291 Programming Assignment #3

William Fedus
Bobak Hashemi
Matthew Burns
May 13, 2015

Abstract

We train two convolutional neural networks on the POFA and NimStim datasets to identify individuals and identify emotions, respectively. In order to train these neural networks, we use two separate optimization procedures, the minFunc package and stochastic gradient descent. The minFunc optimization package achieved a 95.8% on the training set and achieved a 90.0% accuracy on the partitioned test set. Stochastic gradient descent achieved a 99.4% accuracy on the training set and a 89.7% accuracy on the partitioned test set. The minFunc optimization packaged achieved a 91.7% on the training set and achieved a 78.6% accuracy on the partitioned test set. Stochastic gradient descent achieved a 91.7% accuracy on the training set and a 77.2% accuracy on the partitioned test set.

1 Introduction

In this paper, we train a multilayer neural network to identify individuals and to recognize facial expressions. We were provided two datasets, the POFA dataset, which was used to develop a facial expression system, and the NimStim dataset which was used to develop an identity recognition system. We train two neural networks by using the minFunc package provided and by using stochastic gradient descent on a cross entropy loss function.

2 Preprocessing

In both datasets, we begin by partitioning the randomly permuted examples into a training set (75% of examples) and a test set (25% of examples). Vectorizing the images by taking each pixel intensity as a dimension, an image can be viewed as a vector in some high dimensional space. The purpose of preprocessing is to reduce the dimensionality of this space to a more manageable size for injection into our neural network, thereby increase training and testing efficiency. We use a three step process to achieve this reduction.

First, we convert color images to grayscale and scale the images using the built-in image rescaling provided by matlab to 64x64 pixels. Then we apply gabor filter convolution, using a stride size of one, to the images generating a vector in 2560 dimensions for each image. Finally, these high dimensional vectors are rotated and truncated to 40 components by special matrices found using principal component analysis on the training data sets.

Gabor filters

The essential preprocessing step involves taking the convolution of the input image with Gabor filters, shown in figure 2. The filters are generated using the Gabor filter bank provided by [1]. These filters convolved with the scaled input images provide the initial 2560 features that are rejiggered via z-scoring and principle component analysis into 40 per input image.

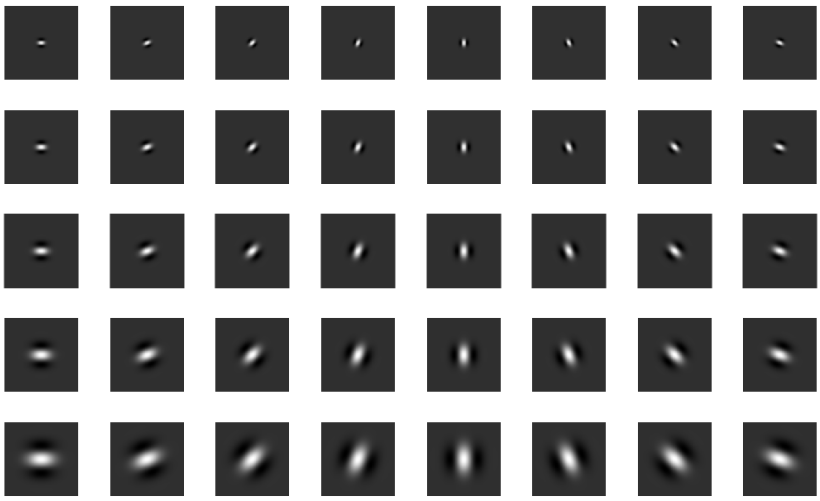

Figure 1: Gabor filters resized to 64x64 pixel images

Each scale contains 8 orientations of the Gabor filter. A single filter convolution provides N numbers which characterize the image so that the vector which contains all data from the Gabor filtering for a single scale has dimension $8N$. We partition our filtered image data into five scale separated 8N dimensional vectors per image using this process. Then, using the training data set, we construct five projectors from the $8N$ dimensional spaces to 8 dimensional subspace spanned by the principal axes of the training data set.

