Simple Linear Model for Handwritten Digit Classification: A Report

Khuong Thanh-Gia-Hieu
Alex-Răzvan Ispas
Philippe Masouf

Abstract

In this report, we explore the development and evaluation of deep multi-layer neural networks for handwritten digit classification using the MNIST dataset. We begin by implementing a linear network and gradually build upon it to create a deep network with multiple layers and non-linear activation functions. Our investigation covers various aspects of the training process, including forward and backward passes, optimization techniques, and parameter initialization. We compare the performance of different network architectures, activation functions, and optimization algorithms, highlighting the impact of these choices on the final classification accuracy.

1. Introduction

Handwritten digit recognition is a fundamental task in computer vision and pattern recognition, often serving as a benchmark problem for machine learning algorithms. In this report, we present a simple linear model for handwritten digit classification using the MNIST dataset. The MNIST dataset is a widely-used dataset containing 70,000 images of handwritten digits (0-9) with 60,000 images for training and 10,000 images for testing. The objective of this report is to observe how changes in different parameters of a simple neural network affect the learning process.

2. Simple Linear Model

2.1. Architecture

Our model is a single-layer neural network [4], which can be considered as an affine transformation followed by a softmax function. The affine transformation maps the input vector (an image of a handwritten digit) to a vector of logits. The softmax function then converts the logits into probabilities for each class. The architecture can be summarized as:

1. Input: a flattened MNIST image (28 x 28 pixels) represented as a 784-dimensional vector.
2. Affine transformation: a matrix multiplication followed by a bias addition.
3. Output: a 10-dimensional vector representing the probability distribution of the 10 classes (digits 0-9).

2.2. Training

The training process involves several steps, including forward pass, backward pass, and parameter updates using Momentum Stochastic Gradient Descent (SGD)[6].

2.2.1. Forward Pass

In the forward pass, the input vector x is transformed using the affine transformation and softmax function to generate a probability distribution over the 10 classes. This is achieved by computing:

$$z = Wx + b$$
$$y = \text{softmax}(z)$$

In the steps above, W is the weight matrix, b is the bias vector, and y is the output probability distribution. The formula of the softmax function is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$$

2.2.2. Backward Pass

The backward pass computes the gradient of the negative log-likelihood (NLL) loss with respect to the model parameters. The NLL loss is defined as:

$$y = \text{softmax}(z)$$
$$\mathcal{L} = -\log(y_{gold})$$

The gold index is the index of the true class. Firstly, we need to calculate the gradient with respect to the output of the affine transformation $z = Wx + b$. Calculating the gradient by using the chain rule is computationally demanding. Instead of using the chain rule, we can take a shortcut by calculating the loss only for the $gold$ value. The negative log-likelihood with respect to the gold value is defined as:

$$\mathcal{L}(x, gold) = -\log \frac{\exp(x_{gold})}{\sum_j \exp(x_j)} = -x_{gold} + \log \sum_j \exp(x_j)$$

We can now take the partial derivative of this loss with respect to a general component z_i:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{\partial - z_{gold} + \log(\sum_{j=1}^{n} exp(z_j))}{\partial z_i}$$
$$\frac{\partial \mathcal{L}}{\partial z_i} = -gold \cdot \delta_i^{gold} + \frac{exp(z_i)}{\sum_{j=1}^{n} exp(z_j)}$$
$$\frac{\partial \mathcal{L}}{\partial z_i} = -gold \cdot \delta_i^{gold} + \text{softmax}(z_i)$$

The δ_i^{gold} symbol is Kronecker's delta[1] which is 1 if $i = gold$ and 0 if the contrary. Now, the gradient of the negative log likelihood with respect to z is equal to:

$$\frac{\partial \mathcal{L}}{\partial z_i} = -y_{gold} + \hat{y}$$

where y_{gold} is the one hot encoded vector of the gold value and $\hat{y}$ contains the computed values of the softmax.

In order to update the parameters of the model, we also need to calculate the gradient with respect to the weights W and the bias b. The chain rules for the two gradients are:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial W} = (-y_{gold} + \hat{y}) \cdot x$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b} = (-y_{gold} + \hat{y}) \cdot 1$$

In the algorithm, the gradient is calculated with the following functions:

$$g_x = \text{backward\_nll}(x, \text{gold}, 1)$$
$$(g_W, g_b) = \text{backward\_affine\_transform}(W, b, x, g_x)$$

2.3. Experiment and Results

We trained the simple linear model on the MNIST dataset for 5 epochs using a step size of 0.01. The model achieved a test accuracy of approximately 92%, demonstrating the effectiveness of the simple linear model for handwritten digit classification.

