Deep Learning Lab1 Report

110511118 陳孟頡

Task 1.

I. Layer Implementation

1. Fully Connected Layer

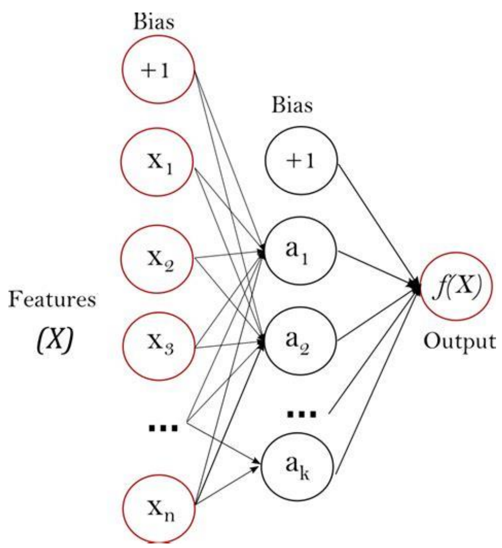
(1) __init__():

宣告全連接層中使用到的參數，包含 weight 以及 bias。

(2) forward():

如圖示，將輸入與 weight 進行 dot product 後，與 bias 進行相加，即可傳至輸出。

```
self.input = input
output = np.dot(input, self.weight) + self.bias
return output
```



(3) backward():

先將包含輸入梯度、輸出特徵的矩陣與 weight 矩陣進行 dot product，來得到該曾輸入的 loss 梯度；接著在和輸入進行 dot product，即可得到 weight 梯度。

```
input_grad = np.dot(output_grad, self.weight.T)
self.weight_grad = np.dot(self.input.T, output_grad)
```

而 bias 梯度的計算，則是由整個 batch 的輸出梯度加總。

```
self.bias_grad = np.sum(output_grad, axis=0, keepdims=True)
```

(4) update():

以計算所得梯度及給予的 learning rate 更新參數。

```
self.weight -= lr * self.weight_grad
self.bias -= lr * self.bias_grad
```

2. Convolution Layer

(1) __init__():

宣告卷積層參數，包含 channel 數量、weight、bias 以及 stride、padding 數量。

(2) forward():

首先，按公式計算輸出的維度

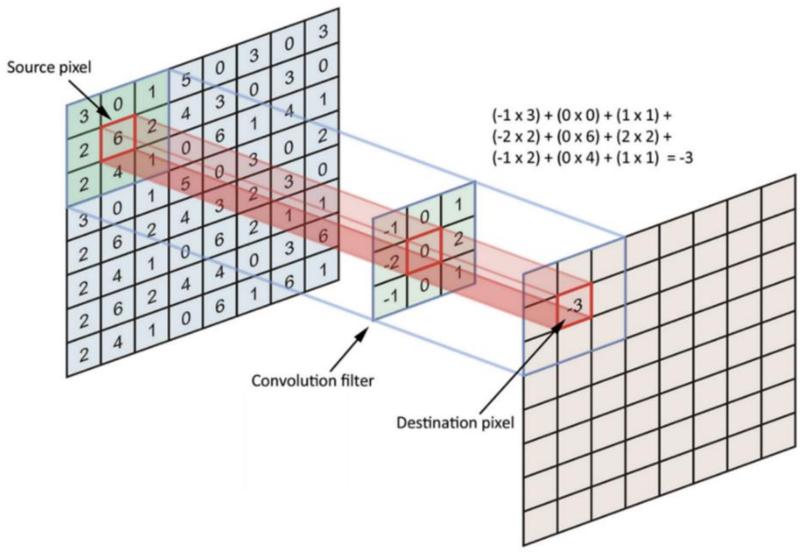
```
output_height = (input_height - self.kernel_size + 2 * self.padding) // self.stride + 1
output_width = (input_width - self.kernel_size + 2 * self.padding) // self.stride + 1
```

$$Output_{\ size} = \frac{N \ - F + (2 * padding)}{stride} + 1$$

接著依照 padding 數量進行周圍的填補後，以三層 for-loop 來計算 convolution。

```
for i in range(output_height):
    for j in range(output_width):
        input_slice = padded_input[:, :, i*self.stride:i*self.stride+self.kernel_size, j*self.stride:j*self.stride+self.kernel_size]
        for k in range(self.out_channels):
            self.output[:, k, i, j] = np.sum(input_slice * self.weights[k], axis=(1, 2, 3)) + self.biases[k]
```

