Convolutional Neural Networks for Classifying Handwritten Digits

Favyen Bastani
fbastani@perennate.com

Zhengli Wang
wzl@mit.edu

December 5, 2014

1 Introduction

Image recognition algorithms have become pervasive: on Google Maps Street View, image recognition protects privacy by automatically blurring faces and license plate numbers [1]; robots use object recognition to carry out various tasks; meanwhile, individuals routinely employ optical character recognition and facial recognition. Over the last decade, image recognition approaches have changed dramatically. Initially, feature extraction techniques such as HOG and FAST were typically combined with support vector machine (SVM) classifiers, with research often focusing on selection of SVM kernels and defining features for specific applications. Recently, though, neural networks have received renewed attention and achieved high classification performance.

Cells in the mammalian visual cortex are able to identify local patterns in the visual field [2]. Convolutional neural networks (CNNs) aim to do the same through *convolutional and pooling layers*. While traditional neural networks are composed of densely connected layers, where each neuron in the previous layer connects to each neuron in the next layer, convolutional layers restrict filters to be applied on local sub-regions of an input image. By training these layers through back-propogation, these filters eventually detect specific features in the image that help with classification.

In this project, we implement a fast CPU-based CNN in Python using the `numpy` scientific computing library, and apply it to the MNIST handwritten digits database [3]. We evaluate the performance of various neural network parameters (including layer configuration, output layer activation function, and training algorithm) in terms of classification accuracy and speed, and compare CNN to feature extraction approaches with SVM. We find that ...

In Sections 2 and 3, we detail convolution and pooling functions, the convolutional neural network structure and training algorithm, and our implementation. In Section 4, we experiment with various neural network parameters, and in Section 5, we compare with SVM-based algorithms.

2 Neural networks

An artificial neural network (ANN) consist of neuron units that, when composed in a layered network, compute some output function from an input vector $y = h(x)$. Each neuron has a specific set of input values (which may be from the input vector of the network, or from other neurons) and a single output. The output is generally computed as $n_{w,b}(z) = f(w \cdot z + b)$ for some *activation function* f, weight vector w, and bias parameter b. The sigmoid function is often used as the activation function, i.e. $f(l) = \frac{1}{1+e^{-l}}$, as it mimics the hard threshold activation pattern of a biological neuron.

Neurons are connected to form a neural network. Typically, the network is composed of layers of neurons, where neurons in the first layer accept inputs from the input vector x, and neurons in each following layer take inputs from the outputs of the neurons in the preceding layer. Then, the network output vector y consists of the outputs from neurons in the last layer of the network. In fully connected neural networks, a neuron takes inputs from every neuron in the previous layer.

Forward propogation refers to the process of propogating some input vector x through each layer of the neural network to compute $y = h(x)$. In a fully connected neural network, this can be done using efficient matrix operations (which can take advantage of fast matrix libraries such as `numpy`). Let Z_l be the output vector of the lth neural network layer (where $Z_0 = x$) and n_{li} be the ith neuron in layer l; then, $Z_{l+1} = f(W^{\mathrm{T}} Z_l + B)$ where W_{ij} is the weight from n_{li} to $n_{l+1,j}$ and B_j is the bias parameter for neuron $n_{l+1,j}$.

Neural networks can be trained using back propogation. At a high level, we compute a cost function on the outputs from the neural network given some input and desired output, and propogate the

error (in the form of partial derivatives of the cost) starting at the last layer to the first layer. These error terms are then used to compute the derivative with respect to specific weight and bias parameters, which can then be updated via gradient descent.

2.1 ANN for multi-label classification

Artificial neural networks are often used for multi-label classification. However, the sigmoid activation function is not well suited for this purpose, where we want to assign exactly one label c for each input vector x. Instead, we use *softmax activation function*, defined as

$$f(L, i) = \frac{e^{L_i}}{\sum_j e^{L_j}}$$

Here, L is a vector containing inputs to the activation function for each neuron in the softmax layer. For back propogation, the cost function given a desired output vector o is

$$E = -\sum_i o_i \log(y_i)$$

2.2 Stochastic gradient descent

In normal gradient descent neural network training, we update weights based on performing averaging derivatives from back propogation across all training samples. Stochastic gradient descent instead randomly selects mini-batches of a predetermined size from the training samples on each iteration and updates based on the selected samples (the batch size is generally selected to yield optimal performance with matrix libraries, although in our implementation we perform back propogation separately for each sample). This increases solution sparsity, as well as convergence speed.

