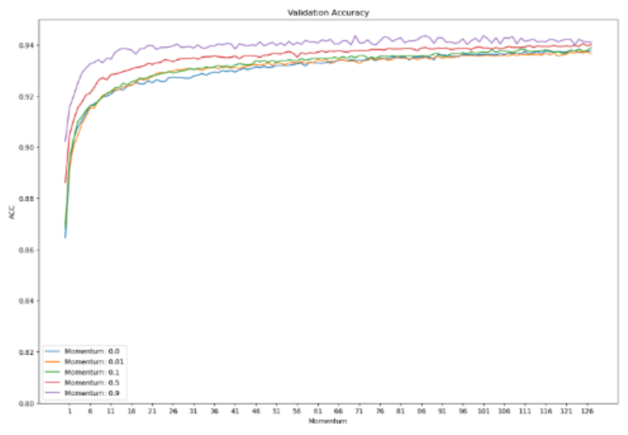
In practice, the coefficient of momentum is initialized at 0.5, and gradually annealed to 0.9 over multiple epochs. The experiment will, however, use the same momentum across all epochs[9]. Momentum tested are: 0.0, 0.01, 0.1, 0.5, 0.9.

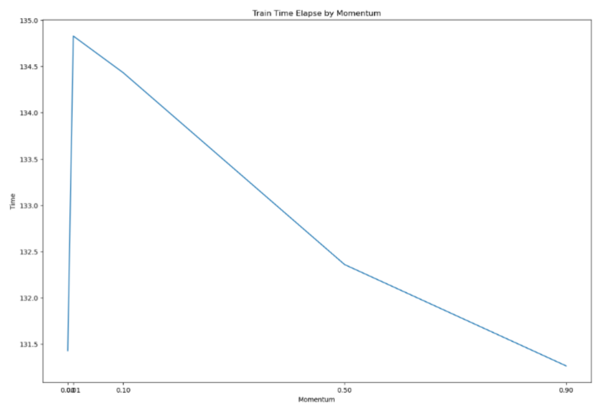












Observation:

Train time increases massively when the momentum increases from 0.0(disable) to 0.01.

As expected, the greater the momentum is, the faster the convergence happens, the smaller the loss gets and the better the accuracy of the model prediction will be. Our experiment does not have obvious overfitting, but we could still see a sign of it from the curves when momentum is set to 9 and when epoch is above 70, as the loss starts to have a minor upward trend.

4 Test Set Result

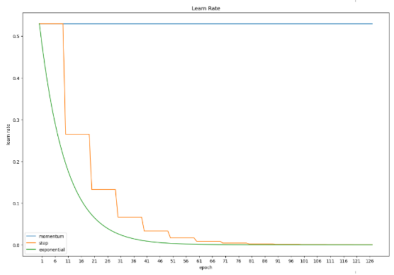
In the previous section, we chose a softmax regression model with batch size 5000, max epoch 128, learning rate 0.53 and momentum 0.9 to be our final model. On the test set, the loss reaches 0.2681 and accuracy gets to 0.9253.

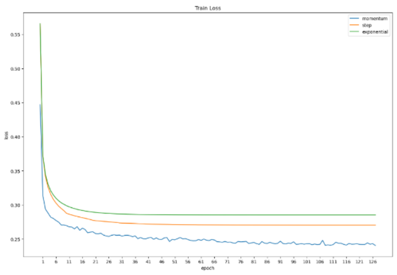
5 Learning Rate Schedules

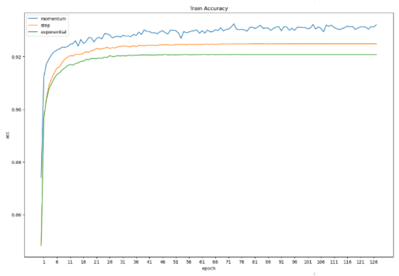
Learning rate schedules seek to adjust the learning rate during training by reducing the learning rate according to a predefined schedule. Common learning rate schedules include time-based decay, step decay and exponential decay[10].

