SYDE 552 Final Project

By

Danny Kong 20747252

Emad Ahmed 20619763

For

April 16th, 2020

Introduction

Many of the advances throughout the history of machine learning have come as a result of mimicking the learning processes of organisms observed in biology. With respect to humans in particular, this can be seen in the use of image recognition and classification as it relates to the human visual system. While it is always difficult to imitate the human visual system since there is so much that is still undiscovered, it is clear that recent advances like the proliferation of convolutional neural networks (CNN) build on the mechanisms observed in the human vision system [1].

However, there are a number of deep learning techniques that don't necessarily carry over naturally to the visual system. Much of deep learning is still based empirically on what is efficient or accurate for a given problem. Convolution kernels loosely model the idea of a receptive field, but not necessarily very close to the way that it appears in the human visual system.

This project explores the effectiveness of image classification methods with both traditional deep learning methods and biological methods that mimic the human visual system. In particular, the biological methods explored were center-surround receptive fields which is a topic that was explored in SYDE 552. While it is expected that traditional deep learning methods will perform better, the goal is to explore how the biological methods differ in performance and why this occurs. The dataset used was the Cifar-10 dataset which contains 60 000 images with 10 classes. Throughout the process of testing different networks, the effect of manipulating different parameters was observed.

Background

We draw inspiration from the human eye, which contains the first large layer of neurons in the visual system. The eye draws information from a layer of rods and cones, which are sensitive to light and colors. Similarly, the data that we feed to our artificial neural network is a matrix of pixel values. However, the eye is not that clear cut. Although there are bipolar neurons that carry the information to higher areas in the system, there are also inter-layer connections. Horizontal cells, which were studied in class, for example, affect neighbouring neurons by inhibition. The result is a receptive field that is not

necessarily one-to-one mapping from the input (rods and cones) to the output (neurons).

Deep learning attempts to solve this problem with convolution. The model slides a window over the image, and applies a function to each group of inputs in the window, as opposed to the value in the center only. This produces a functional structure that makes it explicit that spatially neighbouring areas are in some way similar. However, this structure does not imitate the integration at the first layer of bipolar neurons [2].

One of the common receptive field layouts of bipolar cells in this area are sometimes referred to as on-center/off-surround. That is, there are two regions in the receptive field, which are considered to be concentric rings. The inner ring links directly to the next layer of neurons. The other ring does not, but still affects the output via a horizontal cell as the intermediary. In layman's terms, the horizontal cells inhibit the output proportional to the average of the inputs over the entire receptive field.

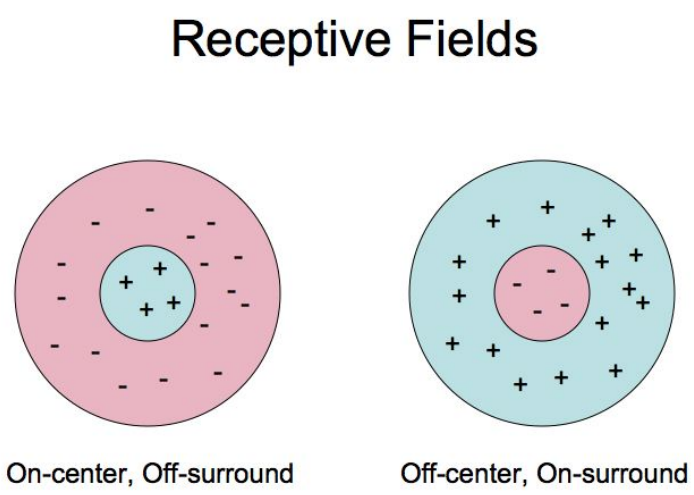

Figure 1: Visual representation of center-surround fields [3]

We may model this in the following way: on every 3X3 window over the image, we take the center pixel as the output. However, we subtract from the output the average of the pixel outputs on the entire window, up to some proportional constant which is a learnable parameter. We also introduce a bias term here, another learnable parameter. The pixel values are 0-255, with 0 being no input.

If the center pixel is non-zero, with all others being zero, we expect the largest output, so we double the output of the center neuron. This way, for a reasonable weight of 1, the output is exactly the input.

