深度視覺 HW6

M113040064 李冠宏

由於這次篇幅過長，因此只取重點，程式碼只截取我們實作的部分。

Data preprocessing

這邊進行了資料的事前準備

Convolutional Networks

首先是 forward，首先設定 shape、stride 和 padding，接下來設定 output 的長和寬，最後透過位移的方式做累乘加以得到 output

```python
# set the variables
N, C, H, W = x.shape
F, C, HH, WW = w.shape
stride = conv_param["stride"]
padding = conv_param["pad"]

# Compute output tensor size
output_height = int((H + 2 * padding - HH) / stride + 1)
output_width = int((W + 2 * padding - WW) / stride + 1)

# padding
input_tensor_padded = torch.nn.functional.pad(x, (padding, padding, padding, padding))

# Initialize output tensor and add bias
out = torch.zeros((N, F, output_height, output_width), device = input_tensor_padded.device, dtype=torch.float64)

# Convolution operation
for i in range(N):
    for c_out in range(F):
        for h_out in range(output_height):
            for w_out in range(output_width):
                h_in = h_out * stride
                w_in = w_out * stride
                out[i, c_out, h_out, w_out] = torch.sum(w[c_out] * input_tensor_padded[i, :, h_in:h_in+HH, w_in:w_in+WW]) + b[c_out]
```

```
Testing Conv.forward
difference:  1.0141824738238694e-09
```

再來是 backward，最主要要做的事情是算出偏微分，kernel 的梯度是透過 dout * x_pad 取得的，而 input 的梯度是透過 dout * kernel 來取得，bias 的計算則相對簡單

```python
# 取得cache的內容
x, w, b, conv_param = cache

# 設定 padding 和 stride
padding = conv_param["pad"]
stride = conv_param["stride"]

# 設定 x_pad
x_pad = torch.nn.functional.pad(x, (padding, padding, padding, padding))

#設定維度
N, C, H, W = x_pad.shape
F, _, HH, WW = w.shape

# 初始化梯度
dx_pad = torch.zeros_like(x_pad)
dw = torch.zeros_like(w)

# 計算db
db = torch.sum(dout, axis=(0, 2, 3))

for n in range(N):
    for f in range(F):
      for i in range(0, H - HH + 1, stride):
        for j in range(0, W - WW + 1, stride):
          # dout[n, f, i, j]
          dx_pad[n, :, i : i + HH, j : j + WW] += dout[n, f, i//stride, j//stride] * w[f]
          dw[f] += dout[n, f, i//stride, j//stride] * x_pad[n, :, i : i + HH, j : j + WW]

dx = dx_pad[:, :, padding:-padding, padding:-padding]
```

```
Testing Conv.backward function
dx error:  2.496440948929281e-09
dw error:  9.222783256472505e-10
db error:  1.201214303148521e-09
```

Max-pooling

首先是 forward，我們的目的是要將選定的範圍取一個最大的，因此先用變數 pool_field 來選取此回合之範圍，再用 torch.max 來取最大值

```python
# set the variables
pool_height = pool_param["pool_height"]
pool_width = pool_param["pool_width"]
stride = pool_param["stride"]

# set the dimension
N, C, H, W = x.shape
out_height = int((H - pool_height) / stride) + 1
out_width = int((W - pool_width) / stride) + 1

# Initialize the output tensor
out = torch.zeros(N, C, out_height, out_width, dtype=torch.float64).to(x.device)

# Apply max pooling
for n in range(N):
    for c in range(C):
        for i in range(out_height):
            for j in range(out_width):
                pool_field = x[n, c, i*pool_height:i*pool_height+pool_height, j*pool_width:j*pool_width+pool_width]
                out[n, c, i, j] = torch.max(pool_field)
```

```
Testing MaxPool.forward function:
difference:  5.921052675939009e-09
```