3. Deep Multi-layer model

3.1. Architecture

For the multi-layer model[2] we conducted experiments on an architecture with three hidden states. The number of neurons for each hidden states is respectively: 100, 50 and 20. The input size is 28*28 and the output size remains 10. Each hidden state performs an affine transformation where the input is the activation of the previous hidden state. For the activation function, we use ReLU[5] and for the readout function we use softmax. The general formula for computing the activation of a hidden state is:

$$z^{(k)} = \text{ReLU}(W^{(k)} \cdot z^{(k-1)} + b^{(k)})$$

For the first layer, $z^{(k-1)}$ will be the input X and for the final one, the activation function will be softmax. The ReLU function is defined as:

$$Relu(z) = max(0, z)$$

3.2. Training

The training process will cover the forward pass, the backward pass, the Momentum Stochastic Gradient Descent, the Adam optimizer and the cosing learning rate scheduler.

3.2.1. Forward Pass

The forward pass is a chain of three affine transformations that have ReLU as activation function, ending up with a softmax readout. This is achieved by computing:

$$
\begin{aligned}
z^{(1)} &= W^{(1)} \cdot x + b^{(1)} \\
a^{(1)} &= \text{ReLU}(z^{(1)}) \\
z^{(2)} &= W^{(2)} \cdot a^{(1)} + b^{(2)} \\
a^{(2)} &= \text{ReLU}(z^{(2)}) \\
z^{(3)} &= W^{(3)} \cdot a^{(2)} + b^{(3)} \\
a^{(3)} &= \text{ReLU}(z^{(3)}) \\
z^{(4)} &= W^{(4)} \cdot a^{(3)} + b^{(4)} \\
p &= softmax(z^{(4)})
\end{aligned}
$$

3.2.2. BACKWARD PASS

In the backward pass, we utilise the chain rule to obtain the gradient. The gradient with respect to the weights of the kth layer W_k and the bias b_k knowing that model has in total n layer, together with the output, one is:

$$
\frac{\partial \mathcal{L}}{\partial W^{(k)}} = \frac{\partial \mathcal{L}}{\partial z^{(n)}} \cdot \frac{\partial z^{(n)}}{\partial a^{(n-1)}} \cdot \frac{\partial a^{(n-1)}}{\partial z^{(n-1)}} \cdots \frac{\partial a^{(k)}}{\partial z^{(k)}} \cdot \frac{\partial z^{(k)}}{\partial W^{(k)}}
$$

$$
\frac{\partial \mathcal{L}}{\partial b^{(k)}} = \frac{\partial \mathcal{L}}{\partial z^{(n)}} \cdot \frac{\partial z^{(n)}}{\partial a^{(n-1)}} \cdot \frac{\partial a^{(n-1)}}{\partial z^{(n-1)}} \cdots \frac{\partial a^{(k)}}{\partial z^{(k)}} \cdot \frac{\partial z^{(k)}}{\partial b^{(k)}}
$$

Recalculating the chain rule for each weight and bias is demanding a lot of computation. Therefore, we are going to make the computation less redundant by implementing a computation graph which is going to calculate the backward propagation in topological order. Each node will be able to persist its gradient such that the previous parameters can use it to compute their backward gradient.

Let $z^{(k)} = W^{(k)} * z^{(k-1)} + b^{(k)}$ be the affine transformation from hidden layer k whose gradient was aleady computed and it is equal to $\frac{\partial \mathcal{L}}{\partial z^{(k)}} = g_{z^{(k)}}$. The gradient with respect to $W^{(k)}$ and with respect to $b^{(k)}$ can be written as:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W^{(k)}} &= \frac{\partial \mathcal{L}}{\partial z^{(k)}} \cdot \frac{\partial z^{(k)}}{\partial W^{(k)}} = g_{z^{(k)}} \cdot \frac{\partial z^{(k)}}{\partial W^{(k)}} = g_{z^{(k)}} \cdot a^{(k-1)} \\
\frac{\partial \mathcal{L}}{\partial b^{(k)}} &= \frac{\partial \mathcal{L}}{\partial z^{(k)}} \cdot \frac{\partial z^{(k)}}{\partial b^{(k)}} = g_{z^{(k)}} \cdot \frac{\partial z^{(k)}}{\partial b^{(k)}} = g_{z^{(k)}} \cdot 1
\end{aligned}
$$

