COMP36212 Assignment EX3. Gradient-Based Optimisation

Vladislav Yotkov

1 Problem Statement

In this assignment we explore the optimisation methods of an Artificial Neural Network (ANN) for image classification. The task is to train a multi-layer perceptron, applied to classify images of handwritten digits (0-9) from the MNIST dataset (Lecun et al. 1998), aiming to achieve maximum classification accuracy. The process involves the computation of a prediction from a sample input (i.e., an image), the comparison of the network prediction with the true label to formulate an objective function for optimisation (cross-entropy loss, Eq. 1), and the use of target optimisation approaches to minimise the objective function with respect to the network parameters.

$$L = -\sum_{k}^{N} \hat{y}_k \log(P_k) \tag{1}$$

1.1 Dataset

The MNIST dataset (Lecun et al. 1998) is a collection of 70,000 images of handwritten digits (0-9), each of which is a 28x28 pixel image, with pixel intensity values ranging from 0-255. For input into the ANN, it is divided into 60,000 training and 10,000 testing images, while each sample is flattened into an array of 784 elements, which is further normalised to the range $0 \leq x_i \leq 1$.

1.2 Artificial Neural Network (ANN)

The ANN is a multi-layer perceptron, consisting of five layers with a total of 784 neurons in the input layer, three hidden layers containing 300, 100, and 100 neurons respectively, and an output layer of 10 neurons (Figure 1) for a total of $276,200$ trainable parameters.

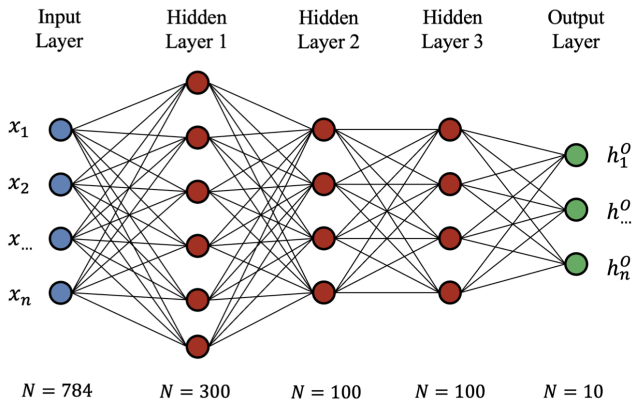


Figure 1: Artificial Neural Network (ANN) architecture, taken from the assignment specification.

The activation function used for the hidden layers is the Rectified Linear Unit (ReLU, Eq. 2), while the output layer uses the softmax function (Eq. 3), which normalises the output logits to a probability distribution over the 10 classes.

$$ReLU(x) = \max(0, x) \tag{2}$$

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{3}$$

2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent is one of the most significant and widely used optimisation algorithms used in machine learning. It is a variant of the Gradient Descent (GD) algorithm, which is used to find the local minimum of a function by iteratively moving in the direction of the negative gradient. However, the main difference is that SGD uses a random sample of the training

data in order to compute the gradient, unlike the GD algorithm which is applied over the entire dataset - leading to high computational costs. In our work, we will be exploring the **on-line** (i.e., stochastic) and **mini-batch SGD** (where $m \ll N$, such that m is the size of the mini-batch, and N is the size of the training dataset) variants of the algorithm. We can define more formally this optimisation method in Algorithm 1, while the update rule is given by Eq. 4.

$$w = w - \frac{\eta}{m} \sum_{i=1}^{m} \nabla L_i(x_i) \tag{4}$$

Algorithm 1 Mini-batch Stochastic Gradient Descent (SGD)

Require: Training dataset D
Require: Learning rate η
Require: Mini-batch size m
Ensure: Optimized model parameters w
1: Initialize model parameters w
2: **while** stopping criteria not met **do**
3: Sample an m-sized mini-batch from D: $\{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$
4: Compute gradient: $\nabla L = \frac{1}{m} \sum_{i=1}^{m} \nabla L_i(x_i, y_i)$
5: Update model weights: $w = w - \eta \nabla L$
6: **end while**
7: **return** Optimized model parameters w

We must further note a few key benefits of using an SGD over general GD:

1. On-line (i.e., stochastic) learning requires a smaller step size (i.e., learning rate) to counteract inherent noise, resulting in smoother but slower adaptation (Wilson and Martinez 2003). Despite the increased runtime, small batches offer a beneficial regularization effect and over time, noise averages out, driving the weights towards the true gradient (Goodfellow, Bengio, and Courville 2016).
2. Large batch sizes exhibit a degradation in quality (Mishkin, Sergievskiy, and Matas 2017) and a low generalisation capability and are prone to getting stuck in local minima due to their convergence to sharp minimizers of the training function (Keskar et al. 2017).

