Assignment 2

Qiang Zhao (2814236)

November 2023

1 Answers

question 1

The computation graph is:

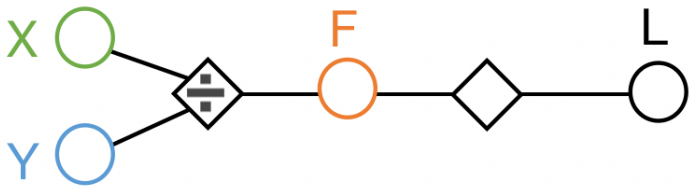

Figure 1: The computation graph of element-wise matrix division

The gradient of L with respect to X is:

$$
\begin{aligned}
X^{\nabla}_{ij} &= \frac{\partial L}{\partial X_{ij}} \\
&= \sum_{ab} \frac{\partial L}{\partial F_{ab}} \frac{\partial F_{ab}}{\partial X_{ij}} \\
&= \sum_{ab} F^{\nabla}_{ab} \frac{\partial F_{ab}}{\partial X_{ij}} \\
&= \sum_{ab} F^{\nabla}_{ab} \frac{\partial (\frac{X_{ij}}{Y_{ij}})}{\partial X_{ij}} \\
&= \sum_{ab} F^{\nabla}_{ab} \frac{1}{Y_{ij}} \\
&= \frac{1}{Y_{ij}} F^{\nabla}_{ij}
\end{aligned}
$$

The gradient of L with respect to Y is:

$$
\begin{aligned}
Y^{\nabla}_{ij} &= \frac{\partial L}{\partial Y_{ij}} \\
&= \sum_{ab} \frac{\partial L}{\partial F_{ab}} \frac{\partial F_{ab}}{\partial Y_{ij}} \\
&= \sum_{ab} F^{\nabla}_{ab} \frac{\partial F_{ab}}{\partial Y_{ij}} \\
&= \sum_{ab} F^{\nabla}_{ab} \frac{\partial (\frac{X_{ij}}{Y_{ij}})}{\partial Y_{ij}} \\
&= \sum_{ab} F^{\nabla}_{ab} \frac{-X_{ij}}{(Y_{ij})^2} \\
&= -\frac{-X_{ij}}{(Y_{ij})^2} F^{\nabla}_{ij}
\end{aligned}
$$

The relevant tensor operation in the code is :

```
class Div(Op):
    @staticmethod
    def forward(context,x,y):
    # x and y are matrices
    context['x'], context['y'] = x, y
    return x/y

    @staticmethod
    def backward(context,output):
    x, y = context['x'], context['y']
    deriv_x = (1 / y ) * output
    deriv_y = (-x / (y ** 2)) * output
    return deriv_x,deriv_y
```

question 2

The computation graph is:

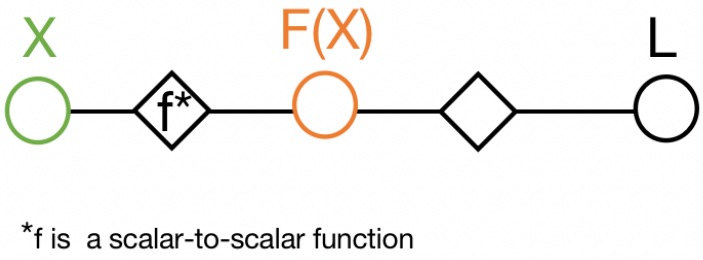

Figure 2: The computation graph of tensor to tensor function

$$
\begin{aligned}
X^{\nabla}_{ijk} &= \frac{\partial L}{\partial X_{ijk}} \\
&= \frac{\partial L}{\partial F_{abc}} \frac{\partial F_{abc}}{\partial X_{ijk}} \\
&= \sum_{abc} F^{\nabla}_{abc} \frac{\partial F_{abc}}{\partial X_{ijk}} \\
&= \sum_{ijk} F^{\nabla}_{ijk} \frac{\partial F_{ijk}}{\partial X_{ijk}} + \sum_{abc \neq ijk} F^{\nabla}_{abc} \frac{\partial F_{abc}}{\partial X_{ijk}} \\
&= F^{\nabla}_{ijk} \frac{\partial F_{ijk}}{\partial X_{ijk}} + 0 \\
&= F^{\nabla}_{ijk} \frac{\partial F_{ijk}}{\partial X_{ijk}}
\end{aligned}
$$

Since for each element of X, we apply f function to them, thus:

$$= F^{\nabla}_{ijk} \frac{\partial f_{ijk}}{\partial X_{ijk}}$$

Take the **sigmoid** function as an example of the f function. Thus, the derivation can be:

$$= F^{\nabla}_{ijk} \sigma(x_{ijk})(1 - \sigma(x_{ijk}))$$

The relevant tensor operation in the code is :

```
class Sigmoid(Op):
    @staticmethod
    def forward(context, x):
        #x is a tensor of any shape
        sigx =  1 / (1 + np.exp(-x))
        context['sigx'] = sigx # store the sigmoid of x for the backward pass
        return sigx
```

```
@staticmethod
def backward(context, goutput):
    sigx = context['sigx'] # retrieve the sigmoid of x
    return output * sigx * (1 - sigx)
```

question 3
The computation graph is:

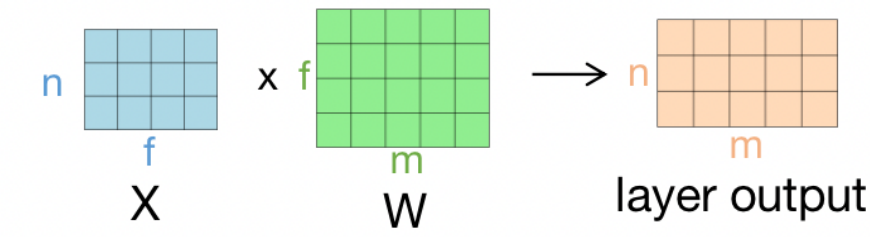

Figure 3: The computation graph of the operation of weight and input

Given the dimension of W is [f,m] and the dimension of X is [n,f], the matrix multiplication of the X and the W will be performed to compute the layer outputs. Assign the output to be K. The dimension of K was as follows:

$$K = XW$$

The gradient of L with respect to X is:

$$X^{\nabla}_{ij} = \frac{\partial L_{ab}}{\partial X_{ij}}$$

$$= \frac{\partial L_{ab}}{\partial K_{ab}} \frac{\partial K_{ab}}{\partial X_{ij}}$$

$$= \sum_{ab} K^{\nabla}_{ab} \frac{(\partial X_{ab} W_{ab})}{\partial X_{ij}} = K^{\nabla}_{ij} W^{T}_{i\,j}$$

The gradient of L with respect to W is:

$$W^{\nabla}_{ij} = \frac{\partial L_{ab}}{\partial W_{ij}}$$

$$= \frac{\partial L_{ab}}{\partial K_{ab}} \frac{\partial K_{ab}}{\partial W_{ij}}$$

$$= \sum_{ab} K^{\nabla}_{ab} \frac{(\partial X_{ab} W_{ab})}{\partial W_{ij}} = X^{T}_{i\,j} K^{\nabla}_{ij}$$

The relevant tensor operation in the code is :

```python
class Linear(Op):
    @staticmethod
    def forward(context, matrix, vectors):
        context['matrix'] = matrix
        context['vectors'] = vectors
        return np.matmul(vectors, matrix)
    @staticmethod
    def backward(context, go):
        matrix, vectors = context['matrix'], context['vectors']
        return np.matmul(go, matrix.T), np.matmul(vectors.T, go)
```

question 4

The computation graph is:

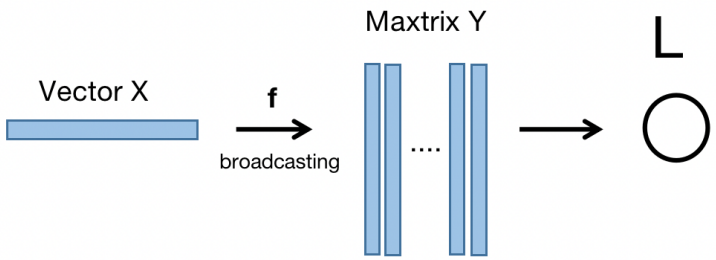

Figure 4: The computation graph of broadcasting

The gradient of L with respect to each element in the vector X is contributed by the sum of the gradient of each element in one row of matrix Y. Thus:

$$X_i^{\nabla} = \sum_{j=1}^{k} \frac{\partial L}{\partial Y_{ij}}$$

Since the k is the number of columns in matrix Y, thus:

$$= \sum_{j=1}^{16} \frac{\partial L}{\partial Y_{ij}}$$

$$= \begin{bmatrix} \sum_{j=1}^{16} \frac{\partial L}{\partial Y_{1j}} \\ \sum_{j=1}^{16} \frac{\partial L}{\partial Y_{2j}} \\ \vdots \\ \sum_{j=1}^{16} \frac{\partial L}{\partial Y_{nj}} \end{bmatrix}$$