進行 convolution 計算時，按照 stride 數移動 kernel 與輸入的特徵圖相乘，並加總於輸出特徵圖，迭代完成整張圖。



(3) backward():

為確保維度一致，先進行輸入以及輸入梯度的 padding，接著以迴圈計算 weight、bias 梯度，最後移除 padding，回傳計算完梯度。

```
for i in range(output_height):
    for j in range(output_width):
        h_start = i * self.stride
        h_end = h_start + self.kernel_size
        w_start = j * self.stride
        w_end = w_start + self.kernel_size

        input_slice = padded_input[:, :, h_start:h_end, w_start:w_end]

        for k in range(self.out_channels):
            weights_grad[k] += np.tensordot(input_slice, output_grad[:, k, i, j], axes=((0), (0)))
            padded_input_grad[:, :, h_start:h_end, w_start:w_end] += self.weights[k] * output_grad[:, k, i, j][:, np.newaxis, np.newaxis, np.newaxis]

        biases_grad[:, 0] += np.sum(output_grad[:, :, i, j], axis=0)
```

(4) update():

以計算所得梯度及給予的 learning rate 更新參數。

3. Batch Normalization

(1) __init__():

宣告 normalization 使用參數，其中 gamma 與 beta 代表 normalized 後數值的 scale 以及 shift。

$$y = \gamma\hat{x} + \beta$$

(2) forward():

首先將輸入的資料轉換為 4D 陣列，由於此次作業中使用到卷積層以及全連接層，需判斷其尺寸並進行統一。

```
if len(input.shape) == 2:
    input = input.reshape(input.shape[0], self.num_features, 1, 1)
```

若沒有此步驟，可能出現大小不相符錯誤訊息。

```
AxisError: axis 2 is out of bounds for array of dimension 2
```

在 training 過程中，計算整個 batch 的 mean 以及 variance。

```
batch_mean = np.mean(input, axis=(0, 2, 3), keepdims=True)
batch_var = np.var(input, axis=(0, 2, 3), keepdims=True)
```

接著進行 normalization，並以 beta 及 gamma 進行 scale 及 shift。

```
normalized = (input - batch_mean) / np.sqrt(batch_var + self.eps)
output = self.gamma * normalized + self.beta
```

最後以 batch mean 及 variance 更新 running mean 及 variance，完成 normalization。

```
self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * batch_mean
self.running_var = self.momentum * self.running_var + (1 - self.momentum) * batch_var
```

(3) backward():

依序計算 beta 及 gamma 梯度、normalized input 梯度、variance 及 mean 梯度、輸入梯度。

```
self.gamma_grad = np.sum(output_grad * self.normalized, axis=(0, 2, 3), keepdims=True)
self.beta_grad = np.sum(output_grad, axis=(0, 2, 3), keepdims=True)

normalized_grad = output_grad * self.gamma

var_grad = np.sum(normalized_grad * (self.input - self.batch_mean), axis=(0, 2, 3), keepdims=True) * -0.5 * np.power(self.batch_var + self.eps, -1.5)

mean_grad = np.sum(normalized_grad, axis=(0, 2, 3), keepdims=True) * -1 / np.sqrt(self.batch_var + self.eps) + var_grad * \
    np.mean(-2 * (self.input - self.batch_mean), axis=(0, 2, 3), keepdims=True)

N = self.input.shape[0] * self.input.shape[2] * self.input.shape[3]
input_grad = normalized_grad / np.sqrt(self.batch_var + self.eps) + var_grad * 2 * (self.input - self.batch_mean) / N + mean_grad / N
```

