AI6126 Advanced Computer Vision

Homework Assignment 1

School of Computer Science and Engineering

Programme: MSAI

Date: 13 February 2024

Authored By: Tan Jie Heng Alfred (G2304193L)

Question 1

Let h and w be the predicted height (in metres) and weight (in kilograms) of a person, given $\mathbf{x}$. We further assume that this follows the supervised training scheme (i.e., we are given ground truth labels).

Given the units, we expect $h \ll w$ in general. If we were to train our multivariate regression model using gradient descent, the gradient of the parameters, $\boldsymbol{\beta}$, with respect to h will be much smaller than that with respect to w:

$$\nabla_{\boldsymbol{\beta}} h \ll \nabla_{\boldsymbol{\beta}} w$$

Because of this, the training will be unstable (i.e., the parameters $\boldsymbol{\beta}$ would likely oscillate) and the model will struggle to converge.

To address this, we could normalize the values of h and w with schemes such as z-score normalization or min-max scaling. After performing normalization, we would obtain $\tilde{h}$ and $\tilde{w}$ that are within a similar range of values, say $(0,1)$. Then, we can train $\boldsymbol{\beta}$ using $\tilde{h}$ and $\tilde{w}$ instead. Finally, given a new data instance $\mathbf{x}^*$ at inference time, we would first compute the corresponding $\widetilde{h^*}$ and $\widetilde{w^*}$, before 'un-normalizing' to obtain h^* and w^* which we are interested in.

Another solution would be, instead of using stochastic gradient descent as our optimizer, we could use other optimizing scheme such as Adam. In particular, we use an optimizer that incorporates momentum which dampens the oscillation by including historical gradients.

Question 2

Number of epochs $= \frac{100 \times 100{,}000}{4000} = 2{,}500$

Question 3

Note that activation function layers (e.g., ReLU and softmax) and pooling layers (e.g., max pooling) do not have any parameters. Therefore, layers 2, 3, 5, 6, 8, 10, and 12 do not have any parameters. The remaining layers are calculated as follows:

Layer	Number of Parameters
Conv2d-1	We start with $1 \times 32 \times 32$ input shape and ends with $6 \times 28 \times 28$ output shape. This suggests that the kernel we use is of the shape $6 \times 1 \times 5 \times 5$ (i.e., 6 kernels of volume $1 \times 5 \times 5$), as the stride = 1 and padding = 0. Since each kernel has a bias, we end up with the following number of parameters: $6(1 \times 5 \times 5 + 1) = 6 \times 26 = 156$
Conv2d-4	The output of layer 3 is of the shape $6 \times 14 \times 14$, suggesting our input shape (for layer 4) should be the same. Now, our output of layer 4 has shape $16 \times 10 \times 10$, suggesting that the kernel is of the shape $16 \times 6 \times 5 \times 5$. Similar to layer 1, we calculate the number of parameters as: $16(6 \times 5 \times 5 + 1) = 16 \times 151 = 2416$
Conv2d-7	The reasoning here is the same as the above layers. We can deduce that the kernel shape is $120 \times 16 \times 5 \times 5$. Hence, we the number of parameters we have is: $120(16 \times 5 \times 5 + 1) = 120 \times 401 = 48120$

Linear-9	Input shape here is $120 \times 1 \times 1$ (which is equivalent to a vector of 120 dimensions) and output shape is 84×1. Since this is a fully connected layer, and each output neuron has one bias, the number of parameters is: $$120(84+1) = 120 \times 85 = 10200$$
Linear-11	Similar to Linear-9, we have input shape 84×1 and output shape 10×1, the number of parameters is: $$84(10+1) = 84 \times 11 = 924$$
Total	Therefore, the entire network has a total of: $$156 + 2416 + 48120 + 10200 + 924 = \mathbf{61816\ parameters}$$

