HW2

2021320071_허천윤

[ ]:
```
#7.1 From Fully Connected Layers to Convolutions
# Sample code
import numpy as np

# Example input for fully connected layer
input_data = np.random.rand(1, 784)  # Flattened 28x28 image

# Weights for the fully connected layer
weights = np.random.rand(784, 128)

# Compute output
output = np.dot(input_data, weights)
print("Output from Fully Connected Layer: ", output)

# Discussion Points:

# The computational complexity of fully connected layers grows linearly with
 ↪the input feature dimensions.
# Fully connected layers can lead to overfitting in image processing due to a
 ↪high number of parameters.
```

[1]:
```
#7.2 Convolutions for Images
import torch
import torch.nn.functional as F

# Define input image and convolution kernel
input_image = torch.rand(1, 1, 28, 28)  # 1 image, 1 channel, 28x28 pixels
kernel = torch.rand(1, 1, 3, 3)  # 3x3 convolution kernel

# Perform convolution operation
output_image = F.conv2d(input_image, kernel)
print("Output from Convolution:", output_image.shape)

# Discussion Points:

# Convolution effectively extracts features from images.
# By learning convolution kernels, models can automatically extract
 useful ↪features.
```

```
Output from Convolution: torch.Size([1, 1, 26, 26])
```

[2]:
```
# 7.3 Padding and Stride
# Convolution with padding and stride output_image_padded =
F.conv2d(input_image, kernel, padding=1, stride=2)
print("Output with Padding and Stride:",
output_image_padded.shape)


# Discussion Points:

# Padding controls the spatial dimensions of the output.
# Stride affects how the convolution kernel moves over the input
data,␣
↪influencing feature map size.
```

```
Output with Padding and Stride: torch.Size([1, 1, 14, 14])
```

[8]:
```
# 7.4 Multiple Input and Output Channels

import torch
import torch.nn.functional as F

# Define a random input tensor with 3 channels
input_image_multi = torch.rand(1, 3, 28, 28)  # Batch size 1, 3 channels, 28x28 ␣
↪pixels

# Define a kernel with 1 output channel and 3 input channels, kernel size 3x3
kernel_multi = torch.rand(1, 3, 3, 3)  # 1 output channel, 3 input channels, ␣
↪3x3 kernel size

# Perform convolution operation
output_image_multi = F.conv2d(input_image_multi, kernel_multi)
print("Output with Multiple Channels:", output_image_multi.shape)


# Discussion Points:

# Multi-channel input processes color images, enhancing the model's expressive ␣
↪power.
# Each channel's convolution kernel learns features independently.
```

```
Output with Multiple Channels: torch.Size([1, 1, 26, 26])
```

[7]:
```
# 7.5 Pooling

# Pooling example
pool = F.max_pool2d(input_image, kernel_size=2)
print("Output after Max Pooling:", pool.shape)
```

```
# Discussion Points:

# Pooling reduces dimensionality, lowering computational complexity
  and risk of overfitting.
# Max pooling and average pooling have distinct characteristics; max
  pooling is commonly preferred.
```

```
Output after Max Pooling: torch.Size([1, 1, 14, 14])
```

[4]:
```
# 7.6 Convolutional Neural Networks (LeNet)
import torch.nn as nn

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate and print model
model = LeNet()
print(model)


# Discussion Points:

# LeNet is one of the earliest convolutional neural networks, suitable for
  handwritten digit recognition.
# The structure contains convolutional layers, pooling layers, and fully
  connected layers.
```

```
LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=120, bias=True)
```

```
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

[5]:

```python
# 8.2 Networks Using Blocks (VGG)
class VGG(nn.Module):
    def __init__(self):
        super(VGG, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # More layers...
        )

    def forward(self, x):
        x = self.conv_layers(x)
        return x

# Instantiate and print model
vgg_model = VGG()
print(vgg_model)



# Discussion Points:

# VGG has a simple structure with a deeper network, increasing depth using
 ↪small convolutional kernels.
# It provides a more powerful feature extraction capability.
```

```
VGG(
  (conv_layers): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1),
    padding=(1, 1))
    (1): ReLU()
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
                                                                 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
)
```

[6]:

```python
# 8.6 Residual Networks (ResNet)

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
 ↪padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
 ↪padding=1)

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.relu(out)
        out = self.conv2(out)
        out += identity  # Adding the input residual
        out = self.relu(out)
        return out

# Instantiate a residual block and test
res_block = ResidualBlock(64, 64)
print(res_block)


# Discussion Points:

# Residual connections help address the vanishing gradient problem in deep ␣
 ↪networks.
# They enable direct information flow across layers.
```

```
    ResidualBlock(
(conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
                                                                    1))
      (relu): ReLU()
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
    padding=(1, 1)) )
```

[ ]: