بسمه تعالی

HW4- Deep Neural Networks

Supervisor:

Prof. Johari Majd

Student:

Morteza Bigdeli - 40261662001

Tarbiat Modares University

Spring 2024

Table of Contents

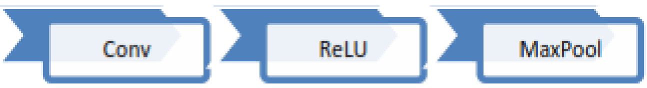
Table of Figures

1. Pen and Paper Exercises

a) CNN Operations

I.

i) In the following, we investigate a tiny CNN. It consists of a 2D convolutional layer, followed by a ReLU, followed by 2D max-pooling. The convolutional layer has a kernel w of size 3 (and no bias term), a stride of 1, and a zero-padding of 1 (i.e., the boundary is padded with 0's). The max-pooling has a kernel size of 2 (the size of the window to take a max over). The input x is a 2D matrix (or an image with a single channel, i.e., gray image). We want to transform the image into the target t. To measure the difference between x and t, we use the mean absolute error (or L1 loss). For simplicity, we do not use a regularization term. $\mathcal{L}_{L1}$.



$$x = \begin{pmatrix} 3 & 1 & 9 & 2 \\ 0 & 0 & 3 & 3 \\ 0 & 3 & 1 & 1 \\ 8 & 2 & 8 & 7 \end{pmatrix}, w = \begin{pmatrix} 0.2 & 0.3 & -0.1 \\ 0.3 & -1.1 & 1.0 \\ -1.0 & 1.1 & 0.3 \end{pmatrix}, t = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$$

Perform a forward pass through the tiny CNN to calculate $\mathcal{L}_{L1}$.

$$X = \begin{bmatrix} 3 & 1 & 9 & 2 \\ 0 & 0 & 3 & 3 \\ 0 & 3 & 1 & 1 \\ 8 & 2 & 8 & 7 \end{bmatrix}, W = \begin{bmatrix} 0.2 & 0.3 & -0.1 \\ 0.3 & -1.1 & 1.0 \\ -1.0 & 1.1 & 0.3 \end{bmatrix}, t = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}$$

Padding, $P=1$, and Applying 3x3 filter with Stride=1

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 9 & 2 & 0 \\ 0 & 0 & 0 & 3 & 3 & 0 \\ 0 & 0 & 3 & 1 & 1 & 0 \\ 0 & 8 & 2 & 8 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$w = 3\times 3$

$$\begin{bmatrix} 0.2 & 0.3 & -0.1 \\ 0.3 & -1.1 & 1.0 \\ -1.0 & 1.1 & 0.3 \end{bmatrix}$$

the output size is : $(6-3+1)(6-3+1) = 4\times 4$

$y_{11} = (0\times 0.2) + (0\times 0.3) + (0\times -0.1) + (0\times 0.3) + (3\times -1.1) + (1\times 1) + 0$

$\Rightarrow y_{11} = -3.3 + 1 = -2.3$

$y_{12} = 0 + (3\times 0.3) + (1\times -1.1) + (9\times 1) + (3\times 0.3) = 9.7$

$y_{13} = 0 + (1\times 0.3) + (9\times -1.1) + (2\times 1) + 0 + (3\times 1.1) + (3\times 0.3)$

$\Rightarrow y_{13} = -3.4$

$y_{14} = 0 + (9\times 0.3) + (2\times -1.1) + 0 + 0 + (3\times 1.1) + (3\times 0.3)$

$\Rightarrow y_{14} = 4.7$

$y_{21} = (0\times0.2)+(3\times0.3)+(1\times-0.1)+0+0+0+(5\times0.3)$

$\Rightarrow y_{21} = 1.7$

$y_{22} = (3\times0.2)+(1\times0.3)+(9\times-0.1)+0+0+(5\times1)+0+(5\times1.1)$
$+(1\times0.3)$

$\Rightarrow y_{22} = 6.6$

$y_{23} = (1\times0.2)+(9\times0.3)+(2\times-0.1)+0+0+(3\times1)+(3\times-1)$
$+(1\times1.1)+(1\times0.3) \Rightarrow y_{23} = 4.1$

