深度視覺 HW4

M113040064 李冠宏

Data Preparation and Visualization

這邊的工作和 HW4 是一樣的，因此不再贅述

Layer Definition

其他地方都和 HW4 一樣，除了 FullyConnectedLayer 的 backward，新增了 dx 的計算，因為 out 的值是由 x*w+b 來的，因此對 x 微分會變成 w，因此 dx 的算法為 backprop 乘上 w 的轉置

```
def backward(self, backprop):
    N, F = backprop.shape
    dx = None
    dw = None
    db = None

    ####################
    # TODO:
    # 1. Derive the gradient of bias, i.e., d(out)/d(b)
    # 2. Multiply d(out)/d(b) with `backprop` due to chain rule, and
then store the AVERAGED result in variable `db`
    # 3. Then, derive the gradient of weights, i.e., d(out)/d(w)
    # 4. Multiply d(out)/d(w) with `backprop` due to chain rule, and
then store the AVERAGED result in variable `dw`
    # 5. Finally, derive the gradient of the input, i.e., d(out)/d(x)
    # 6. Multiply d(out)/d(x) with `backprop` due to chain rule, and
then store the result in variable `dx`
    # HINT: Shapes of `db`, `dw` and `dx` should be (F,), (D, F) and
(N, D) respectively
    ####################
    # -----START OF YOUR CODE-----
    x, w, b, out = self.cache
    db = torch.mean(backprop, dim=0)
    dw = torch.div(torch.matmul(torch.transpose(x, 0, 1), backprop),
x.shape[0])
```

```
    dx = torch.matmul(backprop, torch.transpose(w, 0, 1))
    # ------END OF YOUR CODE------

    return dx, dw, db
```

執行結果: (通過測試)

```
Results of fully connected layer forward and backward tests: All passed.
```

而 ReLU 和 Softmax_CrossEntropy 的部分也和 HW4 一樣。

這邊則是在做 Network 的測試，會印出每層權重和偏移量(bias)的 shape

```
from _utils import Network_Test

def test_classifier(x, y, dim_in, dim_out, dim_hidden):
  assert isinstance(dim_hidden, list), 'Parameter dim_hidden should b
e a list'
  np.random.seed(0)
  dim_hidden.append(dim_out)
  fc_i = 1
  params = {f'w{fc_i}': torch.from_numpy(np.random.randn(dim_in, dim_
hidden[0])*0.8),
        f'b{fc_i}': torch.zeros(dim_hidden[0])}
  net = [FullyConnectedLayer()]
  for layer_i in range(len(dim_hidden)-1):
    fc_i += 1
    params[f'w{fc_i}'] = torch.from_numpy(np.random.randn(dim_hidden[
layer_i], dim_hidden[layer_i + 1])*0.8)
    params[f'b{fc_i}'] = torch.zeros(dim_hidden[layer_i + 1])
    net.append(ReLU())
    net.append(FullyConnectedLayer())
  loss_func = Softmax_CrossEntropy()

  print('Network architecture and shapes of parameters:')
  fc_i = 1
  layer_out = net[0].forward(x, params[f'w{fc_i}'], params[f'b{fc_i}'
])
```

```
  print(f'  {net[0].__name__()}: w1({params[f"w{fc_i}"].shape}), b1({params[f"b{fc_i}"].shape})')
  for layer_i in range(1, len(net), 2):
    # ReLU
    layer_out = net[layer_i].forward(layer_out)
    print(f'  {net[layer_i].__name__()}')
    # FullyConnectedLayer
    fc_i += 1
    layer_out = net[layer_i + 1].forward(layer_out, params[f'w{fc_i}'], params[f'b{fc_i}'])
    print(f'  {net[layer_i + 1].__name__()}: w{fc_i}({params[f"w{fc_i}"].shape}), b{fc_i}({params[f"b{fc_i}"].shape})')
  _, loss = loss_func.forward(y, layer_out)

  params_grad = {}
  dout = loss_func.backward()
  dout_dx, dout_dw, dout_db = net[len(net) - 1].backward(dout)
  params_grad[f'w{fc_i}'] = dout_dw; params_grad[f'b{fc_i}'] = dout_db
  for layer_i in range(len(net)-2, 0, -2):
    # ReLU
    dout = net[layer_i].backward(dout_dx)
    # FullyConnectedLayer
    fc_i -= 1
    dout_dx, dout_dw, dout_db = net[layer_i - 1].backward(dout)
    params_grad[f'w{fc_i}'] = dout_dw; params_grad[f'b{fc_i}'] = dout_db
  Network_Test(params, params_grad)

dim_in = 5; dim_out = 3
dim_hidden = [8, 4, 2]

np.random.seed(0)
x = torch.from_numpy(np.random.randn(2, dim_in))
y = np.array([0, 1])
test_classifier(x, y, dim_in, dim_out, dim_hidden)
```

