HW6 成果分析

```
//TODO

Fix bugs of Batchnorm backprop
```

本次作業實作CNN神經網路的建構

架構

Convolutional Layer

先建立最主要的卷積層。根據輸入輸出的 dimensions 還有 kernal 的設定，我們可以用巢狀的迴圈進行卷積。

```
# Forward

    for nn in range(x_shape[0]):
      for ff in range(w_shape[0]):
        for hh in range(output_H):
          for ww in range(output_W):
            conv_window_h = hh * stride
            conv_window_w = ww * stride
            out_tmp = x_pad[nn, :, conv_window_h:conv_window_h+w_shape[2], 
conv_window_w:conv_window_w+w_shape[3]] * w[ff, :, :, :]
            out[nn, ff, hh, ww] = torch.sum(out_tmp) + b[ff]
```

我的作法是先框出一個 kernal window，後續的所有運算只要專注在這個 window 當中即可。

```
#Backward

    for nn in range(x_shape[0]):
      for ff in range(w_shape[0]):
        for hh in range(output_H):
          for ww in range(output_W):
            conv_window_h = hh * stride
            conv_window_w = ww * stride
            dx_pad[nn, :, conv_window_h:conv_window_h+w_shape[2], 
conv_window_w:conv_window_w+w_shape[3]] += dout[nn, ff, hh, ww] * w[ff, :, :, 
:]
            dw[ff, :, :, :] += dout[nn, ff, hh, ww] * x_pad[nn, :, 
conv_window_h:conv_window_h+w_shape[2], 
conv_window_w:conv_window_w+w_shape[3]]
            db[ff] += dout[nn, ff, hh, ww]

    dx = dx_pad[:, :, pad:pad+x_shape[2], pad:pad+x_shape[3]]
```

在反向傳播中，因為 \(out=x*w\) 所以 \(dx=dout*dw\)。又因為有 rolling windows 的關係，所以矩陣內的每個元素都有可能會被重複運算到，故使用 `+=` 運算子，將所有的變量疊加起來。

Maxpooling

和上面同理，先專注在一個 pooling window 的運算上，再用巢狀迴圈 rolling 。Maxpooling 是在一個 pooling window 中，輸出一個最大值。

```
# Forward
    for nn in range(x_shape[0]):
      for cc in range(x_shape[1]):
        for hh in range(output_H):
          for ww in range(output_W):
            pool_window_h = hh * stride
            pool_window_w = ww * stride
            out_tmp = x[nn, cc, pool_window_h:pool_window_h+pool_height,
pool_window_w:pool_window_w+pool_width]
            out[nn, cc, hh, ww] = torch.max(out_tmp)


# Backward

    for nn in range(x_shape[0]):
      for cc in range(x_shape[1]):
        for hh in range(output_H):
          for ww in range(output_W):
            pool_window_h = hh * stride
            pool_window_w = ww * stride
            out_tmp = x[nn, cc, pool_window_h:pool_window_h+pool_height,
pool_window_w:pool_window_w+pool_width]
            out_tmp_max = torch.max(out_tmp)
            max_mat = (out_tmp == out_tmp_max)
            dx[nn, cc, pool_window_h:pool_window_h+pool_height,
pool_window_w:pool_window_w+pool_width] += (dout[nn, cc, hh, ww]*max_mat)
```

Maxpooling window 中，梯度只會從最大值的元素流過，因為輸出只和輸入的最大值有關係。
\[out = max(\mathbf{x}) \\ dout = \begin{cases} dx_i, & \text{if}\ max(\mathbf{x}) = x_i \\ 0, & \text{otherwise} \end{cases} \]
在反向傳播中，又運算了一次 max ，其實可以在正向傳播完成時直接把 out 存到 cache，增加效率。(不過比大小應該耗不掉多少效率)

Batch Nornalization

以 mini-batch 為單位，依照各個 mini-batch 來進行正規化。如此便可以減少模型過度依賴預設值。

- 調整最佳化過程可能只對最後幾層有影響，因為神經網路的複雜度讓梯度傳至前面可能已經使特徵發生變化。
- 確保每一層的特徵相同，所以在每一層結束後都進行正規化。

根據論文：

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

反向傳播：

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

加入我們的 DNN 當中。

```
Before batch normalization:
  means:  ['52.046', '11.122', '10.243']
  stds:   ['34.646', '30.732', '39.429']

After batch normalization (gamma=1, beta=0)
  means:  ['-0.000', '-0.000', '-0.000']
  stds:   ['1.000', '1.000', '1.000']

After batch normalization (gamma= [1.0, 2.0, 3.0] , beta= [11.0, 12.0, 13.0] )
  means:  ['11.000', '12.000', '13.000']
  stds:   ['1.000', '2.000', '3.000']
```

