CNN 1D—Code Understanding line by line

Code:

```
class CustomCNN_V1(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(CustomCNN_V1, self).__init__()
        self.conv1 = nn.Conv1d(input_dim, 16, kernel_size=3)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool1d(kernel_size=2)
        self.fc1 = nn.Linear(16 * 101, 64)
        self.relu2 = nn.ReLU()
        self.fc2 = nn.Linear(64, output_dim)

    def forward(self, x):
        x = x.unsqueeze(1) # Add channel or batch dimension
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu2(x)
        x = self.fc2(x)
        return x
```

In this example, the CNN model consists of a 1D convolutional layer (nn.Conv1d), a ReLU activation layer (nn.ReLU), a max pooling layer (nn.MaxPool1d), and a fully connected layer (nn.Linear). The input dimension of the convolutional layer (input_dim) should match the number of features in your dataset and the output dimension (output_dim) should match the number of classes you want to predict.

Line by Line:

1. nn.Module is the base class for all neural network modules in PyTorch. By inheriting this class, our CustomCNN_V1 class becomes a PyTorch module.
2. super(CustomCNN_V1, self).__init__() calls the constructor of the parent class nn.Module to initialize the module.
3. self.conv1 = nn.Conv1d(input_dim, 16, kernel_size=3) defines the first convolutional layer. It takes an input with input_dim channels and applies 16 filters, each with a kernel size of 3. The output of this layer will have 16 channels.
4. self.relu1 = nn.ReLU() creates an instance of the ReLU activation function. ReLU (Rectified Linear Unit) is a non-linear activation function that introduces non-linearity to the model by setting negative values to zero.
5. self.pool1 = nn.MaxPool1d(kernel_size=2) defines the max pooling layer. It performs max pooling with a kernel size of 2. Max pooling reduces the spatial dimensions (length) of the input by taking the maximum value within each kernel window.

Spatial dimensions: In the context of image data, spatial dimensions typically represent the width, height, and depth (or channels) of the image. For example, in a 2D grayscale image, the spatial dimensions are typically represented as width and height. Each pixel in the image corresponds to a specific location in these spatial dimensions. The width represents the number of pixels in each row of the image, and the height represents the number of rows or lines in the image.

In the case of a color image or a multi-channel image, there is an additional spatial dimension called the channel dimension. The channel dimension represents different channels or color planes of the image, such as red, green, and blue channels in an RGB image. In the context of a 1D convolutional neural network (CNN) like the one in the code you provided, the spatial dimension refers to the length of the input sequence or signal. Each element in the sequence represents a position or location along the spatial dimension. The convolutional operation is applied along this spatial dimension, capturing local patterns or features in the input sequence.

6. self.fc1 = nn.Linear(16 * 101, 64) creates a fully connected (linear) layer. It takes an input of size 16 * 101 (from the previous layer) and produces an output of size 64. This layer performs a linear transformation on the input data.
7. self.relu2 = nn.ReLU() creates another instance of the ReLU activation function.
8. self.fc2 = nn.Linear(64, output_dim) creates the final fully connected layer. It takes an input of size 64 and produces an output of size output_dim, which corresponds to the number of classes in the classification task
9. x = x.unsqueeze(1) adds a channel dimension to the input tensor. The input tensor x has a shape of (batch_size, input_dim), and we add a new dimension to make it (batch_size, 1, input_dim). This is required because the convolutional layer expects an input with a channel dimension.

10. x = self.conv1(x) applies the first convolutional layer to the input. The output shape will be (batch_size, 16, conv_output_length), where conv_output_length depends on the input length, kernel size, and padding.

11. x = self.relu1(x) applies the ReLU activation function to the output of the first convolutional layer element-wise.

12. x = self.pool1(x) performs max pooling on the output of the ReLU activation layer. The output shape will be (batch_size, 16, pooled_length), where pooled_length is the length after max pooling.

13. x = x.view(x.size(0), -1) reshapes the tensor by flattening it. It converts the tensor from shape (batch_size, 16, pooled_length) to shape (batch_size, 16 * pooled_length).

    In PyTorch, the view() function is used to reshape tensors. It allows you to specify the desired shape while keeping the same underlying data.

The line x = x.view(x.size(0), -1) has two arguments: x.size(0) and -1.

x.size(0) returns the size of the first dimension of x, which is 2 in this case. This argument specifies that we want to keep the same size for the first dimension of the reshaped tensor. -1 is used as a placeholder for the size of the second dimension. When -1 is used in the view() function, PyTorch automatically infers the size based on the other dimensions and the total number of elements in the tensor.