(4) update():

以計算所得梯度及給予的 learning rate 更新參數。

4. Max Pooling

(1) __init__():

宣告參數。

(2) forward():

首先計算輸出大小。

```
pool_height = (height - self.pool_size) // self.stride + 1
pool_width = (width - self.pool_size) // self.stride + 1
```

依照設定 pooling size 進行取值。

```
for i in range(pool_height):
    for j in range(pool_width):
        h_start = i * self.stride
        h_end = h_start + self.pool_size
        w_start = j * self.stride
        w_end = w_start + self.pool_size

        pool_region = input[:, :, h_start:h_end, w_start:w_end]
        self.output[:, :, i, j] = np.max(pool_region, axis=(2, 3))
        self.max_indices[:, :, i, j] = np.argmax(pool_region.reshape(batch_size, channels, -1), axis=2)
```

(3) backward():

先在傳入的 output 梯度迭代計算對應的 input region。

```
for i in range(output_grad.shape[2]):
    for j in range(output_grad.shape[3]):
        h_start = i * self.stride
        h_end = min(h_start + self.pool_size, height)
        w_start = j * self.stride
        w_end = min(w_start + self.pool_size, width)
```

找到最大值的 index 後，將梯度進行回傳。

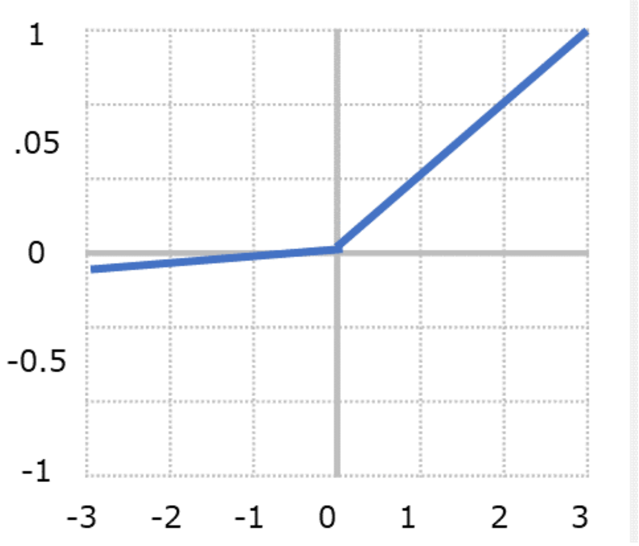
```
for b in range(batch_size):
    for c in range(channels):
        max_idx = self.max_indices[b, c, i, j]

        h_idx = h_start + max_idx // (w_end - w_start)
        w_idx = w_start + max_idx % (w_end - w_start)

        if h_idx < height and w_idx < width:
            input_grad[b, c, h_idx, w_idx] += output_grad[b, c, i, j]
```

5. Leaky ReLU

實作 leaky ReLU activation function，其中 alpha 為資料<0 時的斜率。



6. Softmax with loss

(1) forward():

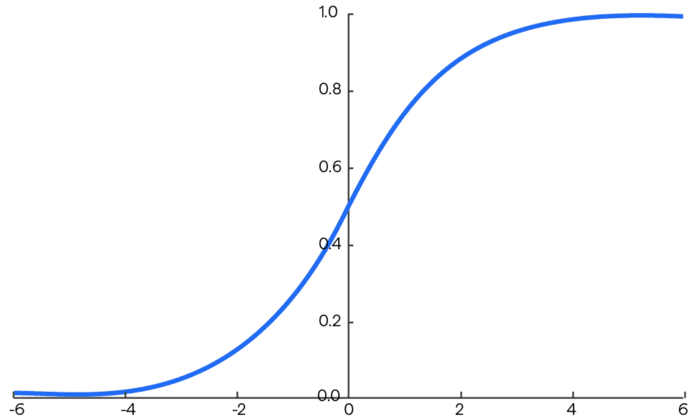
使用 softmax 公式計算各個輸入的可能輸出。

```
exp_input = np.exp(input - np.max(input, axis=1, keepdims=True))
self.predict = exp_input / np.sum(exp_input, axis=1, keepdims=True)
```

