Jodie Simkoff

Homework 6 – Deep Learning for handwritten digit classification

Introduction

Deep learning methods such as convolutional neural networks (CNNs) have proven in recent years to be incredibly powerful tools for a number of tasks, including image classification. CNNs are particularly well-suited to image recognition tasks because (depending on the implementation) they can be nearly translation invariant, recognizing images based on their features regardless of location in the image frame. One very well-studied data set in this realm is the MNIST data set, which comprises a series of handwritten digits that must be classified as a digit in the range of 0-9. In this work, we utilize the MATLAB® Neural Network Toolbox to develop, train and test a CNN for classification of the MNIST data set **(Track II)**. In particular, we are interested in the effect of architecture and hyperparameter selection on the performance of the network.

Overview of Convolutional Neural Networks

As with any neural network, a convolutional neural network consists of a sequence of layers which work together to make a prediction about some function of the inputs they receive. In classification efforts, the output of such a network is a prediction as to the class that the input data corresponds to. The fundamental units (layers) of a CNN are described here in relation to the image classification task:

- Input: The 28x28 pixel images. The MNIST images are black/white, so there is only one channel, but for colored images the input is generally has an additional dimension with 3 channels (RGB).
- Convolution layer: In this layer, a number of filters of lower dimension than the image are convolved across the image. In each local region, the output of the filter is the dot product of the filter matrix (whose entries can be thought of as the weights) and input values in that region. So, each filter maps the input to a reduced dimension output. Parameters of the Convolution layer include the size of the filters, number of filters, stride length and padding (to be discussion in the next section).
- Activation layer: In this layer, the outputs of the convolution layers are passed through an activation function such as ReLU (rectified linear unit: f(x) = max{0,x}) or a sigmoid function. In the case of classification tasks, we can think of these functions as essentially marking the outputs of the previous layer as “on” or “off.” The primay decision related to this layer is the choice of activation function.
- Pooling layer: Since the dimension of the convolution of the image with possibly many filters can be quite large, pooling layers are inserted to downsample, or reduce the dimension, of the outputs of previous layers. This works to aggregate regions of high or low activation in an image before passing to the next convolutional layer. Hyperparameters in this layer include pooling filter size and stride, and the mathematical pooling function is another design choice (max-pooling is most common but average pooling and other options are available).
- Fully-connected layer: this layer is connected to all neurons in the previous layer and computes the class score, i.e. the digit label for this case. The choice of objective function is a design choice: for multiple, discrete classes, soft-max is a popular choice.

- Softmax layer for classification: the softmax function is often used as the activation function for classification problems with multiple classes. It takes the following form:

$$\sigma(x) = P(y = j|x) = \frac{e^{w_j^T x}}{\sum_{k=1}^{K} e^{w_k^T x}}$$

The above layers are all typically found in a CNN, but their arrangement and repetition – the architecture of the network – is also up to the design of the programmer. In general, a series of convolution and ReLU layers are typically followed by a pooling layer, and this arrangement may be repeated multiple times to extract features while reducing the image dimension. Typically moving closer to the output there will be one or multiple Fully Connected layers, possibly with activation layers between them. The last fully connected layer corresponds to the final classification of the image.

As with any NN, the network is trained on a training data set and the weights updated via stochastic gradient descent with backpropagation.

Methods

For this work, the Neural Network Toolbox® in MATLAB® was employed. The goal was to explore the effect of network architecture and hyperparameter settings on performance. Specifically, performance was defined in terms of the average percent of test data correctly classified for a fixed amount of training data. Since there are a large number of design choices and hyperparameters in the CNN, we focused on the following:

1. Number of Convolution-ReLU layers
    a. More layers imply the ability to classify more features, but too many layers can lead to overfitting and increases computational burden.
2. Filter size and stride length in the max-pooling layer
    a. These parameters determine to what extent the dimension of the inputs is reduced in that layer, so the setting for these hyperparameters is often linked in part to the number of layers discussed previously. For a fixed output dimension, the filter size and stride length can also be varied such to achieve overlapping or non-overlapping layers.
3. Max-pooling vs. average pooling
    a. As the name implies, max-pooling consists of convolving a filter over the output of the previous layer and selecting the maximum single value at each filter position. Average pooling instead computes the mean value of the outputs at each filter positions.
4. Learning rate (step-size in stochastic gradient descent): constant value vs. layer-dependent learning rate
    a. The default setting in the MATLAB® toolbox is 0.01, but we will explore a few different values for this parameter. We will also investigate whether improvements are possible when changing this parameter over time, starting with a large learning rate early in training and decreasing it later in training as the network should be converging.