In our experiments, we test: 1. Step decay that drops the learning rate by half every 10 epochs; 2. Exponential decay with decay scalar set to 0.1. Both are common hyperparameters for decay.

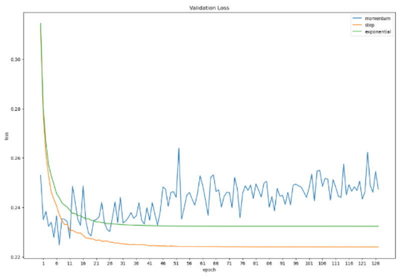
Batch Size: 500

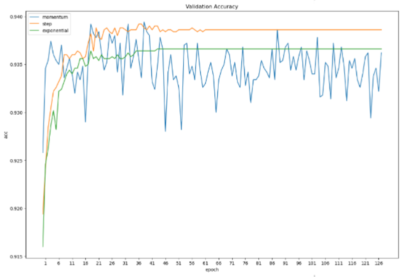




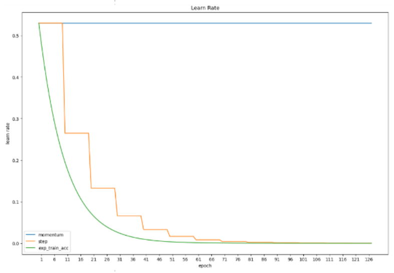


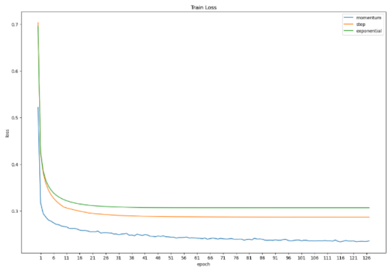
Decay	Train Time(second)
Momentum	134.5563
Step	179.1222
Exponential	280.7093

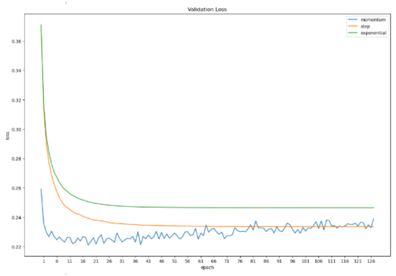




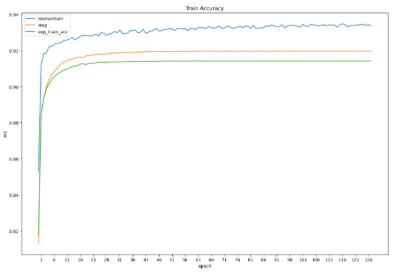
Batch Size: 1000

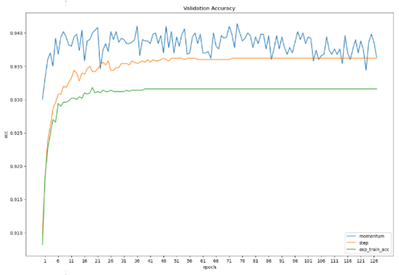




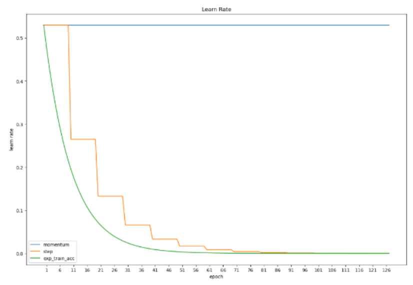


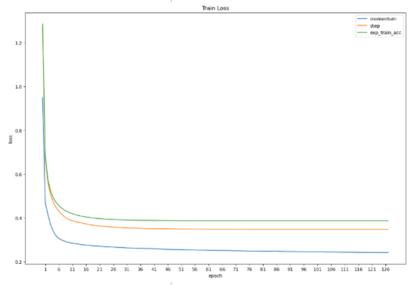
Decay	Train Time(second)
Momentum	130.8779
Step	127.6038
Exponential	127.1685

