Neural Network Architecture

○ Fully Convolutional Network（ FCN ）

這次的作業，我使用的是最基本、最簡單的**FCN**網路架構。它主要用於圖像分割任務，通過卷積和反卷積操作來實現圖元級別的分割預測。之後到**89% accuracy**的時候遇到了瓶頸，想試試看寫**CNN**來提升**accuracy**但寫到一半還是放棄了。

```python
class FullyConnected(_Layer):
    def __init__(self, in_features, out_features):
        self.weight = np.random.randn(in_features, out_features) * 0.01
        self.bias = np.zeros((1, out_features))
        self.input = None

    def forward(self, input):
        self.input = input
        output = np.dot(input, self.weight) + self.bias
        return output

    def backward(self, output_grad):
        input_grad = np.dot(output_grad, self.weight.T)
        self.weight_grad = np.dot(self.input.T, output_grad)
        self.bias_grad = np.sum(output_grad, axis=0, keepdims=True)
        return input_grad
```

Activation Function（1）

○ ReLU（Rectified Linear Unit）

我一開始使用的**Activation Function**和助教一樣是**ReLU**，它引入了非線性特性，使神經網路能夠學習複雜的資料模式。有試過改成**sigmoid**和**tanh**，但**accuracy**反而下降了許多，這可能是因爲他們不能抑制梯度消失的問題。相較於**sigmoid**和**tanh**，**ReLU**對於梯度的傳播更加有利，它可以減輕梯度消失問題，使神經網路更容易訓練。

```
class ActivationReLU(_Layer):
    def __init__(self):
        self.input = None

    def forward(self, input):
        self.input = input
        output = np.maximum(0, input)
        return output

    def backward(self, output_grad):
        input_grad = output_grad * (self.input > 0)
        return input_grad
```

First accuracy： 0.8391 ☐

Loss Function

○ Softmax Cross-Entropy Loss

和助教一樣使用這個**loss function**是因爲這次的**Lab**主要是要做圖形分類，而這個**loss function**的強項就是**classification**，它會說明神經網路模型學習正確的分類。

```python
class SoftmaxWithLoss(_Layer):
    def __init__(self):
        self.softmax_output = None
        self.target = None

    def forward(self, input, target):
        self.target = target
        exp_input = np.exp(input - np.max(input, axis=1, keepdims=True))
        softmax_output = exp_input / np.sum(exp_input, axis=1, keepdims=True)
        self.softmax_output = softmax_output
        batch_size = input.shape[0]
        loss = -np.sum(np.log(softmax_output[np.arange(batch_size), target])) / batch_size
        return softmax_output, loss

    def backward(self):
        batch_size = self.target.shape[0]
        input_grad = self.softmax_output.copy()
        input_grad[np.arange(batch_size), self.target] -= 1
        input_grad /= batch_size
        return input_grad
```

Parameter

- Epoch
- Batch Size

我每一輪的Epoch（訓練次數）都設定1000，然後再從1000次裡挑Accuracy最高的一次作爲我那一輪的Epoch值。Batch Size我都是從2，4，8，16.......,2的冪次方來挑最好的作爲我的batch size值。Batch Size設定太小他會跳出Runtime Warning， 訓練得太久，而且也會導致梯度更新產生的noise， 使訓練過程不穩定，模型收斂到不穩定的局部最小值。 Batch Size設定太大，會使模型更容易跳過局部最小值，梯度更新可能會更加平滑，導致模型陷入不太理想的局部最小值。

- Learning Rate
- Validation Data Size

Learning rate我是從0.1，0.01，0.001，0.0001中來挑跑的最高Accuracy的那一次。至於Validation Data Size，我是用類似於最普遍的dataset split ratio，也就是訓練與測試比為0.7：0.3。我這邊用0.65：0.35，60000*0.35 = 21000。

```
EPOCH = 243
Batch_size = 16
Learning_rate = 0.001
```

```
val_image_num=21000
```

Second accuracy: 0.8669

Regularization

- **L1 Regularization**
- **L2 Regularization**

之後爲了更進一步增加**accuracy**，我增加了網路層，模型結構變得更複雜了，所以使用**L1**和**L2**正規化來控制模型的複雜度，減少過擬合的風險，並提高模型的泛化能力。至於正規化係數，我通過交叉驗證，找到最佳的正規化強度。

```python
class FullyConnected(_Layer):
    def __init__(self, in_features, out_features, weight_decay=0.0):  # 添加weight_decay参数
        self.weight = np.random.randn(in_features, out_features) * 0.01
        self.bias = np.zeros((1, out_features))
        self.input = None
        self.weight_decay = weight_decay  # 正则化系数

    def forward(self, input):
        self.input = input
        output = np.dot(input, self.weight) + self.bias
        return output

    def backward(self, output_grad):
        input_grad = np.dot(output_grad, self.weight.T)
        self.weight_grad = np.dot(self.input.T, output_grad)
        self.bias_grad = np.sum(output_grad, axis=0, keepdims=True)

        # 添加正则化项的梯度
        self.weight_grad += 2 * self.weight_decay * self.weight
        return input_grad
```