If we want to compute the gradient with respect to the affine transformation $z^{(k)}$, by knowing the gradient of the loss with respect to $z^{(k+1)}$, we can compute the following chain:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z^{(k)}} &= \frac{\partial \mathcal{L}}{\partial z^{(k+1)}} \cdot \frac{\partial z^{(k+1)}}{\partial a^{(k)}} \cdot \frac{\partial a^{(k)}}{\partial z^{(k)}} \\
\frac{\partial \mathcal{L}}{\partial z^{(k)}} &= g_{z^{(k+1)}} \cdot W^{(k+1)} \cdot \frac{\partial a^{(k)}}{\partial z^{(k)}}
\end{aligned}
$$

The partial derivative of $a^{(k)}$ with respect to $z_i^{(k)}$ is:

$$
\frac{\partial a^{(k)}}{\partial z_i^{(k)}} = \mathbb{1}[a_i^{(k)} > 0]
$$

3.2.3. MOMENTUM STOCHASTIC GRADIENT DESCENT (SGD)

Momentum SGD is a variation of the standard stochastic gradient descent algorithm that incorporates the concept of momentum to achieve faster convergence and better stability. The idea behind momentum is inspired by the physical interpretation of learning, where the optimization process can be imagined as a ball rolling down a hill. As the ball rolls, it accumulates momentum, enabling it to overcome local minima and traverse the optimization landscape more efficiently.

In the context of optimization, momentum is implemented by maintaining a moving average of the gradients and using this accumulated momentum to update the weights. This approach reduces the oscillations in the parameter updates and helps the optimizer converge more quickly.

The update rule for Momentum SGD can be formulated mathematically as follows:

$$
v_{t+1} = \mu \cdot v_t + (1 - \mu) \cdot \nabla_J(W_t) \tag{1}
$$

$$
W_{t+1} = W_t - \eta \cdot v_{t+1} \tag{2}
$$

where:

- W_t represents the model parameters at iteration t.
- $J(W_t)$ is the loss function evaluated at θ_t.
- $\nabla_W J(\theta_t)$ denotes the gradient of the loss function with respect to the model parameters W_t.
- η is the learning rate.
- v_t is the velocity vector at iteration t, initialized to zero.
- μ is the momentum coefficient, a hyperparameter typically set to a value between 0 and 1.

After training the deep network with MomentumSGD for 5 epochs, we obtain the following results:

- Train accuracy: 98.75%
- Dev accuracy: 97.45%

Compared to the standard SGD, MometontumSGD offers several advantages, such as faster convergence at the beginning of the training process and better final performance (97.45% dev accuracy compared to 97.24%). Since the gradient with a batch size of 1 is very noisy, the momentum helps decrease the noisiness, leading to more efficient optimization and higher accuracy.

3.3. Adam

Adam (Adaptive Moment Estimation)[3] is an optimization algorithm used in deep learning to update the weights of a neural network during training. Adam combines ideas from both RMSprop and momentum to adaptively adjust the learning rate for each parameter. The main advantage of Adam over other optimization algorithms is that it requires very little memory to run efficiently on large datasets.

The Adam update equation is given by:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t, \tag{3}$$

where w_t and w_{t+1} are the weights at time step t and $t+1$, respectively, α is the learning rate, $\hat{m}_t$ and $\hat{v}_t$ are estimates of the first and second moments of the gradients, and ϵ is a small constant to prevent division by zero. The estimates are computed as:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{4}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{5}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{6}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}, \tag{7}$$

where g_t is the gradient at time step t, and β_1 and β_2 are the exponential decay rates for the first and second moments, respectively. The algorithm uses bias correction to adjust the estimates of the moments, which helps to reduce their bias towards zero at the beginning of training.

3.4. Cosine learning rate scheduler

The cosine learning rate scheduler is popular because it does not decrease too quickly at the beginning or end of training. This allows the model to learn quickly at the start and more efficiently toward the end.

