# CSCE 636: Deep Learning Assignment 2

Fatemeh Doudi

March 8, 2024

## 1 Programming Assignment

### Part a:

I have acquired and am utilizing the Python variant of the CIFAR-10 dataset.

### Part b:

In completing the "imageUtils.py", as outlined in Figure 1, the following steps were taken: Initially, a padding of 4 units is applied to all sides of the image, effectively enlarging the dimensions to (32 + 2 * padding) in both directions. The key task then involves randomly cropping a 32x32 segment from the padded image. To achieve this, one must identify the upper-left starting point for the crop, which should be strategically chosen within the bounds of (0, 0) to (2 * padding, 2 * padding) to ensure the final image size of 32x32.

### Part c:

In this section, we delve into the implementation of "NetWork.py". The dataset images originate from 3-channel data, and per the assignment's requirements, we must transform them into 16-channel data using a convolutional layer. Details of this implementation can be found in Figure 2.

In the subsequent step, we will focus on implementing the "norm-relu" layer (Figure 4), which is specific to ResNet version 1. This layer is a crucial component exclusive to this version, as its functionality is inherently integrated within the architecture of version 2 blocks.

In our endeavor to implement the two distinct blocks, we turn our attention first to the Standard block. For this block, I have replicated the structure depicted in Figure 4(a) from reference [2]. This implementation is segmented into two layers. The first layer comprises a convolutional layer, succeeded by batch normalization and then a ReLU activation. The second layer consists of the weights and batch normalization, with the application of ReLU following the addition of the residual connection. It's crucial to highlight that at the beginning of each stack, a downsampling by a factor of 2 is necessary, thereby the stride for the first layer of the first block in each stack is set to 2, with the exception of stack 0. Detailed information on the implementation can be found in 5.

For the Bottleneck block, we integrate the pre-activation architecture with Figure 5-Left from reference [1], leading to the derived structure presented in **??**. This configuration consists of three layers, each initiating with batch normalization and a ReLU

function, followed by a convolution layer. At the conclusion of the third layer, the residual connection is added. The initial 1x1 layer reduces the dimensionality of the feature maps, effectively compressing the number of features. The second layer, utilizing a 3x3 filter, processes the data, engaging in the core convolution operation. Finally, the third layer, also a 1x1 convolution, restores the number of feature maps to their original dimensionality, ensuring that the output matches the input's depth for the residual connection. Similar to the Standard block, downsampling occurs at the start of each stack, with the exception of the first one. Additionally, within the first layer of stack 0, the number of filters remains unchanged.

In the next step, our focus shifts to layer stacking. A critical aspect of this process involves specifying the projection based on the input and output dimensions of each block. The arrangement for stacking the standard blocks is illustrated in Figure 6. Notably, in this phase, it is important to mention that there is no requirement for projection in the first block of stack0, as the input and output dimensions remain consistent.However, stacking the bottleneck blocks, as shown in Figure 7, presents a more complex scenario. Specifically, within stack0, the initial block requires a projection to increase the feature map depth from 16 to 64 to match the output specifications. For the first block in subsequent stacks, the projection needs to adjust the feature map depth from filter*2 to filter*4. This adjustment ensures that the input and output dimensions of these blocks align properly, facilitating effective integration within the network's architecture.

## Part d:

In this segment of the work, I employ the Adam optimizer for optimization purposes(Although Adam did not exist at that time, I think we should appreciate the advancements in optimization techniques:) ), alongside cross-entropy as the criterion for evaluating the performance of my model. It's important to highlight that the Adam optimizer eliminates the need for manual weight decay adjustments and dynamically modifies the learning rate, offering a more adaptive approach to optimization compared to traditional methods like SGD.

## Part e:

### Result - Version1:

Fine-tuning for version 1 involved experimenting with various parameters, including ResNet size and batch size. Adjustments to the batch size did not yield significant improvements. The learning curve for different ResNet sizes is illustrated in 8, while the outcomes for the validation set in Checkpoint 200, are presented in Table 1.

