Small Project Build a ResNet20 Models

Khaula Sayyidatunadia

May 15, 2024

1 Introduction about ResNet

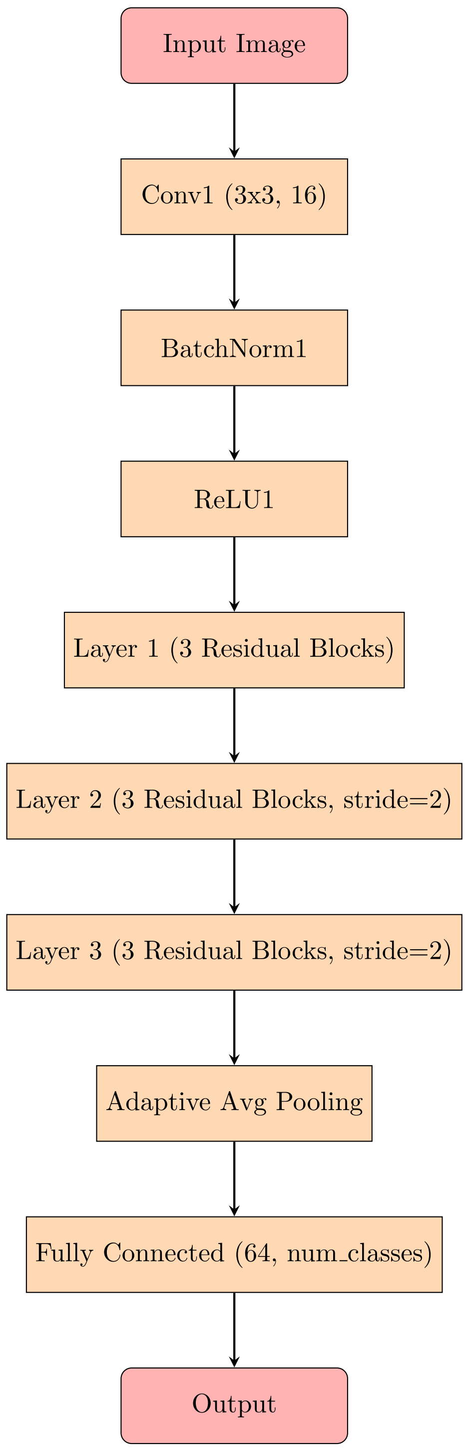
ResNet uses VGG's structure of 3x3 convolutional layers. Each residual block in ResNet has two 3x3 convolutional layers with the same number of output channels. After each convolutional layer, there is a batch normalization layer and a ReLU activation function. The key idea is the shortcut connection: the input skips the two convolutional layers and is added directly to their output just before the final ReLU activation. This design requires the input and output to have the same shape so they can be added together. If we need to change the number of channels, we add a 1x1 convolutional layer to adjust the input to the correct shape for the addition.

2 Model ResNet-20

```
[language=Python, caption=ResNet20 Model]
import torch
import torch.nn as nn

class ResNet20(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet20, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)

        # Residual blocks
        self.layer1 = self._make_layer(in_channels=16, out_channels=16, num_blocks=3, stride=1)
        self.layer2 = self._make_layer(in_channels=16, out_channels=32, num_blocks=3, stride=2)
        self.layer3 = self._make_layer(in_channels=32, out_channels=64, num_blocks=3, stride=2)

        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc_out = nn.Linear(64, num_classes)

        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def _make_layer(self, in_channels, out_channels, num_blocks, stride):
        layers = []
        layers.append(ResidualBlock(in_channels, out_channels, stride))
        for _ in range(1, num_blocks):
            layers.append(ResidualBlock(out_channels, out_channels, stride=1))
        return nn.Sequential(*layers)

```

```
    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc_out(x)
        return x

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        shortcut = self.shortcut(x)
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        x += shortcut
        x = self.relu(x)
        return x

# Instantiate the model
model = ResNet20()
print(model)
```

2.1 Architectureof my ResNet20



3 Explanation about ResNet20

Conv1 Layer: Converts the input image (3 channels, e.g., RGB) into 16 feature channels using a 3x3 filter.

Batch Normalization: Normalizes the output from Conv1.

ReLU Activation: Non-linear activation applied after batch normalization.

Residual Blocks: Series of blocks, each with two 3x3 convolutional layers and shortcut connections.

Adaptive Average Pooling: Reduces the feature map size to 1x1.

Fully Connected Layer: Produces the final class predictions.

Residual Blocks

Residual blocks are the core of the ResNet architecture. In this implementation:

- **Conv1 and Conv2**: Each residual block contains two 3x3 convolutional layers.
- **Batch Normalization and ReLU**: Applied after each convolutional layer.
- **Shortcut Connection**: Provides a direct path from the input to the output of the residual block, helping to prevent the vanishing gradient problem.

Shortcut Connection

If the number of channels or spatial dimensions changes (due to changes in stride or filter size), the shortcut connection uses a 1x1 convolutional layer to adjust the dimensions to match the output of the convolutional layers.

_make_layer Function

This function constructs a residual layer composed of multiple residual blocks. If the stride is not 1 or if the number of input channels differs from the output channels, the shortcut connection will use a 1x1 convolution to adjust the dimensions.