14. x = self.fc1(x) applies the fully connected layer to the flattened tensor. The output shape will be (batch_size, 64).

15. x = self.relu2(x) applies the ReLU activation function to the output of the fully connected layer.

16. x = self.fc2(x) applies the final fully connected layer to obtain the logits. The output shape will be (batch_size, output_dim).

17. The logits are returned as the final output from the forward method. The logits represent the raw predicted scores for each class

Formula and Calculation of Each Layer:

Let's go through each layer of the CustomCNN_V1 model and calculate the output shape for a given input_dim of 204. We'll also provide the necessary mathematical formulas and explanations for each layer.

Input Layer:

- The input tensor shape is (batch_size, input_dim).
- For our example, the input dimension is 204.

Convolutional Layer (self.conv1):

- The nn.Conv1d layer applies a 1-dimensional convolution to the input.
- It takes an input tensor of shape (batch_size, in_channels, input_length) and produces an output tensor of shape (batch_size, out_channels, output_length).
- In our case, in_channels is equal to input_dim (204), out_channels is 16, and input_length is the number of time steps (length of the sequence).
- The kernel_size parameter represents the size of the convolutional kernel (filter). Here, it is set to 3.
- The output length can be calculated using the formula: output_length = input_length - kernel_size + 1.

ReLU Activation (self.relu1):

- The ReLU activation function applies an element-wise activation operation, setting negative values to zero.
- It doesn't change the shape of the tensor, so the output shape remains the same.

Max Pooling Layer (self.pool1):

- The nn.MaxPool1d layer performs 1-dimensional max pooling on the input.
- It takes an input tensor of shape (batch_size, channels, input_length) and produces an output tensor of shape (batch_size, channels, output_length).
- The kernel_size parameter specifies the size of the pooling window.
- The output length can be calculated using the formula: output_length = input_length // kernel_size.

Flatten Layer (x.view(x.size(0), -1)):

- This line reshapes the tensor by flattening it.
- The view() function is used to reshape the tensor.
- x.size(0) returns the size of the first dimension (batch size), and -1 is used as a placeholder for the size of the second dimension, which is inferred based on the remaining dimensions and the total number of elements.
- The output shape will be (batch_size, flattened_length).

Fully Connected Layer (self.fc1):

- The nn.Linear layer applies a linear transformation to the input.
- It takes an input tensor of shape (batch_size, input_features) and produces an output tensor of shape (batch_size, output_features).
- Here, input_features is equal to 16 * 101 because the previous layer's output had 16 channels and 101 flattened units.
- The output shape will be (batch_size, 64).

ReLU Activation (self.relu2):

- Similar to the previous ReLU activation, it doesn't change the shape of the tensor.

Fully Connected Layer (self.fc2):

- The final fully connected layer takes the input tensor of shape (batch_size, 64) and produces an output tensor of shape (batch_size, output_dim).
- Here, output_dim is the desired output dimension specified during the model initialization.

Each line of code in the forward() method represents a transformation applied to the input tensor (x). These transformations are performed sequentially in the order listed above.

Let's summarize the output shape after each line:

- Input shape: (batch_size, input_dim)
- After self.conv1: (batch_size, 16, output_length1)
- After self.relu1: (batch_size, 16, output_length1)
- After self.pool1: (batch_size, 16, output_length2)
- After x.view(x.size(0), -1): (batch_size, flattened_length)
- After self.fc1: (batch_size, 64)
- After self.relu2: (batch_size, 64)
- After self.fc2: (batch_size, output_dim)

Note: The output length (output_length1 and output_length2) and flattened length depend on the specific input length, kernel size, and pooling size. These values can be calculated using the formulas mentioned earlier.

Let's walk through an example with a small input and print the input and output shapes at each line of the code.

Assuming input_dim = 4 and output_dim = 2 for simplicity, let's follow the forward pass and track the shapes:

```
input_dim = 4
output_dim = 2
model = CustomCNN_V1(input_dim, output_dim)

# Create a sample input tensor
input_tensor = torch.randn(2, input_dim)
print("Input shape:", input_tensor.shape)

# Perform the forward pass
x = input_tensor
x = x.unsqueeze(1)
print("After unsqueeze(1) - Input shape:", x.shape)

x = model.conv1(x)
print("After conv1 - Output shape:", x.shape)

x = model.relu1(x)
print("After relu1 - Output shape:", x.shape)
```

```
x = model.pool1(x)
print("After pool1 - Output shape:", x.shape)

x = x.view(x.size(0), -1)
print("After view - Output shape:", x.shape)

x = model.fc1(x)
print("After fc1 - Output shape:", x.shape)

x = model.relu2(x)
print("After relu2 - Output shape:", x.shape)

x = model.fc2(x)
print("After fc2 - Output shape:", x.shape)
```