Table 1: V1-Resnet size

| ResNet size | Accuracy |
|---|---|
| 18 | 0.8984 |
| 16 | 0.8750 |
| 8 | 0.8596 |

This establishes the base model as the optimal choice. After retraining, the performance on the test set reached 89.81 percent. The validation and test result for the

selected model has been shown in 10. Also, a log for best v1 model is in the submission folder. It only has the epoch loss, so it is relatively small.

**Result - Version2:**

Like version 1, for version 2, the learning curve for different ResNet sizes is illustrated in 9. we got the following accuracy results for different batch size on validation set:

Table 2: V2-Resnet size

| ResNet size | Accuracy |
|---|---|
| 18 | 0.8764 |
| 16 | 0.8724 |
| 8 | 0.8662 |

This establishes the base model as the optimal choice. After retraining, the performance on the test set reached 87.29 percent. The validation and test result for the selected model has been shown in 11. Also, a log for best v2 model is in the submission folder. It only has the epoch loss, so it is relatively small.

# 2 Figures

```python
def preprocess_image(image, training):
    """ Preprocess a single image of shape [height, width, depth].

    Args:
        image: An array of shape [32, 32, 3].
        training: A boolean. Determine whether it is in training mode.

    Returns:
        image: An array of shape [32, 32, 3].
    """
    if training:
        ### YOUR CODE HERE
        # Resize the image to add four extra pixels on each side.
        ### YOUR CODE HERE
        pad = 4
        image = np.pad(image, ((pad, pad), (pad, pad), (0, 0)), mode='constant')
        ### YOUR CODE HERE
        # Randomly crop a [32, 32] section of the image.
        # HINT: randomly generate the upper left point of the image
        coordinates = np.random.randint(0 , 2*pad+1 , size=2)
        sub_image = image[coordinates[0]:coordinates[0]+32 , coordinates[1]:coordinates[1]+32 , :]
        ### YOUR CODE HERE

        ### YOUR CODE HERE
        image = np.fliplr(sub_image) if (np.random.rand() > 0.5) else sub_image
        ### YOUR CODE HERE

    ### YOUR CODE HERE
    mean = np.mean(image , axis=(0, 1))
    std = np.std(image , axis=(0, 1))
    image = (image - mean) / std
    ### YOUR CODE HERE

    return image
```

Figure 1: ImageUtils.py

```python
### YOUR CODE HERE
# define conv1
self.start_layer = nn.Conv2d(in_channels = 3, out_channels = self.first_num_filters, kernel_size = 3, stride = 1, padding = 1)
### YOUR CODE HERE
```

Figure 2: Starting Layer

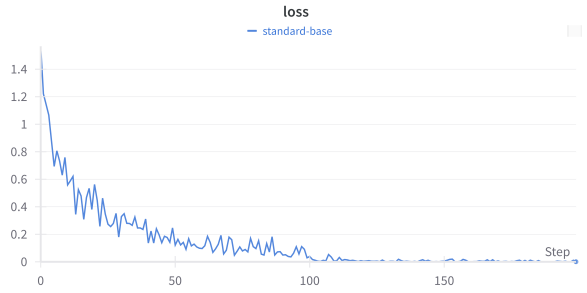


Figure 3: 5-b

```python
class batch_norm_relu_layer(nn.Module):
    """ Perform batch normalization then relu.
    """
    def __init__(self, num_features, eps=1e-5, momentum=0.997) -> None:
        super(batch_norm_relu_layer, self).__init__()
        ### YOUR CODE HERE
        self.batch_norm_relu = nn.Sequential(
            nn.BatchNorm2d(num_features, eps=eps, momentum=momentum) ,
            nn.ReLU(inplace=True)
        )
        ### YOUR CODE HERE
    def forward(self, inputs: Tensor) -> Tensor:
        ### YOUR CODE HERE
        return self.batch_norm_relu(inputs)
        ### YOUR CODE HERE
```