可以發現，在加入 Batchnorm 之後，每層的標準差和平均數都可以被很好的控制。

```
Running check with reg =  0
W1 max relative error: 7.554828e-03
W2 max relative error: 9.331857e-03
W3 max relative error: 1.078155e-02
W4 max relative error: 1.501991e-09
b1 max relative error: 9.999999e-01
b2 max relative error: 1.000000e+00
b3 max relative error: 1.110223e-06
b4 max relative error: 2.382429e-09
beta1 max relative error: 7.290488e-03
beta2 max relative error: 2.329975e-02
beta3 max relative error: 2.572281e-09
gamma1 max relative error: 6.621865e-03
gamma2 max relative error: 2.271405e-02
gamma3 max relative error: 1.686124e-09

Running check with reg =  3.14
W1 max relative error: 1.145399e-03
W2 max relative error: 1.503421e-03
W3 max relative error: 1.830862e-03
W4 max relative error: 2.206159e-08
b1 max relative error: 1.942890e-06
b2 max relative error: 1.804112e-06
b3 max relative error: 1.110223e-06
b4 max relative error: 9.149199e-08
beta1 max relative error: 6.879257e-03
beta2 max relative error: 7.083755e-03
beta3 max relative error: 6.922886e-08
gamma1 max relative error: 6.652254e-03
gamma2 max relative error: 8.533660e-03
gamma3 max relative error: 4.155353e-08
```

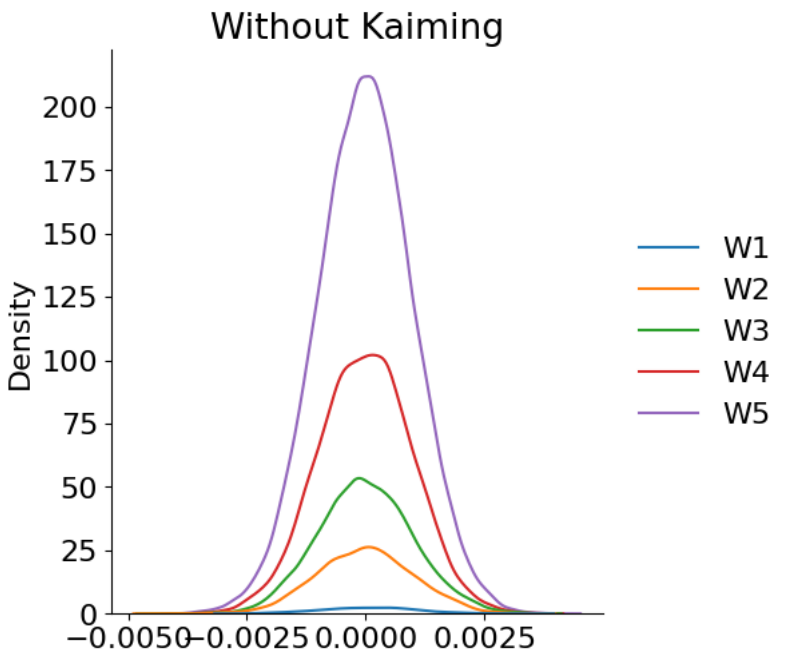
只是不知道為何，算出來的結果一直有誤差，檢查了幾次都發現不出問題，上網查閱相關的範例
也是一樣。日後多查一些資料後再進行改善。