<u>Question 4</u>

<u>First layer</u>
Since we have input features maps of dimensions $256 \times 32 \times 32$, we can deduce that $d_1 = 256$, in order to match the volume of the input feature maps. Now, to deduce n_1, we look to the second layer. In particular, the number of output channels from layer 1, determined by n_1, has to be 64, since we use a $64 \times 3 \times 3$ convolutional layer in layer 2. Hence, $n_1 = 64$.
The output feature maps from layer 1 will therefore be of dimensions:
$$64 \times (32 - F_1 + 1) \times (32 - F_1 + 1) = 64 \times (33 - F_1) \times (33 - F_1)$$

<u>Second layer</u>
Since we have 32 filters of size $64 \times 3 \times 3$ with padding $= 1$, the output feature maps will have the following dimensions:
$$\begin{aligned} &32 \times (33 - F_1 - 3 + 2(1) + 1\,) \times (33 - F_1 - 3 + 2(1) + 1\,) \\ &= 32 \times (33 - F_1) \times (33 - F_1) \end{aligned}$$

<u>Third layer</u>
Again, we can deduce $d_2 = 32$, following the dimensions of the output features maps from layer 2. Now, since we have an element-wise addition subsequently, our output feature maps have to be of the dimension $256 \times 32 \times 32$, matching that of the initial input feature maps. Hence, $n_2 = 256$.
To find F_1 and F_2, note that after applying the third convolution layer, our output feature maps are of the dimensions:
$$\begin{aligned} &256 \times (33 - F_1 - F_2 + 1) \times (33 - F_1 - F_2 + 1) \\ &= 256 \times (34 - F_1 - F_2) \times (34 - F_1 - F_2) \end{aligned}$$
Since we need the output feature maps to match $256 \times 32 \times 32$, we therefore have:
$$\begin{aligned} 34 - F_1 - F_2 &= 32 \\ \Rightarrow F_1 + F_2 &= 2 \end{aligned}$$
Since $F_1, F_2 \in \mathbb{Z}^+$, we must have $F_1 = 1 = F_2$.
Altogether, we have $n_1 = 64, d_1 = 256, F_1 = 1$ and $n_2 = 256, d_2 = 32,$ and $F_2 = 1$.

Question 5

Yes, batch normalization can have a regularization effect.

Let a $\{\mathbf{x}_k\}_{k=1}^N$ be a mini-batch of N feature maps. We perform batch normalization across each channel, c, as such,

$$\widetilde{\mathbf{x}_{k,c}} = \frac{\mathbf{x}_{k,c} - \mu_{\mathbf{x},c}}{\sqrt{\sigma_{\mathbf{x},c}^2 + \epsilon}},$$

for some small $\epsilon > 0$, and for all $1 \le k \le N$. Here,

$$\mu_{\mathbf{x},c} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_{i,c}$$
$$\sigma_{\mathbf{x},c}^2 = \frac{1}{N}\sum_{i=1}^{N} \left(\mathbf{x}_{i,c} - \mu_{\mathbf{x},c}\right)^2.$$

To improve re-introduce more non-linearity, we further evaluate,

$$\mathbf{y}_{k,c} = \gamma_{\mathbf{x},c}\widetilde{\mathbf{x}_{k,c}} + \beta_{\mathbf{x},c},$$

where $\gamma_{\mathbf{x},c}$ and $\beta_{\mathbf{x},c}$ are learnable parameters. Then, we use all the $\mathbf{y}_{k,c}$ as the input feature maps for the subsequent layer (or an activation function), for all c.

Note that $\mu_{\mathbf{x},c}$ and $\sigma_{\mathbf{x},c}^2$ are both statistics of the minibatch $\{\mathbf{x}_k\}_{k=1}^N$ – their values are dependent on the minibatch. In other words, given another unique minibatch $\{\mathbf{x}_i'\}_{i=1}^N$, each of the $\mu_{\mathbf{x}',c}$ and $\sigma_{\mathbf{x}',c}^2$ are going to be different from $\mu_{\mathbf{x},c}$ and $\sigma_{\mathbf{x},c}^2$ above. Specifically, the intermediate feature maps from a particular training example are normalized by different amounts, depending on the other examples in the minibatch, hence producing different activations in different epoch. This added noise prevents the model from memorizing the training examples, therefore reducing overfitting, and improving generalization.