If all the values are large, or if only the center value is small, the output is suppressed (center-off). Since the output should not be smaller than the baseline if light hits only the surrounding region, we use a ReLu nonlinearity here.

To summarize, the layer groups inputs by sliding a square 3X3 window over the inputs with padding. If the original input was size 32X32, we would have 32*32 = 1024 of 3X3 images, centered at each pixel. The output is the center of each image, subtracted by the weight times the average of the whole image, plus the bias.

Methods

The testing methodology is simple, we train a traditional CNN on the CIFAR-10 dataset, and see how it performs. Once we have a well performing neural network, we reset the parameters (to avoid any lottery effects), and add a single center/surround layer to the first layer. We designed the layer to be able to act as a drop-in replacement.

The training parameters remain the same over both training runs, but one additional control we could have placed is to have the exact same training data. The order and augmentations are randomly assigned, but in hindsight there may be a way to seed the input batches so that they are the same.

The first experiment is based on the trainability of the networks. We test whether adding this layer makes it more difficult to train (requires more data), by training on small amounts of it (10 epochs), and looking at the differences. The traditional network with early stopping can train slowly but steadily through hundreds of epochs, given data augmentation.Some state of the art networks don’t reach their maximum accuracy until over 100 epochs. However, the largest differences between networks is often between how quickly it reaches a lower threshold, such as 60/70%. A larger network may train slower, but reach the same accuracy faster since it requires fewer epochs.

The second experiment is based on the best performance of the network, trained on as much data as possible. We train for 50 epochs, without early stopping, and determine if adding a center/surround layer is better overall. If we are able to reach a higher

accuracy over a large number of epochs, it would be a definitive example of a better network.

Finally, between these experiments, we make observations on the time it takes to train (regardless of data consumed), as well as any other noticeable effects.

Results

The first experiment was simply to create a small but effective artificial neural network that performs well on the CIFAR-10 dataset. This is done so that we have a baseline to compare against. The network is small so that we can make multiple training runs, since we had limited time to run this project. The network that we built achieved good results. Trained over just 10 epochs, we were able to establish a test accuracy of anywhere around 60%. We selected the particular network configuration and initial parameters that trained to exactly 60% accuracy in 5 epochs, reset it's parameters back to initial, and added a center/surround layer. The resulting network trained to an accuracy of 63% in 10 epochs. The difference between the 63% and 60% in accuracy can only be attributed to the center/surround layer since every all parameters between the two models were the same including initialization.

While this is not a great enough improvement to signify any real difference between the two, it does indicate that adding this layer doesn't negatively affect the ability of our optimizer to train the network. We also notice that while training on cpu, it is not significantly slower to train the center/surround network.

Training for a longer duration helps both models. After 10 epochs, our small baseline model peaks at 82% accuracy, and does not improve much further. However, our model with a Center/Surround layer as the very first layer is unable to reach that accuracy, only attaining roughly 74% throughout, even after 30 epochs.

Additionally, the optimizer was more sluggish in optimizing for the loss function in the center/surround network. The loss decreased slower, and the resulting accuracy was lower.

The main benefit to our model is that it does train very fast. Evaluation of our model is also very fast, since our forward pass consists of one copy and one average call, which are costly, as well as a few transposes and views which are comparatively less

intensive. Compared to evaluating the first convolution layer, our layer takes 17 seconds to evaluate 100,000 times, compared to 7 seconds for the convolution. Since we did not implement our layer very low level in cudnn, we cannot approach the performance of predefined layers.

Discussion

Although the first experiment was very promising, after training for a longer period of time, the traditional quickly outperforms ours, while ours eventually hits a cap. This could be due to a number of reasons, but from our observations, the weights after training were very small. The resulting effect of adding this layer is very small, though it reduces overall performance. Aside from problems in our layer design methodology, this could be a problem with the initialization, or with the gradient calculation. There has been a great amount of evidence to suggest that the performance of a network is tied very closely to the initial parameters. However, since we only used one set of initial values, it may be that they favour the traditional network architecture. It may also be that since the initial values are zero, there are problems with the gradient calculation of the ReLu non-linearity.