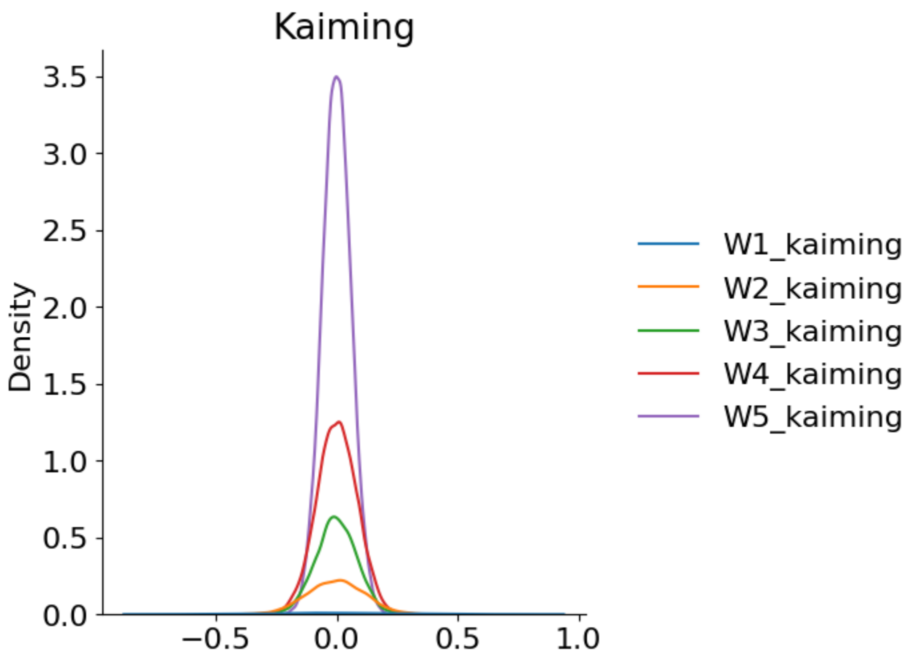
backprop of BN:

```
dx error:  0.07613132307158305
dgamma error:  3.087149574058185e-10
dbeta error:  2.919336030277982e-10
```

Kaiming Initialization

- ReLU 活化函數會拋棄小於0的值
- 若每層的權重是常態分佈 (\(\mu=0\)) 的話，等於說砍掉了一半的值。
- 我的理解是，Kaiming 針對這個分佈曲線做了平移和縮放，讓權重有不同的變化





3 Layers CNN

實作三層的 cnn 神經網路，採用剛才的 API ，並且新增 ReLU 的運算就好了。

```
# ReLU

out = torch.max(torch.zeros_like(out), out) # Forward
dout = (dout>=torch.zeros_like(dout)).long() # Backward
```

```
# Forward

    out, cache1 = Conv.forward(X, W1, b1, conv_param)
    cache2 = out
    out = torch.max(torch.zeros_like(out), out)
    out, cache3 = MaxPool.forward(out, pool_param)
    out, cache4 = Linear_ReLU.forward(out, W2, b2)
    scores, cache5 = Linear.forward(out, W3, b3)


# Backward

    loss, dout = softmax_loss(scores, y)
    loss += self.reg * ((torch.sum(W1 * W1) + torch.sum(W2 * W2) +
torch.sum(W3 * W3)))

    dh, grads['W3'], grads['b3'] = Linear.backward(dout, cache5)
    grads['W3'] += 2 * self.reg * W3
    dh, grads['W2'], grads['b2'] = Linear_ReLU.backward(dh, cache4)
    grads['W2'] += 2 * self.reg * W2
    dh = MaxPool.backward(dh, cache3)
    dh[cache2 < 0] = 0
    _, grads['W1'], grads['b1'] = Conv.backward(dh, cache1)
    grads['W1'] += 2 * self.reg * W1
```