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Loss 部分則以 log 計算得(cross entropy)。

```
self.target = target.astype(int)
your_loss = -np.mean(np.log(self.predict[range(self.target.shape[0]), self.target]))
```



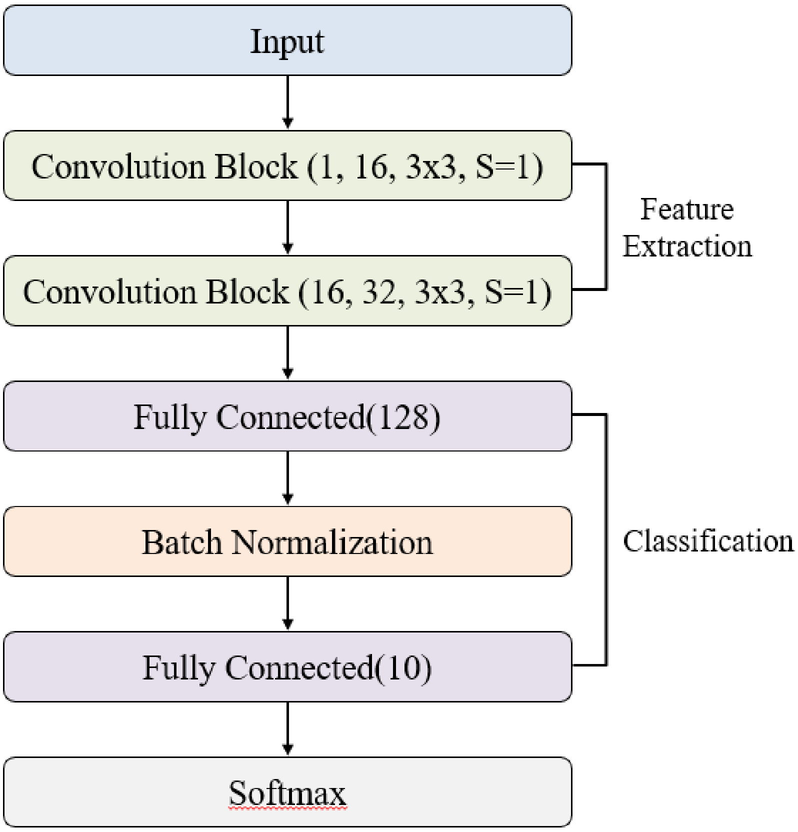
(2) backward():

使用 loss 來計算輸入梯度。

```
input_grad = self.predict.copy()
input_grad[range(self.target.shape[0]), self.target] -= 1
input_grad /= self.target.shape[0]
```