Output:

```
Input
shape: torch.Size([2, 4])
After
unsqueeze(1) - Input
shape: torch.Size([2, 1, 4])
After
conv1 - Output
shape: torch.Size([2, 16, 2])
After
relu1 - Output
shape: torch.Size([2, 16, 2])
After
pool1 - Output
shape: torch.Size([2, 16, 1])
After
view - Output
shape: torch.Size([2, 16])
After
fc1 - Output
shape: torch.Size([2, 64])
After
relu2 - Output
shape: torch.Size([2, 64])
After
fc2 - Output
shape: torch.Size([2, 2])
```

In this example, we have a batch size of 2, and the input tensor has a shape of (2, 4).

- After applying x = x.unsqueeze(1), the shape becomes (2, 1, 4). We add a channel dimension to the input tensor.
- model.conv1 performs a 1D convolution on x, resulting in an output shape of (2, 16, 2) (assuming output_length1 = 2).
- The ReLU activation function model.relu1 doesn't change the shape, so the output shape remains (2, 16, 2).
- The max pooling operation model.pool1 with kernel_size=2 reduces the length dimension, resulting in an output shape of (2, 16, 1) (assuming output_length2 = 1).
- x.view(x.size(0), -1) flattens the tensor, converting it to (2, 16).
- The first fully connected layer model.fc1 transforms the input into (2, 64).
- Applying the ReLU activation function model.relu2 doesn't change the shape, so the output shape remains (2, 64).
- Finally, model.fc2 produces the desired output shape of (2, 2).
- Please note that the exact values for output_length1 and output_length2 will depend on the specific input size and the chosen kernel size and pooling size

After applying x = x.unsqueeze(1), the shape becomes (2, 1, 4). We add a channel dimension to the input tensor.

What does mean this line?

The line x = x.unsqueeze(1) is used to add a channel dimension to the input tensor. In PyTorch, convolutional layers expect inputs with multiple channels, even if the input is 1D. By default, convolutional layers assume the input has a shape of (batch_size, channels, length).

In the context of the provided code, the input tensor has a shape of (batch_size, input_dim) before unsqueezing. After applying x = x.unsqueeze(1), the shape becomes (batch_size, 1, input_dim). The unsqueeze operation inserts a new dimension at index 1, effectively adding a channel dimension to the input tensor.

Adding the channel dimension is necessary for the subsequent convolutional layer (self.conv1) to operate correctly. In this case, the convolutional layer expects an input with shape (batch_size, 16, input_dim) where 16 represents the number of output channels from the convolution.

In PyTorch, the convention for representing images and multidimensional data is (batch_size, channels, height, width) or (batch_size, channels, length) depending on the dimensionality of the data.

In the original input tensor of shape (batch_size, input_dim), we only have the batch dimension and the feature dimension. By unsqueezing at index 1, we expand the tensor to have a shape of (batch_size, 1, input_dim). This means that each input sample now has a single channel (in this case, representing the features of the data).

Python view()

Example:

```
q = torch.tensor([[1,2,3], [4,5,6]])
print(q)

size_0 = q.size(0)
size_1 = q.size(1)
print(f"size_0: {size_0}")
print(f"size_1: {size_1}")

view_0 = q.view(size_0, -1)
view_1 = q.view(size_1, -1)

print(f"view_0: {view_0}")
print(f"view_1: {view_1}")
```

Output:

```
tensor([[1, 2, 3],
        [4, 5, 6]])

size_0: 2
size_1: 3

view_0: tensor([[1, 2, 3],
        [4, 5, 6]])

view_1: tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

In PyTorch, the -1 argument in the view() function is used to infer the size of that dimension based on the other dimensions and the total number of elements in the tensor.

In the provided example, we have a tensor q with shape (2, 3). When we apply view_0 = q.view(size_0, -1), it means we want to reshape q into a tensor with size_0 rows and an inferred number of columns.

Since -1 is used for the number of columns, PyTorch automatically calculates the appropriate number of columns based on the total number of elements in the tensor and the given size_0. In this case, q has 6 elements (2 rows x 3 columns), and size_0 is 2, so the resulting tensor view_0 will have shape (2, 3).

Similarly, when we apply view_1 = q.view(size_1, -1), it means we want to reshape q into a tensor with size_1 columns and an inferred number of rows. PyTorch again calculates the appropriate number of rows based on the total number of elements in the tensor and the given size_1. In this case, q has 6 elements, and size_1 is 3, so the resulting tensor view_1 will have shape (3, 2).