The relevant tensor operation in the code is :

```
class Broadcasting(Op):
    # the input is a vector with shape [n,1]
    # In this case, the parameter dim is 1.
    @staticmethod
    def forward(context, input, *, dim, repeats):
        context['dim'] = dim
        return np.repeat(input, repeats, axis=dim)
    @staticmethod
    def backward(context, goutput):
        dim = context['dim']
        return goutput.sum(axis=dim, keepdims=True)
```

question 5
1)c.value contains the result of forward as follows:

```
array([[ 0.27222761,  2.72484507],
       [-0.26814385, -1.73085836]])
```

The result was obtained by the forward operation (+) using a magic operation function __add__ in Python.
2)The c.source is as follows:

```
<vugrad.core.OpNode at 0x7fb060723eb0>
```

It represents an instantiated object of the OpNode class, which is used to produce the TensorNode. The relevant code is as follows:

```
opcode = OpNode(cls, context, inputs)
outputs = [TensorNode(value=output, source=opnode)
for output in outputs_raw]
```

3)The c.source.inputs[0] is as follows:

```
<vugrad.core.OpNode at 0x7fb060723eb0>
```

It refers to the object a, which is one of the inputs of the forward operation.
4)a.grad is a zero-initialized array created during the initialization of TensorNode, which is subsequently used for storing gradients.The current value is as follows:

```
array([[0., 0.],
       [0., 0.]])
```

It is created by the following code:

```
self.grad = np.zeros(shape=value.shape)
```

question 6
1)The following code in the class **Op** indicates the practical usage of OpNode:

```
opcode = OpNode(cls, context, inputs)
```

The **cls** here refers to the class itself. Because classes such as **Add, Sub, Multiply, and MatrixMultiply** inherit from the Op class, in practical usage, the **cls** refers to these classes themself.
2)In the forward function of **Add** class, the practical of add operation is implemented, the code is as follows:

```
def forward(context, a, b):
    assert a.shape == b.shape, f'Arrays not the same sizes
    ({a.shape} {b.shape}).'
    return a + b
```

3)The reason why the output node(s) is left None at first is to avoid the risk of a circular reference, in which **None** acts as a placeholder.

In the following codes of the OpNode class, the OpNode was connected to the output nodes:

```
# extract the gradients over the outputs (these have been
# computed already)
goutputs_raw = [output.grad for output in self.outputs]
```

question 7
The following code from the **OpNode** class leads the actual **Op** operation to do the actual computation.

```
# compute the gradients over the inputs
ginputs_raw = self.op.backward(self.context, *goutputs_raw)
```

question 8
We chose the Ops **Squeeze** from the **ops.py** to prove the implementation is correct; the objective of this function is to remove a specific **singleton** dimension without changing the value in the input tensor.
Suppose we have a tensor **X**, the d-th dimension is 1, thus the following dimension is as follows:

$$(N_1, N_2, N_3, .., N_{d-1}, 1, N_{d+1}, ..., N_n)$$

The forward process is to remove the dimension which is equal to 1, thus the dimension of the tensor **Y** will be:

$$(N_1, N_2, N_3, ...N_{d-1}, N_{d+1}, ..., N_n)$$

The backward process is to recover the dimension.

$$X^{\nabla} = \frac{\partial L}{\partial X}$$

$$= \frac{\partial L}{\partial Y}(N_1, N_2, N_3, .., N_{d-1}, 1, N_{d+1}, ..., N_n)$$

The practical test of **Squeeze** function is as follows:

```
#Initialization
x = np.arange(10).reshape(1,2,5)
ctx = {}
a = vg.TensorNode(x)

#forward
b = vg.Squeeze.forward(ctx,x)

#Assign the l be the loss, which have the same shape with
#the output of forward.
l = np.full_like(b, fill_value=1)

#backward
c = vg.Squeeze.backward(ctx,l)

#Print the process of shape change
print("The shape of the input is", x.shape)
print("The shape of the input after the forward is",b.shape)
print("The shape of the output after the backward is",c.shape)
```