$y_{24} = (9\times0.2)+(2\times0.3)+0+(3\times0.3)+(3\times-1.1)+0+$
$(1\times-1)+(1\times1.1)+0 \Rightarrow y_{24} = 0.1$

$y_{31} = 0+0+0+(3\times1)+0+(8\times1.1)+(2\times0.3) \Rightarrow y_{31} = 12.4$

$y_{32} = (3\times-0.1)+0+(3\times-1.1)+(1\times1)+(8\times-1)+(2\times1.1)+$
$(8\times0.3) \Rightarrow y_{32} = -6$

$y_{33} = 0+(3\times0.3)+(3\times-0.1)+(3\times0.3)+(1\times-1.1)+(1\times1)+$
$(2\times-1)+(8\times1.1)+(7\times0.3) \Rightarrow y_{33} = 10.3$

$y_{34} = (3\times0.2)+(3\times0.3)+0+(1\times0.3)+(1\times-1.1)+0+(8\times-1)$
$+(7\times1.1)+0 \Rightarrow y_{34} = 0.4$

$y_{41} = 0+0+(3\times-0.1)+0+0(8\times-1.1)+(2\times1)+0$

$\Rightarrow y_{41} = -7.1$

$$y_{42} = 0 + (3\times 0.3) + (1\times -0.1) + (8\times 0.3) + (2\times -1.1) + (8\times 1)$$

$$+ 0 \Rightarrow \boxed{y_{42} = 9}$$

$$y_{43} = (3\times 0.2) + (1\times 0.3) + (1\times -0.1) + (2\times 0.3) + (8\times -1.1)$$

$$+ (7\times 1) \Rightarrow \boxed{y_{43} = -4.4}$$

$$y_{44} = (1\times 0.2) + (1\times 0.3) + 0 + (8\times 0.3) + (7\times -1.1) + 0 + 0$$

$$\Rightarrow \boxed{y_{44} = -4.8}$$

$$\Longrightarrow \tilde{X} = \begin{bmatrix} -2.3 & 9.7 & -3.4 & 4.7 \\ 1.7 & 6.6 & 4.1 & 0.1 \\ 12.4 & -6 & 10.3 & 0.4 \\ -7.1 & 9 & -4.4 & -4.8 \end{bmatrix}$$

$$\xrightarrow[\max(C,0)]{\text{Relu}} \begin{bmatrix} 0 & 9.7 & 0 & 4.7 \\ 1.7 & 6.6 & 4.1 & 0.1 \\ 12.4 & 0 & 10.3 & 0.4 \\ 0 & 9 & 0 & 0 \end{bmatrix}$$

$$\xrightarrow[\text{pooling}]{\text{max}} \begin{bmatrix} 9.7 & 4.7 \\ 12.4 & 10.3 \end{bmatrix}$$

$$L_1 = |9.7 - 1| + |4.7 - 2| + |12.4 - 3| + |10.3 - 1|$$

$$\boxed{L_1 = 21.3}$$

II.

ii) Compute the analytical gradient for the (2D) max-pool layer. Hint: You already calculated the gradient for a ReLU back in exercise 2 :

$$\frac{\partial \text{ReLU}(\mathbf{x})}{\partial \mathbf{x}} = \begin{cases} 1 & \mathbf{x} > 0 \\ 0 & \mathbf{x} \leq 0 \end{cases}$$

The idea here is similar - for some entries the gradient will be backpropagated, for some it won't.

The gradient for a max function with respect to a single x_i will be

$$\frac{\partial \max(\mathbf{x})}{\partial x_i} = \begin{cases} 1 & x_i = \max(\mathbf{x}) \\ 0 & otherwise \end{cases}$$

We can see that the gradient will only be backpropagated through maximum values; for non-maximum values, the gradient is zero.

Max-pooling applies the max operations to several patches in the input. We split the input $\mathbf{x}$ into N patches $\mathbf{z}_k$. Then, the gradient for a single patch $\mathbf{z}_k$ wrt. to an input x_i is

$$\frac{\partial \max(\mathbf{z}_k)}{\partial x_i} = \begin{cases} 1 & x_i = \max(\mathbf{z}_k) \\ 0 & otherwise \end{cases}$$

We sum up the gradients for overlapping patches

$$\frac{\partial \text{max\_pool}(\mathbf{x})}{\partial x_i} = \sum_{\Sigma_{k \in \mathcal{R}(x_i)}} \frac{\partial \max(\mathbf{z}_k)}{\partial x_i}$$

where $\mathcal{R}(x_i)$ indexes all patches for which x_i is in the receptive field. This is necessary, as a single x_i can be the maximum of several patches, e.g. for a max pool operation with a kernel size 3 and stride 1.