To explore the effects of the above listed hyperparameter and architectural specifications, each network was trained on a randomly generated subset of 2000 handwritten digit images; 500 randomly selected

images were used as validation data. Each training was allowed to run for ten epochs (ten passes over the entire data set) since this was found to work quite well while still allowing for some delineation in performance between different strategies. The training progress for one such trial is plotted in Fig. 1.

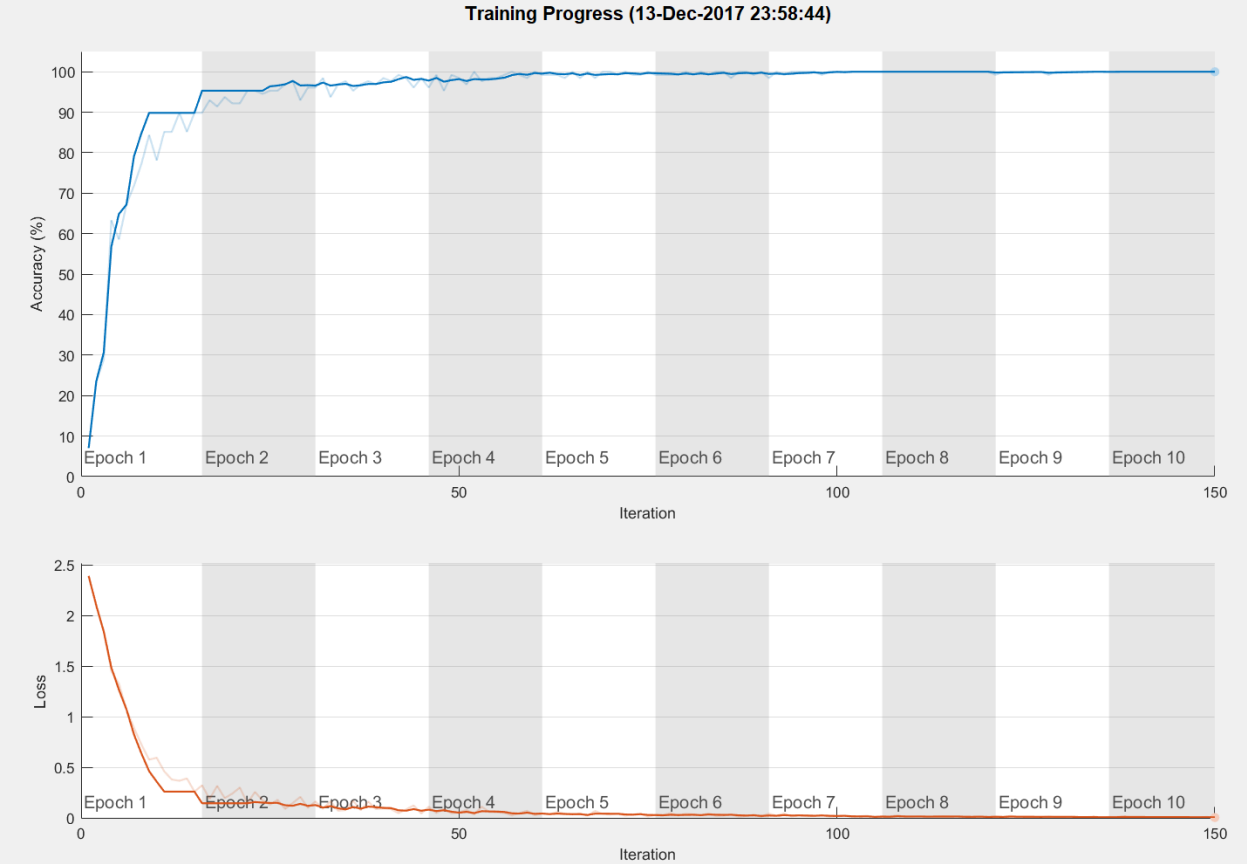

Figure 1. Example training progress: accuracy and loss-function value over ten epochs, defined as passes over the entire training data set