The printed result is as follows:

```
The shape of the input is (1, 2, 5)
The shape of the input after the forward is (2, 5)
The shape of the output after the backwards is (1, 2, 5)
```

The test results showed the corrected result as expected.

question 9

The adapted code, including three steps of the training script, is as follows:
1)Add a new **ReLu** class in the **ops.py**:

```
class Relu(Op):
    @staticmethod
    def forward(context,input):
        context['raw_input'] = input
        return np.maximum(0, input)

    @staticmethod
    def backward(context,gouput):
        return gouput * (context['raw_input'] > 0)
```

2)Add a new **relu** function in the **functions.py**:

```
def relu(x):
    return Relu.do_forward(x)
```

3)Replace the **sigmoid** with **relu** in the **train_mlp.py**:

```
    #hidden = vg.sigmoid(hidden)
    #sigmoid --> relu
    hidden = vg.relu(hidden)
```

The running result of Sigmoid versus ReLU is compared as follows:

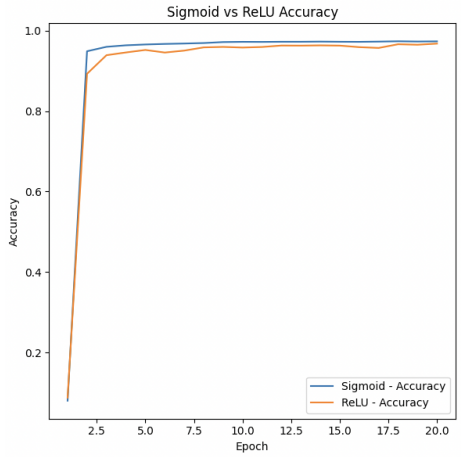

Figure 5: The accuracy comparison between Sigmoid and ReLu

Figure 5 indicates that the Sigmoid function provided an improved accuracy than the ReLU function on the synthetic data.

question 10
We adapted the network architecture with the following aspects:

a)The number of units:

The Test was implemented using the MNIST dataset with a fixed **learning rate (0.0001)** and the default architecture; the change with the number of units was by adjusting the **hidden_mult** parameter in the class **MLP**.

The **Figure 6** show the effect of different numbers of units on the accuracy (**Fig.6A**) and the loss(**Fig.6B**) of the model; the results indicate that as the number of units in the network increases, the model's loss decreases earlier, and the accuracy improves slightly.

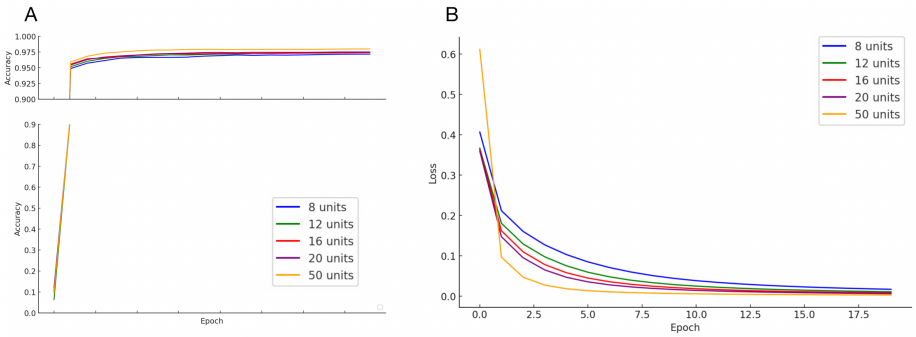

Figure 6: The accuracy and loss comparison with respect to different numbers of units

b)The number of layers:

The Test was implemented using the MNIST dataset with a fixed **learning rate(0.0001)** and the default number of **units(8)**.

Figure 7 shows the effect of different layers on the accuracy and the loss of the model; interestingly, in this model, increasing the number of layers actually led to a decrease in model performance, including a reduction in accuracy**(Fig.7A)** and an increase in loss**(Fig.7B)**. The possible cause of this phenomenon may be related with the vanishing gradient problem, which remains to explore.