執行結果: (通過測試)

```
Network architecture and shapes of parameters:
  FullyConnectedLayer: w1(torch.Size([5, 8])), b1(torch.Size([8]))
  ReLU
  FullyConnectedLayer: w2(torch.Size([8, 4])), b2(torch.Size([4]))
  ReLU
  FullyConnectedLayer: w3(torch.Size([4, 2])), b3(torch.Size([2]))
  ReLU
  FullyConnectedLayer: w4(torch.Size([2, 3])), b4(torch.Size([3]))
Test of network: Passed.
```

Optimizer

這邊我們是要建立各種不同的優化器，用來更新權重和偏移量(bias)

首先是 SGD (Stochastic Gradient Descent)，是最基本的優化器，他的更新方式也最簡單，對於每個權重和偏移量，每次更新扣掉 learning rate * 梯度即可

```
class SGD(object):
  def __init__(self, learning_rate=1e-3):
    self.lr = learning_rate
  def step(self, params, params_grad):
    for key, val in params.items():
      if key not in params_grad.keys():
        raise KeyError(f'params_grad[\'{key}\'] is missing')
      if val.shape != params_grad[key].shape:
        raise ValueError(f'Expected shape of params_grad[\'{key}\'] {params[key].shape} but got {params_grad[key].shape}')

      ####################
      # TODO:
      # 1. Update each parameter `params[key]` with its gradient `params_grad[key]`
      ####################
      # -----START OF YOUR CODE-----
      params[key] -= self.lr * params_grad[key]
      # ------END OF YOUR CODE------
```

執行結果: (通過測試)

```
Test of SGD: Passed.
```

再來是 SGD + momentum，他透過加上一個所謂的動能來使得更新的方向能更精確，更新方式是先算出動能，動能是每次更新時乘上一個 momentum 的值再減去 learning rate * 梯度，最後再將權重加上動能

```python
class SGD_Momentum(object):
  def __init__(self, learning_rate=1e-3, momentum=0.9):
    self.lr = learning_rate
    self.momentum = momentum
    self.velocity = {}
  def step(self, params, params_grad):
    for key, val in params.items():
      if key not in params_grad.keys():
        raise KeyError(f'params_grad[\'{key}\'] is missing')
      if val.shape != params_grad[key].shape:
        raise ValueError(f'Expected shape of params_grad[\'{key}\'] {params[key].shape} but got {params_grad[key].shape}')
      if key not in self.velocity.keys():
        self.velocity[key] = torch.zeros_like(params[key])

      ####################
      # TODO:
      # 1. Update each velocity `self.velocity[key]` of each parameter
      # 2. Update each parameter `params[key]` with its velocity
      ####################
      # -----START OF YOUR CODE-----
      self.velocity[key] = self.momentum * self.velocity[key] - self.lr * params_grad[key]
      params[key] += self.velocity[key]
      # ------END OF YOUR CODE------
```