Forward Pass

In the forward method, the input image is processed through the following layers:

- **Conv1 → BN → ReLU**: Convolutional layer followed by Batch Normalization and ReLU activation.
- **Layer1, Layer2, Layer3**: Each layer consists of several residual blocks.
- **Adaptive Average Pooling**: Performs adaptive average pooling.
- **Flattening**: Converts the tensor from spatial dimensions to a 1D vector.
- **Fully Connected Layer**: Produces the final output probabilities for each class.

Weight Initialization

The _initialize_weights function ensures that all convolutional layer weights are initialized using the He (Kaiming He) normal distribution, and the biases in the batch normalization layers are set to their default values.

4 Results of ResNet20 Model on CIFAR-10

Below are the results of training and testing the ResNet20 model on the CIFAR-10 dataset:

Epoch	Train Loss	Test Loss	Accuracy
1	1.6119	1.3691	51.46
2	1.1025	1.0943	62.34
3	0.8830	0.9076	68.53
4	0.7622	0.8337	71.87
5	0.6960	0.9722	68.84
6	0.6524	0.7677	74.38
7	0.6258	0.8284	72.07
8	0.5992	0.7076	76.23
9	0.5837	0.9341	71.19
10	0.5719	1.0154	68.15
11	0.5541	0.7721	74.30
12	0.5456	0.9238	71.53
13	0.5368	0.9986	69.60
14	0.5229	0.8773	72.49
15	0.5206	0.7474	74.69
16	0.5190	0.8156	73.03
17	0.5045	0.7561	76.30
18	0.4981	0.6690	77.31
19	0.5028	0.6678	78.01
20	0.4880	0.6730	77.49
21	0.4884	0.5488	81.24
22	0.4815	0.7730	75.66
23	0.4828	0.6952	75.90
24	0.4762	0.6977	77.00
25	0.4752	0.6214	79.67
26	0.4650	0.7763	73.94
27	0.4711	0.5647	81.19
28	0.4674	0.6357	78.99
29	0.4641	0.5644	80.36
30	0.4602	0.6145	79.28
31	0.4563	0.6713	78.19
32	0.4548	1.2811	63.33
33	0.4603	0.6551	78.88
34	0.4551	0.6778	77.84
35	0.4546	0.6242	78.40
36	0.4472	0.5960	80.14
37	0.4499	0.6229	80.46
38	0.4552	0.5904	80.24
39	0.4467	0.5905	80.00
40	0.4460	0.6393	79.73
41	0.4427	0.5497	81.32
42	0.4423	0.5452	81.00
43	0.4446	0.5627	81.12
44	0.4415	0.6919	77.95
45	0.4385	0.5950	80.22
46	0.4348	0.5841	80.85
47	0.4367	0.5577	82.02
48	0.4366	0.9322	73.32
49	0.4357	0.6615	78.58
50	0.4365	0.6775	77.85
51	0.4301	0.6880	78.57

To calculate the average error across all epochs, you can follow these steps:

1. Sum all the errors from each epoch.
2. Divide the total error by the total number of epochs.

Let's compute it:
The total number of epochs provided is 51.

$$\text{Total error} = (1 - 0.5146) + (1 - 0.6234) + \ldots + (1 - 0.7857)$$

$$\text{Total error} = 51 - (0.5146 + 0.6234 + \ldots + 0.7857)$$

$$\text{Total error} = 51 - 39.74$$

$$\text{Total error} = 11.26$$

Now, let's calculate the average error:

$$\text{Average error} = \frac{\text{Total error}}{\text{Total number of epochs}}$$

$$\text{Average error} = \frac{11.26}{51}$$

$$\text{Average error} \approx 0.2208$$

Hence, the average error across all epochs is approximately 22.08%.

Table 1: Comparison of ResNet20 Performance

Model	**#Layer**	**#Params**	**Error**	**Accuracy (%)**
ResNet20 (Paper)	20	0.27M	8.75%	90%
ResNet20 (Khaula)	20	272,474	22.08%	81.24%

5 Conclusion

5.1 Comparison Between My ResNet Model and Reference ResNet Model

In this document, we compare the ResNet model that I developed with the reference ResNet model proposed by Kaiming He et al. We highlight the differences in architecture and training approaches.

5.2 Architecture Differences

- My model allows for ReLU activation after each convolutional layer or after each block, while the reference model applies ReLU after each convolutional layer.
- I use average pooling after the final block, whereas the reference model uses max pooling after each block.

5.3 Modifications in Training and Testing

- I set weight decay to 5×10^{-4} in my model.
- Unlike the reference model, I did not use $t_{\max}$ for scheduling steps due to encountering errors multiple times.
- The long computation time was due to limitations of the device, resulting in lengthy computations.

5.4 Proposed Solutions

- To address long computation time, I propose reducing the model size by using convolution with a 3×3 kernel and stride 2, or employing model pruning.
- Additionally, I suggest reducing the number of epochs from 200 to 100 to mitigate the risk of disruptions during training and testing.

While my ResNet model offers flexibility in activation functions and pooling methods, it faces challenges in terms of computational resources and stability during training. However, by implementing proposed solutions, these challenges can be addressed effectively.