事實上，也可以改用 `torch.nn.conv2d` 平行處理，大幅增加運算速度。直接引用下方的 FastConv API 就可以了。

Deeper CNN

用 for 處理深層神經網路。

```
# Forward

for i in range(self.num_layers-1):

      # Conv forward
      h, cache = FastConv.forward(h, self.params['W'+str(i+1)],
self.params['b'+str(i+1)], conv_param)
      caches.append(cache)

      # BN forward
      if self.batchnorm:
        h, cache = SpatialBatchNorm.forward(h, self.params['gamma'+str(i+1)],
self.params['beta'+str(i+1)], self.bn_params[i])
        caches.append(cache)

      # ReLU forward
      h, cache = ReLU.forward(h)
      caches.append(cache)

      # Pool forward
      if i in self.max_pools:
        h, cache = FastMaxPool.forward(h, pool_param)
        caches.append(cache)

    scores, cache_final = Linear.forward(h,
self.params['W'+str(self.num_layers)], self.params['b'+str(self.num_layers)])
```

```
# Backward

    loss, dout = softmax_loss(scores, y)
    loss += self.reg * (torch.sum(self.params['W'+str(self.num_layers)]**2))

    dx, dw, db = Linear.backward(dout, cache_final)
    grads['W'+str(self.num_layers)] = dw + (2 * self.reg *
self.params['W'+str(self.num_layers)])
    grads['b'+str(self.num_layers)] = db

    for i in range(self.num_layers-1, 0, -1):
      #print(str(i)+"---")

      # Pool backprop
      if i-1 in self.max_pools:
        dx = FastMaxPool.backward(dx, caches[-1])
        #print(len(caches[-1]))
        caches.pop()

      # ReLU backprop
      dx = ReLU.backward(dx, caches[-1])
      caches.pop()

      # BN backprop
      if self.batchnorm:
        dx, dgamma, dbeta = SpatialBatchNorm.backward(dx, caches[-1])
        caches.pop()
        grads['gamma'+str(i)] = dgamma
        grads['beta'+str(i)] = dbeta

      # Conv backprop
      dx, dw, db = FastConv.backward(dx, caches[-1])

      #print(len(caches[-1]))
      caches.pop()
      grads['W'+str(i)] = dw + (2 * self.reg * self.params['W'+str(i)])
      grads['b'+str(i)] = db
      loss += self.reg * (torch.sum(self.params['W'+str(i)]**2))
```

這部份做了很多次的調整，才形成現在這個版本。
最一開始在做反向傳播時是直接取用 `cache[i]` ，但有些層不需經過池化，所以造成在正向傳播時沒有池化的層我也必須 append 一個佔位用的假 cache 給他，非常不漂亮，到後來增加了 Batchnorm 之後更是如此。
所以，最後才改成以上的版本，反向傳播中用完一個 cache 就 pop 掉，每次讀取 `cache[-1]` 即可。

模型訓練

> Train the best convolutional model that you can on CIFAR-10, storing your best model in the best_model variable. We require you to get at least 71% accuracy on the validation set using a convolutional net, within 60 seconds of training.

在一分鐘訓練出至少71%準確度的模型。以下是我的嘗試：

第一個版本：

```
#from fc_networks import adam, sgd_momentum

weight_scale = 'kaiming'
learning_rate = 0.0035
reg = 0.001

model = DeepConvNet(input_dims=(3, 32, 32), num_classes=10,
                    num_filters=[32, 80],
                    max_pools=[0, 1],
                    reg=reg,
                    weight_scale=weight_scale,
                    dtype=dtype,
                    device=device)
solver = Solver(model, data_dict,
                num_epochs=30,
                batch_size=128,
                update_rule=adam,
                optim_config={
                  'learning_rate': learning_rate,
                },
                print_every=10000,
                device=device)
```

```
val_acc = solver.check_accuracy(data_dict['X_val'], data_dict['y_val'])
test_acc = solver.check_accuracy(data_dict['X_test'], data_dict['y_test'])

print(f'Validation set accuracy: {"{:.4f}".format(val_acc*100)}%')
print(f'Test set accuracy: {"{:.4f}".format(test_acc*100)}%')
```

```
Validation set accuracy: 71.1700%
Test set accuracy: 70.3900%
```