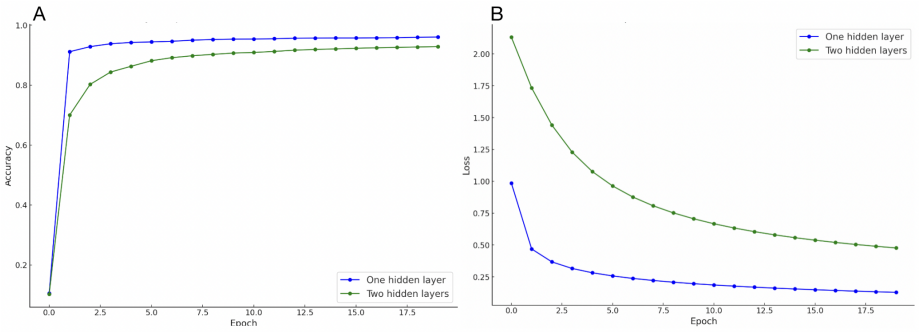

Figure 7: The accuracy and loss comparison with respect to different numbers of layers

c) The initialization of parameters:

The test was implemented using the MNIST dataset with a fixed **learning rate(0.0001)** and the default number of **units(8)**. We tested the **zero initial-**

ization and **standard normal distribution** separately.

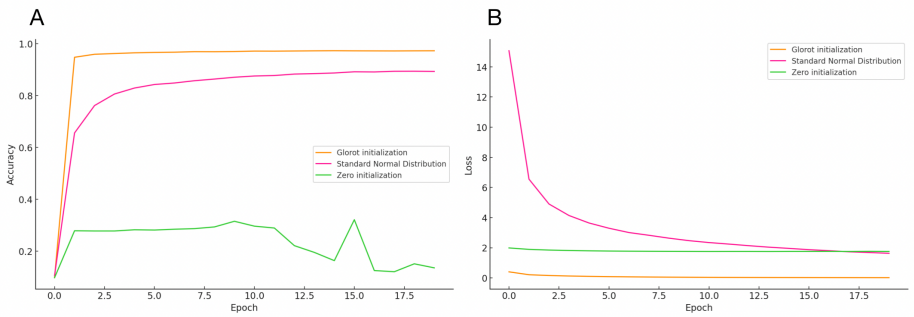

Figure 8: The accuracy and loss comparison with respect to different initialization methods

Figure 8 shows that the accuracy(**Fig.8A**) of models using Glorot initialization is better than those initialized with standard normal distribution and is significantly better than zero initialization. The loss (**Fig.8B**)of Glorot remains at a lower level compared with standard normal distribution and zero initialisation.

question 11

The architecture of the neural network we implemented using Pytorch was as follows:

Our dataset was derived from **CIFAR10**. The dataset has a total of 60,000 labelled colour images, 32*32 in size and divided into ten classes of 6,000 images each. Out of this, 50,000 are used for training with 5,000 images per class, and another 10,000 are used for testing with 1,000 images per class.The test of learning rate(**Fig.9A**),the number of epochs(**Fig.9B**),batch size(**Fig.9C**) was shown as follows:

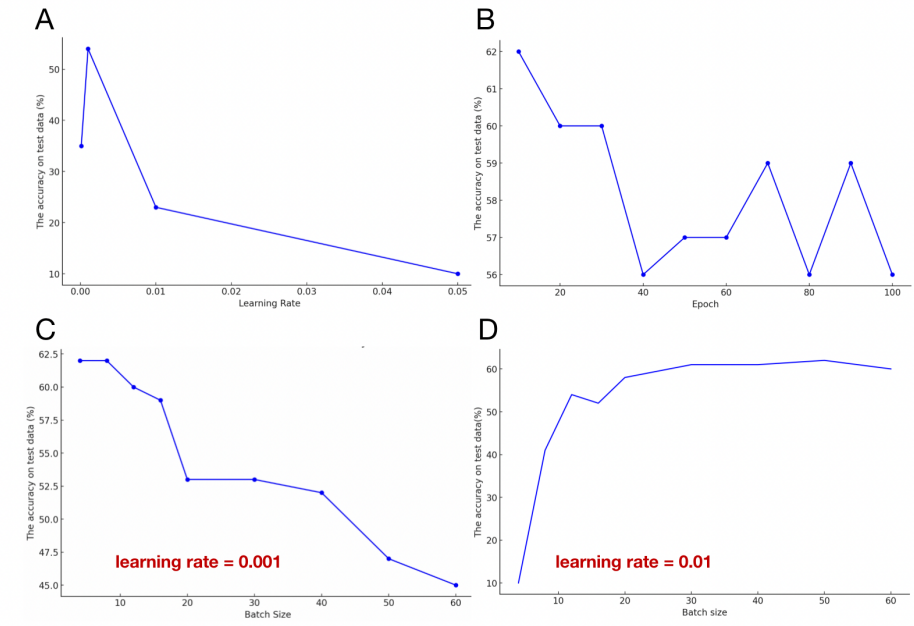

Figure 9: The effect of different hyperparameters on CIFAR10 dataset

a)For the test of the learning rate, the batch size was fixed at **4**, and the epoch was set as **2**.