Question 6

In the ResNet architecture, batch normalization (BN) is done on output feature maps from the convolution layers, and before inputting into the ReLU activation function[1]. This helps the model achieve faster convergence, reduced sensitivity to learning rate, and improved generalization. An alternative placement of BN, however, would be after the ReLU activation function (whose input are the output feature maps from the convolution layers).

Performing BN before ReLU helps prevent the "dying" ReLU problem, where the activations from the ReLU is perpetually zero. This can happen during training when a gradient update is so large that the preceding convolution layer produces output that are negative for most of the input feature maps. This not only "kills" the activation, but also prevents the model from learning, as the gradient is also zero.

BN, before ReLU, is able to address this, as the negative outputs from the convolution would be normalized to values that are potentially non-negative. Therefore, the expected value and variance of the input into the ReLU function become,

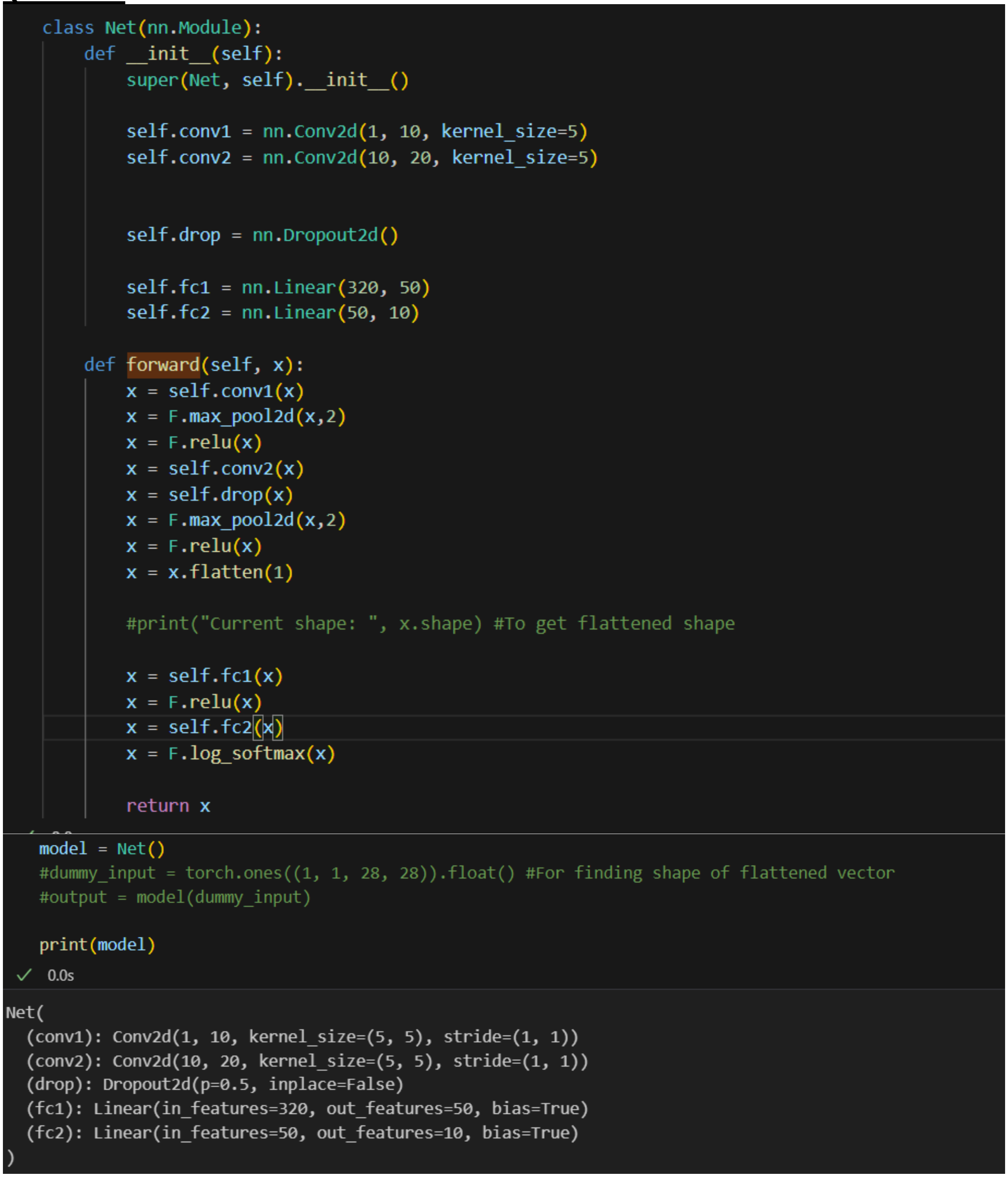
$$\begin{aligned}
\mathbb{E}(\mathbf{y}_{k,c}) &= \mathbb{E}(\gamma_{\mathbf{x},c}\widetilde{\mathbf{x}_{k,c}} + \beta_{\mathbf{x},c}) \quad \text{(following notation in Q5)} \\
&= \gamma_{\mathbf{x},c}\mathbb{E}(\widetilde{\mathbf{x}_{k,c}}) + \beta_{\mathbf{x},c} \\
&= \beta_{\mathbf{x},c} \ (\because \mathbb{E}(\widetilde{\mathbf{x}_{k,c}}) = 0) \\
\mathrm{Var}(\mathbf{y}_{k,c}) &= \mathrm{Var}(\gamma_{\mathbf{x},c}\widetilde{\mathbf{x}_{k,c}} + \beta_{\mathbf{x},c}) \\
&= \gamma_{\mathbf{x},c}^2\mathrm{Var}(\widetilde{\mathbf{x}_{k,c}}) \\
&= \gamma_{\mathbf{x},c}^2 \ \ (\because \mathrm{Var}(\widetilde{\mathbf{x}_{k,c}}) = 0)
\end{aligned}$$