2.1 Configuration Experiments

We experimented with different hyperparameters for the SGD algorithm, in order to find the optimal configuration for our ANN classifier:

1.**Learning rate**: The learning rate η is the step size of the gradient descent algorithm, and is the most important hyper-parameter (Goodfellow, Bengio, and Courville 2016). Setting it to larger values risks rapid changes and overshooting the minimum, while converging to a suboptimal solution. On the other hand, a smaller learning rate requires more training epochs to converge, and is more likely to get stuck in a local minimum. In our work on SGD, we experimented with the following values $\eta = \{0.1, 0.01, 0.001\}$.

2.**Batch size**: The batch size m is the number of training samples used to compute the gradient at each iteration. The mini-batch SGD is said to follow the gradient of the true generalization error (Goodfellow, Bengio, and Courville 2016) by computing an unbiased estimate (implying data is sampled randomly). When $m = 1$, the algorithm is called **on-line** (i.e., stochastic) learning, whereas for $m = N$ it is called **batch** learning. In our work, we experimented with the following batch size values $m = \{1, 10, 100\}$.

As requested, we track the convergence of the training process by plotting the loss and test accuracy over all epochs for learning rates $\eta = 0.1$ and batch sizes $m = 10$ in Fig. 2. We can see that there is a **rapid fall in the loss** function, and a corresponding **jump in the test accuracy** in the first epoch, which is caused by the large learning rate (Goodfellow, Bengio, and Courville 2016). Furthermore, the training process is **stable** with a monotonically decreasing loss function, which demonstrates the theoretical correctness of the SGD implementation. Regarding the **generalization capability**

of the model, we can see that the test accuracy **does not suffer from over-fitting**, as it is constantly increasing over the epochs, and reaches a value of 98% at the last (i.e., tenth) epoch.

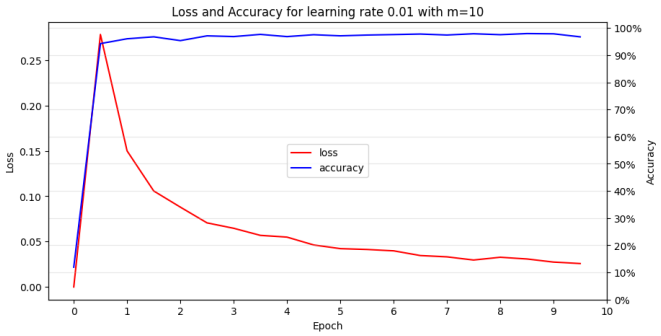


Figure 2: Loss and test accuracy for lr=0.1 and $m = 10$.

Additionally, while this hyperparameter configuration somewhat **exhibits a zig-zagging behaviour** from Epoch 1 onwards, which is common to stochastic GD approaches (Wilson and Martinez 2003), the oscillations are limited in magnitude, but we cannot yet confirm if the model has reached the global optimum. To address this, we rerun this experiment over 20 epochs (Fig 3) and find that the model does not further improve the performance, while the zig-zagging effect is now not only present but also more pronounced between the 10th and 20th epochs - implying that the step size is too large to find the global optimum.

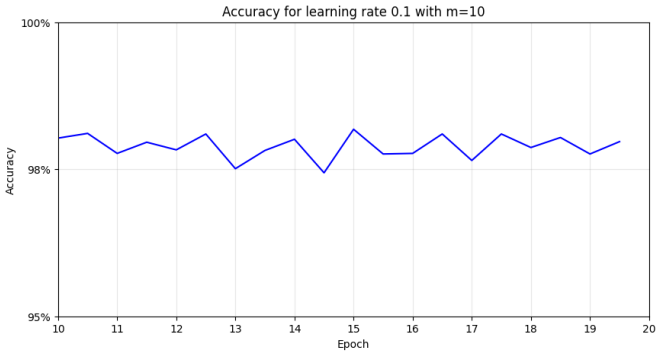


Figure 3: Test accuracy for lr=0.1 and $m = 10$ over 20 epochs.