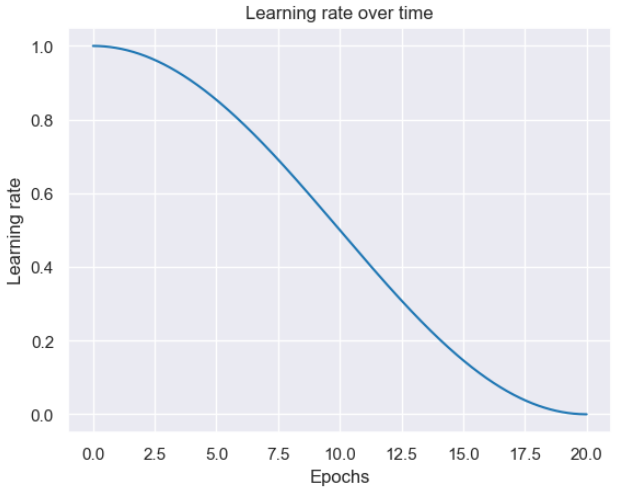

Figure 1. Cosine learning rate scheduler

3.5. Experiments and Results

To investigate the networks further, we made the following improvements on top of the original code:

- A hyperparameters grid search for the best hyperparameters
- Learning rate scheduler
- Mini Batch Gradient Descent instead of Stochastic Gradient Descent
- Adam optimizer instead of MomentumSGD
- A plot of loss and accuracy over time

3.5.1. The best hyperparameters

We performed 54 experiments with multiple parameters for a deep neural network of 3 layers with hidden sizes of $(100, 50, 20)$, respectively. The details of the result can be found in Table (1). The best model so far gives 98.4% dev accuracy with the following hyperparameters:

- lr = 0.001
- batch size = 16
- epochs = 20
- learning rate scheduler = 'cosine'

Here are the loss and accuracies over time for this particular model setting.

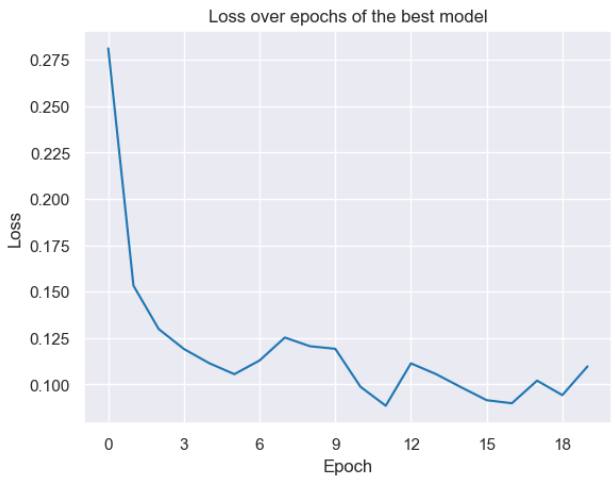

Figure 2. Training loss over time of the best model.

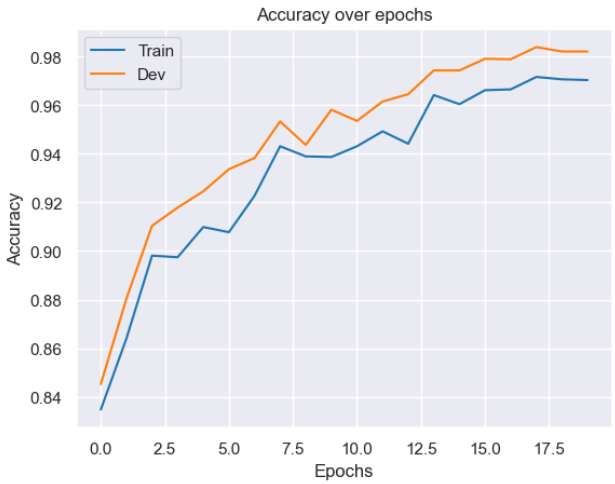

Figure 3. Train and dev accuracy over time of the best model.

3.5.2. The effect of learning rate scheduler

With the cosine learning rate scheduler, the model converges better at the end, and the whole training procedure is more stable, almost always reaching a better optimum.