[1] https://doi.org/10.48550/arXiv.1512.03385

From this, we see that the learned parameters, β and γ, controls the activations from ReLU to some extent. For example, if $\beta \gg 0$ and $\gamma \approx 0$, then majority of $\mathbf{y}$ will not be clipped by ReLU and will be propagated forward. Hence, performing BN prior to activation function affords this control to the model, which in turn addresses the "dying" ReLU problem.
Furthermore, BN helps mitigate the internal covariate shift, which refers to the change in distribution of intermediate feature maps during training. Addressing this is important as the shift makes it challenging for deeper layers to learn stable representation.
As seen above, BN maintains a constant statistics (i.e., $\mu = \beta$ and $\sigma^2 = \gamma$) in each layer, across the minibatch, allowing deeper layers to learn representations in a more stable landscape. Compare this to if ReLU is performed before BN, then the distribution of the feature maps would be fundamentally changes, as they are necessarily non-negative. This defeats the purpose of performing BN to standardize the distribution of the feature maps before further propagation.
However, some empirical studies[2] show that different placement of BN could be beneficial for different networks, and could be dependent on the downstream task as well.

[2] https://doi.org/10.1109/ICSPIS48872.2019.9066113

Question 7



```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)


        self.drop = nn.Dropout2d()

        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.max_pool2d(x,2)
        x = F.relu(x)
        x = self.conv2(x)
        x = self.drop(x)
        x = F.max_pool2d(x,2)
        x = F.relu(x)
        x = x.flatten(1)

        #print("Current shape: ", x.shape) #To get flattened shape

        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.log_softmax(x)

        return x
```

```python
model = Net()
#dummy_input = torch.ones((1, 1, 28, 28)).float() #For finding shape of flattened vector
#output = model(dummy_input)

print(model)
```

✓ 0.0s

```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```