From the experiments on both learning rates and batch sizes we reach the following conclusions:

1. **Instability for** $lr = 0.1, m = 1$: From Fig. 8a it becomes clear that a combination of a large learning rate and a small batch size leads to a very unstable optimization process, with a nan loss value and a constant test accuracy of 0.098 (i.e., random guessing) as visible in the table below. These results are expected due to the erratic stochastic updates, and the high variance in the gradient estimates (Goodfellow, Bengio, and Courville 2016). In contrast, just by decreasing the learning rate to $\eta = 0.01$ or by increasing the batch size to $m = 10$ we can achieve a stable and an accurate solution ($\geq 95\%$).
2. **Slow convergence for large** m: While a larger batch size leads to a more stable learning because of the averaging effect, it can suffer from a lack of generalization caused by the convergence to sharp minimizers and the inability to escape them post factum (Keskar et al. 2017). This is evident for $(lr = 0.001, m = 10)$ and $(lr = 0.001, m = 100)$ from Fig. 8b and 8c, respectively, where the test accuracy convergence slows down over all 10 epochs.
3. **Striking a good balance**: The optimal SGD solution depends on the trade-off between the learning rate and the batch size. However, the following configurations: $(lr = 0.1, m = 100)$, $(lr = 0.01, m = 10)$, and $(lr = 0.001, m = 1)$ seem to strike a good balance between the two hyperparameters, achieving low average loss and high test accuracy. It is also not unreasonable to claim that the learning rate and the batch size seem inversely correlated in terms of model performance.

2.2 Analytical Validation

To validate the correctness of the provided analytical gradient calculation, we approximate the derivative using three different finite-difference methods (Ford 2015):

1. Forward difference: $\frac{f(x+h)-f(x)}{h}$
2. Backward difference: $\frac{f(x)-f(x-h)}{h}$

Avg Loss	Test Acc	η	m	Time (s)
nan	0.098	0.1	1	3533s
0.025	0.977	0.1	10	3268s
0.017	0.976	0.1	100	3268s
0.027	0.977	0.01	1	3557s
0.019	0.976	0.01	10	3338s
0.17	0.950	0.01	100	3301s
0.019	**0.978**	0.001	1	3553s
0.16	0.952	0.001	10	3332s
0.53	0.872	0.001	100	3288s

Table 1: SGD performance: Loss and Accuracy

3. Central difference: $\frac{f(x+h)-f(x-h)}{2h}$

where we set the step size h to 10^{-8}.

Due to the significant computational cost of the all these methods (each takes ~1s per sample), we only compute the numerical gradients for a single sample at the end of the first training epoch and compare them to the analytical ones. We carry out our experiments using a learning rate of $\eta = 0.1$ and a mini-batch size of $m = 10$. The results shown in Figure 4 and Figure 5 indicate that the analytical and numerical gradients differ by less than $3 \times 10^{-4}\%$ (central difference) on average, which goes to show that the analytical gradient calculation is correct. As expected the central difference method provides a better approximation and a lower truncation error than the forward and backward difference ones due to the fact that it is second-order accurate, while the others are first-order accurate. This is clearly visible from the lower mean relative error and the narrower distribution of the central difference method. And while these results demonstrate our trust in the analytical solution, we note that the numerical algorithms are too computationally expensive to be used in practice over the range of all training data.

3 Improving Convergence

In order to counteract the zig-zagging effect of the large learning rate, but still maintain fast convergence, we will explore two different techniques: learning rate decay and momentum.