4. Conclusions

4.1. Effect of batch size

Based on the results so far, we can clearly see the effect of batch size and learning rate. The lower the batch size, the lower the learning rate should be. For example:

- The relationship between batch size and learning rate is an important aspect to consider when training deep neural networks. When the batch size is smaller, the gradient estimate is noisier, and the optimization algorithm can update the parameters with more frequency. However, with a larger batch size, the gradient estimate is more accurate, and the optimization algorithm updates the parameters less frequently.
- In the experiments conducted, it was observed that with a learning rate of 0.01, only batch sizes of 64 produced promising results, while some with batch sizes of 4 did not converge at all. This indicates that for a high learning rate, a larger batch size is necessary to stabilize the training process. On the other hand, with a learning rate of 0.001, only batch sizes of 16 and some of 4 produced good results. This indicates that with a smaller learning rate, a smaller batch size is preferable to avoid overfitting and converge to a better optimum.

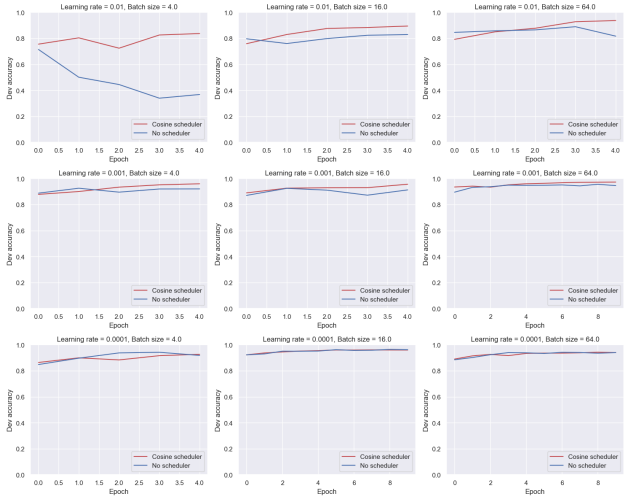

Figure 4. The difference between no learning rate scheduler and cosine learning rate scheduler

4.2. The effect of cosine learning rate scheduler

The learning rate is a hyperparameter that determines the step size at which the optimizer adjusts the weights of the model during training. If the learning rate is too high, the model may not converge to the optimal solution, and if the learning rate is too low, the model may take too long to converge or get stuck in a local minimum.

In our experiments, we observed that the choice of learning rate had a significant impact on the model's performance. Specifically, we found that:

With a high learning rate (e.g., 0.01), the model may not converge at all, especially with smaller batch sizes. For ex-

ample, some models with a batch size of 4 did not converge at all with a learning rate of 0.01. With a lower learning rate (e.g., 0.001), the model was able to converge, but the choice of batch size was also important. For example, we found that a batch size of 16 gave better results than a batch size of 4 or 64 with a learning rate of 0.001. With an even lower learning rate (e.g., 0.0001), the model may take longer to converge, but it can also give better results than with a higher learning rate. Overall, choosing the right learning rate is important to achieve good results with a deep neural network, and it often depends on other hyperparameters like the batch size and the optimizer used. Experimentation and tuning of hyperparameters are essential to find the best combination of settings for a given task.

4.3. The best hyperparameters

The best model found in the experiments had a dev accuracy of 98.4% with the hyperparameters of lr=0.001, batch size=16, epochs=20, and learning rate scheduler='cosine'. The cosine learning rate scheduler was chosen as it allows the model to learn quickly at the start of training while also enabling efficient learning towards the end. The results obtained suggest that this model is suitable for the given task and outperforms other models with different hyperparameters. However, further experiments could be conducted to investigate whether other combinations of hyperparameters could yield even better results.

References

[1] Milton Abramowitz and Irene Stegun. *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables*. Dover Publications, 1965.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[5] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. *ICML*, 2010.

[6] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. *Proceedings of the 30th international conference on machine learning*, 28(3):1139–1147, 2013.