Results

Number of convolution-ReLU layers

For consistency, in what follows we hold a number of hyperparameters constant:

- In the convolution layers
  - Filter size = 3x3
  - Number of filters per layer = 16
  - Stride = 1
  - Zero padding = 1 pixel at each edge (so that convolution doesn't reduce dimension)
- In the max-pooling layers
  - Filter size = 2x2
  - Stride = 2

The following six configurations were considered, differing primarily in the number and location of convolution-ReLU layer pairs. The highlighted layers emphasize changes from one architecture to the next.

Architecture 1: Single convolution layer

Conv→ReLU→MaxPool→FC→Softmax

Architecture 2: A second Conv/ReLU layer

Conv→ReLU→MaxPool→ Conv→ReLU→MaxPool→FC→Softmax

Architecture 3: Adding Conv/ReLU at the end

Conv→ReLU→MaxPool→ Conv→ReLU→MaxPool→Conv→ReLU→FC→Softmax

Architecture 4: Adding conv/ReLU at the end (dimension is low, 7x7, so no additional max-pooling)

Conv→ReLU→MaxPool→ Conv→ReLU→MaxPool→ Conv→ReLU→Conv→ReLU→FC→Softmax

Architecture 5: Adding Conv/ReLU at the beginning

Conv→ReLU→ Conv→ReLU→MaxPool→ Conv→ReLU→MaxPool→Conv→ReLU→FC→Softmax

Architecture 6: Adding another Conv/ReLU in the middle

Conv→ReLU→ Conv→ReLU→MaxPool→ Conv→ReLU→ Conv→ReLU→MaxPool→
Conv→ReLU→FC→Softmax

The results are presented in Table 1 below:

	Mean validation accuracy (10 trials)
Architecture 1	92.64%
Architecture 2	96.66%
Architecture 3	97.20%
Architecture 4	96.76%
Architecture 5	97.54%
Architecture 6	97.44%

Table 1. Mean validation accuracy for networks differing in number of convolution-ReLU layers

As evidenced by the results in Table 1, even a simple network can do quite well at the MNIST digit classification task. There is a large jump in accuracy when moving from one convolution layer to two (Architecture 1 vs. all others), but the returns are diminishing after that. Contrasting the performance of Architecture 4 with that of 5 and 6, we note that adding additional convolutional layers at the beginning rather than at the end has more marginal benefit. This makes sense since it implies further filtering while the data is still higher dimensional. Once it has been max-pooled twice and reduced to a 7x7 image, the benefit of additional convolution layers as in Architecture 4 is negligible or even deleterious.

Max-pooling hyperparameters

We now use Architecture 3 from the previous trials as a base case for considering the effect of the max-pooling hyperparameters on performance. All convolution layer hyper parameters were held the same as before.

Architecture 3a: Base case

Conv→ReLU→ →MaxPool { f=2x2, s=2 } → Conv→ReLU→MaxPool { f = 2x2, s=2 } →Conv→ReLU→FC→Softmax

Dimension after first MaxPool: 14x14

Dimension after second MaxPool: 7x7

Architecture 3b: Reduce dimension, earlier

Conv→ReLU→ MaxPool { f=4x4, s=4 } → Conv→ReLU→MaxPool { f = 3x3, s=2 } →Conv→ReLU→FC→Softmax

Dimension after first MaxPool: 7x7

Dimension after second MaxPool: 3x3

Architecture 3c: Reduce dimension, later

Conv→ReLU→ →MaxPool { f=2x2, s=2} → Conv→ReLU→MaxPool { f = 4x4, s=3 } →Conv→ReLU→FC→Softmax

Dimension after first MaxPool: 14x14

Dimension after second MaxPool: 4x4

As seen in Table, 2, the 2x2 filter with stride = 2 seems to be the best combination of hyperparameters for this example. Reducing the dimension of the outputs earlier via a larger filter in the first max-pooling layer seems to have the most negative effect, reducing performance by nearly a percent compared with the base case. Keeping the same granularity in the first layer and then reducing the dimension slightly more in the second max-pooling layer leads to some apparent performance decrease but it's not nearly as severe. This fits well with the observation that adding additional convolution layers at the end does not improve performance: most of the important features for digit classification have already been extracted early in the network.