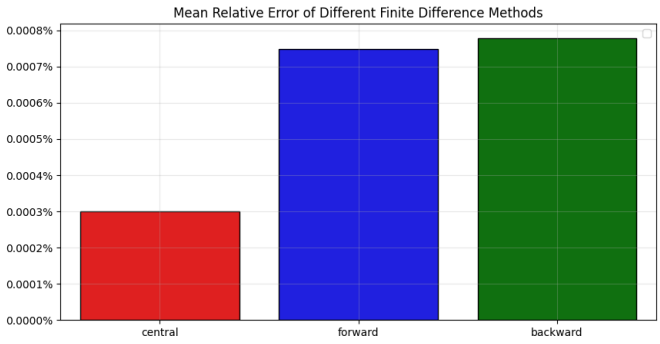


Figure 4: Mean relative error of the finite-difference methods.

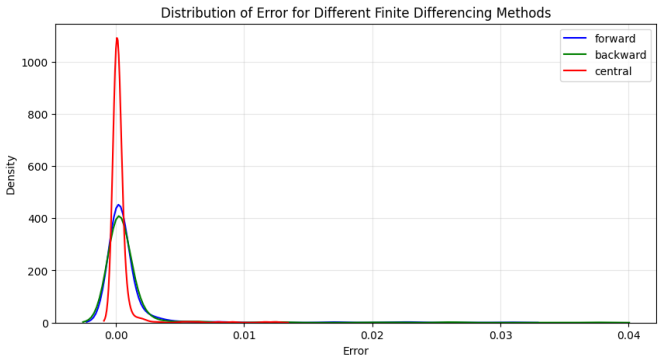


Figure 5: Relative error distribution of the finite-difference methods.

3.1 Learning Rate Decay

Learning rate decay is a common technique often useful in practice (Wilson and Martinez 2003) that makes more aggressive steps at the beginning and then gradually decrease the learning rate over time, allowing the model to make larger updates initially, followed by smaller and more fine-grained ones towards the end. More formally, we define the learning rate decay as follows:

$$\eta_k = \eta_0(1-\alpha) + \alpha\eta_N, \quad \text{where} \quad \alpha = \frac{k}{N} \tag{5}$$

where η_0 is the initial learning rate, η_N is the final learning rate, k is the current epoch, and N is the total number of epochs.

Some researchers even claim that using learning rate decay is equivalent to increasing the batch size (Smith et al. 2018) in terms of model performance, where the latter leads to significantly fewer parameter updates (i.e., improved

parallelism and shorter training times).

Nevertheless, throughout our experiments we explore various combinations of the initial and the final learning rates, and we vary the number of batches per epoch $m = \{1, 10, 100\}$. We argue that it is reasonable to use a more aggressive learning rate at the beginning of the training process to explore the parameter space, and then slowly calibrate it so as not to overshoot the optimal solution which was the case for SGD in Section 2.1.

The results shown in the table below indicate that:

1. the decay technique improves the final average loss and test accuracy for $m = 10$ because of the aggressive initial learning rate and the batch averaging effect;
2. reducing the final learning rate η_N leads to a better convergence for $m = 10$, eliminating the zig-zagging effect;
3. however, this method also exhibits some of SGD's shortcomings from Section 2.1 when $m = 1$ (i.e., on-line learning), which we believe is once again caused by the erratic high-variance gradient updates.

Avg Loss	Test Acc	η_0	η_N	m
2.303	0.098	0.1	0.001	1
0.002	0.983	0.1	0.001	10
0.031	0.977	0.1	0.001	100
0.001	**0.984**	0.1	0.0001	10
0.027	0.977	0.01	0.001	10

Table 2: SGD performance with learning rate decay: Loss and Accuracy

We further plot the loss and test accuracy for the best batch size (i.e., $m = 10$) in Figure 6 to demonstrate the effect of the learning rate decay on the zigzag effect common to SGD (Wilson and Martinez 2003).