II. Network Architecture

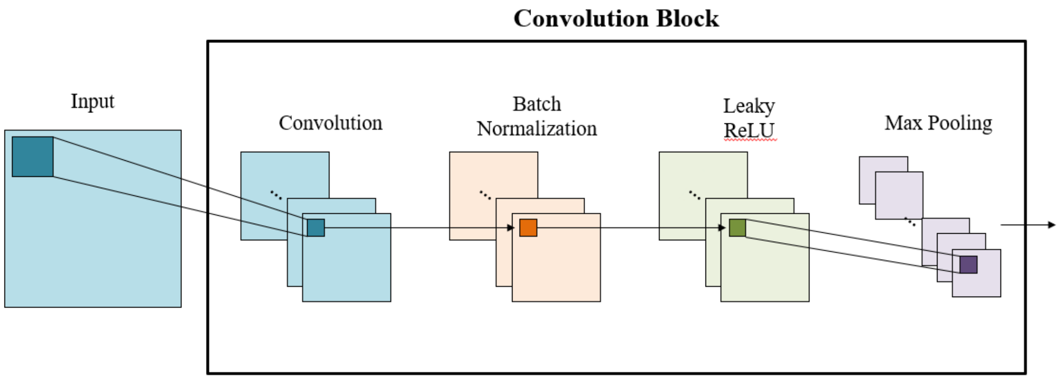
1. Block diagram



2. Structure details

本次作業使用兩層卷積層、兩層全連接層作為主骨架，卷積層負責進行特徵萃取，全連接層進行分類任務。

其中，convolution block 內詳細架構如下圖:



(1) Convolution:

主要萃取特徵層。相較於全連接層，卷積層更能夠提取出圖像中的特徵，但 CPU 計算負擔比全連接層來的大，因此，增加 channel 數或卷積層數雖可提升準確度，同時也會使訓練時間大幅增加，需視應用進行取捨。

(2) Batch Normalization:

將輸入資料進行標準化，使訓練更加穩定，同時縮短訓練時間。

(3) Leaky ReLU:

相較於 ReLU，leaky ReLU 在輸入為負值時能維持一定的輸出，解決 dead neuron 問題。也因此，leaky ReLU 再進行梯度更新時傳遞會較 ReLU 流暢。

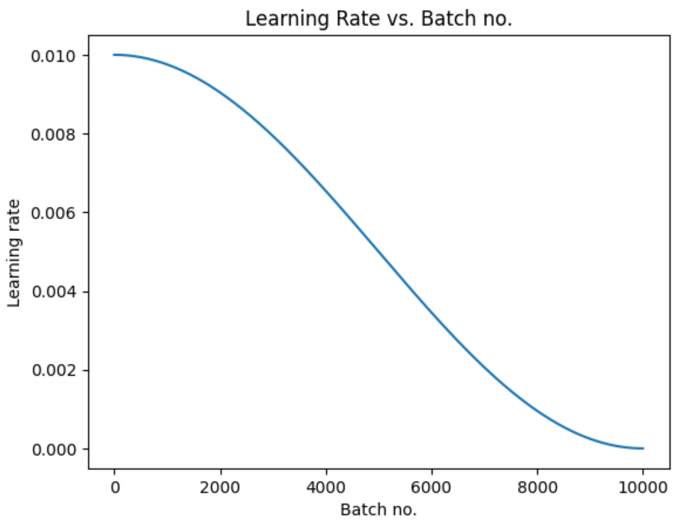
(4) Max Pooling:

進一步增加此架構的 receptive field，使模型能更準確將輸入進行非類任務。

III. Training Method

1. CosineAnnealing Scheduler

訓練中使用課程所教的 learning rate scheduler “Cosine Annealing”，其 learning rate 變化如圖:



一般模型訓練時會逐漸逼近 error surface 的最低點，此時若 learning rate 太大，會使 loss 在最小值附近震盪，因此使用 Cosine Annealing 動態變化 learning rate，在接近訓練尾端調降，來使模型能更佳的收斂至 loss 最小值。

2. Visualizing Training Process

當模型中包含卷積層，訓練時間會增加數十倍，為了明確了解此時訓練進度，使用 tqdm library 將訓練進度視覺化，如下圖:

```
Epoch 12/20
Epoch 12/20:   53%|█████      | 266/500 [08:08<07:23,  1.90s/batch, Train Loss=0.446, Train Acc=91.6]
```

3. Saving Best Model

訓練過程中可能因為 overfitting 或 learning 太大，使模型準確度在後面 epoch 準確度反而下降，因此使用 pickle library 將最佳的模型參數進行 serialization 儲存下來，並在 testing 時以最佳參數來驗證。

```
if total_val_loss < best_val_loss:
    best_val_loss = total_val_loss
    print(f"New best model found at epoch {epoch}, saving model with Val_loss: {avg_val_loss:.4f}")
    with open(best_model_path, 'wb') as f:
        pickle.dump(net, f)
```

Task 2.

