Internship Report

The proposed goal of the internship was to use deep morphological network implementations on agricultural image processing tasks. However, because of the complexity of the task I was only able to proceed to some trials on classification tasks with implementations of Deep Morphological Networks. Furthermore, I tried to compare this model to a similar CNN model and a hybrid model.

What is a Deep Morphological Network?

CNNs performed well on most of the image processing tasks. However, the linear convolution filters may not capture complex structures in the images. Mathematical morphology using some methods such as Erosion, Dilation, Opening and Closing etc. can do better on some tasks. In the deep morphological networks, we replace convolutional filters with learnable morphological networks. However, we should also note that morphological networks are less efficient than the CNNs. From the comparisons I made CNNs run faster than Morphological networks. However, this can be because of CNN methods were optimized for GPU and I had hand-crafted methods in Morphological networks.

Basic Implementation Architecture:

I have been inspired by the paper “An Introduction to Deep Morphological Networks” by Nogueira et al. In this paper authors propose to do depth wise convolution by applying separate filters to each input channel then do a depth wise pooling which acts like the morphological operation by taking the min or max for erosion and dilation and finally stacks the found filters into one by pointwise convolution.

```
class MaxBinarize(torch.autograd.Function):
    @staticmethod
    def forward(ctx, weight):
        max_val, _ = weight.view(weight.size(0), -1).max(dim=1)
        binarized_weight = (weight == max_val.view(-1, 1, 1, 1)).float()
        return binarized_weight
```

In the MaxBinarize class we binarize the morphological weights to either 1 or 0 by taking the maximum value as 1 and others as 0.We will use it to mimic dilation in positive space and erosion in negative space.

```python
# Depthwise Morphological Layer with depthwise pooling
class DepthwiseMorphological(nn.Module):
    def __init__(self, in_channels, kernel_size=3):
        super(DepthwiseMorphological, self).__init__()
        self.depthwise_conv = nn.Conv2d(in_channels, in_channels,
                                        kernel_size=kernel_size, groups=in_channels,
                                        padding=kernel_size // 2)

    def forward(self, x):
        binarized_weight = MaxBinarize.apply(self.depthwise_conv.weight)
        x = F.conv2d(x, binarized_weight, groups=x.size(1), padding=self.depthwise_conv.padding)
        return x
```

In the DepthwiseMorphological class we aim to perform a depth wise convolution that is convolving the binarized weights(filters) with the input per channel. Because of that we have filters for each input channel.

```python
# Erosion Layer (min-pooling after depthwise convolution)
class ErosionLayer(nn.Module):
    def __init__(self, in_channels, kernel_size=3):
        super(ErosionLayer, self).__init__()
        self.erosion = DepthwiseMorphological(in_channels, kernel_size)

    def forward(self, x):
        return F.max_pool2d(-self.erosion(-x), kernel_size=3, stride=1, padding=1)
```

In the ErosionLayer class we take the the smallest values in the neighbor by negating the input first and maxpooling on the negated output.

```python
# Dilation Layer (max-pooling after depthwise convolution)
class DilationLayer(nn.Module):
    def __init__(self, in_channels, kernel_size=3):
        super(DilationLayer, self).__init__()
        self.dilation = DepthwiseMorphological(in_channels, kernel_size)

    def forward(self, x):
        return F.max_pool2d(self.dilation(x), kernel_size=3, stride=1, padding=1)
```

In the DilationLayer class we take the largest values in neighbour by max pooling the filtered output.

```python
# Combined morphological operations (erosion followed by dilation)
class MorphologicalLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3):
        super(MorphologicalLayer, self).__init__()
        self.erosion = ErosionLayer(in_channels, kernel_size)
        self.dilation = DilationLayer(in_channels, kernel_size)
        self.pointwise_conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)
        self.batch_norm = nn.BatchNorm2d(out_channels)  # Batch normalization

    def forward(self, x):
        x_erosion = self.erosion(x)
        x_dilation = self.dilation(x)
        x_combined = self.pointwise_conv(x_erosion + x_dilation)
        x_combined = self.batch_norm(x_combined)
        return F.relu(x_combined)  # Apply ReLU for non-linearity
```

In MorphologicalLayer class we do 1 erosion layer, 1 dilation layer and 1 pointwise convolution to stack the found outputs. At the end we do batch normalization to stabilize and converge faster. And finally a ReLu layer to provide more non-linearity.

The Opening and Closing Layers can be easily defined by incorporating erosion and dilation layers. For the opening we do erosion followed by dilation and for the closing we do dilation followed by erosion.

```python
# Full DeepMorphNet architecture
class DeepMorphNet(nn.Module):
    def __init__(self, num_classes=10):
        super(DeepMorphNet, self).__init__()
        # Composed layers with different morphological operations
        self.layer1 = MorphologicalLayer(3, 64, kernel_size=5)
        self.layer2 = MorphologicalLayer(64, 128, kernel_size=5)
        self.layer3 = OpeningLayer(128, kernel_size=3)
        self.layer4 = ClosingLayer(128, kernel_size=3)


        self.flatten_size = None  # To be set dynamically

        # Fully connected layers
        self.fc1 = None  # Will be initialized later
        self.fc2 = nn.Linear(1024, 512)
        self.dropout = nn.Dropout(0.5)  # Dropout for regularization
        self.fc3 = nn.Linear(512, num_classes)
```

In the DeepMorphNet class we propose 2 of Morphological Layers defined before, 1 Opening Layer followed by 1 Closing Layer. Therefore, in total we have 4 sequence of erosion layers followed by dilation layers. We calculate the input to Fully Connected Layer and we have 3 fully connected layers. We have a dropout layer to prevent overfitting.

The rest of the code is train and test loops applied to this architecture. The performance of this architecture is 67% accuracy on Cifar10 Dataset. Corresponding CNN architecture gave me 77% accuracy on same dataset.

Hybrid Model:

We proposed a hybrid network model which incorporates both morphological layers and Cnn layers.

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

        # CNN Block 1
        self.cnn1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=5, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Morphological Block
        self.morph_block = MorphologicalLayer(64, 128, kernel_size=3)

        # CNN Block 2
        self.cnn2 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

In this architecture we have 1 convolutional layer then a morphological layer defined before and 1 convolutional layer at the end. This architecture results in 76% Test accuracy.The morphological network and the hybrid network both configured to work on 50 epochs, 0.001 learning rate and 64 batches.

On the Mnist dataset with 2 erosion layers followed by dilation layers performed 98% accuracy.

Results:

Architecture	Dataset	Performance
4 Erosion and Dilation Layers	Cifar10	67%
1 Cnn + 1 Erosion and Dilation Layers + 1 Cnn	Cifar10	76%
3 Cnn	Cifar10	77%
2 Erosion and Dilation Layers	Mnist	98%