III.

iii) Derive the gradient for a 1D convolutional layer

where the input is $\mathbf{x} = [x_1, x_2, x_3, x_4, x_5]^\top$, the weight is $\mathbf{w} = [w_1, w_2]^\top$, and the output is $\mathbf{y} = \text{conv}(\mathbf{x}, \mathbf{w})$. Derive the gradient for the filter weights $\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$.

Q1) iii) $X = [x_1, x_2, x_3, x_4, x_5]^T$, $W = [w_1, w_2]$

$\Rightarrow y = \mathrm{Conv}(X, W)$

$\Rightarrow y_1 = x_1 w_1 + x_2 w_2$, $y_2 = x_2 w_1 + x_3 w_2$

$y_3 = x_3 w_1 + x_4 w_2$, $y_4 = x_4 w_1 + x_4 w_2$

$y_5 = x_4 w_1 + x_5 w_2$

Gradient calculation: $\frac{\partial y_i}{\partial w_1} = x_i$, $\frac{\partial y_i}{\partial w_2} = x_{i+1}$

$\frac{\partial y_1}{\partial w_1} = x_1$, $\frac{\partial y_1}{\partial w_2} = x_2$

$\frac{\partial y_2}{\partial w_1} = x_2$, $\frac{\partial y_2}{\partial w_2} = x_3$

$\frac{\partial y_3}{\partial w_1} = x_3$, $\frac{\partial y_3}{\partial w_2} = x_4$

$\frac{\partial y_4}{\partial w_1} = x_4$, $\frac{\partial y_4}{\partial w_2} = x_4$

$\frac{\partial y_5}{\partial w_1} = x_4$ $\frac{\partial y_5}{\partial w_2} = x_5$

$$\Rightarrow \frac{\partial y}{\partial w} = \begin{bmatrix} x_1 & x_2 \\ x_2 & x_3 \\ x_3 & x_4 \\ x_4 & x_5 \end{bmatrix}$$

IV.

iv) $\frac{\partial y}{\partial w}$ with an upstream gradient $\frac{\partial L}{\partial y}$ to calculate $\frac{\partial L}{\partial w}$:

Perform matrix multiplication:

$$\frac{\partial L}{\partial w} = \frac{\partial y}{\partial w}^T \cdot \frac{\partial L}{\partial y}$$

if $\frac{\partial L}{\partial y} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ \delta_4 \end{bmatrix}$

$$\Longrightarrow \frac{\partial L}{\partial w} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_2 & x_3 & x_4 & x_5 \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ \delta_4 \end{bmatrix} = \begin{bmatrix} x_1\delta_1 + x_2\delta_2 + x_3\delta_3 + x_4\delta_4 \\ x_2\delta_1 + x_3\delta_2 + x_4\delta_3 + x_5\delta_4 \end{bmatrix}$$

*The result can be interpreted as a convolution between the flipped input x and the upstream gradient $\frac{\partial L}{\partial y}$, showing that the backward pass is also a convolution.

b) CNN Arithmetic

b) CNN Arithmetics: We aim to calculate several properties for different configurations of 2D convolutional and max-pooling layers. For the layers, we use the same notation as PyTorch - input channels C_{in}, output channels C_{out} , kernel size K, stride S, and padding P. We use square kernels and equal stride/padding in each dimension; hence, we only need to specify scalars.

To avoid clutter, we use the following notation:

$\text{Conv}(C_{\text{in}} = 3, C_{\text{out}} = 16, K = 5, S = 2, P = 1) \equiv \text{Conv}(3{,}16{,}5{,}2{,}1)$

$\text{MaxPool}\ (K = 2, S = 1, P = 0) \equiv \text{MaxPool}\ (2{,}1{,}0)$

$\text{FC}(C_{\text{in}} = 4096, C_{\text{out}} = 1000) \equiv \text{Linear}(4096{,}1000)$

ReLU, Tanh, LeakyReLU have no arguments and are elementwise operations.

