Final Project, CS559

Robert Gale

June 2020

1 Intro

Deep neural networks (DNNs) have taken machine learning by storm. According to the Universal Approximation Theorem, with enough parameters, a single linear layer combined with a nonlinear activation function is capable of modeling basically any function. For years, the barriers for practically applying DNNs to complex problems were computational feasibility and sufficient quantities of data. With the introduction of high speed GPU hardware, as well as the application of the back propagation algorithm for training efficiency, computational feasibility is no longer a bottleneck in DNN training.

A typical DNN architecture consists of linear approximation layers with nonlinear activation functions. Convolution — inspired by biology and widely used in signal processing — is another type of layer capable of powerful latent feature extraction with a relatively tiny number of parameters.

In this project, I wanted to apply convolutional networks to speech data, treating a simplified ASR task as an image recognition problem. However, due to constraints in time for implementation, hyperparameter tuning, and training (on a CPU, since I'm not trying to implement GPU code in two weekends), my experiments with speech data didn't make it very far. Instead, I'm sharing the experiments that finished using the MNIST data set of handwritten digits. [3]

1.1 Back Propagation

Taking inspiration from experience with PyTorch and Keras, I wanted a flexible interface so my network architecture could be easily reshaped at the top level of my program.

Listing 1: Instantiating a deep neural network and a convolutional network

```
dnn = Sequential(
    Flatten(),
    Linear(784, 256),
    Sigmoid(),
    Linear(256, len(dictionary)),
)
```

```
cnn = Sequential(
    Reshape((1, 28, 28)),
    Convolution2d((3, 3), in_channels=1,
        out_channels=16, pad='same'),
    ReLU(),
    Convolution2d((3, 3), in_channels=16,
        out_channels=32),
    ReLU(),
    Flatten(),
    Linear(25088, 10),
)
```

The core concept of backpropagation involves recursively feeding an error gradient backward through a neural network to determine the gradient of each layer's weights. The key is the use of the chain rule of derivatives. As summarized by [4], the rules for back propagation can be seen in four main equations. First is the error δ^L, which is the gradient of the error coming out of the final layer of a network:

$$\delta^L = \Delta_a C \odot \sigma'(z^L)$$

where $\Delta_a C$ is the gradient of the cost function, $\odot$ is the elementwise (Hadamard) product operator, and $\sigma'(z^L)$ is the output of the final activation function. Moving backward, each other layer's gradient δ^l is found with:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

where w^{l+1}) is the weights of the following layer. These two gradients allow us to calculate the gradient with respect tolayer's weights w^l and bias b^l, which, for a linear layer with a nonlinear activation looks like:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

where a is the incoming data used in the forward pass.

My implementation generalized a bit further to allow for each layer, activation, or cost function to operate on the same input/output interface, simplifying the recursive portion of the algorithm so any layers and any activations could be stacked together arbitrarily. The key point was separating the activation functions from the linear (or other) layers. If we split the equation for δ^l into two parts, like so:

$$\delta^\sigma = \delta^{l+1} \odot \sigma'(z^l)$$
$$\delta^l = w^{l+1}\delta^\sigma$$

we can then think in terms of *modules* rather than *layers*, decoupling the implementation for an activation from the layer it activates. Listing 2 shows the implementation of linear and sigmoid layers, illustrating how the forward and backward methods have identical interfaces, using the exact same inputs and outputs.

Listing 2: Implementation of linear and sigmoid modules. Note how the forward and backward methods of each class have identical interfaces, i.e. the same kinds of inputs and outputs.

```
class Linear(Function):
    # ...
    def forward(self, X):
        return np.dot(X, self.weights) + self.
            bias

    def _backward(self, X, f_X, dy):
        dx = np.dot(dy, self.weights.T)
        dw = np.dot(X.T, dy)
        db = np.sum(dy, axis=(0)).reshape(self.
            bias.shape)
        return dx, dw, db

class Sigmoid(Function):
    def forward(self, X):
        return  1 / (1 + np.exp(-X))

    def _backward(self, X, f_X, dy):
        gradient = np.multiply(f_X, (1 - f_X))
        dx = dy * gradient
        return dx, None, None
```

With this framework in place, the recursive portion of back propagation is a piece of cake, and can be handled in a single, simple loop. It also allowed me to add modules for the convolutional portion of my network without any need to consider how it would interplay with any activation I'd like to use.