第二個版本：

增加神經數量， capacity 變高，所以效果變好，但如果仔細看訓練過程，其實是 overfitting 的。

```python
weight_scale = 'kaiming'
learning_rate = 0.0025
reg = 0.001

model = DeepConvNet(input_dims=(3, 32, 32), num_classes=10,
                    num_filters=[32, 128],
                    max_pools=[0, 1],
                    reg=reg,
                    weight_scale=weight_scale,
                    # batchnorm=True,
                    dtype=dtype,
                    device=device)
solver = Solver(model, data_dict,
                num_epochs=30,
                batch_size=128,
                update_rule=adam,
                optim_config={
                  'learning_rate': learning_rate,
                },
                print_every=10000,
                device=device)
```

```python
val_acc = solver.check_accuracy(data_dict['X_val'], data_dict['y_val'])
test_acc = solver.check_accuracy(data_dict['X_test'], data_dict['y_test'])

print(f'Validation set accuracy: {"{:.4f}".format(val_acc*100)}%')
print(f'Test set accuracy: {"{:.4f}".format(test_acc*100)}%')
```

```
Validation set accuracy: 71.5700%
Test set accuracy: 71.5200%
```

```
(Time 0.01 sec; Iteration 1 / 9360) loss: 2.712113
(Epoch 0 / 30) train acc: 10.90%; val_acc: 12.28%
(Epoch 1 / 30) train acc: 60.80%; val_acc: 59.32%
(Epoch 2 / 30) train acc: 66.70%; val_acc: 63.31%
(Epoch 3 / 30) train acc: 70.20%; val_acc: 66.83%
(Epoch 4 / 30) train acc: 73.20%; val_acc: 68.54%
(Epoch 5 / 30) train acc: 72.00%; val_acc: 68.26%
(Epoch 6 / 30) train acc: 71.80%; val_acc: 69.91%
(Epoch 7 / 30) train acc: 74.80%; val_acc: 69.54%
(Epoch 8 / 30) train acc: 74.40%; val_acc: 69.44%
(Epoch 9 / 30) train acc: 76.90%; val_acc: 70.68%
(Epoch 10 / 30) train acc: 74.60%; val_acc: 69.40%
(Epoch 11 / 30) train acc: 75.80%; val_acc: 70.76%
(Epoch 12 / 30) train acc: 75.90%; val_acc: 69.35%
(Epoch 13 / 30) train acc: 77.40%; val_acc: 70.00%
(Epoch 14 / 30) train acc: 77.50%; val_acc: 68.26%
(Epoch 15 / 30) train acc: 77.10%; val_acc: 70.59%
(Epoch 16 / 30) train acc: 75.00%; val_acc: 69.11%
(Epoch 17 / 30) train acc: 79.70%; val_acc: 71.57%
(Epoch 18 / 30) train acc: 75.90%; val_acc: 70.72%
(Epoch 19 / 30) train acc: 79.90%; val_acc: 71.43%
(Epoch 20 / 30) train acc: 79.80%; val_acc: 71.28%
(Epoch 21 / 30) train acc: 78.00%; val_acc: 71.07%
(Epoch 22 / 30) train acc: 80.20%; val_acc: 71.10%
(Time 60.00 sec; Iteration 7158 / 9360) loss: 0.991407
End of training; next iteration will exceed the time limit.
```

第三個版本：

加入 Batchnorm

```
weight_scale = 'kaiming'
learning_rate = 0.0025
reg = 0.001

model = DeepConvNet(input_dims=(3, 32, 32), num_classes=10,
                    num_filters=[32, 100],
                    max_pools=[0, 1],
                    reg=reg,
                    weight_scale=weight_scale,
                    batchnorm=True,
                    dtype=dtype,
                    device=device)
solver = Solver(model, data_dict,
                num_epochs=30,
                batch_size=128,
                update_rule=adam,
                optim_config={
                  'learning_rate': learning_rate,
                },
                print_every=10000,
                device=device)
```

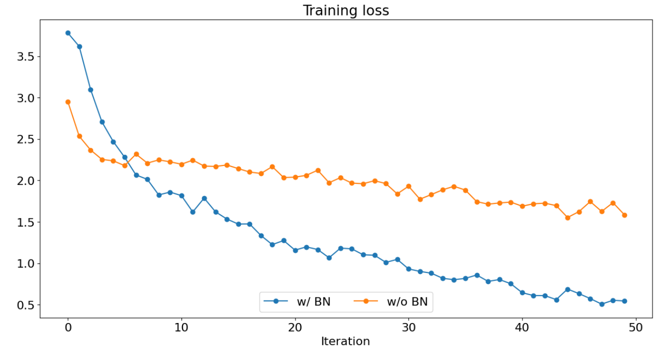
```
(Time 0.02 sec; Iteration 1 / 9360) loss: 3.577355
(Epoch 0 / 30) train acc: 10.70%; val_acc: 11.50%
(Epoch 1 / 30) train acc: 61.40%; val_acc: 58.19%
(Epoch 2 / 30) train acc: 68.40%; val_acc: 64.61%
(Epoch 3 / 30) train acc: 71.20%; val_acc: 66.40%
(Epoch 4 / 30) train acc: 71.80%; val_acc: 66.55%
(Epoch 5 / 30) train acc: 71.60%; val_acc: 66.70%
(Epoch 6 / 30) train acc: 76.00%; val_acc: 68.18%
(Epoch 7 / 30) train acc: 79.80%; val_acc: 70.36%
(Epoch 8 / 30) train acc: 79.00%; val_acc: 68.78%
(Epoch 9 / 30) train acc: 74.70%; val_acc: 67.19%
(Epoch 10 / 30) train acc: 80.10%; val_acc: 69.83%
(Epoch 11 / 30) train acc: 82.00%; val_acc: 70.29%
(Epoch 12 / 30) train acc: 79.60%; val_acc: 70.23%
(Time 59.99 sec; Iteration 3985 / 9360) loss: 0.696833
End of training; next iteration will exceed the time limit.
```