再來是 backward 的部分，要計算 input 的梯度，我們只要將 dout 乘上範圍中最大的值即可

```python
# set the variables
x, pool_param = cache
N, C, H, W = x.shape
pool_height = pool_param["pool_height"]
pool_width = pool_param["pool_width"]
stride = pool_param["stride"]
_, _, out_height, out_width = dout.shape

# initialization
dx = torch.zeros_like(x)

# Compute gradient for each element in the output
for n in range(N):
    for c in range(C):
        for i in range(out_height):
            h_start = i * stride
            h_end = h_start + pool_height
            for j in range(out_width):
                w_start = j * stride
                w_end = w_start + pool_width

                # find the slice of the input volume
                x_slice = x[n, c, h_start:h_end, w_start:w_end]
                # find the maximum value in the slice
                mask = (x_slice == torch.max(x_slice))
                # assign the gradients to the maximum value in the slice
                dx[n, c, h_start:h_end, w_start:w_end] += mask * dout[n, c, i, j]
```

```
Testing MaxPool.backward function:
dx error:  6.653155794014975e-10
```

Three-layer convolutional network

這部分我們要實作一個三層的卷積網路，首先我們要先初始化參數

```
C, H, W = input_dims
self.params['W1'] = torch.normal(mean = 0.0, std = weight_scale, size = (num_filters, C, filter_size, filter_size), dtype=self.dtype, device=device)
self.params['b1'] = torch.zeros(num_filters, dtype=self.dtype, device=device)
self.params['W2'] = torch.normal(mean = 0.0, std = weight_scale, size = (num_filters * H * W // 4, hidden_dim), dtype=self.dtype, device=device)
self.params['b2'] = torch.zeros(hidden_dim, dtype=self.dtype, device=device)
self.params['W3'] = torch.normal(mean = 0.0, std = weight_scale, size = (hidden_dim, num_classes), dtype=self.dtype, device=device)
self.params['b3'] = torch.zeros(num_classes, dtype=self.dtype, device=device)
```

再來是 forward 的部分，我們可以呼叫已經定義好 class 中的 method 來直接實作 forward 的部分

```
x, conv_cache = Conv_ReLU_Pool.forward(X, W1, b1, conv_param, pool_param)
x, hidn_cache = Linear_ReLU.forward(x, W2, b2)
x, clas_cache = Linear.forward(x, W3, b3)

scores = x
```

Backward 也是呼叫寫好的 class methods，並且累加梯度和 loss，最後算出來並當回傳值

```
l_softmax, dout = softmax_loss(x, y)

dl_dx, grads['W3'], grads['b3'] = Linear.backward(dout, clas_cache)
grads['W3'] += self.reg * W3
loss += torch.sum(torch.pow(W3, 2))

dl_dx, grads['W2'], grads['b2'] = Linear_ReLU.backward(dl_dx, hidn_cache)
grads['W2'] += self.reg * W2
loss += torch.sum(torch.pow(W2, 2))

dl_dx, grads['W1'], grads['b1'] = Conv_ReLU_Pool.backward(dl_dx, conv_cache)
grads['W1'] += self.reg * W1
loss += torch.sum(torch.pow(W1, 2))

loss = l_softmax + 0.4 * self.reg * loss
```

Deep convolutional network

首先是初始化參數，這邊已經包含了 kaiming，因此分為兩個部分，這邊主要是將 kernel 和 bias，以及 gamma 和 beta 參數做初始化

```python
# 初始化參數
C, H, W = input_dims
filter_size = 3

if weight_scale == 'kaiming':
  # 第一層
  self.params['W1'] = kaiming_initializer(C, num_filters[0], K=filter_size, relu=True, dtype=dtype, device=device)
  self.params['b1'] = torch.zeros(num_filters[0], dtype=dtype, device=device)

  # 中間
  for i in range(2, self.num_layers):
      self.params[f'W{i}'] = kaiming_initializer(num_filters[i-2], num_filters[i-1], K=filter_size, relu=True, dtype=dtype, device=device)
      self.params[f'b{i}'] = torch.zeros(num_filters[i-1], dtype=self.dtype, device=device)

  # 最後一層
  self.params[f'W{self.num_layers}'] = kaiming_initializer(num_filters[-1] * (H // (2 ** len(max_pools))) * (W // (2 ** len(max_pools))), num_classes, relu=False, dtype=dtyp
  self.params[f'b{self.num_layers}'] = torch.zeros(num_classes, dtype=self.dtype, device=device)

  if self.batchnorm:
      for i in range(1, self.num_layers):
          self.params[f'gamma{i}'] = torch.ones(num_filters[i-1], dtype=self.dtype, device=device)
          self.params[f'beta{i}'] = torch.zeros(num_filters[i-1], dtype=self.dtype, device=device)

else:
  # 第一層
  self.params['W1'] = torch.randn(num_filters[0], C, filter_size, filter_size, dtype=dtype, device=device) * weight_scale
  self.params['b1'] = torch.zeros(num_filters[0], dtype=dtype, device=device)

  # 中間
  for i in range(2, self.num_layers):
      self.params[f'W{i}'] = torch.randn(num_filters[i-1], num_filters[i-2], filter_size, filter_size , dtype=dtype, device=device) * weight_scale
      self.params[f'b{i}'] = torch.zeros(num_filters[i-1], dtype=self.dtype, device=device)

  # 最後一層
  self.params[f'W{self.num_layers}'] = torch.randn(num_filters[-1] * (H // (2**len(max_pools))) * (W // (2**len(max_pools))), num_classes, dtype=self.dtype, device=device) *
  self.params[f'b{self.num_layers}'] = torch.zeros(num_classes, dtype=self.dtype, device=device)

  if self.batchnorm:
      for i in range(1, self.num_layers):
          self.params[f'gamma{i}'] = torch.ones(num_filters[i-1], dtype=self.dtype, device=device)
          self.params[f'beta{i}'] = torch.zeros(num_filters[i-1], dtype=self.dtype, device=device)
```

接下來是 forward pass，雖然較前面複雜，但我們也是透過呼叫已經定義好 class 的 method 來實踐 forward pass

```python
# 設定 output varable 和 cache
out = X
cache = {}

# forward pass
for i in range(1, self.num_layers):
    if self.batchnorm:
      # batch norm + max pooling
      if i-1 in self.max_pools:
        out, cache[f'ch{i}both'] = Conv_BatchNorm_ReLU_Pool.forward(out, self.params[f'W{i}'], self.params[f'b{i}'], self.params[f'gamma{i}'], self.params[f'beta{i}'], conv_p
      # batch norm
      else:
        out, cache[f'ch{i}batch'] = Conv_BatchNorm_ReLU.forward(out, self.params[f'W{i}'], self.params[f'b{i}'], self.params[f'gamma{i}'], self.params[f'beta{i}'], conv_param
    else:
      # max pooling
      if i-1 in self.max_pools:
        out, cache[f'ch{i}pool'] = Conv_ReLU_Pool.forward(out, self.params[f'W{i}'], self.params[f'b{i}'], conv_param, pool_param)
      # none
      else:
        out, cache[f'ch{i}none'] = Conv_ReLU.forward(out, self.params[f'W{i}'], self.params[f'b{i}'], conv_param)

# liear layer
scores, cache[f'ch{self.num_layers}linear'] = Linear.forward(out, self.params[f'W{self.num_layers}'], self.params[f'b{self.num_layers}'])
```

Backward 也是，只要記得把正確的參數傳給正確的 function 即可

```python
loss_soft, dout = softmax_loss(scores, y)

# backward pass

# linear layer
lx_dx, grads[f'W{self.num_layers}'], grads[f'b{self.num_layers}'] = Linear.backward(dout, cache[f'ch{self.num_layers}linear'])
grads[f'W{self.num_layers}'] += 2 * self.reg * self.params[f'W{self.num_layers}']

for i in reversed(range(1, self.num_layers)):
    if self.batchnorm:
        # batch norm + max pooling
        if i-1 in self.max_pools:
            lx_dx, grads[f'W{i}'], grads[f'b{i}'], grads[f'gamma{i}'], grads[f'beta{i}'] = Conv_BatchNorm_ReLU_Pool.backward(lx_dx, cache[f'ch{i}both'])
        # bacth norm
        else:
            lx_dx, grads[f'W{i}'], grads[f'b{i}'], grads[f'gamma{i}'], grads[f'beta{i}'] = Conv_BatchNorm_ReLU.backward(lx_dx, cache[f'ch{i}batch'])
    else:
        # max pooling
        if i-1 in self.max_pools:
            lx_dx, grads[f'W{i}'], grads[f'b{i}'] = Conv_ReLU_Pool.backward(lx_dx, cache[f'ch{i}pool'])
        # none
        else:
            lx_dx, grads[f'W{i}'], grads[f'b{i}'] = Conv_ReLU.backward(lx_dx, cache[f'ch{i}none'])

    # 累加 loss 和 梯度
    grads[f'W{i}'] += 2 * self.reg * self.params[f'W{i}']
    loss += torch.sum(self.params[f'W{i}'] ** 2)

# 計算 loss
loss = self.reg * loss + loss_soft
```