Appendix

Table 1: Hyperparameters grid search.

experiment	lr	n.o. epochs	batch size	learning rate scheduler	train accuracy(%)	last epoch dev accuracy(%)	best epoch dev accuracy(%)	training time(s)
0	0.0100	5	4	cosine	0.835800	0.837300	0.837300	26.056545
1	0.0100	5	4	no	0.355600	0.367800	0.714500	26.466447
2	0.0100	5	16	cosine	0.880200	0.895000	0.895000	49.146702
3	0.0100	5	16	no	0.806700	0.829400	0.829400	35.355222
4	0.0100	5	64	cosine	0.885770	0.937301	0.937301	11.146196
5	0.0100	5	64	no	0.778373	0.817178	0.889829	11.197112
6	0.0100	10	4	cosine	0.586920	0.595200	0.706900	26.273140
7	0.0100	10	4	no	0.437740	0.460700	0.644000	28.059290
8	0.0100	10	16	cosine	0.911040	0.914700	0.914700	110.613092
9	0.0100	10	16	no	0.654760	0.667000	0.817500	78.591252
10	0.0100	10	64	cosine	0.897498	0.939391	0.939391	34.630888
11	0.0100	10	64	no	0.835398	0.831011	0.905951	26.520456
12	0.0100	20	4	cosine	0.370180	0.385200	0.614000	26.923552
13	0.0100	20	4	no	0.208880	0.197600	0.591700	26.980327
14	0.0100	20	16	cosine	0.707680	0.753400	0.847500	41.739444
15	0.0100	20	16	no	0.704340	0.726500	0.821300	57.544172
16	0.0100	20	64	cosine	0.896180	0.933121	0.933121	58.291984
17	0.0100	20	64	no	0.861113	0.897492	0.899881	75.231702
18	0.0010	5	4	cosine	0.971040	0.966000	0.967600	26.140442
19	0.0010	5	4	no	0.952000	0.959200	0.959200	25.762881
20	0.0010	5	16	cosine	0.951540	0.960000	0.960000	39.940569
21	0.0010	5	16	no	0.913380	0.920700	0.925900	50.126463
22	0.0010	5	64	cosine	0.913063	0.956708	0.956708	25.088587
23	0.0010	5	64	no	0.881474	0.912918	0.925060	21.322497
24	0.0010	10	4	cosine	0.981200	0.973200	0.973200	53.285697
25	0.0010	10	4	no	0.949920	0.946000	0.956400	50.052248
26	0.0010	10	16	cosine	0.968000	0.966400	0.971200	83.049522
27	0.0010	10	16	no	0.938860	0.950400	0.950400	76.957153
28	0.0010	10	64	cosine	0.928828	0.962978	0.962978	23.850039
29	0.0010	10	64	no	0.913623	0.920283	0.961982	21.587300
30	0.0010	20	4	cosine	0.974640	0.966000	0.966800	53.430004
31	0.0010	20	4	no	0.951840	0.951600	0.962400	86.218170
32	0.0010	20	16	cosine	0.970420	0.982200	0.984000	119.080513
33	0.0010	20	16	no	0.923080	0.935000	0.953300	97.080804
34	0.0010	20	64	cosine	0.927310	0.937898	0.945163	45.219236
35	0.0010	20	64	no	0.887208	0.899781	0.925955	24.520635
36	0.0001	5	4	cosine	0.945920	0.954000	0.954400	26.635183
37	0.0001	5	4	no	0.958320	0.961200	0.961200	26.519762
38	0.0001	5	16	cosine	0.934520	0.936700	0.943100	27.425412
39	0.0001	5	16	no	0.920500	0.932800	0.932800	46.873546
40	0.0001	5	64	cosine	0.898118	0.926752	0.926752	16.824355
41	0.0001	5	64	no	0.883032	0.918093	0.942377	24.439442
42	0.0001	10	4	cosine	0.963840	0.959600	0.960800	56.327372
43	0.0001	10	4	no	0.968160	0.962400	0.964400	52.821643
44	0.0001	10	16	cosine	0.948160	0.939900	0.943100	82.969690
45	0.0001	10	16	no	0.940520	0.939900	0.942100	89.553592
46	0.0001	10	64	cosine	0.920037	0.959295	0.962281	42.352957

experiment	lr	n.o. epochs	batch size	learning rate scheduler	train accuracy(%)	last epoch dev accuracy(%)	best epoch dev accuracy(%)	training time(s)
47	0.0001	10	64	no	0.899297	0.929538	0.931628	31.960392
48	0.0001	20	4	cosine	0.974160	0.969200	0.969200	69.678566
49	0.0001	20	4	no	0.974400	0.968000	0.969600	75.262942
50	0.0001	20	16	cosine	0.960120	0.961600	0.963200	144.611784
51	0.0001	20	16	no	0.937000	0.940100	0.943000	81.988964
52	0.0001	20	64	cosine	0.895540	0.927846	0.940983	24.688917
53	0.0001	20	64	no	0.908568	0.973328	0.980792	59.940841

