DL(CS60010) ASSIGNMENT 1
Omkar Bhandare 22CS30016

Introduction

This assignment aims to implement custom deep learning components, namely CustomBatchNorm2d, CustomReLU, CustomDropout and CustomMaxPooling2d and integrate them into a modified VGG16 model for image classification. In the assignment, we have used the well-known MNIST dataset, which includes 60,000 training images and 10,000 test images, each of which is 28x28 pixels (and single channel, i.e. grayscale), each of which consists of handwritten digits from 0-9. The dataset images are themselves size-normalized and centred.
Building custom components from scratch offers multiple advantages:

1. Enhances the understanding of core mathematics used in DL concepts
2. It provides flexibility to modify behaviour layerwise
3. Allows experimentation with various parameters, which may not be possible if equivalent library functions are used
4. It helps in gaining insights and understanding the underlying training dynamics

Implementing these components from scratch helped us understand how these components work internally.
The assignment also let us explore the concepts of Data Augmentation and Learning Rate Scheduling, which, as we will see later, helped improve the final results.

Implementation Details

1. **Data Augmentation**

The Data was augmented while loading the Dataset itself on the go, a random rotation from -15 to +15 degrees was applied along with a 10% translation in both horizontal and vertical direction. The augmented images were converted to tensors and then normalised using the mean and variance of MNIST Pixel values taken from the internet. Note that the augmentation was applied only to the Training images in the dataset.

```
train_transform = transforms.Compose([
       transforms.RandomRotation(15),
       transforms.RandomAffine(degrees = 0, translate=(0.1, 0.1)),
       transforms.ToTensor(),
       transforms.Normalize((0.1307,), (0.3081,))
   ])
```

2. **Custom Dropout**

   Bernoulli Distribution is used to generate the mask for the dropout; the distribution produces 0 with a probability of p and 1 with a probability of 1-p; the input is then element-wise multiplied by the produced to drop the neurons where the produced mask is 0. Finally, the masked input is divided by 1-p to keep the expected value of activations unchanged.

```
if self.training:
            mask = torch.bernoulli(torch.ones_like(x) * (1 - self.p))
            return x * mask / (1 - self.p)
```

3. **Custom BatchNorm2d**

At first, mean and variance were calculated per channel, and the running values were maintained. Using the calculated mean and variance, the input x was normalised, and finally, the normalized x was returned with scaling and shifting with the parameters gamma and beta. Gamma allows the network to control the normalised activation scale, while Beta enables the network to move the normalized values by a learned offset. Without Gamma and Beta, BatchNorm would force the output to always have zero mean and unit variance. The momentum is used to control how much the running statistics are updated with new batch statistics; this helps to maintain a more stable estimate of the statistics.

```
if self.training:
            mean = x.mean([0, 2, 3])
            var = x.var([0, 2, 3], unbiased=False)
            self.running_mean=(1-self.momentum)*self.running_mean   +
self.momentum * mean
                self.running_var=(1-self.momentum)*self.running_var  +
self.momentum * var
        else:
            mean = self.running_mean
            var = self.running_var

    x_normalized=(x-mean[None,  :,  None,  None])/torch.sqrt(var[None,
:, None, None] + self.eps)
        return    self.gamma[None,    :,    None,    None]*x_normalized+
self.beta[None, :, None, None]
```

4. **Custom ReLU**

Implemented ReLU activation ReLU(x) = max(x, 0). Here element-wise operation was implemented for efficient computation

```
def forward(self, x):
        return torch.max(torch.zeros_like(x), x)
```

5. **Custom MaxPooling2d**

   The output dimensions were calculated as per the formula discussed in the theory O = floor ((I + 2P - K) / S) + 1, where O: single output dimension, I: single input dimension, P: padding size, K: kernel single dimension, S: stride value. The torch.nn.functional unfold method was used to convert the input tensor into sliding windows. The unfolded tensor was reshaped so the max could easily be calculated along the last dimension. Then, the final computed tensor was reshaped to the required output dimensions.

```
def forward(self, x):
        N, C, H, W = x.shape
        out_H = (H - self.kernel_size) // self.stride + 1
        out_W = (W - self.kernel_size) // self.stride + 1

        x_unfolded = x.unfold(2, self.kernel_size, self.stride)
    x_unfolded = x_unfolded.unfold(3, self.kernel_size, self.stride)

        x_windows = x_unfolded.contiguous()
        x_pooled = x_windows.view(N, C, out_H, out_W, -1).max(-1)[0]
        return x_pooled
```

6. **Custom VGG16 Model**

   Following is the layer-wise information of custom-implemented VGG16 architecture:

Block 1
Conv2D : P = 1, K = 3 x 3, number of filters = 64
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 64
BatchNorm2D
ReLU
MaxPool : K = 2 x 2, S = 2

Block 2
Conv2D : P = 1, K = 3 x 3, number of filters = 128
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 128
BatchNorm2D
ReLU
MaxPool : K = 2 x 2, S = 2