I.

i) The receptive field is defined as the region of all pixels in the input that produces a feature in the feature map f_k. We define the receptive field size $\mathcal{R}_k$ as the width of this input region (in this exercise width and hight of the receptive field are equal). Express the receptive field size $\mathcal{R}_k$ of a convolutional or pooling layer as function of stride S, kernel size K, and the receptive field size of the previous layer $\mathcal{R}_{k-1}$. Let $\mathcal{R}_0 = 1$. Then, $\mathcal{R}_1$ corresponds to the receptive field size of the first layer.

Hint: Sketch a 1D grid and apply several consecutive

۲

1D convolutions with $K = \{2,3\}$ and $S = \{1,2\}$. For each feature, answer how many other features influence its value. Based on these examples, you can see a recursive formula emerge.

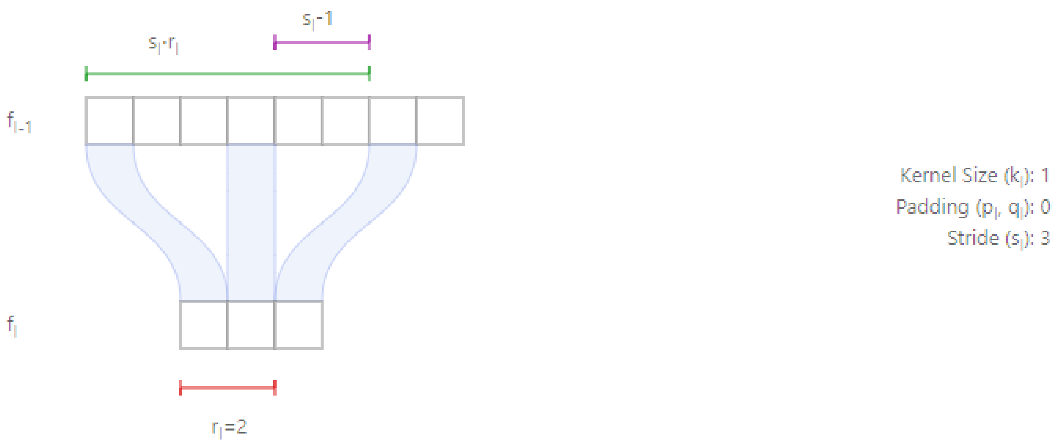
Define r_l as the receptive field size of the final output feature map f_L, with respect to feature map f_l. In other words, r_l corresponds to the number of features in feature map f_l which contribute to generate one feature in f_L. Note that $r_L = 1$.

As a simple example, consider layer L, which takes features f_{L-1} as input, and generates f_L as output. Here is an illustration:

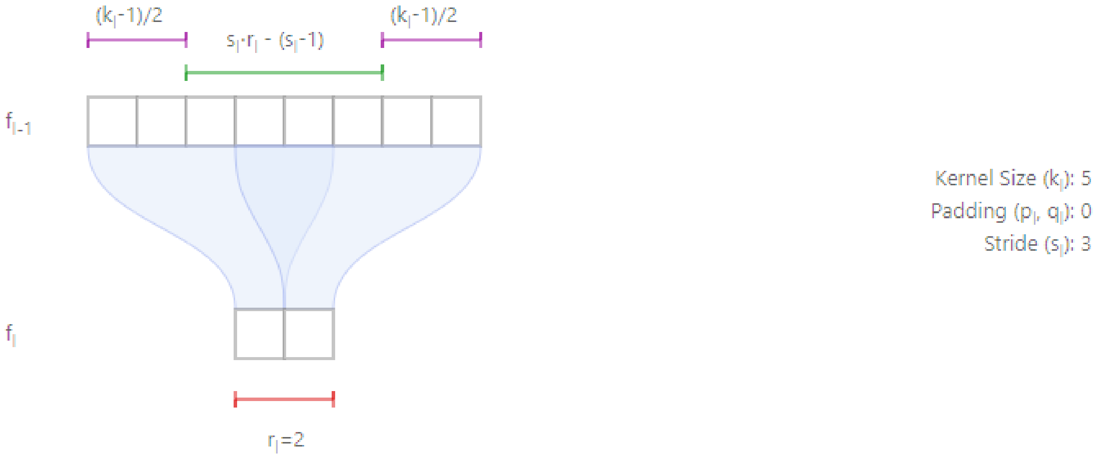
It is easy to see that k_L features from f_{L-1} can influence one feature from f_L, since each feature from f_L is directly connected to k_L features from f_{L-1}. So, $r_{L-1} = k_L$.