Further reducing the final learning rate η_N to 0.00001 eliminates the zigzag effect completely towards the end of the training, as shown in Figure 7, which proves that this decay technique indeed helps the model converge faster and more

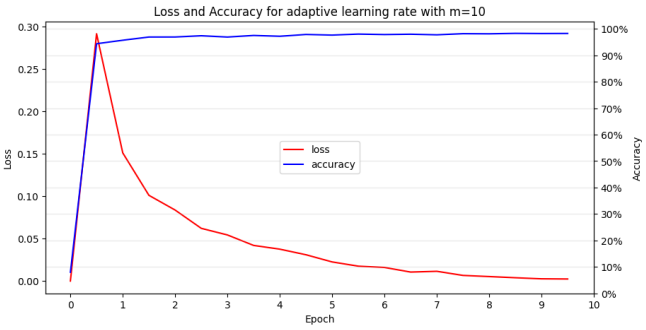


Figure 6: SGD with learning rate decay ($\eta_0 = 0.1$, $\eta_N = 0.001$, $m = 10$): Loss and Accuracy

smoothly to the optimal solution - test accuracy of 98.4%.

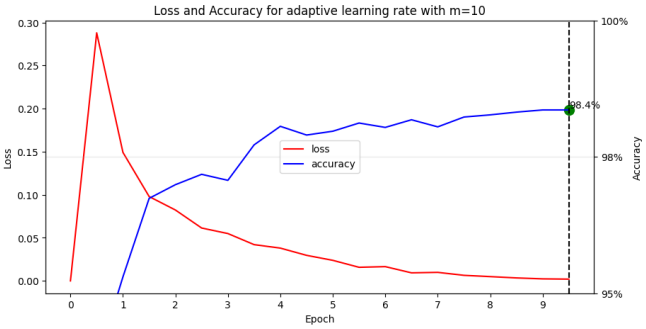


Figure 7: SGD with learning rate decay ($\eta_0 = 0.1$, $\eta_N = 0.0001$, $m = 10$): Loss and Accuracy

3.2 Momentum

4 Adaptive Learning

In this section we will explore the AdaGrad optimizer (Duchi, Hazan, and Singer 2011), defined in Algorithm 2. The main innovation of this technique is that it introduces individual adaptive learning rates (η) for each model weight, whereas before η was a global hyperparameter set uniformly for all parameters. This is possible through the accumulation of squared historical gradients per parameter w_i over time, which are then used for the scaling of w_i's learning rate; more formally it can be described as follows:

where $\odot$ denotes the element-wise product of the two vectors, while $\nabla J(w)$ is the gradient of the loss function $J(w)$ with respect to the model parameters w. The ϵ (usually set to 10^{-8}) term is further added to avoid division by zero.

Algorithm 2 AdaGrad

Require: Learning rate η
Require: Initial model parameters w
Require: Small constant ϵ (e.g., 10^{-8})
Ensure: Optimized model parameters w

Initialize G_0 as an empty diagonal matrix of the same size as w
while stopping criteria not met **do**
    Compute gradient $g_t = \nabla J(w)$ at current parameters w
    Accumulate squared gradient: $G_t = G_{t-1} + g_t^2$
    Compute update: $\Delta w = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
    Update parameters: $w = w + \Delta w$
end while
return Optimized model parameters w

We further note three key aspects of AdaGrad:

1. **Parameter-Specific Learning Rates**: AdaGrad adapts the learning rate so that infrequent parameters (i.e., rare features) receive larger updates, whereas the frequent ones (i.e., common features) receive smaller updates. This is especially useful in computer vision tasks (e.g., image classification) with frequent or repetitive image patterns (e.g., background, main body of digit).
2. **No Manual Learning Rate Tuning**: AdaGrad eliminates the need to manually tune the learning rate by allowing it to adaptively tune for each parameter during training. Nevertheless, we are still required to choose the initial learning rate η.
3. **Rapid Learning Rate Decay**: One issue with AdaGrad is that the learning rate monotonically decreases during training, which may cause premature convergence. To verify this, we can observe that in the AdaGrad update rule, the squared gradients G_t are accumulated over time, effectively shrinking the learning rate (η) on each dimension (Zeiler 2012). Eventually, η becomes infinitesimally small, and the learning stagnates. To address this, two algorithm extensions were proposed, namely: RMSProp (Tieleman and Hinton 2012) and Adam (Kingma and Ba 2017), that attempt to resolve this issue by introducing a decaying average of the historical squared gradients $E[g^2]_t$.

5 Conclusion

Duchi, John, Elad Hazan, and Yoram Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12 (61): 2121–59. http://jmlr.org/papers/v12/duchi11a.html.

