PyTorch

Convolutional Neural Network (CNN) – 1D

Syntax:

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0,
dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)
```

Input Size: (N, Cin, L) == (Batch Size, Input Channel, Length of Signal)

Output Size: (N, Cout, L) == (Batch Size, Output Channel, Length of Signal)

Parameters:

- **in_channels** *(int)* – Number of channels in the input image
- **out_channels** *(int)* – Number of channels produced by the convolution
- **kernel_size** *(int or tuple)* – Size of the convolving kernel
- **stride** *(int or tuple, optional)* – Stride of the convolution. Default: 1
- **padding** *(int, tuple or str, optional)* – Padding added to both sides of the input. Default: 0
- **padding_mode** *(str, optional)* – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'
- **dilation** *(int or tuple, optional)* – Spacing between kernel elements. Default: 1
- **groups** *(int, optional)* – Number of blocked connections from input channels to output channels. Default: 1
- **bias** *(bool, optional)* – If True, adds a learnable bias to the output. Default: True

Now I will create a sequential (1D-CNN) including ReLU, MaxPooling, and Fully Connected Layer and I will the operations in each layer with their basics syntax.

Let’s assume we have a tabular dataset of something where we have a total of 10 feature columns and 1 target column (with two classes 0 and 1). The 10 feature columns are numerical values. Since we have a total of 10 feature columns so we can say that our input length or size or vector is 10.

Let’s start coding……..

Since we will pass the input tensor in different layers so we need a dummy tensor. So let's create a sample tensor. As per the documentation, we know that Conv1D takes (batch, channel, input size) as an input of the layer.

Let's create a sample tensor

```
import torch
import torch.nn as nn

# Let's consider in our dataset(Tabular) feature length 10 so we can define it is
input_size = 10
# Create a random tensor
x = torch.randn(2, 1, input_size)
print(x)
print(f"Input Size Shape: {x.shape}")
```

Output:

```
tensor([[[-0.6610,  -1.8503,  -0.0568,  -
0.0567,  0.0766,  0.4129,  1.2354,
          0.7027,  1.2490, -0.2926]],

        [[-0.7635,  1.4789, -0.7392, -0.1308, -
0.8864, -0.1417,  0.2720,
          -0.2650, -0.4053, -0.6898]]])
```

```
torch.Size([2, 1, 10])
```

@First Block (conv1d -> ReLU -> MaxPooling)

```
# Create first conv1d layer
conv1 = nn.Conv1d(in_channels=1, out_channels=8, kernel_size=2)
# Let's pass the input tensor into the first conv1D layer
out_conv1 = conv1(x)
# Now let's check the output shape from conv1
print(out_conv1)
print(f"The shape after conv1 operation: {out_conv1.shape}")
```

Output:

```
tensor([[[-3.3438e-01, -1.3632e+00, -1.0146e+00, -4.0648e-01, -7.2670e-01,
          -6.5476e-01, -5.3044e-01, -8.9682e-01, -1.6148e-01],
         [ 4.5527e-01, -7.2719e-01, -1.5547e+00, -9.4309e-01, -4.7869e-01,
          -8.2807e-01, -6.8479e-01, -6.7923e-01, -8.3759e-01],
         [-1.5267e-01,  4.5183e-01,  1.1121e+00,  8.1635e-01,  4.1812e-01,
           6.8020e-01,  6.0687e-01,  5.2161e-01,  7.9643e-01],
         [ 4.3820e-01,  5.1978e-01, -9.9143e-01, -1.1451e+00, -1.1422e-01,
          -6.4183e-01, -6.5119e-01, -1.0680e-01, -1.3772e+00],
         [ 7.7126e-01, -2.9751e-01, -1.2291e-01,  4.9548e-01,  2.8995e-01,
           2.9871e-01,  4.2791e-01,  1.1247e-01,  7.2308e-01],
         [-8.6832e-03,  5.2976e-01,  1.2033e-02, -3.3007e-01,  6.4759e-02,
          -9.0837e-02, -1.5579e-01,  1.5242e-01, -5.0635e-01],
         [-9.0034e-01, -1.2793e-01,  2.7488e-01, -1.3442e-01, -3.4443e-01,
          -1.6466e-01, -2.5821e-01, -2.1400e-01, -2.2309e-01],
         [-5.7228e-01,  3.4834e-01,  1.4436e-01, -3.9211e-01, -1.7877e-01,
          -2.0515e-01, -3.1643e-01, -2.6104e-02, -5.9585e-01]],