Now, consider the more general case where we know r_l and want to compute r_{l-1}. Each feature f_l is connected to k_l features from f_{l-1}.

First, consider the situation where $k_l = 1$: in this case, the r_l features in f_l will cover $r_{l-1} = s_l \cdot r_l - (s_l - 1)$ features in in f_{l-1}. This is illustrated in the figure below, where $r_l = 2$ (highlighted in red). The first term $s_l \cdot r_l$ (green) covers the entire region where the features come from, but it will cover $s_l - 1$ too many features (purple), which is why it needs to be deducted. [3]



For the case where $k_l > 1$, we just need to add $k_l - 1$ features, which will cover those from the left and the right of the region. For example, if we use a kernel size of 5 ($k_l = 5$), there would be 2 extra features used on each side, adding 4 in total. If k_l is even, this works as well, since the left and right padding will add to $k_l - 1$. [4]



So, we obtain the general recurrence equation (which is first-order, non-homogeneous, with variable coefficients):

$$r_{l-1} = s_l \cdot r_l + (k_l - s_l) \tag{1}$$

This equation can be used in a recursive algorithm to compute the receptive field size of the network, r_0. However, we can do even better: we can solve the recurrence equation and obtain a solution in terms of the k_l's and s_l's:

$$r_0 = \sum_{l=1}^{L} \left((k_l - 1) \prod_{i=1}^{l-1} s_i \right) + 1 \tag{2}$$

This expression makes intuitive sense, which can be seen by considering some special cases. For example, if all kernels are of size 1, naturally the receptive field is also of size 1. If all strides are 1, then the receptive field will simply be the sum of $(k_l - 1)$ over all layers, plus 1, which is simple to see. If the stride is greater than 1 for a particular layer, the region increases proportionally for all layers below that one. Finally, note that padding does not need to be taken into account for this derivation.

Q2)

i) In more general way to calculate receptive field, we can also see:

$$n_{out} = \frac{n_{i-1} + 2P - K}{S} + 1$$

where n_{out} is the number of output features for the layer i, and n_{i-1} is the number of input feature of layer i.

Then we need to calculate the jump, the jump represents the cumulative stride, we also have:

$$j_i = j_{i-1} \times S$$

where j_{i-1} is the jump of previous layer.

finally, the receptive field in forward calculation in arithmetic definition is calculated by:

$$r_i = r_{i-1} + (k-1) j_{i-1}$$

$$\Rightarrow r_i = r_{i-1} + (k-1) \prod_{k=1}^{i-1} S_k$$

II.

ii) Calculate the receptive field size $\mathcal{R}_k$ at each layer for the following architectures:

1) Conv (32,128,3,1,1) - Relu - Conv (128,128,4,4,1).

2) (Conv (3,3,5,2,2) - Relu) $*$ 6 (The same block 6 times).

3) Conv (3,64,3,2,1) - Relu - MaxPool(4,3,0) - Conv (64,128,3,1,1) - Relu - MaxPool(2, 2,0).

Q2)

ii) from last part formula we have:

$$r_i = r_{i-1} + (k-1) S_{k-1}$$

=> Now we calculate all three parts:

1) Arch 1 → Conv(32,128,3,1,1)-relu-Conv(128,128,4,4,1)

=> First convolution layer, assumming pixel 1×1

$R_0 = 1 \Rightarrow R_1 = 1 + \underbrace{(3-1)}_{\text{kernel}} \times \underbrace{1}_{\text{stride}} = 3$, $k=3$, $S=1$

the relu doesnt effect the size, so, the second convolutional layer:

$\Rightarrow R_1 = 3 \Rightarrow R_2 = 3 + (4-1) \times 1 = 6$, $k=4$, $S=1$

---

2) Arch 2 → (Conv(3,3,5,2,2)-relu)x6

$R_0 = 1 \Rightarrow R_1 = 1 + (5-1) \times 1 = 5$, $k=5$, $S=2$

for block 2 to 6:

for block 2 to 6:

$R_k = R_{k-1} + (5-1) \cdot 2^{k-1}$

$\Rightarrow \begin{cases} R_2 = 5+(5-1)\times 2 = 13 \\ R_3 = 13+(5-1)\cdot 2^2 = 29 \\ R_4 = 29+(5-1)\cdot 2^3 = 61 \\ R_5 = 61+(5-1)\cdot 2^4 = 125 \\ R_6 = 125+(5-1)\cdot 2^5 = 253 \end{cases}$

---

3) Arch3 → Conv(3,64,3,2,1) – ReLU – maxPool(4,3,0) –

– Conv(64,128,3,1,1) – relu – maxPool(2,2,0)

$R_0 = 1 \rightarrow R_1 = 1+(3-1)\cdot 2 = 5$

ReLU does not affect the receptive field size.

⇒ MaxPooling with $k=2$, $S=2$

$\Rightarrow R_1 = 5+(2-1)\times 2 = 7$

⇒ Conv, $k=3$, $S=1$, $P=1$

$\Rightarrow R_2 = 7+(3-1)\times 1 = 9$

Again relu doesn't affect the receptive size,

maxPool(2,2,0), $k=2$, $S=2$

$R_2 = 9+(2-1)\times 2 = 13$

III.

iii) We pass an image of size $C = 3, W = 512, H = 512$ through the network below. Fill in values of question marks below so that the architecture is valid. Then report the tensor size (C_{out}, H, W) after each layer when the image passes through the net:

Conv(?,?,3,1,1) → ReLU → MaxPool (2,2) →

Conv (64,?,3,1,1) → ReLU → MaxPool (2,2) →

Conv (128,256,3,1,1) → ReLU → MaxPool (2,2) →

Conv(?,512,3,1,1) → ReLU → MaxPool (2,2)

Q2)

iii) $C=3$, $W=512$, $H=512$

image size $=3$, $W=512$

1) conv(?, ?, 3, 1, 1) → ReLU → max pool(2, 2)

→ the input has three channels, and since the next convolution layer has 64 input size, so, we have ⟹ conv(3, 64, 3, 1, 1)

it can be implied that:

$C_{out}=64$, and W_{out} & H_{out} can be calculated with: $W_{out}=H_{out}=\left\lceil \frac{W_{in}+2P-k}{S} \right\rceil+1$

considering $k=3$, $S=1$, $P=1$

$$\Longrightarrow W_{out}=H_{out}=\left\lceil \frac{512+2\times1-3}{1} \right\rceil+1=512$$

$\Longrightarrow$ So the result of this convolution is:

$C=3$, $C_{out}=64$, $W_{out}=H_{out}=512$

After maxpooling with $s=2$, $k=2$, $p=0$

$\Longrightarrow H_{out}=512/2=256$, $W_{out}=512/2=256$

the relu is element wise & does not change the sized, Tensor size = (64, 256, 256)

---

2) Conv(64, ?, 3, 1, 1) $\rightarrow$ Relu $\rightarrow$ maxpool(2, 2)

$\rightarrow$ the output will be the next convolution layer, so it is: $C_{out}=128$

with $k=3$, $S=1$, $P=1 \Longrightarrow H_{out}=W_{out}=256$ (formula)

relu does not affect the size

for maxpooling we have: $S=2$, $k=2$, $p=0$

$\Longrightarrow H_{out}=W_{out}=256/2=128$

Tensor size: (128, 128, 128)

---

3) conv (128, 256, 3, 1, 1) → ReLU → maxpool (2, 2)

$C_{out} = 256$

$k=3,\ P=1,\ S=1 \Rightarrow H_{out} = W_{out} = 128$

maxpooling → $k=2,\ S=2,\ P=0$ → $H_{out} = W_{out} = 64$

Tensor size = (256, 64, 64)

---

4) Conv (?, 512, 3, 1, 1) → ReLU → maxPool (2, 2)

$C_{in} = 256$ → input channel due to last layer

$C_{out} = 512$

$k=3,\ P=1,\ S=1 \Rightarrow W_{out} = C_{out} = 64$

maxpooling ⇒ $k=2,\ S=2,\ P=0 \Rightarrow W_{out} = C_{out} = 32$

Tensor size = (512, 32, 32)

IV.

iv) The network architecture above is the (simplified) convolutional part of a network called VGG16 by (Simonyan *et al.*, 2014). As mentioned in the lecture, this stack of convolutional and max-pooling layers is often followed by a few fully-connected layers. The first fully-connected layer of a VGG16 is a `FC(25088, 4096)`. Calculate the number of trainable parameters for 1st and 2nd convolutional layers and the first fully-connected layer.

(Q2)

iv)

1) first convoloutinal layer → Conv(3, 64, 3, 1, 1)

⟹ Paremeters: $(k^2 \times C_{in}+1)C_{out} = (3\times3\times3+1)\times64 = 1792$

---

2) Second Conoloutinal layer → Conv(64, 128, 3, 1, 1)

⟹ Paremeters: $(3\times3\times64+1)\times128 = 73586$

First fully Connected layer.(FC(25088, 4096)x

Parmeters numbers:

⟹ Paremeters: $(C_{in}+1)\times C_{out} = (25088+1)\times4096 = 102764544$

2. MNIST classification with CNN

In this part a CNN has been implemented for MNIST data, loss values and accuracies will be reported by chaning hyperparameters of CNN:

a) Calculate the normalization constants for MNIST

Here is the output for calculated the normalization constants:

```
Min Pixel Value: 0
Max Pixel Value: 255
Mean Pixel Value 33.31842041015625
Pixel Values Std: 78.56748962402344
Scaled Mean Pixel Value 0.13066047430038452
Scaled Pixel Values Std: 0.30810779333114624
```

b) Build a Pytorch dataloader

The code lines for the dataloader considering normalization constants and reducing the size to 14*14:

```
# b) Build a Pytorch dataloader
transform = Compose([Resize((14, 14)), ToTensor(),
Normalize((train_mean,), (train_std,))])

train_dataset = MNIST(root='./data', train=True, download=True,
transform=transform)
test_dataset = MNIST(root='./data', train=False, download=True,
transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

c) Implement the CNN class

Implementing CNN class with determined hyperparameters in the question:

```
# c) Implement the CNN class
```

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, 1, 0)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 32, 3, 1, 0)
        self.relu2 = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2, 0)
        self.dropout1 = nn.Dropout(0.25)
        self.fc1 = nn.Linear(800, 128)
        self.relu3 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 10)
        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool(x)
        x = self.dropout1(x)
        x = x.view(-1, 800)
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        x = self.log_softmax(x)
        return x
```

d) Implement the training loop

Here is the code lines for training loop:

```
# d) Implement the training loop
model = CNN()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
train_losses = []
train_loss_epochs=[]
num_epochs = 150
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
```

```
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        train_losses.append(loss.item())
    train_loss_epochs.append(running_loss / len(train_loader))
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(train_loader)}')
```

e) Reach a test accuracy of > 99% on MNIST with CNN

The loss values over iterations with mentioned hyperparameters in the question has been shown in Fig (1) and over epochs in Fig (2).