As a final step, we paste together the five 8 dimensional vectors for each image generating a vector in 40 dimensions per image. By saving our projection operators, we can apply the same transformation to the test data after applying the gabor filters.The Neural Network is then trained and tested using these 40 inputs.

3 Methods

3.1 Gradient Checking

Neural networks learn through minimization of a "loss" or "cost" function. The process of minimization uses the gradient of the loss function to find local minima in the weight space. If the gradient of the loss function can be derived explicitly, the fastest algorithms will implement them in code directly. Numerical gradient checking is a technique that increases the confidence in your algorithm by checking the hard-coded gradient against a numerical gradient found by evaluating the loss function for small deviations in weight space.

The minFunc optimization package comes with built in gradient checking through the function call

```
derivativeCheck(@LossFunction, function_input, max_derivative_order_to_check, ..)
```

which prints the maximal difference between the numerical gradient and the hard coded gradient (which should be an output of "LossFunction").

3.2 Training a Neural Network Model

In order to train our neural network, we first define the cost function 1.

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\} \log \frac{\exp(\theta^{(k)\top} h_{W,b}(x^{(i)}))}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} h_{W,b}(x)^{(i)}))}\right]. \tag{1}$$

With this objective in mind, we can then train our neural network through the backpropagation algorithm. Backpropagation is a recursive algorithm which computes the gradient of the cost function $J(\theta)$ of a neural network. Assuming the weights are differentiable functions, backpropagation provides a computationally efficient method to compute gradients, with complexity O(W), where W is the number of weights in the network.

The training of the network

1. **Forward Propagation**: compute network activations at all layers based on current inputs.
$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)} \tag{2}$$
$$a^{(l+1)} = f(z^{(l+1)}) \tag{3}$$
where $z^{(l)}$ is total input to each node in layer l, $W^{(l)}$ is the weight from layer l to layer $l+1$, $a^{(l)}$ is the activation of layer l, $b^{(l)}$ is the bias of layer l and f is the activation function for the network, in this case
2. **Compute Output Layer $\delta^{(n_l)}$.**
$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \tag{4}$$
where $\delta^{(n_l)}$ is total input to each node in layer l and y is the weight from layer l to layer $l+1$.
3. **Compute Hidden Layer $\delta^{(l)}$.** Recursively compute $\delta^{(l)}$ for lower layers. $\delta^{(l)}$ can be thought of as the amount that layer l is responsible for error in the output.
$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \tag{5}$$
4. **Compute Gradients.** We compute the gradient of each node in the network by substituting our previous results.
$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{6}$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}. \tag{7}$$
5. **Update Weights and Biases.** With the gradients computed, we may update the parameters of our neural network, that is the weight matrix W and the bias vector b.
$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \nabla_{W^{(l)}} J(W, b) \tag{8}$$
$$b_i^{(l)} = b_i^{(l)} - \alpha \nabla_{b^{(l)}} J(W, b) \tag{9}$$
where α is the learning rate.

Matrix provides an extremely efficient matrix manipulation tool set and so these procedures are vectorized for computational efficiency. For instance, the forward propagation equation $z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$ may be simply expressed as a matrix multiplication of the weights and the activations plus the bias vector $WA + b$.

3.3 Update SGD formula

In order to minimize the loss function 1, we use stochastic gradient descent which is outlined in Algorithm 1. This is essentially standard gradient descent, except that we choose to calculate the gradient only by a single example. This procedure is iterated until the change in the loss function is below a threshold ϵ.

However, it should be noted that since our cost function 1 is non-convex, we have no guarantees of reaching the global minimum.

Algorithm 1 Stochastic Gradient Descent

procedure STOCHASTIC GRADIENT DESCENT
    $\theta \leftarrow 0$
    for t = 0, . . . , T **do**
        Randomly permute m examples order
        for i = 0, . . . , m **do**
            $\theta_{t+1} = \theta_t + \eta_t \nabla J(\theta)$
    return θ

4 Results

We decompose our results of algorithm runtime and performance into the two data sets. In both data sets, the neural network we train generalizes well to new examples and results in commendable performance for each.

4.1 NimStim Results

The training of the NimStim dataset using minFunc took 5.4s for 1000 iterations and using stochastic gradient descent took 69.05s for 100 iterations.