b)For the test of the epoch, the learning rate was fixed at **0.001**, and the batch size was fixed at **4**.

c)For the test of the batch size, the learning rate was fixed at **0.001**, and the epoch was fixed at **6**.

Based on the above tests, we can see that a higher learning rate leads to lower accuracy, and this trend was also presented in the epoch and batch size. However, the decreased accuracy with a higher batch size may be caused by an unmatched learning rate, so we increase the the learning rate to 0.01 to test the effect of batch size under this new condition. **Figure 9D** shows the result, which indicates that batch size and accuracy correlate positively with a higher learning rate. Then, we can conclude the relationship between the batch size and accuracy was affected by the learning rate, which acts as a confounder.

Thus, we choose the final set of hyperparameters as follows:

Learning rate: **0.001**
The number of epochs: **10**

Batch size: **4**

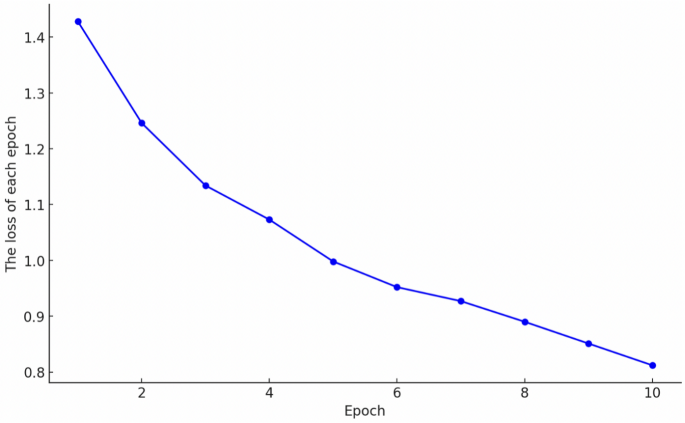

Figure 10: The loss of model using the final set of parameter

The final accuracy of the test data is **0.63,** the loss trend is shown in **Figure 10**. The prevalent codes are attached in the Appendix.

question 12

We adapted the network architecture with the following aspects:

a)The setting of different optimizers:

We tested the different optimizer on the test data; the loss(**Fig.11A**) and accuracy(**Fig.11B**) result was indicated as follows:

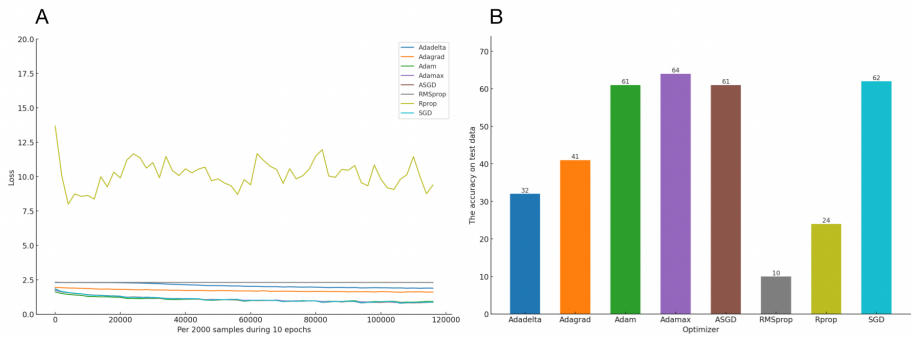

Figure 11: The loss and accuracy comparison with respect to different optimizers

The above results demonstrate that **Adamax** achieves the highest accuracy of 0.64, followed by **SGD**, **Adam**, and **ASGD**, all exceeding an accuracy of 0.6. Additionally, the loss curve reveals that **Rprop** optimization consistently maintains a high loss level, resulting in lower accuracy. In contrast, although **RMSprop** has a lower loss level, its accuracy performance is the poorest. In summary, **Adamax** is the most preferable optimizer.