1.2 Convolutional Networks

Convolution can be understood as a weighted moving average across an input of n dimensions. The weights are referred to as a "kernel" and are the part of the convolutional layers which are trained with input data. Scanning across the input data, a kernel-sized window is multiplied by the kernel weights, then summed to provide a value for each step in the convolutional process.

$$y_{ij} = \sum_k \sum_l w_{kl} x_{i+k-1,j+l-1} + b$$

where w are the weights of the kernel, x is the input, and b is a bias term. To allow the convolutional process to "see" the edges better, and to have control over the output size of a convolutional layer, the input data is often zero-padded around the outside of the dimensions. Step size can also vary, where the window of data from the input might iterate by an index of $s_d >= 1$, with s_d known as the "stride" of each dimension.

A given convolutional layer may have several input channels. For example, photographs might have a red, green, and a blue channel. Output channels are usually more abstract conceptually, and each output channel represents a latent feature, increasing in complexity as more convolutional layers are applied.

For gradients with respect to bias b, weights w, we sum up the layer's output gradient times one, the input x, and w respectively:

$$\frac{\partial C}{\partial b} = \sum_i \sum_j \delta_{ij}^{l+1}$$
$$\frac{\partial C}{\partial w_{mn}} = \sum_i \sum_j \delta_{ij}^{l+1} \dot{x}_{i+k-1,j+l-1}$$
$$\frac{\partial C}{\partial x_{mn}} = \sum_i \sum_j \delta_{ij}^{l+1} \dot{w}_{m-i+1,n-j+1}$$

Full derivation of the equations in Section 1.2 can be found in [5].

2 Methods

The MNIST data set was split 80%/20% into a training set of 56,000 images and a test set of 14,000 images. Each image measured 28 x 28 pixels of monochromatic scans of handwritten digits, and is labeled with the numeral represented in the picture.

For my baseline fully connected (FC) model, I flattened the 28 x 28 pixels into a one-dimensional input of 784 pixels. I fed these values into a DNN built of a linear layer with 256 hidden units, followed by a sigmoid activation, then another linear layer with 256 hidden units. The 10 outputs of the second layer

Model	Train Accuracy	Test Accuracy
Two-layer FC	99.9%	97.5%
Convolutional	99.7%	**98.1%**

Table 1: Accuracy for the fully-connected (FC) and convolutional models.

were activated by the softmax function, and loss was computed with cross entropy. The FC model had 203,530 trainable parameters.

My experimental model was a convolutional network. The network begins with two 2-D convolutional layers — activated with rectified linear units (ReLU) — with a kernel size of 3 x 3 and a stride of 1. The first of these layers applied padding calculated to retain the 28 x 28 image size, and had 16 channels of output. The second had no padding and 32 channels of output. The data was then flattened into one dimension of size 25,088 and run through a linear layer, activated by softmax function, and loss computed with cross entropy. This model had 255,690 total trainable parameters. Both the experimental and baseline architectures are shown in Python code in Listing 1.

The networks were both trained with a learning rate of 0.0001, and although I mentioned a small decay during my presentation, for my final run I used a constant learn rate with no annealing. Each experiment was set up to run 100 epochs. Early stopping rules were in place, and if the test loss increased for three consecutive epochs, training stopped.