最後訓練出來的 model 之 training accuracy 有到 100%

```
(Epoch 25 / 30) train acc: 100.00%; val_acc: 20.12%
(Epoch 26 / 30) train acc: 100.00%; val_acc: 20.16%
(Time 5.42 sec; Iteration 131 / 150) loss: 0.033225
(Epoch 27 / 30) train acc: 100.00%; val_acc: 19.17%
(Epoch 28 / 30) train acc: 100.00%; val_acc: 19.99%
(Time 5.82 sec; Iteration 141 / 150) loss: 0.011002
(Epoch 29 / 30) train acc: 100.00%; val_acc: 18.41%
(Epoch 30 / 30) train acc: 100.00%; val_acc: 19.17%
```

Train a good model!

最後訓練的 training accuracy 為 80.4%，validation accuracy 為 71.94%

```
(Epoch 15 / 20) train acc: 81.70%; val_acc: 72.31%
(Epoch 16 / 20) train acc: 77.20%; val_acc: 71.83%
(Epoch 17 / 20) train acc: 80.10%; val_acc: 72.28%
(Epoch 18 / 20) train acc: 84.20%; val_acc: 73.11%
(Epoch 19 / 20) train acc: 80.40%; val_acc: 72.62%
(Epoch 20 / 20) train acc: 80.40%; val_acc: 71.94%
```

Test your model!

準確率有到 73%

```
Validation set accuracy: 73.1100%
Test set accuracy: 73.2300%
```

Batch Normalization

首先是 forward 的部分

Mode 為 train：

```
sample_mean = torch.mean(x, axis=0)
sample_var = torch.var(x, axis=0)
running_mean = momentum * running_mean + (1 - momentum) * sample_mean
running_var = momentum * running_var + (1 - momentum) * sample_var

x_hat = (x - sample_mean) / (torch.sqrt(sample_var + eps))
out = gamma * x_hat + beta

cache = x, x_hat, sample_mean, sample_var, eps, gamma
```

Mode 為 test：

```
out = gamma*(x - running_mean)/(torch.sqrt(running_var + eps)) + beta
```

接下來是 backward

```
N, D = dout.shape
x, x_hat, sample_mean, sample_var, eps, gamma = cache
dbeta = torch.sum(dout, axis=0)
dgamma = torch.sum(x_hat * dout, axis=0)

dvar = torch.sum(gamma * dout * (x - sample_mean) * (-0.5) * torch.pow(sample_var + eps, -1.5), axis=0)
dmean = torch.sum(gamma * dout * (-1.0) * torch.pow(sample_var + eps, -0.5), axis=0) + dvar * (-2.0 / N) * torch.sum(x - sample_mean, axis=0)
dx = dout * gamma * torch.pow(sample_var + eps, -0.5) + dvar * 2.0 * (x - sample_mean) / N + dmean / N
```

Backward_alternative

```
N, D = dout.shape
x, x_hat, sample_mean, sample_var, eps, gamma = cache
dbeta = torch.sum(dout, axis=0)
dgamma = torch.sum(x_hat * dout, axis=0)
dxhat = dout * gamma
dx = (1. / (N * torch.sqrt(sample_var + eps))) * (N*dxhat - torch.sum(dxhat, axis=0) - x_hat*torch.sum(dxhat*x_hat, axis=0))
```