Ford, William. 2015. "Chapter 12 - Linear System Applications." In *Numerical Linear Algebra with Applications*, edited by William Ford, 241–62. Boston: Academic Press. https://doi.org/https://doi.org/10.1016/B978-0-12-394435-1.00012-0.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Keskar, Nitish Shirish, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." https://arxiv.org/abs/1609.04836.

Kingma, Diederik P., and Jimmy Ba. 2017. "Adam: A Method for Stochastic Optimization." https://arxiv.org/abs/1412.6980.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324. https://doi.org/10.1109/5.726791.

Mishkin, Dmytro, Nikolay Sergievskiy, and Jiri Matas. 2017. "Systematic Evaluation of Convolution Neural Network Advances on the Imagenet." *Computer Vision and Image Understanding* 161 (August): 11–19. https://doi.org/10.1016/j.cviu.2017.05.007.

Smith, Samuel L., Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. "Don't Decay

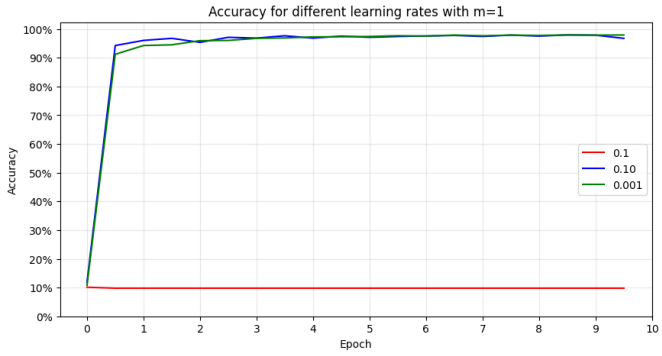
the Learning Rate, Increase the Batch Size." https://arxiv.org/abs/1711.00489.

Tieleman, Tijmen, and Geoffrey Hinton. 2012. "Lecture 6.5-Rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude." *COURSERA: Neural Networks for Machine Learning* 4: 26–31.

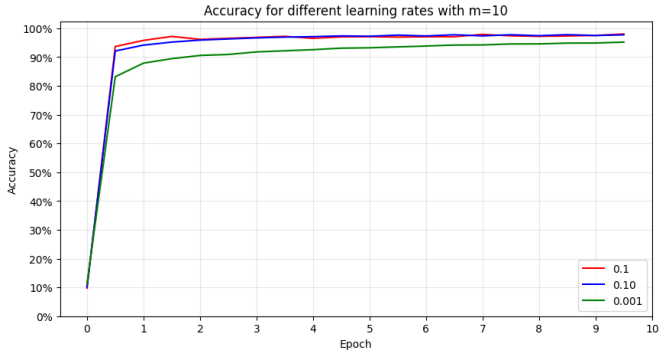
Wilson, D.Randall, and Tony R. Martinez. 2003. "The General Inefficiency of Batch Training for Gradient Descent Learning." *Neural Networks* 16 (10): 1429–51. https://doi.org/https://doi.org/10.1016/S0893-6080(03)00138-2.

Zeiler, Matthew D. 2012. "ADADELTA: An Adaptive Learning Rate Method." https://arxiv.org/abs/1212.5701.

(a) Batch size (m) = 1



(b) Batch size (m) = 10



(c) Batch size (m) = 100

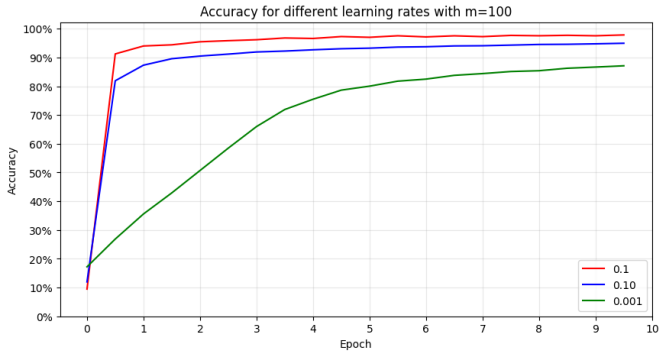


Figure 8: Test accuracy for different learning rates and batch sizes.