Block 3
Conv2D : P = 1, K = 3 x 3, number of filters = 256
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 256
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 256
BatchNorm2D
ReLU
MaxPool : K = 2 x 2, S = 2

Block 4
Conv2D : P = 1, K = 3 x 3, number of filters = 512
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 512
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 512
BatchNorm2D
ReLU
MaxPool : K = 2 x 2, S = 2

Block 5
Conv2D : P = 1, K = 3 x 3, number of filters = 512
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 512
BatchNorm2D
ReLU
Conv2D : P = 1, K = 3 x 3, number of filters = 512
BatchNorm2D
ReLU
MaxPool : K = 2 x 2, S = 2

FCC
Flatten the final dimensions: 512 x 7 x 7
FCC: 4096 (ReLU, Dropout {p = 0.5})
FCC: 4096 (ReLU, Dropout {p = 0.5})
FCC: 10 (CrossEntropyLoss)

7. **Learning Rate Scheduler**
   Along with the Adam optimizer, a Learning Rate Scheduler was also implemented for the same ReduceLROnPlateau is used. In this method, the scheduler.s tep(test_loss) is called after each epoch, and the scheduler monitors the test loss if this metric does not decrease for the ‘patience’ number of epochs; the LR is then multiplied by the provided factor. This specific method is helpful when we want to overcome the plateaus in training, i.e. when np improvement is detected for certain epochs, update the LR and adjust the LR. This helps improve the overall final accuracy.

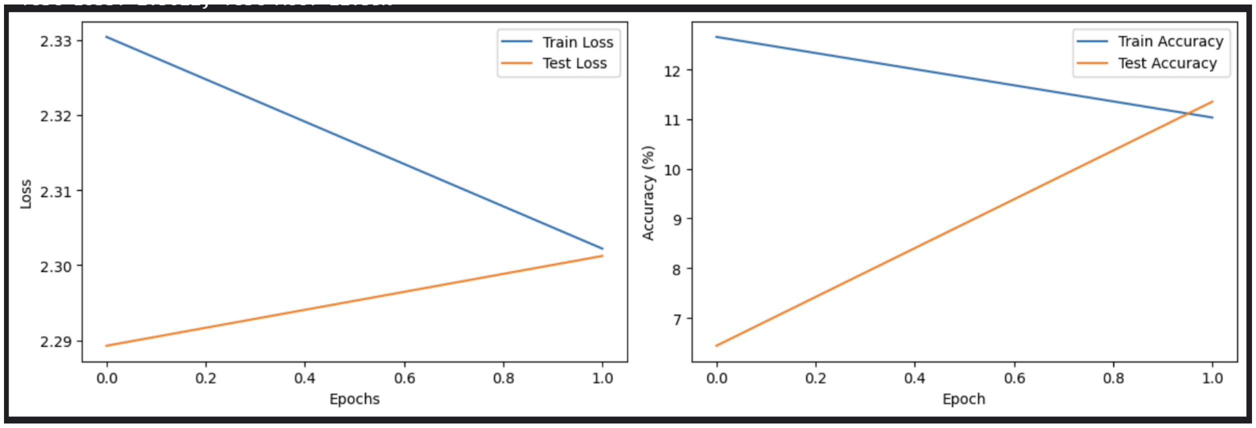
```
scheduler=optim.lr_scheduler.ReduceLROnPlateau(optimizer,mode='min',
factor=0.5, patience=1)

# the below function is called in the epoch loop
scheduler.step(test_loss)
```

Training Strategy and Results

For the training purposes, we will start with the following hyperparameters:
(*please note that since it was computationally expensive, the number of epochs has been reduced to 2, so that it runs within a reasonable amount of time)*
BATCH_SIZE = 16
EPOCHS = 2
LR = 0.001
Optimizer = Adam
Scheduler = ReduceLROnPlateau (factor = 0.5, patience = 1)
For the custom implementation after the 2 epochs, the following statistics were obtained :

```
Train Loss: 2.3022, Train Acc: 11.03%
Test Loss: 2.3012, Test Acc: 11.35%
```



For the PyTorch implementation after the 2 epochs, the following statistics were obtained:

```
Train Loss: 2.3022, Train Acc: 11.01%
Test Loss: 2.3012, Test Acc: 11.33%
```

Conclusion and Future Work

Conclusion

1. Implementing custom layers deepened our understanding of various elements of Deep Learning.
2. While the model closely matched the PyTorch implementation in accuracy, there was a minor gap due to numerical precision and optimization differences.
3. The Batch Normalization played a key role in significantly improving the convergence rates.

Future Work

1. Optimize CustomMaxPooling2d so that it can handle non-divisible kernel sizes more effectively.
2. Try hyperparameter tuning with various hyperparameters present in the current model implementation.
3. Experiment with other normalization techniques apart from the Batch Normalization.