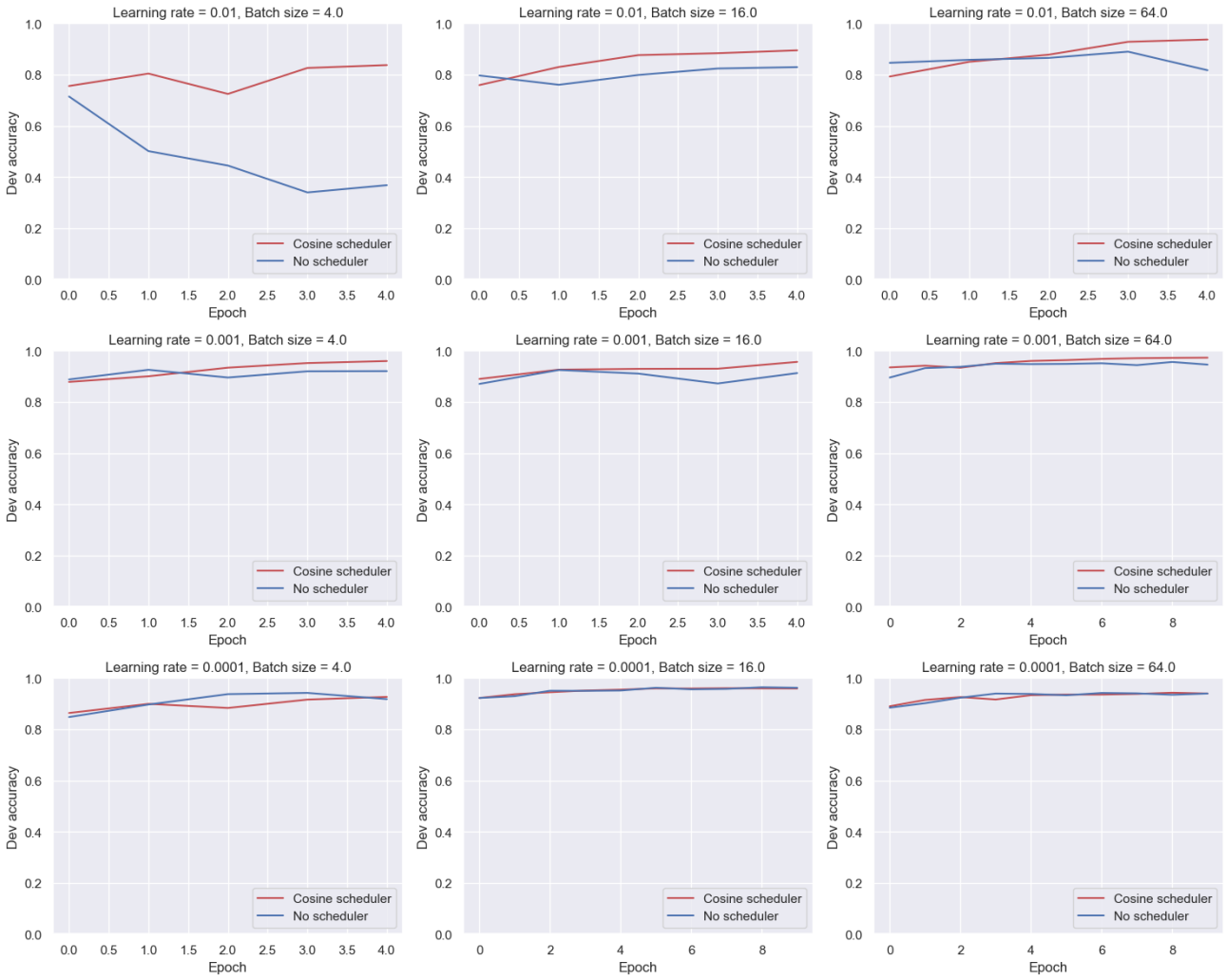

Figure 5. The difference between no learning rate scheduler and cosine learning rate scheduler