For the model itself, it appears that, since the weights and biases of the layers are all small after training (order of 1e-4 or smaller), the resulting output is essentially dilated by a factor of 2. Since our model contains aggressive corrections for overfitting, it may be a cause for our model underperforming. Changing the initialization or simply removing the factor of 2 may help.

Our model did perform very well with small amounts of data. Whereas the traditional model eventually outperforms ours, it is no harder to train in terms of validation accuracy at every epoch than the original. Our model also has the advantage of being very fast to train. The additional layer computation wise is about as costly as two additional convolution layers. Each step in our calculation has well defined gradients, and the model itself consists only of very simple components. Our implementation in pytorch makes use of the automatic differentiation engine and no `for` loops whatsoever.

Since our time was limited, we were unable to improve the network further. Each training run of roughly 30 epochs would still take multiple hours each for both models, meaning only a few changes could be made per day. The results could be more

definitive if our layer was designed to work with CUDA, and also if we had access to cuda cores in the first place.

References

[1] D. Gupta and Dishashree, “Convolutional Neural Network (CNN) for Image recognition,” Analytics Vidhya, 13-Mar-2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/06/architecture-of-convolutional-neural-networks-simplified-demystified/. [Accessed: 17-Apr-2020].

[2] B. Carlson, “Chapter 7 - Special Senses—Vision and Hearing,” *ScienceDirect*. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128042540000077.

[3] “Perception Lecture Notes: Retinal Ganglion Cells,” *Perception Lecture Notes: Retinal Ganglion Cells*. [Online]. Available: https://www.cns.nyu.edu/~david/courses/perception/lecturenotes/ganglion/ganglion.html

Code

The networks were written using the pytorch library. The models are defined as follows using pytorch. Training and evaluating the model is straightforward, since the model makes use of the `nn.Module` class as it’s superclass. In the following code, `nn` is `torch.nn`, and `F` is `torch.nn.functional`, consistent with common import naming conventions.

The first class CenterSurround is the class that defines the center/surround layer. The layer takes two inputs, the first being a channel, and the second being the side dimension of a square input. The convention for this layer is to produce outputs that are the same dimension of the input, but that is not always necessary, we just did not need to implement it for our experiments.

```
class CenterSurround(nn.Module):
    def __init__(self, in_channels, in_shape):
        super(CenterSurround, self).__init__()
        self.in_channels = in_channels
        self.in_shape = in_shape
        self.w = nn.Parameter(torch.zeros(in_channels, in_shape,
in_shape))
        self.b = nn.Parameter(torch.zeros(in_channels, in_shape,
in_shape))
```

```
    def forward(self, x):
        batches = x.shape[0]
        channels = x.shape[1]
        L = x.shape[2]
        W = x.shape[3]
        x_unf =
F.unfold(x,(3,3),padding=1).transpose(1,2).view(batches,-1,channels,3,
3)
        tmp =
x_unf[:,:,:,1,1].transpose(1,2).view(batches,channels,L,W)
        x_unf =
x_unf.mean((3,4)).transpose(1,2).view(batches,channels,L,W)
        return F.relu(torch.add(torch.add(2*tmp, x_unf*self.w),
self.b))
```

We define a base network that represents a small, traditional CNN

```
class Base(nn.Module):
    def __init__(self):
        super(Base, self).__init__()
        self.conv1 = nn.Conv2d(3, 32*3, 3)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.bn1 = nn.BatchNorm2d(96)
        self.conv2 = nn.Conv2d(32*3, 64*3, 3)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.bn2 = nn.BatchNorm2d(192)
        self.conv3 = nn.Conv2d(64*3, 64*3, 3)
        self.fc1 = nn.Linear(64*3*4*4, 64)
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.bn1(x)
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.bn2(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 64*3*4*4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x
```

Finally, we define a new network with a center/surround layer modeling adaptive input.

```
class Net(nn.Module):
    def __init__(self, base):
        super(Net, self).__init__()
        self.cs1 = CenterSurround(3,32)
        self.base = base

    def forward(self, x):
        return self.base(self.cs1(x))
```

We also have pretrained models available.