The minFunc optimization package achieved a 95.8% on the training set and achieved a 90.0% accuracy on the partitioned test set. Stochastic gradient descent achieved a 99.4% accuracy on the training set and a 89.7% accuracy on the partitioned test set.

In order to track convergence, we have plotted the training accuracy rate as a function of iterations in Figure 4.1.

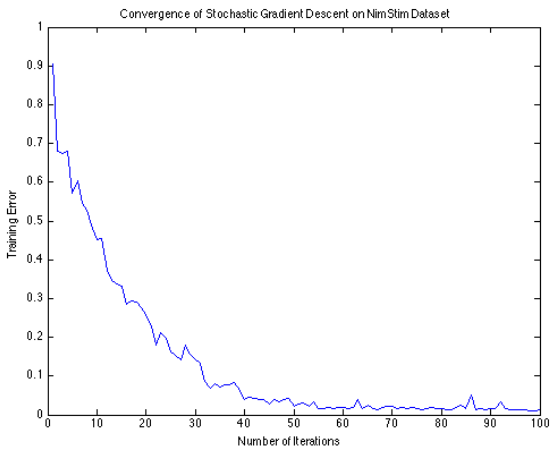

Figure 2: Training accuracy on the NimStim dataset as a function of iterations with stochastic gradient descent.

4.2 POFA Results

The training of the POFA dataset using minFunc took 2.6s for 194 iterations and using stochastic gradient descent took 19.4s for 100 iterations.

The minFunc optimization packaged achieved a 91.7% on the training set and achieved a 78.6% accuracy on the partitioned test set. Stochastic gradient descent achieved a 91.7% accuracy on the training set and a 77.2% accuracy on the partitioned test set.

In order to track convergence, we have plotted the training accuracy rate as a function of iterations in Figure 4.2.

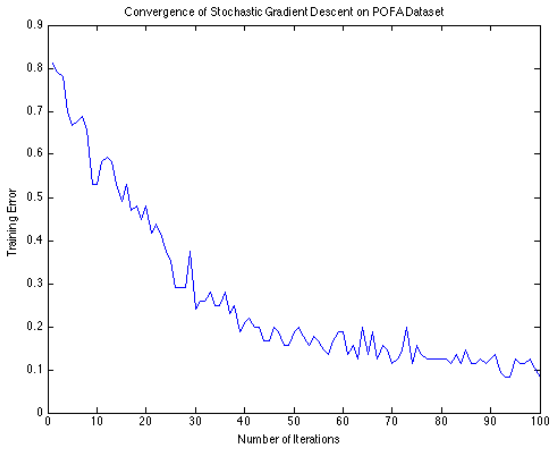

Figure 3: Training accuracy on the POFA dataset as a function of iterations with stochastic gradient descent.

5 Discussion and Conclusions

Our group found that a considerable wide range of preprocessing techniques led to good performance in the identification of individuals and the identification of emotions. In one instance, through an error, the Gabor filters we were initially using were of the wrong dimension (8×8) and as a result, had limited spatial resolution. However, interestingly, the performance on our datasets was not substantially compromised and the construction of the filters was an order of magnitude faster. This potentially indicates that simpler edge detection techniques might prove effective and assist in the training of larger networks with more convolutional units.

6 Author Contributions

William (Liam) wrote the neural network training algorithms, including forward propagation, the cost function, the backpropagation algorithm and computed the gradients of the network with respect to weights and biases. Liam interfaced the code with minFunc and also wrote the stochastic gradient descent method. Finally, Liam assisted with the preprocessing code, in particular, reading in images and gabor convolutional filters.

Bobak was responsible for the preprocessing of the image data and interfacing with the neural network framework provided.

Matthew mirrored Liam's efforts in a class framework. As a result, we have submitted two primary repositories, one for Matt and one developed jointly by Liam and Bobak.

References

[1] Mohammad Haghighat, Gabor Feature Extraction [Source Code]. Available at: http://www.mathworks.com/matlabcentral/fileexchange/44630-gabor-feature-extraction Accessed May 2nd, 2015