        [[-8.8379e-01, -6.7168e-02, -5.8160e-01, -5.7841e-01, -1.5578e+00,
          -1.2965e+00, -6.6203e-01, -1.3227e+00, -5.6621e-02],
         [-3.9891e-01, -8.2424e-01, -1.0364e-01, -6.9514e-01, -9.1932e-01,
          -1.8155e+00, -1.2781e+00, -8.7891e-01, -1.2199e+00],
         [ 3.2705e-01,  8.0796e-01,  1.8615e-01,  6.0352e-01,  5.4398e-01,
           1.2332e+00,  9.9561e-01,  5.6770e-01,  1.0890e+00],
         [ 2.1268e-01, -1.4995e+00,  1.2588e-01, -5.9331e-01,  5.7319e-01,
          -8.9280e-01, -1.1748e+00,  2.7704e-01, -1.9929e+00],
         [ 1.6653e-01,  8.0654e-01,  4.7249e-01,  3.8491e-01, -4.9478e-01,
          -4.0604e-01,  2.2370e-01, -2.8569e-01,  7.5523e-01],
         [ 2.1803e-01, -5.8165e-01,  4.5683e-02, -1.1846e-01,  6.4015e-01,
           1.7677e-01, -2.0761e-01,  4.5442e-01, -6.9849e-01],
         [-3.6735e-01, -2.4247e-01, -5.6606e-01, -2.4642e-01,  1.1015e-03,
           4.5230e-01,  7.9790e-02, -5.1060e-02, -2.9753e-02],
         [-6.1097e-02, -6.7189e-01, -3.2691e-01, -2.7744e-01,  5.1964e-01,
           3.9100e-01, -1.5982e-01,  3.2951e-01, -6.4548e-01]]]
```

The shapes after the conv1 operation:

```
torch.Size([2, 8, 9])
```

Let’s print the weight matrix and bias vector generated by conv1d:

```
print(conv1.weight)
print(conv1.weight.shape)
```

Weight:	Weight Shape:

tensor([[[0.4428, -0.2142]],
 	
        [[-0.1202, 0.4498]],
 	
        [[0.1115, 0.4142]],
 	
        [[0.2567, -0.2503]],
 	
        [[-0.1601, -0.3884]],
 	
        [[-0.3017, 0.0808]],
 	
        [[-0.5680, 0.6448]],
 	
        [[-0.2064, -0.5240]]], requires_grad=True)
	torch.Size([8, 1, 2])

```
print(conv1.bias)
print(conv1.bias.shape)
```

Bias Vector:	Bias Shape:
tensor([0.4137, 0.0427, 0.1060, 0.0064, 0.0981, -0.4472, 0.1271, 0.2429], requires_grad=True)	torch.Size([8])

```
# Create the first ReLU layer
relu1 = nn.ReLU()
# Pass output result from conv1 into the ReLU function
out_relu1 = relu1(out_conv1)
print(out_relu1)
# Now let's check the output shape from relu1
print(f"The shape after relu1 operation: {out_relu1.shape}")
```

Output:

```
tensor([[[0.0682, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.2563, 0.0000, 1.4075, 0.0000, 0.0000, 1.1295, 0.0000, 0.3684,
          0.0000],
         [0.8433, 0.4436, 1.4853, 0.0000, 0.2851, 1.3071, 0.0000, 0.7083,
          0.0000],
         [0.2249, 0.6591, 0.0000, 1.4563, 0.7872, 0.0000, 1.2871, 0.1517,
          1.2422],
         [0.9265, 0.5202, 1.6360, 0.0000, 0.3648, 1.4438, 0.0000, 0.8182,
          0.0000],
         [0.0000, 0.0000, 0.3905, 0.0000, 0.0000, 0.2665, 0.0000, 0.0000,
          0.0000],
         [0.0931, 0.0116, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.1587, 0.0000, 0.5252, 0.2249, 0.0000, 0.4477, 0.0000,
          0.5222]],

        [[0.0000, 0.1025, 0.0000, 0.0000, 0.0000, 0.1007, 0.0000, 0.0000,
          0.0000],
         [0.0000, 1.2946, 0.5470, 0.0824, 0.0000, 0.2233, 0.8936, 0.0000,
          0.0000],
         [0.0000, 1.6017, 0.3830, 0.0000, 0.0000, 0.8474, 1.1101, 0.0000,
          0.0000],
         [1.5351, 0.0000, 0.0554, 0.5549, 1.7774, 0.2531, 0.0000, 0.7394,
          1.3959],
         [0.0000, 1.7274, 0.5467, 0.0491, 0.0000, 0.9264, 1.2396, 0.1097,
          0.0000],
         [0.0000, 0.4780, 0.0000, 0.0000, 0.0000, 0.0000, 0.1279, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2024, 0.0000, 0.0000,
          0.0000],
         [0.7072, 0.0000, 0.0000, 0.0000, 0.7995, 0.0000, 0.0000, 0.0616,
          0.6223]]]
```

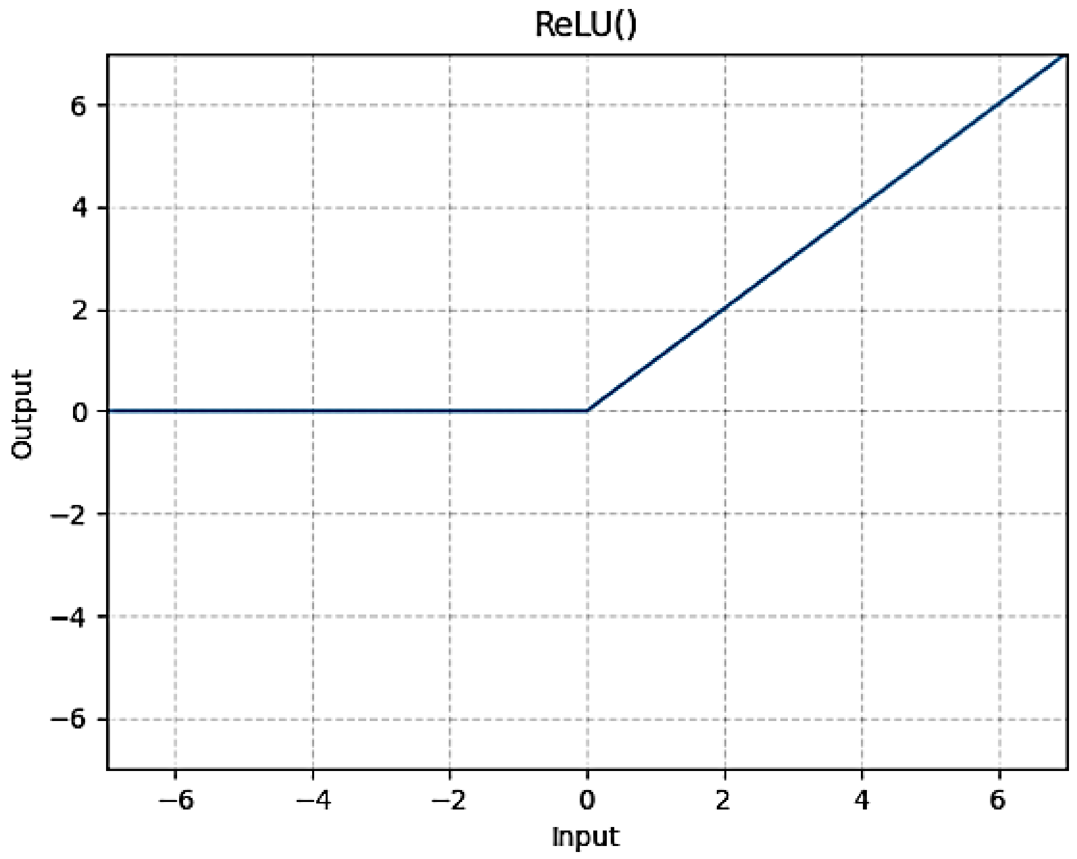
Output Shape from ReLU Activation Function:

```
Torch.Size([2, 8, 9])
```

RELU:

Applies the rectified linear unit function element-wise. ReLU stands for Rectified Linear Unit, and it is an activation function commonly used in neural networks. The ReLU activation function introduces non-linearity to the network by outputting the input directly if it is positive, and 0 otherwise. It helps to alleviate the vanishing gradient problem and can improve the network's ability to learn complex patterns and make nonlinear transformations in the data.

ReLU(x) = max(0, x)



MAXPOOL1D

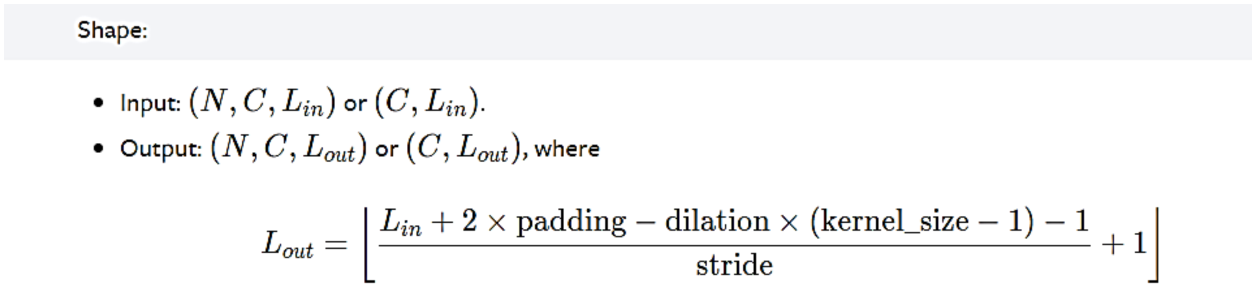
MaxPool1d is a downsampling operation commonly used in convolutional neural networks (CNNs). It reduces the spatial dimensions (width) of the input tensor while retaining the most important features. MaxPool1d operates on a 1-dimensional input tensor and applies a sliding window of a specified size (kernel size) to the input. In each window, it selects the maximum value and outputs it as the pooled value. The sliding window moves with a specified stride, determining the amount of overlap between consecutive windows. MaxPool1d helps in reducing the spatial dimensions of the input, which can be beneficial for reducing computational complexity and extracting the most salient features. By selecting the maximum value within each window, MaxPool1d helps in capturing the most prominent features and promoting translation invariance.

Syntax:

torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)

Applies a 1D max pooling over an input signal composed of several input planes.

- **kernel_size** (*Union[int, Tuple[int]]*) – The size of the sliding window, must be > 0.
- **stride** (*Union[int, Tuple[int]]*) – The stride of the sliding window, must be > 0. The default value is kernel_size.
- **padding** (*Union[int, Tuple[int]]*) – Implicit negative infinity padding to be added on both sides, must be >= 0 and <= kernel_size / 2.
- **dilation** (*Union[int, Tuple[int]]*) – The stride between elements within a sliding window, must be > 0.
- **return_indices** (*bool*) – If True, will return the argmax along with the max values. Useful for torch.nn.MaxUnpool1d later
- **ceil_mode** (*bool*) – If True, will use *ceil* instead of *floor* to compute the output shape. This ensures that every element in the input tensor is covered by a sliding window.


Shape:

- Input: (N, C, L_{in}) or (C, L_{in}).
- Output: (N, C, L_{out}) or (C, L_{out}), where

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

In our code, we also applied MaxPoo1D. Let’s illustrate the output from the MaxPooling layer below

```
# Create first MaxPool layer
pool1 = nn.MaxPool1d(kernel_size=2)
# Pass output result from relu1 into the pool1 function
out_pool1 = pool1(out_relu1)
print(out_pool1)
# Now let's check the output shape from relu1
print(f"The shape after pool1 operation: {out_pool1.shape}")
```

Output:

Output from the MaxPooling Layer:	Output Shape:
tensor([[[0.9669, 0.4090, 0.5684, 0.0691], [1.2436, 0.1224, 0.9627, 0.2316], [0.0000, 0.8166, 0.0000, 0.7562], [0.4214, 1.4040, 0.2128, 1.3628], [0.0000, 0.0000, 0.0000, 0.0000], [0.0726, 0.2576, 0.0000, 0.2702], [0.3871, 0.4063, 0.0838, 0.2945], [0.0654, 0.0000, 0.0872, 0.0000]], [[0.5583, 0.7033, 0.2578, 0.1887], [0.0587, 0.6684, 0.1999, 0.1959], [0.7350, 0.0567, 0.6016, 0.0000], [1.3004, 0.5526, 1.1866, 0.4413], [0.0000, 0.0000, 0.0000, 0.0000], [0.1786, 0.0000, 0.1609, 0.0000], [0.4160, 0.1441, 0.2591, 0.0703], [0.0000, 0.0000, 0.0000, 0.0000]]]	torch.Size([2, 8, 4])

@Second Block (conv1d -> ReLU -> MaxPooling)

In same way I will create another Conv1D, ReLU, and MaxPooling Layer. Then I will create Fully Connected Layer or Dense Layer.

```
# Create second conv1d layer
conv2 = nn.Conv1d(in_channels=8, out_channels=16, kernel_size=2)
# Let's pass output tensor from last layer (MaxPool) into the second conv1D layer
out_conv2 = conv2(out_pool1)
# Now let's check the output shape from conv2
print(f"The shape after conv2 operation: {out_conv2.shape}")

# Create second ReLU layer
relu2 = nn.ReLU()
# Pass output result from conv2 into the ReLU function
out_relu2 = relu2(out_conv2)
# Now let's check the output shape from relu1
print(f"The shape after relu2 operation: {out_relu2.shape}")

# Create second MaxPool layer
pool2 = nn.MaxPool1d(kernel_size=2)
# Pass output result from relu2 into the pool2 function
out_pool2 = pool2(out_relu2)
# Now let's check the output shape from relu1
print(f"The shape after pool2 operation: {out_pool2.shape}")
```

Output Shape from Each Layer:

```
The shape after conv2 operation: torch.Size([2, 16, 3])

The shape after relu2 operation: torch.Size([2, 16, 3])

The shape after pool2 operation: torch.Size([2, 16, 1])
```

@Third Block (Fully Connected Layer -> Prediction)

Fully Connected Layer or Dense Layer or Flatten Layer:

LINEAR Layer:

In PyTorch, the *nn.Linear* layer is a fully connected (dense) layer that performs a linear transformation on the input data. It is commonly used in neural network architectures for tasks such as classification, regression, and dimensionality reduction. The *nn.Linear* layer takes the input tensor and applies a linear transformation to it using a weight matrix and bias vector. The weight matrix defines the linear mapping from the input features to the output features, and the bias vector allows for an additional offset term.

Mathematically, the linear transformation performed by the **nn.Linear** layer can be expressed as:

```
y = xA^T + b
```

where x is the input tensor of shape ***(batch_size, input_size),*** y is the output tensor of shape ***(batch_size, output_size)***, A is the weight matrix of shape ***(output_size, input_size)***, and b is the bias vector of shape ***(output_size,)***.

The output of the *nn.Linear* layer is computed by multiplying the input tensor with the weight matrix, transposed, and adding the bias vector. This operation allows the layer to learn a linear mapping from the input features to the output features.

Syntax:

torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)

Parameters:

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to False, the layer will not learn an additive bias.
  Default: True

How to calculate in_features and out_features in fully connected layer?

To calculate the in_features and out_features for a fully connected layer, we need to consider the shape of the input tensor and the desired output size. In our case, after the second block of operations, the shape of the tensor is (2, 16, 1). To pass this tensor through a fully connected layer, you need to flatten it into a 1D tensor.

The in_features of the fully connected layer would be the total number of elements in the flattened tensor which is 2 * 16 * 1 = 32.

The out_features of the fully connected layer can be any desired size based on your task.

For example,if you want the output size to be 2, you can set out_features=2

Here's an example of how you can calculate the in_features and out_features and create the fully connected layer

# Calculate in_features and out_features

For Input Feature we can calculate it from the last layer before fully connected layer. For example in our case the last layer was maxpool2 output. So we can calculate by following process

in_features = out_pool2.size(1) * out_pool2.size(2) # 16 * 1 = 16

And the output feature will how many classes we have in our dataset.
out_features =2

# Create the fully connected layer
fc1 = nn.Linear(in_features, out_features)

Let's return back in our code............

```
in_features = out_pool2.size(1) * out_pool2.size(2) # 16*1
# Let's we have 2 class
out_features = 2

# Reshape the input tensor from pool2
reshape_out_pool2 = out_pool2.view(out_pool2.size(0), -1)
print(f"Reshape of output tensor from Pool2: {reshape_out_pool2.shape}")
```

Reshape of output tensor from Pool2: torch.Size([2, 16])

Now let's create our first fully connected layer and print weight matrix, matrix shape, bias vector and vector shape.

```
# Create first fully connected layer
fc1 = nn.Linear(in_features=in_features, out_features=out_features)
print(fc1.weight)
print(f"Shape of weight matrix in the linear layer: {fc1.weight.shape}")
print(fc1.bias)
print(fc1.bias.shape)
```

Output of Weight Matrix:

tensor([[0.1878, -0.2169, -0.0292, 0.1976, 0.2029, -0.1147, 0.1643, 0.1783, 0.1441, 0.2140, 0.0915, -0.1305, 0.1165, -0.1847, -0.1039, 0.2235], [0.2486, 0.2447, -0.1078, -0.1333, -0.2055, -0.1383, 0.2324, -0.2196, 0.0803, 0.0077, 0.1533, 0.1777, -0.0734, 0.1822, -0.0248, -0.0613]], requires_grad=True)	Shape: torch.Size([2, 16]) requires_grad = True, means it will update during traing.

Output of Bias Vector:

tensor([0.1636, -0.1120], requires_grad=True)	Shape: torch.Size([2])

```
# Pass out tensor from the last layer (Pool2) into the fc1
out_fc1 = fc1(reshape_out_pool2)
print(out_fc1)
# Now check output shape from fc1
print(f"Output shape from first fully connected layer fc1: {out_fc1.shape}")
```

Output from Fully Connected Layer

tensor([[0.0076, 0.0698], [0.0599, 0.0285]]	torch.Size([2, 2])

Prediction:

To predict class label we apply probabilistic theory. The softmax function will convert the output values into probabilities representing the likelihood of each class.