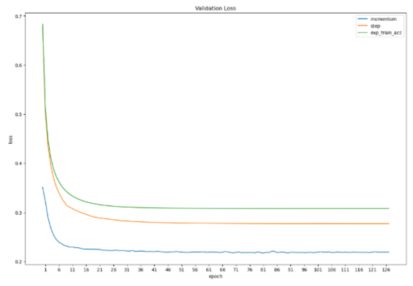




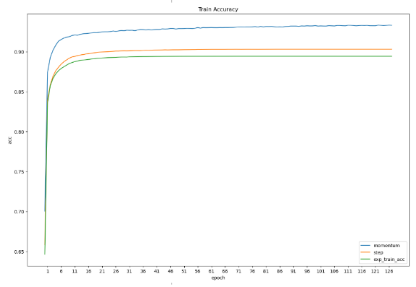
Batch Size: 5000

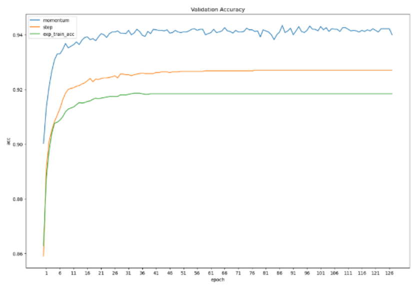






Decay	Train Time(second)
Momentum	128.6791
Step	127.3494
Exponential	127.6608





Observation:

Decay for the learning rate is always the same. This is because we are decaying by epochs(schedules), rather than by training results per iteration. Exponential decay offers a smooth curve on learning rate update, which is associated with smooth curves(less fluctuation) on loss and accuracy. The learning rate stops updating(model stops learning) at around 65 epochs for exponential decay and 90 for step decay, and neither of them shows a trend of overfitting in the first 90 epochs.

When batch size is set to 500, both loss and accuracy on the train set fluctuate greatly with SGD(momentum). As analyzed in Section 1, the smaller the batch size is, the more random the gradient can be. Under this occasion, step decay and exponential decay show much smoother convergence and better accuracy. Step decay is better than exponential decay, and converges faster.

When batch size grows to 1000, fluctuations with SGD become smaller. However, the loss on the train set keeps decreasing whilst following an upward trend on the validation set. This suggests that small batches do not work with a constant learn rate at 0.53 and a momentum at 0.9. Same as when batch size is 500, step decay is better than exponential decay. An interesting observation is that SGD converges fastest and has the lowest loss on the train set. However, although being unstable, it has the largest loss and highest accuracy on the validation set. A potential reason could be that small batches usually means the model can be over-sensitive to noises, that could lead to worse predictions. And with the epoch increasing, the model gets too 'confident' and bad predictions are penalized much more strongly than good predictions are rewarded[11]. Basically, when both accuracy and loss are increasing, the network is starting to overfit, and both phenomena are happening at the same time.

The largest batch we test is of size 5000. SGD's loss and accuracy are stable and have the best performances. Step decay is the second best. We could still see small fluctuations on SGD accuracy scores on validation. One could say that the network is starting to learn patterns only relevant for the training set and not great for generalization. The size of the batch has a huge impact for this significant improvement.

SGD usually requires more time for training, and step decay requires the least in our experiment. However, with comparatively large batch sizes, train time for step and exponential is close.

Based on the analysis above, we could at most stop training when the epoch gets to 90 before the model gets too 'confident' on the train patterns that are not relevant or harmful to generalization. Decay methods have more reliable results, smoother convergence and lesser learning cost.

6 Adaptive Learn Rate -- Adam

The Adaptive Movement Estimation algorithm, or Adam for short, is an extension to gradient descent and a natural successor to techniques like AdaGrad and RMSProp that automatically adapts a learning rate for each input variable for the objective function and further smooths the search process by using an exponentially decreasing moving average of the gradient to make updates to variables[12].