Figure 4: norm relu

```python
def __init__(self, filters, projection_shortcut, strides, first_num_filters) -> None:
    super(standard_block, self).__init__()
    blck_num = first_num_filters
    ### YOUR CODE HERE
    self.projection_shortcut = projection_shortcut if projection_shortcut is not None else None
    if blck_num == 0:
        if filters != 16:
            self.layer1 = nn.Sequential(
                nn.Conv2d(in_channels=filters // 2, out_channels=filters, kernel_size=3, stride = 2, padding=1, bias=False),
                nn.BatchNorm2d(num_features=filters),
                nn.ReLU(inplace=True)
            )
        else:
            self.layer1 = nn.Sequential(
                nn.Conv2d(in_channels=filters, out_channels=filters, kernel_size=3, stride = 1, padding=1, bias=False),
                nn.BatchNorm2d(num_features=filters),
                nn.ReLU(inplace=True)
            )
    elif blck_num >0:
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=filters, out_channels=filters, kernel_size=3, stride = 1, padding=1, bias=False),
            nn.BatchNorm2d(num_features=filters),
            nn.ReLU(inplace=True)
        )

    self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=filters, out_channels=filters, kernel_size=3, stride=1, padding=1, bias=False),
        nn.BatchNorm2d(num_features=filters)
    )
    self.relu = nn.ReLU(inplace=True)

def forward(self, inputs: Tensor) -> Tensor:
    ### YOUR CODE HERE
    res = self.projection_shortcut(inputs) if (self.projection_shortcut is not None) else inputs

    y1 = self.layer1(inputs)
    y2 = self.layer2(y1) + res
    out = self.relu(y2)
    return out
    ### YOUR CODE HERE
```

Figure 5: standard block

# References

```
if block_fn is standard_block:
    if filters != filters_out or strides != 1:
        projection_shortcut = nn.Sequential(
            nn.Conv2d(filters//2, filters_out, kernel_size=1, stride=strides, bias=False),
            nn.BatchNorm2d(filters_out)
        )
    if stack_num>0:
        for i in range(resnet_size):
            if i == 0:
                block = block_fn(
                    filters=filters, projection_shortcut=projection_shortcut, strides=strides, first_num_filters=i
                )
            else:
                block = block_fn(
                    filters=filters, projection_shortcut=None, strides=1, first_num_filters=i
                )
            self.blocks.append(block)
            filters = filters_out
    else:
        for i in range(resnet_size):
            block = block_fn(
                filters=filters, projection_shortcut=None, strides=1, first_num_filters=i
            )
            self.blocks.append(block)
            filters = filters_out
```

Figure 6: Stack Standard Blocks

```
else:
    if stack_num>0:
        for i in range(resnet_size):
            if i == 0:
                projection_shortcut = nn.Sequential(
                nn.Conv2d(
                    filters*2, filters*4, kernel_size=1, stride=strides, bias=False
                ),
                nn.BatchNorm2d(filters_out)
                )

                block = block_fn(
                    filters=filters, projection_shortcut=projection_shortcut, strides=strides, first_num_filters=i
                )
            else:
                block = block_fn(
                    filters=filters, projection_shortcut=None, strides=1, first_num_filters=i
                )
            self.blocks.append(block)
    else:
        for i in range(resnet_size):
            if i == 0:
                projection_shortcut = nn.Sequential(
                    nn.Conv2d(
                        16, 16*4, kernel_size=1, stride=strides, bias=False
                    ),
                    nn.BatchNorm2d(filters_out)
                )
                block = block_fn(
                    filters=filters, projection_shortcut=projection_shortcut, strides=1, first_num_filters=i
                )
            else:
                block = block_fn(
                    filters=filters, projection_shortcut=None, strides=1, first_num_filters=i
                )
            self.blocks.append(block)
```