We implement one variation of stochastic gradient descent that significantly reduces training time in many cases. Rather than update weights directly from the computed gradient Δ, we apply a momentum that retains gradients from previous iterations. For a training rate α and momentum m, we update parameters θ as

$$v = mv + \alpha\Delta$$

$$\theta = \theta - v$$

3 Convolutional neural networks

Images, along with many other data types, exhibit strong local patterns that need to be identified in order to correctly classify images. Fully connected neural networks may ignore these patterns, or training them to identify local patterns will take a significant number of iterations. Convolutional neural networks take an alternate approach, where the localized structure is directly encoded into the structure of the neural network. In this section, we discuss the functionality of convolution and pooling layers, and strategies of incorporating them in neural networks.

3.1 Convolution

Images can often be hierarchically broken down into features and sub-features that support image recognition. For example, a flower may contain a stem and a petal, each of which consist of edges at various positions and angles. Additionally, these features may exist at various locations in the image: if an image is shifted in one direction, an image recognition algorithm should still be able to classify it in the same way.

The idea behind convolutional layers for image recognition is that a feature can be represented as an n by n *filter*, and then convolved two-dimensionally with the m by m input (which in this case would be an image). If regions of the input match up with the filter, then elements of the convolution output corresponding to those regions will generally take higher values. The output from the convolution is added with a bias term and passed through an activation function. Notably, each component of the convolution can be modeled as a neuron: the neuron takes inputs from a sub-region, multiplies the input by a weight matrix (the filter), and then applies the activation function; besides the sparsely connected input, the only other difference is that we require the weight matrix to be the same across multiple neurons. This achieves the local patterns property that we desire, since the filter is restricted to only consider sub-regions of the input, while still incorporating the weights and bias components of other neurons.

In each convolutional layer, we have k filters, each an n by n matrix. Then, for an m by m input, we get a k by $(m - n + 1)$ by $(m - n + 1)$ output.

3.2 Pooling

Pooling layers apply an aggregate function (typically either max or mean) on non-overlapping regions of the output from convolutional layers (this is done separately for each filter). These regions are often equal in size; so if the output from the convolutional layer is 30 by 30, the pooling layer may pool across each of the four 15 by 15 regions.

Pooling accomplishes two purposes. First, we reduce the amount of data that later layers need to consider, speeding up training and reducing overfitting. Second, we get a degree of translation invariance. If the input is shifted slightly, the result after pooling will still be similar. This is desirable since generally image recognition classes shouldn't change under basic transforms.

3.3 Network

Convolution and pooling can be used in isolation, without incorporation in a convolutional neural network. In that case, an autoencoder is typically used to train the filters, where the goal is to have the output layer match the input layer despite a smaller number of neurons in intermediate layers. The trained filters can then be used to extract features from both training and testing data, and then fed into a classifier.

Convolutional neural networks present a more interesting case, where the convolution and pooling operations are combined with classification into a single structure that can be trained in one process. Typical convolutional neural networks consist of one to three convolutional layers followed by fully connected hidden layers; back propogation is carried out across the entire network. When there are several convolutional layers, the first convolutional layer often becomes used by the network as an edge detector, while later convolutional layers in the network detect more and more complex features; for simple datasets, a single convolutional layer can be applied to directly identify features. The hidden layers use final pooling outputs to eventually classify the image.

4 CNN evaluation

We implement a convolutional neural network in Python using the `numpy` package, which provides fast scientific computing operations. We evaluate its performance on the MNIST dataset of handwritten digits in terms of overall training algorithm, network structure, number of filters, size of filters, number of pooling regions, and output layer activation function.

4.1 MNIST dataset

The Mixed National Institute of Standards and Technology (MNIST) dataset contains sixty thousand training images and ten thousand testing images. Its name comes from the NIST dataset that it is an adaptation of. In the NIST dataset, training and testing images were taken from two different sources, while in MNIST, the sources are merged.

Each MNIST image contains a single handwritten digit (from zero to nine). The images are all 28 pixels by 28 pixels and grayscale.

5 Comparison with SVM-based recognition

6 Conclusion