3 Results

As shown in Table 1, the convolutional network had the best test accuracy at 98.1%, as compared to the FC network at 97.5%. Loss plots are shown in Figure 1, and show that the FC network trained at a slow steady curve, as compared to the convolutional network, which had a loss that declined more steeply before a slight, bumpy increase. Both networks triggered the early stopping rules. Both models overfit a bit with train accuracies approaching 100%. The FC network stopped training on the 80^{th} epoch, and the convolutional network stopped on the 52^{nd}

4 Conclusion

The convolutional network outperformed the fully connected network by a fairly small margin, though both models performed quite well. In retrospect, I wish I had better balanced the number of parameters, as the convolutional network had more than 25% more parameters. This might explain why the convolutional network's test loss curve showed a little more of an upward trend toward the end of training.

There were a few items that would have been nice to play around with if time was less of an issue. I didn't get to explore kernel sizes or strides, or really experiment much with the number of channels, or adding more fully connected layers at the top of the network. I didn't implement any sort of weight penalty regularization, which I've gleaned can really help the early layers of convolution yield more generalizable kernels. Also, as we discussed during the presentation, I didn't apply any max pooling layers, which I suspect could have given the convolutional network a little more of an edge.

However, I doubt I really should put my CPU under any more stress for MNIST experiments, so I'll probably do any further work in a GPU-capable library!

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[3] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[4] M. A. Nielsen, "Neural networks and deep learning," 2018. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[5] P. Jaumier, "Backpropagation in a convolutional layer," 2019. [Online]. Available: https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509

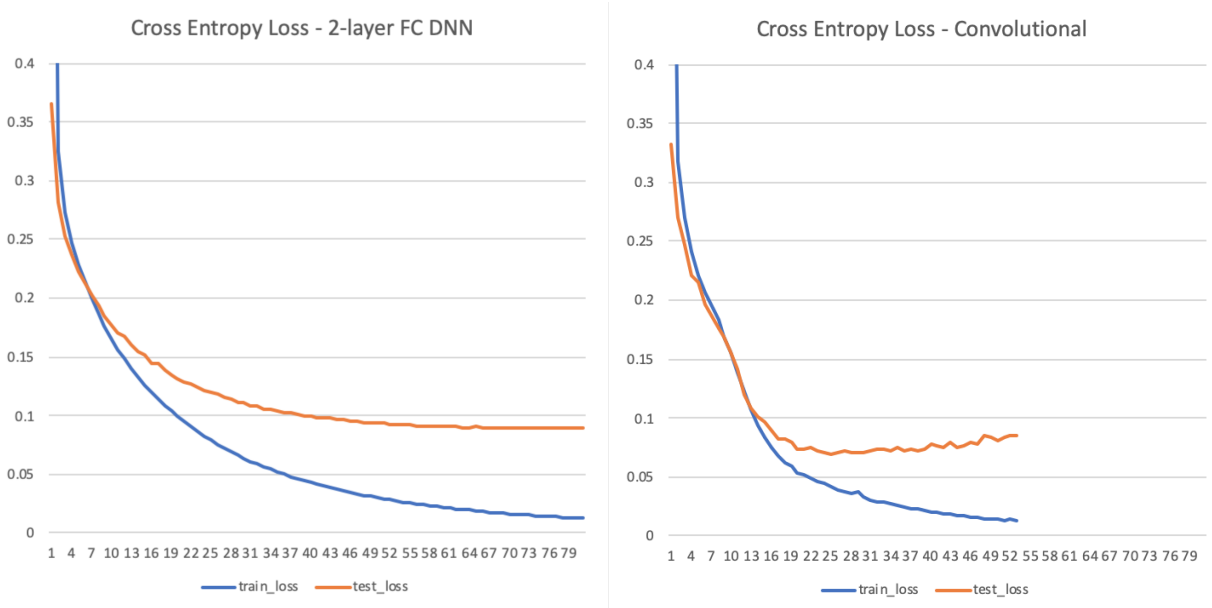

Figure 1: Cross-entropy loss for each epoch during training, for the baseline model (left) and experimental model (right)