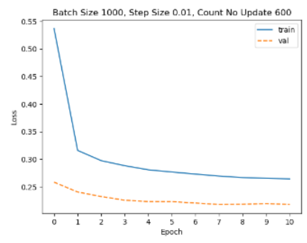
```
**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)
```

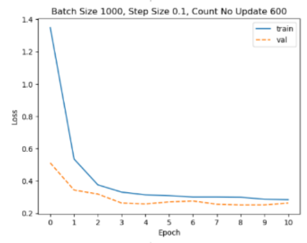
We do not have enough time for parameter tuning. Therefore, we will fix beta1 at 0.9 and beta2 at 0.99. Since they are normally good for most of the data.

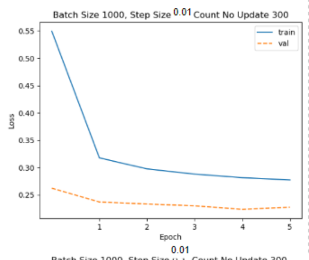
The way we define convergence is that we measure the distances between 1. new weight and weight from last step; 2. new bias and bias from last step. If both distances are lower than a certain threshold for a certain amount of continuous steps, we say the model has converged and the training should end. Below are some code snippets and experiment diagrams.

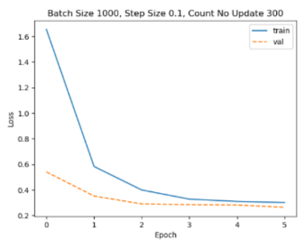
```python
def is_converge(self, new_W, new_b):
    # if delta is smaller than a given threshold for continuous times, we see it as converge
    dist_W = np.linalg.norm(self.old_W - new_W)
    dist_b = np.linalg.norm(self.old_b - new_b)

    if dist_W < self.W_threshold and dist_b < self.W_threshold:
        self.converge_count += 1
    else:
        self.converge_count =0

    if self.converge_count > self.count_threshold:
        return True
    return False
```

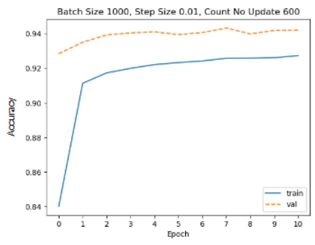
	Converge Epoch	Train Time (second)	Validation Loss Mean	Validation Loss Variance	Validation Accuracy Mean	Validation Accuracy Variance
Batch size: 1000 Step size: 0.01 Max No update count: 600	11	12.195	0.2272057	0.001562	0.939163	0.0001719
Batch size: 1000 Step size: 0.1 Max No update count: 600	11	11.187	0.29643288	0.0595218	0.929963	0.0002457
Batch size: 1000 Step size: 0.1 Max No update count: 300	6	6.0199	0.3352116	0.0548986	0.930	4.6988
Batch size: 1000 Step size: 0.01 Max No update count: 300	6	6.0887	0.233698	0.0005984	0.93596	9.1441
Batch size: 5000 Step size: 0.01 Max No update count: 600	55	57.306	0.232073	0.041393	0.937356	0.004506
Batch size: 5000 Step size: 0.1 Max No update count: 600	55	58.442	0.294389	1.574028	0.935010	0.0071816
Batch size: 5000 Step size: 0.1 Max No update count: 300	28	30.8305	0.35916	1.56766	0.93177	0.00958
Batch size: 5000 Step size: 0.01 Max No update count: 300	28	30.441	0.244630	0.02920	0.9335	0.00325