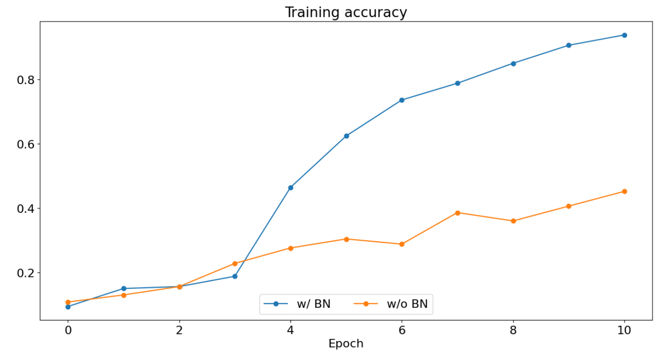
同樣的 `reg` 下，過度擬和的現象減少，但可能是演算法錯誤的關係，讓 testing data 中的準確率極低。

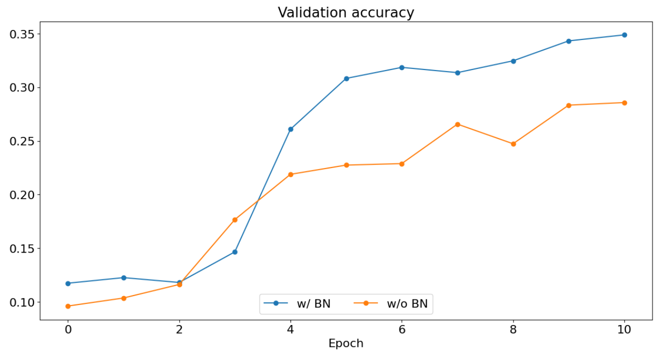
```
Validation set accuracy: 37.7600%
Test set accuracy: 38.3500%
```