Figure 7: Stack Bottleneck Blocks

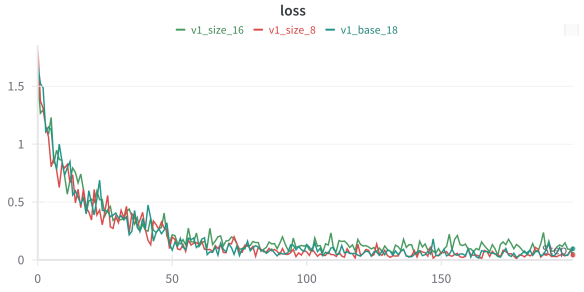


Figure 8: V1-Fine tune over ResNet size

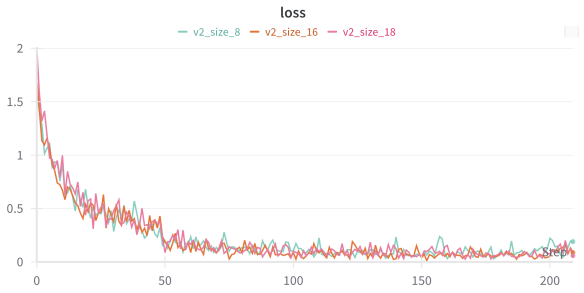


Figure 9: V2-Fine tune over ResNet size

```
### Test or Validation ###
Restored model parameters from model_v1_base_best/model-160.ckpt
Test accuracy: 0.8986
Restored model parameters from model_v1_base_best/model-170.ckpt
Test accuracy: 0.8968
Restored model parameters from model_v1_base_best/model-180.ckpt
Test accuracy: 0.8984
Restored model parameters from model_v1_base_best/model-190.ckpt
Test accuracy: 0.8982
Restored model parameters from model_v1_base_best/model-200.ckpt
Test accuracy: 0.8984
### Training... ###
Epoch 1 Loss 0.008797 Duration 81.667 seconds.
Epoch 2 Loss 0.145293 Duration 81.545 seconds.
Epoch 3 Loss 0.004382 Duration 81.987 seconds.
Epoch 4 Loss 0.067826 Duration 81.550 seconds.
Epoch 5 Loss 0.004615 Duration 75.713 seconds.
Epoch 6 Loss 0.122149 Duration 73.228 seconds.
Epoch 7 Loss 0.062707 Duration 72.873 seconds.
Epoch 8 Loss 0.112204 Duration 73.114 seconds.
Epoch 9 Loss 0.013762 Duration 73.295 seconds.
Epoch 10 Loss 0.097543 Duration 73.954 seconds.
Checkpoint has been created.
### Test or Validation ###
Restored model parameters from model_v1_base_best/model-10.ckpt
Test accuracy: 0.8981
```

Figure 10: V1 best

```
Checkpoint has been created.
### Test or Validation ###
Restored model parameters from model_v2_base_18/model-160.ckpt
Test accuracy: 0.8758
Restored model parameters from model_v2_base_18/model-170.ckpt
Test accuracy: 0.8758
Restored model parameters from model_v2_base_18/model-180.ckpt
Test accuracy: 0.8762
Restored model parameters from model_v2_base_18/model-190.ckpt
Test accuracy: 0.8772
Restored model parameters from model_v2_base_18/model-200.ckpt
Test accuracy: 0.8764
### Training... ###
Epoch 1 Loss 0.077190 Duration 93.654 seconds.
Epoch 2 Loss 0.102857 Duration 93.024 seconds.
Epoch 3 Loss 0.087344 Duration 93.544 seconds.
Epoch 4 Loss 0.136639 Duration 93.307 seconds.
Epoch 5 Loss 0.166942 Duration 93.416 seconds.
Epoch 6 Loss 0.088785 Duration 93.138 seconds.
Epoch 7 Loss 0.201381 Duration 93.048 seconds.
Epoch 8 Loss 0.092959 Duration 93.426 seconds.
Epoch 9 Loss 0.117264 Duration 93.217 seconds.
Epoch 10 Loss 0.059235 Duration 93.472 seconds.
Checkpoint has been created.
### Test or Validation ###
Restored model parameters from model_v2_base_18/model-10.ckpt
Test accuracy: 0.8729
```

Figure 11: V2 best