Max-pooling vs. average-pooling

We again use Architecture 3 as a base case, and now consider average-pooling rather than max-pooling. The results can be found in the last row of Table 2.

Architecture 3d: Base case but with average-pooling

Conv→ReLU→ →AveragePool { f=2x2, s=2 } → Conv→ReLU→AveragePool { f = 2x2, s=2 } →Conv→ReLU→FC→Softmax

It turns out that max-pooling works a bit better than average-pooling for this data set (about 1% better in terms of mean validation accuracy). One explanation for this observation is that the digit images consist of distinct features like lines and edges – average pooling will tend to "blend" these features, leading to lost information passed on to the next layer. If a particular filter is scanning for a feature in the middle of an 8, and this is strongly indicated in one location in the grid, but not in adjacent ones, average filtering will downplay the presence of said feature. By inspection, for the exact same training and test data in a single one run, the average-pooling approach misclassified the following digit (an 8) as a 0, while the max-pooling network got it right:

Figure 2. An example of a digit where max-pooling does better than average-pooling

	Mean validation accuracy (10 trials)
Architecture 3a	97.20%
Architecture 3b	96.16%
Architecture 3c	96.96%
Architecture 3d	96.28%

Table 2. Effect of max-pooling filter size on performance (3a-c), effect of average pooling (3d)

Learning Rate

Architecture 3a was once more used as a base case to study the effect of the learning rate, η, on performance. The default value used in all previous trials was η=0.01. For this experiment we tried values of η=0.001 and η=0.1. We also tried a varying learning rate approach, where the learning rate started at 0.01 and reduced by a factor of 0.8 every epoch so that by the last epoch the learning rate was 0.0013 (i.e. an order of magnitude reduction). The results are shown in Table 3.

	Mean validation accuracy (10 trials)
Architecture 3a, η = 0.01	96.44%
Architecture 3a, η = 0.001	95.64%
Architecture 3a, η = 0.1	95.46%
Architecture 3a, variable η	96.26%

Table 3. Effect of learning rate on performance, note all trials were run using the same randomly picked training images

Interestingly, both the increased and decreased learning rates lead to decreased performance relative to the original parameter value of η=0.01. This can be explained as follows: for a fixed amount of data (recall we limited it to 10 epochs), the lower learning rate networks don't quite converge in terms of their loss function and predictive ability. This is verified by checking the respective plots over the training horizon. So, if these networks had been allowed to train a little longer they would have caught up in performance (at the cost of additional computational effort, though).

On the other hand, the explanation for the dip in performance with a higher learning rate is less clear, since it was visually confirmed that the optimization was still converging to 100% correct prediction of the training data within the ten training epochs. The same randomly generated training and validation data were used for each of the above trials, so the implication must be that somehow the different local minima to which the η=0.01 network converged turned out to be better predictors of the validation data classes.

The variable learning rate scheme had almost no effect compare to keeping the rate fixed at its best value. That said, given the nature of the data set and the computing power available (parallelization on GPU was employed – training took under 10 seconds on average), it was very fast and easy to converge the network to 100% training data prediction. For more difficult problems or where computational resources are limited, a variable learning rate scheme could prove important.

Conclusion

In summary, the MATLAB® Neural Network Toolbox was used to train a convolutional neural network to classify the MNIST data set of handwritten digits, with very good performance achieved in short times. A number of different architectures and hyperparameter values were tested. In particular, the findings include:

- More convolutional layers toward the front of the network were relatively more beneficial than adding them to the end.
- A small network consisting of 3-4 convolutional layers is sufficient for this data set.
    - In fact, even a network with just one convolutional layer can achieve > 92% validation accuracy.
- A small max-pooling filter (2x2) is preferable to larger filters, which reduce dimension faster.
- Max-pooling does better than average-pooling for this data set.
- The best training rate was η=0.01; increasing or decreasing it caused small losses in validation accuracy.