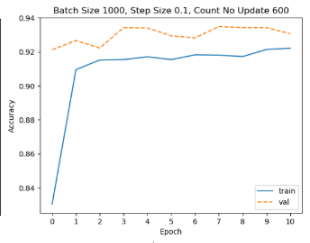


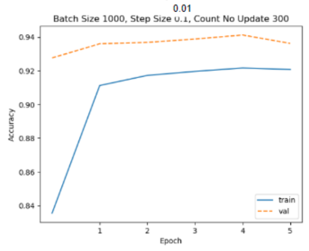


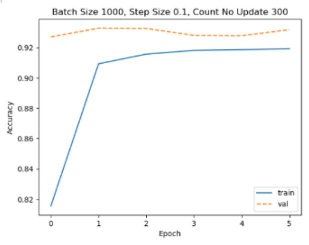


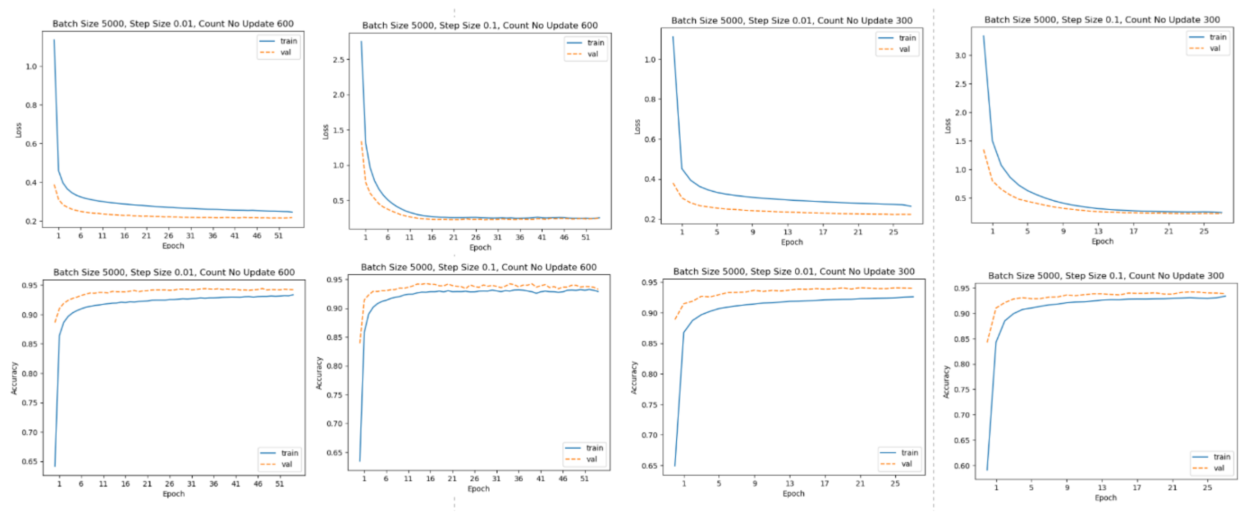












When the batch size is set to 1000, all four models we test have fluctuations on validation loss and accuracy, especially when the decay step size is set to 0.1 and max step count for no update is 600. With this model, we could see the validation loss bumps at epoch 2, 3, 6, 8. Small batch sizes could easily lead to high noise sensitivity. And large step size makes the weight and bias jump around the optimal but never get to the best solution(local oscillations). Also, at least 600 steps with no update to stop training is a strict rule. We basically keep emphasizing the noises and rules that can hurt generalization just to reach this 600 limit. This is the reason why reducing the step size or the count limit can both produce much smoother curves(loss and accuracy) and smaller fluctuations.

By comparing the variance and mean for loss and accuracy on epochs, step size 0.01 and count limit 600 works the best on batch size 1000. Both the loss and accuracy variances are the smallest, and the mean being the highest.

For batch size 5000, all curves are smoother than with batch size 1000. Step size 0.01 works better than 0.1, with less fluctuation, smaller mean and variance on loss, and higher mean and lower variance on accuracy. This phenomenon is analyzed above. With this batch size, step size at 0.01 and count limit at 600 produce the best result.

In our experiment, the amount of epochs to take to converge is the same if the batch size and count limit are the same. We could also see from the curves that for a comparatively good performance, the loss on the train set starts to get stable at about the first 30% epochs. Our computing gives small updates on weights and biases. And if we set a high count limit, there could be 2 situations: (1) with a small step size, the regression gets closer and closer to the optimal solution with minor changes; (2) with a large step size, the regression skips the optimal solution and bouncing around this point, due to the decay is too significant to lower the changes on weights and biases.

Overall, Adam can give reliable and statistical significance good learning results, with smooth loss and fast convergence, stable accuracy and less learning cost. It performs larger updates for more sparse parameters and smaller updates for less sparse parameters. However, it can be proven too aggressive and stops learning too early when training deep neural networks.