I. Implementation

1. Network architecture

使用 Pytorch 函式庫進行模型建構:

(1) __init__():

直接使用 torch.nn、torch.nn.function 宣告模型各層參數

```
def __init__(self):
    super(Net, self).__init__()
    # First convolutional layer
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
    self.bn1 = nn.BatchNorm2d(16)
    self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

    # Second convolutional layer
    self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
    self.bn2 = nn.BatchNorm2d(32)
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

    # Fully connected layer
    self.fc1 = nn.Linear(in_features=7 * 7 * 32, out_features=128)
    self.bn3 = nn.BatchNorm1d(128)

    # Output layer
    self.fc2 = nn.Linear(in_features=128, out_features=10)
```

(2) forward():

從輸入至輸出撰寫 forward 路徑

```
def forward(self, x):
    # Pass through convolutional layers
    x = self.pool1(F.leaky_relu(self.bn1(self.conv1(x))))
    x = self.pool2(F.leaky_relu(self.bn2(self.conv2(x))))

    # Flatten the tensor for fully connected layers
    x = x.view(x.size(0), -1)

    # Pass through fully connected layers with dropout
    x = F.leaky_relu(self.bn3(self.fc1(x)))
    x = self.fc2(x)

    return x
```

2. Training method

(1) Criterion, optimizer and scheduler

此次作業為分類任務，使用 cross entropy 來計算 loss；optimize 採用 SGD，scheduler 使用 CosineAnnealingLR。

```
import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=EPOCH)
criterion = criterion = nn.CrossEntropyLoss()
```

(2) Training acceleration

使用 cuda 來加速訓練過程。

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print('Device : ', device)
net = Net()
net = net.to(device)
```

```
inputs = train_data_tensor[it * Batch_size:(it + 1) * Batch_size].view(-1, 1, 28, 28).to(device)
labels = train_label_tensor[it * Batch_size:(it + 1) * Batch_size].to(device)
labels_onehot = train_label_onehot_tensor[it * Batch_size:(it + 1) * Batch_size].to(device)
```

II. Comparison to Task 1

1. Training Speed

由於使用 GPU 進行加速，task 2 中訓練速度較 task 1 快了約 200 倍。

Task 1:

```
Epoch 1/20
Epoch 1/20: 100%|██████████| 500/500 [15:30<00:00,  1.86s/batch, Train Loss=1.95, Train Acc=59.6]
Validation: 100%|██████████| 100/100 [00:45<00:00,  2.19it/s]
Task1  | Epoch:   1 | Train Loss:   1.9479 | Train Acc: 59.6140 | Val Loss:   1.6214 | Val Acc: 72.0500
New best model found at epoch 1, saving model with Val_loss: 1.6214
Epoch 2/20
Epoch 2/20: 100%|██████████| 500/500 [14:32<00:00,  1.74s/batch, Train Loss=1.38, Train Acc=74.7]
Validation: 100%|██████████| 100/100 [00:46<00:00,  2.14it/s]
Task1  | Epoch:   2 | Train Loss:   1.3809 | Train Acc: 74.7100 | Val Loss:   1.1471 | Val Acc: 79.4900
New best model found at epoch 2, saving model with Val_loss: 1.1471
Epoch 3/20
Epoch 3/20: 100%|██████████| 500/500 [15:00<00:00,  1.80s/batch, Train Loss=1.01, Train Acc=81.4]
Validation: 100%|██████████| 100/100 [00:46<00:00,  2.15it/s]
Task1  | Epoch:   3 | Train Loss:   1.0079 | Train Acc: 81.4120 | Val Loss:   0.8570 | Val Acc: 84.6700
New best model found at epoch 3, saving model with Val_loss: 0.8570
```

Task 2:

```
Epoch 1/20: 100%|██████████| 500/500 [00:04<00:00, 109.12batch/s, Train Loss=0.289, Train Acc=92.2]
Task2  | Epoch:   1 | Train Loss: 0.2890 | Train Acc: 92.1820 | Val Loss: 0.1162 | Val Acc: 96.8500
New best model saved with validation loss: 0.1162
Epoch 2/20: 100%|██████████| 500/500 [00:02<00:00, 184.91batch/s, Train Loss=0.0871, Train Acc=97.8]
Task2  | Epoch:   2 | Train Loss: 0.0871 | Train Acc: 97.8240 | Val Loss: 0.0845 | Val Acc: 97.6100
New best model saved with validation loss: 0.0845
Epoch 3/20: 100%|██████████| 500/500 [00:02<00:00, 191.74batch/s, Train Loss=0.0506, Train Acc=98.9]
Task2  | Epoch:   3 | Train Loss: 0.0506 | Train Acc: 98.8880 | Val Loss: 0.0743 | Val Acc: 97.8100
New best model saved with validation loss: 0.0743
```

2. Loss Decreasing Speed

從上圖也可看出，task 2 在第一個訓練 epoch 中即可達到 92% training accuracy、96% validation accuracy，但 task 1 中卻只有 59% training acceleration 及 72% validation accuracy。推測造成此現象原因如下:

(1) Pytorch 函式庫經優化，因此能對模型進行最佳化的訓練。
(2) Task 2 中額外使用了 optimizer “SGD”，也會提升訓練品質。

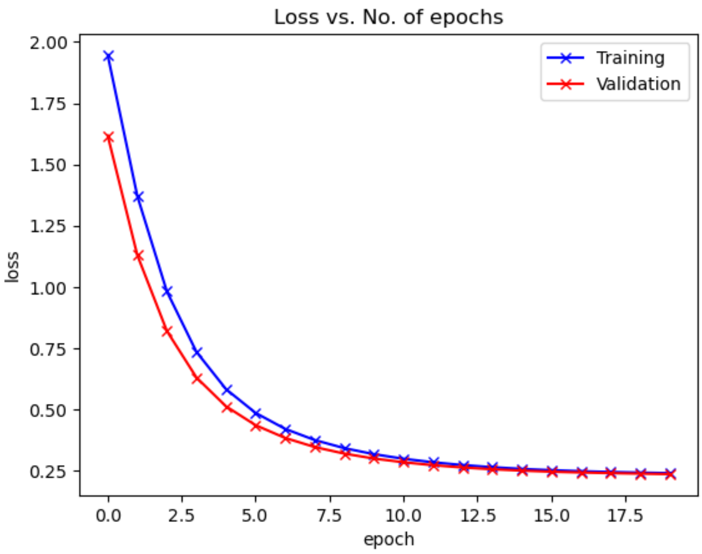
<table>
<tr><th colspan="2"></th><th>Task 1</th><th>Task 2</th></tr>
<tr><td colspan="2">Training Speed</td><td>900s/epoch</td><td>4s/epoch</td></tr>
<tr><td rowspan="2">Accuracy at Epoch 1</td><td>Training Accuracy</td><td>59.61%</td><td>92.18%</td></tr>
<tr><td>Validation Accuracy</td><td>72.05%</td><td>96.85%</td></tr>
</table>

Result

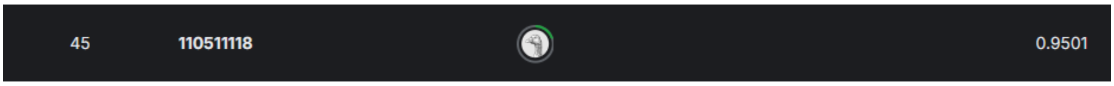
I. Task 1

Training accuracy: 95.26%

Validation accuracy: 95.25%



Kaggle accuracy: 95.01%



II. Task 2

Training accuracy: 100.00%

Validation accuracy: 98.29%