Spatial Batch Normalization

Forward

```
N, C, H, W = x.shape
x = x.permute(0, 2, 3, 1).reshape(N * H * W, C)
# x = x.reshape(N * H * W, C)
out, cache = BatchNorm.forward(x, gamma, beta, bn_param)
out = out.reshape(N, H, W, C).permute(0, 3, 1, 2)
# out = out.reshape(N, C, H, W)
```

Backward

```
N, C, H, W = dout.shape
dout = dout.permute(0, 2, 3, 1).reshape(N * H * W, C)
# dout = dout.reshape(N * H * W, C)
dx, dgamma, dbeta = BatchNorm.backward_alt(dout, cache)
dx = dx.reshape(N, H, W, C).permute(0, 3, 1, 2)
# dx = dx.reshape(N, C, H, W)
```

Batchnorm for deep convolutional networks

最後是將 batch normalization 套用到深度卷積網路中

有 batch normalization 的

```
Solver with batch norm:
(Time 0.02 sec; Iteration 1 / 50) loss: 3.140225
(Epoch 0 / 10) train acc: 9.80%; val_acc: 11.75%
(Epoch 1 / 10) train acc: 15.80%; val_acc: 12.03%
(Epoch 2 / 10) train acc: 14.80%; val_acc: 11.45%
(Epoch 3 / 10) train acc: 19.00%; val_acc: 14.61%
(Epoch 4 / 10) train acc: 45.60%; val_acc: 26.68%
(Time 2.05 sec; Iteration 21 / 50) loss: 1.279115
(Epoch 5 / 10) train acc: 61.80%; val_acc: 31.48%
(Epoch 6 / 10) train acc: 72.00%; val_acc: 32.59%
(Epoch 7 / 10) train acc: 77.80%; val_acc: 32.43%
(Epoch 8 / 10) train acc: 83.80%; val_acc: 33.54%
(Time 3.73 sec; Iteration 41 / 50) loss: 0.787794
(Epoch 9 / 10) train acc: 88.20%; val_acc: 35.39%
(Epoch 10 / 10) train acc: 91.80%; val_acc: 36.17%
```