7 Conclusion

The optimal model we choose has the parameters as following: learning rate 0.53, momentum 0.9, batch size 5000 and max epochs 128. The accuracy on the test set reaches to 0.9253 and has average loss at 0.2663.

The balance between batch size, max epoch, learning rate and momentum, has decisive influences over the learning results. Small batch sizes could lead to underfitting or over-sensitivity on the data noise, large batch sizes with large epochs commit to more stable loss but they can also decrease the generalization ability of the model. They also affect the train cost and convergence. Learning rate and momentum also have a great impact on how a model converges. Large values for learning rates can help the model converge faster. However, it is also possible to result in minimal optimal or local oscillations around the optimal solutions.

Learning rate decay and adaptive learning rate are common methods to tackle these issues. In our experiment, we test step decay and exponential decay. Due to the limitation of time, the compare and contrast stage does not invoke a parameter tuning stage. Therefore, it would be too reckless to conclude which one is better for our data and tasks. But we could say that decaying can lead to faster convergence and smoother loss and accuracy, especially for exponential decay. SGD only becomes the best until the batch size becomes huge. Small fluctuations still happen on this 'SGD' best performance. It could be that the model starts to only learn and use patterns that are good for generalization.

As an extended interest, Adam(Adaptive Movement Estimation algorithm) is implemented. The best model among the ones we test is: batch size 5000, decay for mean 0.9, decay for variance 0.99, step size 0.01, and the model will stop learning if updates on biases and weights are lower than a small threshold for 600 continuous steps. The mean accuracy and mean loss for this model on the test set prediction are 0.9264 and 0.2643. Overall, Adam can give reliable and statistical significance good learning results, with smooth loss and fast convergence, stable accuracy and less learning cost.

There is an article “Without-Replacement Sampling for Stochastic Gradient Methods”[14] addressing that sampling without replacement is proven to be better. If time applies, it could be interesting to investigate how the model performs on with/without-replacement sampling mini batches. Other topics we could look into would be more systematic tests on parameter tuning for adaptive learning rate and learning rate schedule decay. Also, the way we measure convergence in the Adam implementation can be better.

8 References

[1] MINST database http://yann.lecun.com/exdb/mnist/

[2] Disadvantages of small batch sizes and how to balance out batch size
https://blog.csdn.net/weixin_45928096/article/details/123643006

[3] Fluctuating loss during training for text binary classification
https://stackoverflow.com/questions/63743557/fluctuating-loss-during-training-for-text-binary-classification

[4] What is Epoch in Machine Learning?
https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-epoch-in-machine-learning

[5] How to know if a model has converged? https://blog.csdn.net/qq_36230981/article/details/129216625

[6] Understand the Impact of Learning Rate on Neural Network Performance
https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/#:~:text=A%20learning%20rate%20that%20is,carefully%20selecting%20the%20learning%20rate

[7] Understanding Learning Rate
https://towardsdatascience.com/https-medium-com-dashingaditya-rakhecha-understanding-learning-rate-dd5da26bb6de

[8] Optimization in Deep Learning
https://medium.com/ai%C2%B3-theory-practice-business/optimization-in-deep-learning-5a5d263172e#:~:text=Momentum%20or%20SGD%20with%20momentum,determine%20the%20direction%20to%20go

[9] What is momentum in Machine Learning?
https://www.tutorialspoint.com/what-is-momentum-in-machine-learning

[10] Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning
https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

[11] How is it possible that validation loss is increasing while validation accuracy is increasing as well
https://stats.stackexchange.com/questions/282160/how-is-it-possible-that-validation-loss-is-increasing-while-validation-accuracy

[12] How to implement an Adam Optimizer from Scratch
https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc

[13] Kingma, D.P. and Ba, J. (2017) Adam: A method for stochastic optimization, arXiv.org. Available at: https://arxiv.org/abs/1412.6980 (Accessed: 27 October 2023).

[14] Shamir, O. (2016) Without-replacement sampling for stochastic gradient methods: Convergence results and application to distributed optimization, arXiv.org. Available at: https://arxiv.org/abs/1603.00570 (Accessed: 27 October 2023).