The relevant parameter settings were listed as follows:

```
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
optimizer = optim.Adam(net.parameters(), lr=0.001,weight_decay=_0.001)
optimizer = optim.Rprop(net.parameters(), lr=0.001)
optimizer = optim.Adadelta(net.parameters(), lr=0.001, rho=0.9, eps=1e-06, weight_decay=0)
optimizer = optim.Adagrad(net.parameters(), lr=0.001, lr_decay=0, weight_decay=0)
optimizer = optim.Adamax(net.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)
optimizer = optim.ASGD(net.parameters(), lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight_decay=0)
optimizer = optim.RMSprop(net.parameters(), lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False)
```

Figure 12: The parameter settings for different optimizers

b)The setting of different loss function: We compared two types of loss functions on the test data; the loss and accuracy result was indicated as follows:

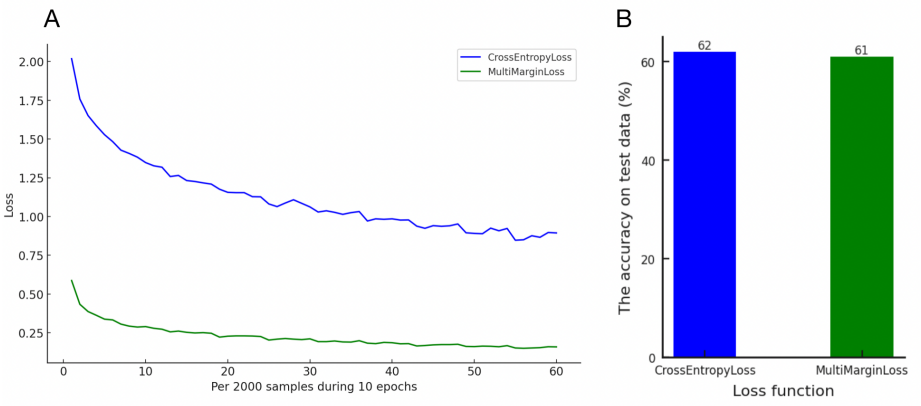

Figure 13: Enter Caption

The relevant parameter was listed as follows:

```
criterion = nn.CrossEntropyLoss(weight=None, reduction='mean')
criterion = nn.MultiMarginLoss(p=1, margin=1, weight=None, reduction='mean')
```

From the result, we can conclude the **CrossEntropyLoss** function is the more suitable one for multi-category classification problems than **MultiMarginLoss** function, as it combines the **Softmax** activation function with the tradi-

tional **cross-entropy** loss, providing an efficient way to deal with competing relationships between categories.

A Appendix

question 11
The test code for different hyperparameters was as follows:

```
import torch
import torchvision
import torchvision.transforms as transforms
import os

data_dir = '/Users/crystal_zhao/PycharmProjects/ASSIGN2/'
os.chdir(data_dir)

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=0)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')


#Define a test function used for testing different hyperparameters
#The batch size is fixed at 4.
def hyper_test(lr, epoch):
    import torch.nn as nn
    import torch.nn.functional as F

    class Net(nn.Module):
        def __init__(self):
            super().__init__()
```

```
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
# Define a loss function and optimize
import torch.optim as optim
criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0.9)  # learning rate

# Train the network
for epoch in range(epoch):  # loop over the dataset multiple times # epoch
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

# Test on the whole dataset
```

```
    correct = 0
    total = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            # calculate outputs by running images through the network
            outputs = net(images)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
    return 100 * correct // total


#1)Test with learning rate
#lr
lr_accuracy = []
lr = [0.0001,0.001,0.01,0.05]
for i in lr:
    lr_accuracy.append(hyper_test(i,2))

#2)Test with epoch
epoch_accuracy = []
epoch = list(range(10,101,10))
for i in epoch:
    epoch_accuracy.append(hyper_test(0.001,i))

#03)Test with batch size
batch_accuracy = []
batch = [4,8,12,16,20,30,40,50,60]
for i in batch:
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                            download=True, transform=transform)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=i,
                                              shuffle=True, num_workers=0)

    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                           download=True, transform=transform)
    testloader = torch.utils.data.DataLoader(testset, batch_size=i,
                                             shuffle=False, num_workers=0)

    batch_accuracy.append(hyper_test(0.001,6))

#04)Final set of hyperparameters
```

```
hyper_test(0.001,10)
```