Activation Function（2）

○ Leaky ReLU（Leaky Rectified Linear Unit）

爲了提高accuracy，我嘗試把ReLU這個Activation Function改成Leaky ReLU，並成功地提高了差不多1%的accuracy。與標準的ReLU啟動函數不同，Leaky ReLU在輸入為負數時不會將啟動值設為零，而是保留一個小的斜率，使其變得略微線性。 相較於ReLU更多地減輕了梯度消失問題，使得神經網路更容易訓練。

```
class ActivationLeakyReLU(_Layer):
    def __init__(self, alpha_1=0.0001):
        self.input = None
        self.alpha = alpha_1  # Leaky ReLU 的小斜率参数，默认为 0.01

    def forward(self, input):
        self.input = input
        output = np.where(input > 0, input, self.alpha * input)
        return output

    def backward(self, output_grad):
        input_grad = output_grad * np.where(self.input > 0, 1, self.alpha)
        return input_grad
```

Third accuracy: 0.8836 ☐

Increasing Network Layer & Capacity

- **Layer**
- **Capacity**

通過增加更多的網路層可以增加網路模型的複雜度，允許模型更深層次地學習特徵，從而更好地捕獲資料的抽象表示。而增加更多的容量可以使模型具有更大的表示能力，更容易適應複雜的資料模式。在**88% accuracy**的時候，每次增加容量，都能提升大約**0.5%**的**accuracy**。

```
beta = 0.5   # L2正则化的超参数

class Network(object):
    def __init__(self, weight_decay=0.0):
        self.fc1 = FullyConnected(28*28, 1024, weight_decay=weight_decay)
        self.act1 = ActivationLeakyReLU(0.01)  # 使用 Leaky ReLU 激活函数
        self.fc2 = FullyConnected(1024, 512, weight_decay=weight_decay)
        self.act2 = ActivationLeakyReLU(0.01)  # 使用 Leaky ReLU 激活函数
        self.fc3 = FullyConnected(512, 10, weight_decay=weight_decay)
        self.loss_layer = SoftmaxWithLoss()

    def forward(self, input, target):
        h1 = self.fc1.forward(input)
        h1_activated = self.act1.forward(h1)
        h2 = self.fc2.forward(h1_activated)
        h2_activated = self.act2.forward(h2)
        pred, loss = self.loss_layer.forward(self.fc3.forward(h2_activated), target)
        return pred, loss
```

Optimizer

○ SGD（Stochastic Gradient Descent）

爲了加速模型的訓練過程和提升**accuracy**，我在網路模型裡增加了**SGD optimizer**。它的主要功能是最小化模型的損失函數，通過調整模型的參數，使損失函數的值不斷減小，從而使模型更適應訓練資料。**SGD optimizer**還具有隨機性，在每次反覆運算中隨機選擇一個小批次（**mini-batch**）的訓練樣本來計算梯度，從而避免陷入局部最小值，並且可以加速訓練過程。**SGD**的隨機性和雜訊也能夠防止過擬合，使模型更容易泛化到未見過的資料。

```
    def update(self, lr):
        self.sgd_optimizer(lr)

    def sgd_optimizer(self, lr):
        self.fc1.weight -= lr * (self.fc1.weight_grad + alpha * np.sign(self.fc1.weight) + beta * 2 * self.fc1.weight_decay *
self.fc1.weight)
        self.fc1.bias -= lr * self.fc1.bias_grad
        self.fc2.weight -= lr * (self.fc2.weight_grad + alpha * np.sign(self.fc2.weight) + beta * 2 * self.fc2.weight_decay *
self.fc2.weight)
        self.fc2.bias -= lr * self.fc2.bias_grad
        self.fc3.weight -= lr * (self.fc3.weight_grad + alpha * np.sign(self.fc3.weight) + beta * 2 * self.fc3.weight_decay *
self.fc3.weight)
        self.fc3.bias -= lr * self.fc3.bias_grad
```

Final accuracy： 0.9028