```
# Apply softmax function along the appropriate dimension (assuming dimension 1)
probs = nn.functional.softmax(out_fc1, dim=1)

# Get the predicted class labels
_, predicted_labels = torch.max(probs, dim=1)

# Print the predicted labels
print(predicted_labels)
```

Output:

```
tensor([1, 1])
```

Let's go through the calculations step by step:

The logits tensor out_fc1 is

```
tensor([[0.2803, 0.0191],
        [0.2526, 0.1101]], grad_fn=<AddmmBackward0>)
```

We apply the softmax function along dimension 1 to obtain class probabilities. The softmax function normalizes the logits and maps them to a probability distribution:

```
probs = nn.functional.softmax(out_fc1, dim=1)
```

The resulting **probs** tensor is:

```
tensor([[0.9322, 0.0678],
        [0.6981, 0.3019]], grad_fn=<SoftmaxBackward>)
```

The values in each row now sum up to 1, representing the probabilities of each class.

Next, we use **torch.max** to find the class with the highest probability for each sample along dimension 1:

```
_, predicted_labels = torch.max(probs, dim=1)
```

The **predicted_labels** tensor is:

```
tensor([0, 0])
```

This means that for the first sample, the model predicts class 0 with the highest probability, and for the second sample, it also predicts class 0 with the highest probability.

In summary, after applying softmax to the logits, we obtain class probabilities. We then use **torch.max** to find the class with the highest probability as the predicted class label. In this case, both samples are predicted as class 0 with the highest probability.

Summary:

```
input_size = 10
x = torch.randn(2, 1, input_size)
print(f"Input Size Shape: {x.shape}")
conv1 = nn.Conv1d(in_channels=1, out_channels=8, kernel_size=2)
out_conv1 = conv1(x)
print(f"The shape after conv1 operation: {out_conv1.shape}")
relu1 = nn.ReLU()
out_relu1 = relu1(out_conv1)
print(f"The shape after relu1 operation: {out_relu1.shape}")
pool1 = nn.MaxPool1d(kernel_size=2)
out_pool1 = pool1(out_relu1)
print(f"The shape after pool1 operation: {out_pool1.shape}")
# """-----------------------Second Block-----------------------"""
conv2 = nn.Conv1d(in_channels=8, out_channels=16, kernel_size=2)
out_conv2 = conv2(out_pool1)
print(f"The shape after conv2 operation: {out_conv2.shape}")
relu2 = nn.ReLU()
out_relu2 = relu2(out_conv2)
print(f"The shape after relu2 operation: {out_relu2.shape}")
pool2 = nn.MaxPool1d(kernel_size=2)
out_pool2 = pool2(out_relu2)
print(f"The shape after pool2 operation: {out_pool2.shape}")
# """-----------------------Third Block-----------------------"""
in_features = out_pool2.size(1) * out_pool2.size(2) # 16*1
out_features = 2
reshape_out_pool2 = out_pool2.view(out_pool2.size(0), -1)
print(f"Reshape of output tensor from Pool2: {reshape_out_pool2.shape}")
fc1 = nn.Linear(in_features=in_features, out_features=out_features)
out_fc1 = fc1(reshape_out_pool2)
print(f"Output shape from first fully connected layer fc1: {out_fc1.shape}")
```

Output

```
Input Size Shape: torch.Size([2, 1, 10])

The shape after conv1 operation: torch.Size([2, 8, 9])

The shape after relu1 operation: torch.Size([2, 8, 9])

The shape after pool1 operation: torch.Size([2, 8, 4])

The shape after conv2 operation: torch.Size([2, 16, 3])

The shape after relu2 operation: torch.Size([2, 16, 3])

The shape after pool2 operation: torch.Size([2, 16, 1])

Reshape of output tensor from Pool2: torch.Size([2, 16])

Output shape from first fully connected layer fc1: torch.Size([2, 2])
```

Source Links:

PyTorch Official: https://pytorch.org/

Github: https://github.com/Istiak665

Scripted By:

Md. Istiak Ahammed

Graduate Researcher

Kyungpook National University, South Korea

Email: ahammed.istiakbd@gmail.com