執行結果: (通過測試)

```
Test of SGD_Momentum: Passed.
```

最後是 Adam，三者中最複雜的方式，基本上它是 momentum + RMSprop，再加上"bias corrections"的方法。首先要先算出 momentum 和動能，接下來透過 bias corrections 的方式算出 m_hat 和 v_hat，最後用這兩個變數去更新權重。

```python
class Adam(object):
  def __init__(self, learning_rate=1e-3, beta1=0.9, beta2=0.999, epsilon=1e-8):
    self.lr = learning_rate
    self.beta1 = beta1
    self.beta2 = beta2
    self.epsilon = epsilon
    self.momentum = {}
    self.velocity = {}
    self.t = 0

  def step(self, params, params_grad):
    self.t += 1
    for key, val in params.items():
      if key not in params_grad.keys():
        raise KeyError(f'params_grad[\'{key}\'] is missing')
      if val.shape != params_grad[key].shape:
        raise ValueError(f'Expected shape of params_grad[\'{key}\'] {params[key].shape} but got {params_grad[key].shape}')
      if key not in self.momentum.keys():
        self.momentum[key] = torch.zeros_like(params[key])
      if key not in self.velocity.keys():
        self.velocity[key] = torch.zeros_like(params[key])
      ####################
      # TODO:
      # 1. Update each momentum `self.momentum[key]` and velocity `self.velocity[key]` of each parameter
      # 2. Calculate the bias-corrected momentum and velocity, this step should NOT change the value in `self.momentum` nor `self.velocity`
      # 3. Update each parameter `params[key]` with its bias-corrected momentum and velocity
      ####################
      # -----START OF YOUR CODE-----
      self.momentum[key] = self.beta1 * self.momentum[key] + ((1-self.beta1) * params_grad[key])
```

```
        self.velocity[key] = self.beta2 * self.velocity[key] + ((1-self
.beta2) * (params_grad[key] ** 2))
        m_hat = self.momentum[key] / (1 - (self.beta1 ** self.t))
        v_hat = self.velocity[key] / (1 - (self.beta2 ** self.t))
        params[key] -= self.lr * (m_hat / ((v_hat ** (1/2)) + self.epsi
lon))
        # ------END OF YOUR CODE------
```

執行結果: (通過測試)

```
Test of Adam: Passed.
```

Network Definition

這個部分是定義 model，因程式碼冗長因此就不放上來

Test Optimizers

這個部份是要測試優化器

```
from _utils import load_small_dataset
small_train_dataset, small_valid_dataset = load_small_dataset()

BATCH_SIZE = 16
DIM_HIDDENS = [128]
LR = 2e-4
REG = 0.0

optimizers = {'SGD': SGD(learning_rate=LR), 'SGD_Momentum': SGD_Momen
tum(learning_rate=LR), 'Adam': Adam(learning_rate=LR)}
train_loss = {}; train_acc = {}
valid_loss = {}; valid_acc = {}
for optim_name, optim in optimizers.items():
  model = Classifier(IN_DIM, OUT_DIM, DIM_HIDDENS, device=DEVICE)
  small_train_dataloader = Dataloader(small_train_dataset, batch_size
=BATCH_SIZE)
  small_valid_dataloader = Dataloader(small_valid_dataset, batch_size
=BATCH_SIZE)
```

```python
  tl, ta, vl, va = model.training(small_train_dataloader, small_valid_dataloader, epochs=100, optimizer=optim, reg_lambda=REG, verbose=101)
  idx = np.int32(np.array(vl).argmin())
  print(f'{optim_name}: Training accuracy: {format((ta[idx]), ".2f")}%, validation accuracy: {format((va[idx]), ".2f")}%\n\tTraining loss: {format((tl[idx]), ".4f")}, validation loss: {format((vl[idx]), ".4f")}')
  train_loss[optim_name] = tl; train_acc[optim_name] = ta
  valid_loss[optim_name] = vl; valid_acc[optim_name] = va
```

執行結果:

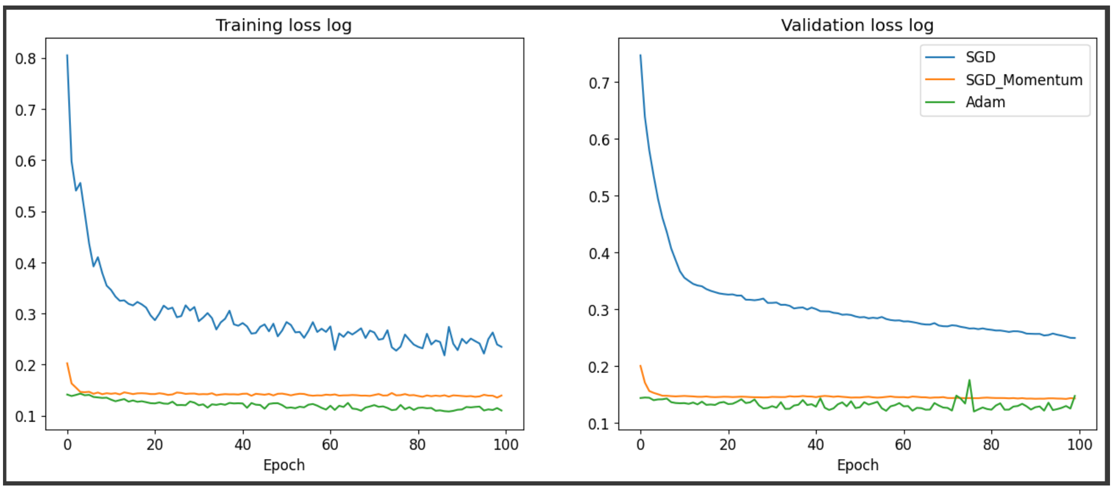
用以下的結果來看，效果是 Adam > SGD_Momentum > SGD

```
Files already downloaded and verified
SGD: Training accuracy: 14.55%, validation accuracy: 10.28%
        Training loss: 0.2347, validation loss: 0.2497
SGD_Momentum: Training accuracy: 20.50%, validation accuracy: 13.51%
        Training loss: 0.1391, validation loss: 0.1425
Adam: Training accuracy: 30.25%, validation accuracy: 28.53%
        Training loss: 0.1210, validation loss: 0.1204
```

接下來是 training loss 和 validation loss 的測試

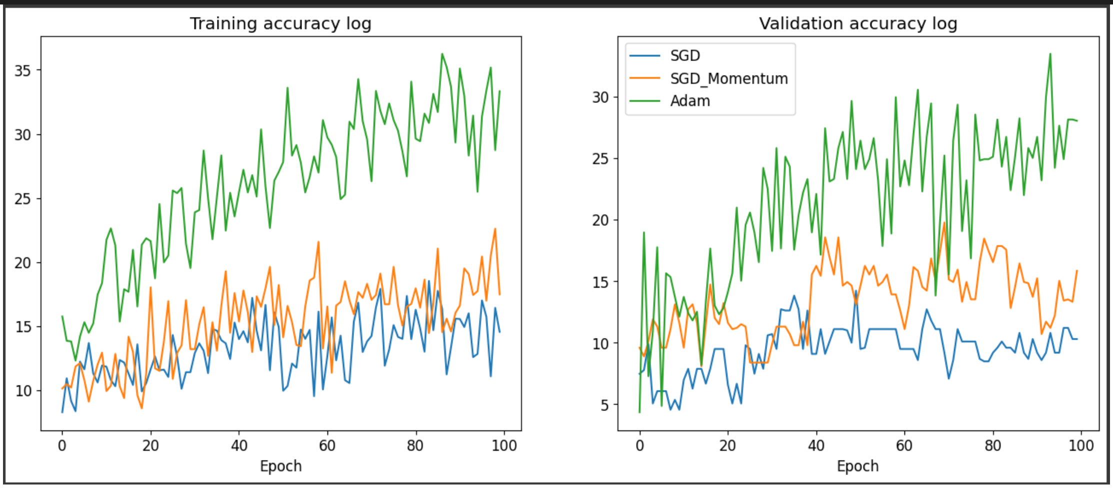
```python
cand = list(optimizers.keys())

train = list(train_loss.values()); valid = list(valid_loss.values())
plot_curves(cand, 'loss', train, valid)
```



training accuracy 和 validation accuracy 的測試

```
train = list(train_acc.values()); valid = list(valid_acc.values())
plot_curves(cand, 'accuracy', train, valid)
```