我最後採用第二個版本。

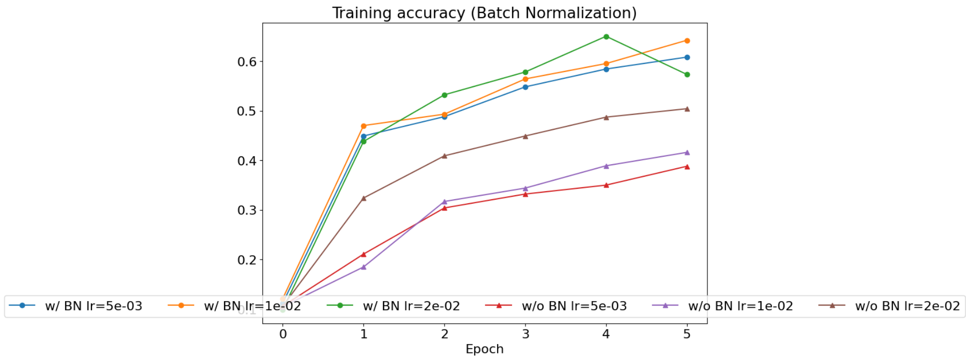
訓練結果分析

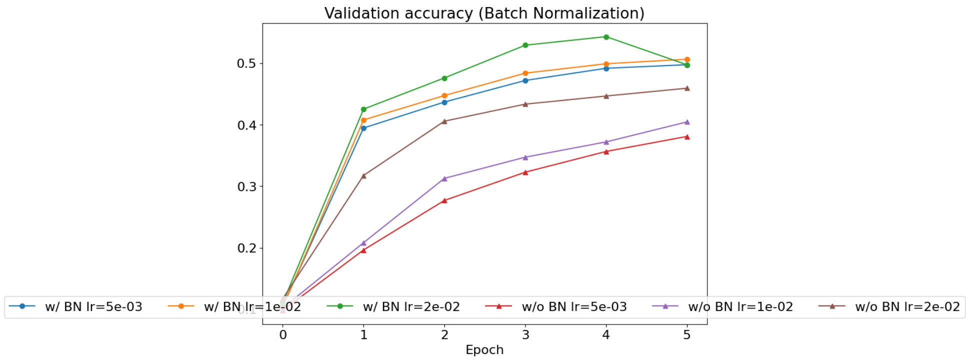






Batchnorm 在訓練上有很大的提升，但如同上方所述，可能在算法中出現一些錯誤導致 testing 準確率極低。





發現最好的組合是 lr=2e-2 w/BN (表中綠色曲線)，只是到最

後 overfitting，可以用 Early stop 避免準確率下降。
若要穩定成長的話，可以稍微降低學習率至 lr=1e-2 (表中橘色曲線)。

其他分析

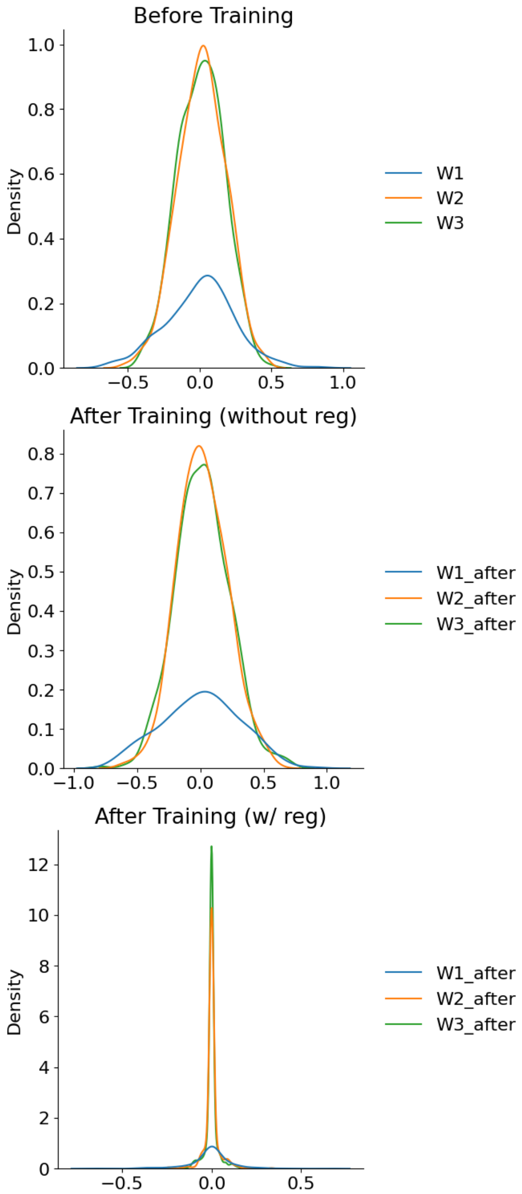
為什麼 Regularization 可以防止過度擬合

$$Cost\ Function\ C = L(\theta) + \lambda\Omega(W)$$
$$\frac{\partial C}{\partial W} = \frac{\partial L(\theta)}{\partial W} + \lambda\frac{\partial\Omega(W)}{\partial W}$$
$$\frac{\partial C}{\partial b} = \frac{\partial L(\theta)}{\partial b} + 0$$

L2 正則化只會對 W 產生影響。

$$\lambda\frac{\partial\Omega(W)}{\partial w_i} = \frac{\lambda}{n}w_i$$
$$w_i \leftarrow w_i - \eta\frac{\partial C}{\partial w_i}$$
$$= w_i - \eta\frac{\partial L(\theta)}{\partial w_i} - \eta\frac{\lambda}{n}w_i$$
$$= (1-\eta\frac{\lambda}{n})w_i - \eta\frac{\partial L(\theta)}{\partial w_i}$$

其中，$(1-\eta\frac{\lambda}{n}) < 1$，所以會讓權重減小，不會放大而讓模型發散。這就是為何正則化可以防止過度擬合。
我們畫個圖證明一下：



Before Training
Density
1.0
0.8
0.6
0.4
0.2
0.0
−0.5
0.0
0.5
1.0
W1
W2
W3
After Training (without reg)
Density
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−1.0
−0.5
0.0
0.5
1.0
W1_after
W2_after
W3_after
After Training (w/ reg)
Density
12
10
8
6
4
2
0
−0.5
0.0
0.5
W1_after
W2_after
W3_after