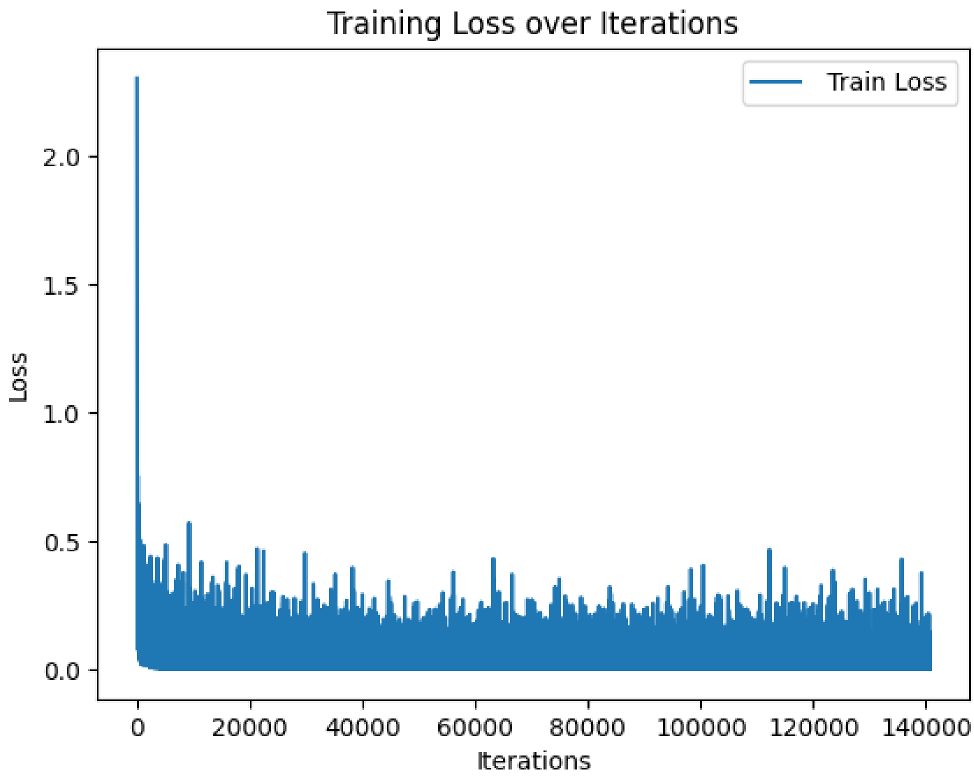


Figure 1. Loss values over iterations

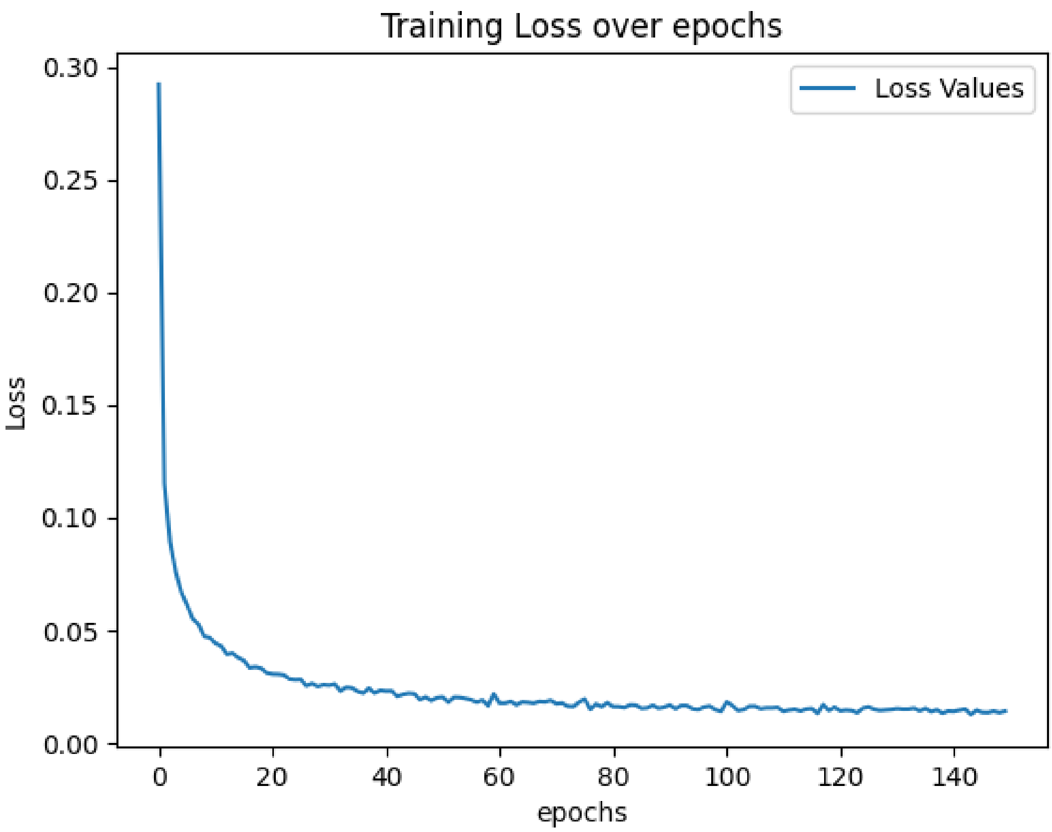


Figure 2. Loss values over epochs

The output of test accuracy after 150 epochs for downscaled data has been obtained as follows:

```
Accuracy of the network on the 10000 test images: 98.8%
```