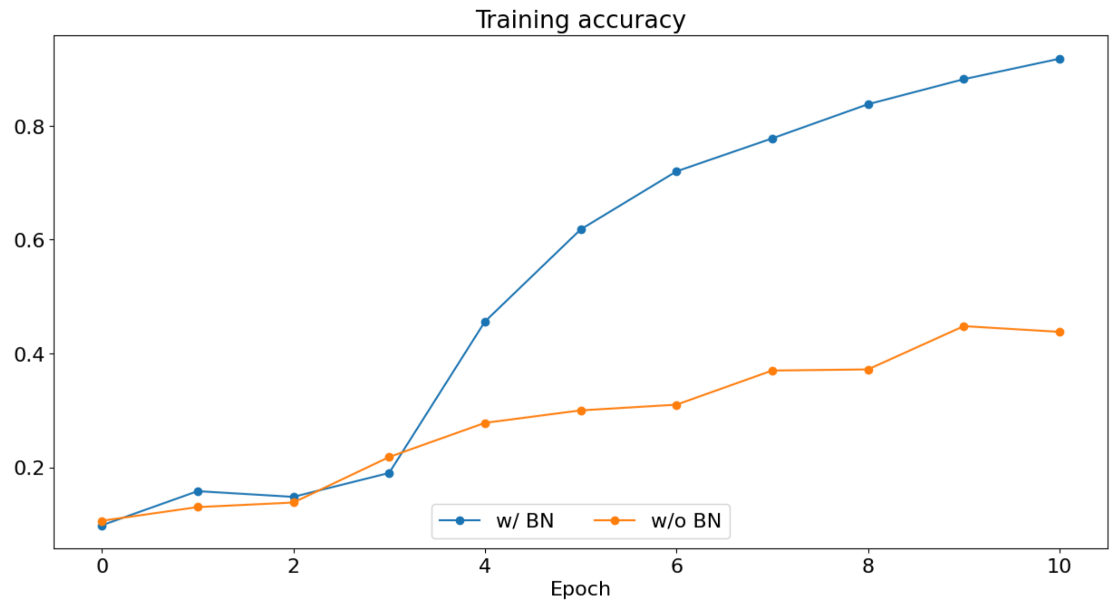
沒有 batch normalization 的

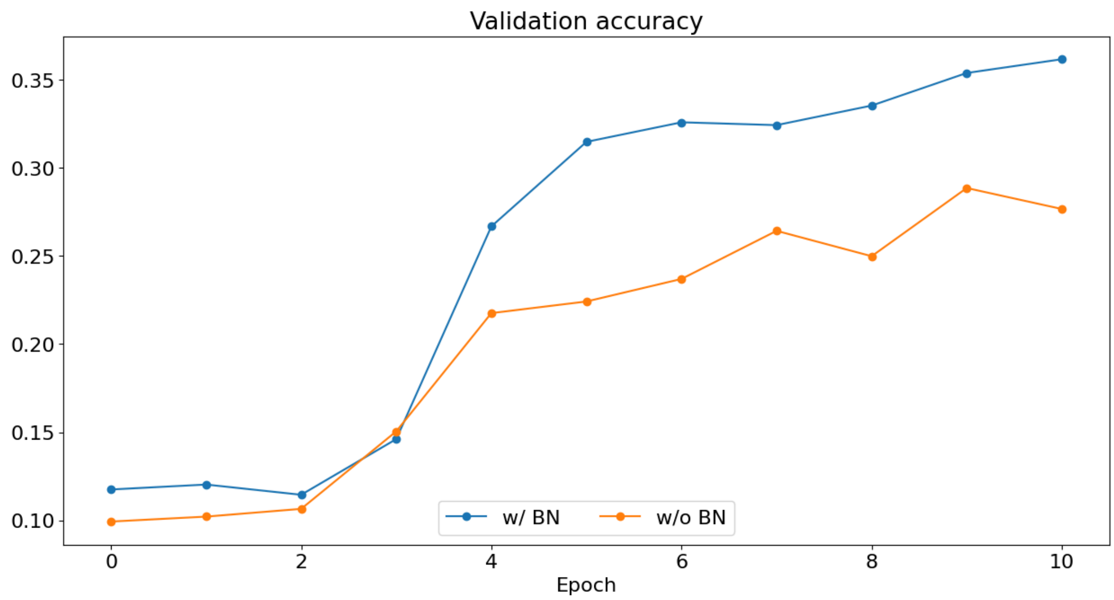
```
Solver without batch norm:
(Time 0.01 sec; Iteration 1 / 50) loss: 2.666604
(Epoch 0 / 10) train acc: 10.60%; val_acc: 9.93%
(Epoch 1 / 10) train acc: 13.00%; val_acc: 10.21%
(Epoch 2 / 10) train acc: 13.80%; val_acc: 10.65%
(Epoch 3 / 10) train acc: 21.80%; val_acc: 15.05%
(Epoch 4 / 10) train acc: 27.80%; val_acc: 21.76%
(Time 1.41 sec; Iteration 21 / 50) loss: 2.047068
(Epoch 5 / 10) train acc: 30.00%; val_acc: 22.42%
(Epoch 6 / 10) train acc: 31.00%; val_acc: 23.70%
(Epoch 7 / 10) train acc: 37.00%; val_acc: 26.43%
(Epoch 8 / 10) train acc: 37.20%; val_acc: 24.99%
(Time 2.56 sec; Iteration 41 / 50) loss: 1.654234
(Epoch 9 / 10) train acc: 44.80%; val_acc: 28.86%
(Epoch 10 / 10) train acc: 43.80%; val_acc: 27.67%
```

我們可以發現準確率差異很大，另外，loss 也影響很大


Training loss
3.0
2.5
2.0
1.5
1.0
w/ BN
w/o BN
0
10
20
30
40
50
Iteration

準確率提升了很多


Training accuracy
0.8
0.6
0.4
0.2
w/ BN
w/o BN
0
2
4
6
8
10
Epoch



最後則是不同 learning rate 的影響