Training with Best Hyperparameters

Hyperparameters

```
# Please set your hyperparameters here
BATCH_SIZE = 8
LR = 0.01
DIM_HIDDENS = [256, 64]
REG = 0.0001
OPTIMIZER = SGD(learning_rate=LR)
train_dataloader = Dataloader(train_dataset, batch_size=BATCH_SIZE)
valid_dataloader = Dataloader(valid_dataset, batch_size=BATCH_SIZE)
model = Classifier(IN_DIM, OUT_DIM, DIM_HIDDENS, device=DEVICE)
model.print_param_shape()
```

執行結果:

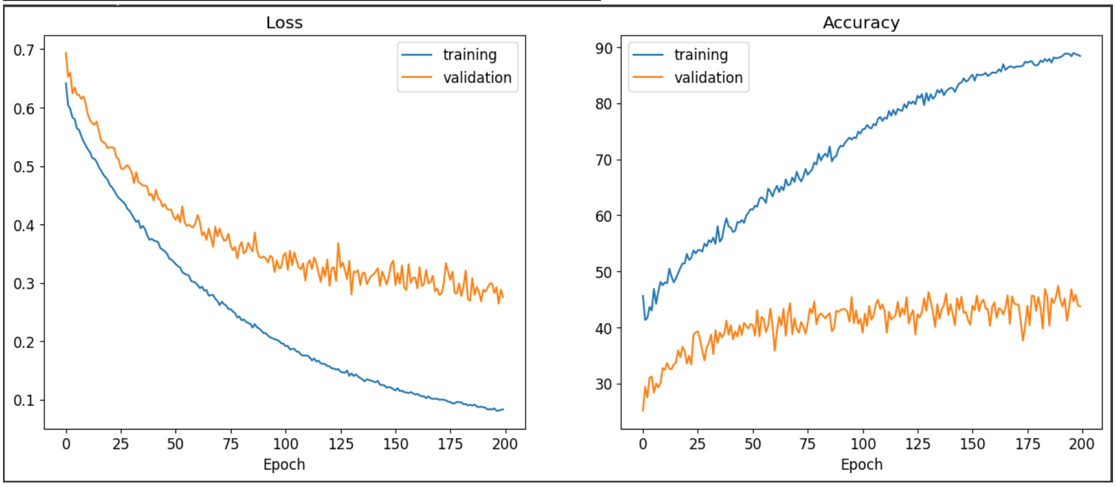
```
w1 torch.Size([3072, 256])
b1 torch.Size([256])
w2 torch.Size([256, 64])
b2 torch.Size([64])
w3 torch.Size([64, 10])
b3 torch.Size([10])
```

訓練

```
train_loss, train_acc, valid_loss, valid_acc = model.training(train_dataloader, valid_dataloader, optimizer=OPTIMIZER, reg_lambda=REG)
plot_result(train_loss, train_acc, valid_loss, valid_acc)
```

執行結果: (只取最後幾個 EPOCHS)

```
Epoch 198:
[MODEL SAVED]
Training accuracy: 88.71%, loss: 0.0812
Validaion accuracy: 45.85%, loss: 0.2643
Epoch 199:
Training accuracy: 88.65%, loss: 0.0824
Validaion accuracy: 43.97%, loss: 0.2885
Epoch 200:
Training accuracy: 88.43%, loss: 0.0833
Validaion accuracy: 43.77%, loss: 0.2755
```



最後將 test data 當作 input 的準確率為 46.52%

```
test_dataloader = Dataloader(test_dataset)
model.load_model(MODEL_PATH)
model.test(test_dataloader)
```

```
Got 4652 correct prediction in 10000 test data, accuracy: 46.52%
```