Neural Networks

We implemented a simple 2-layer regularized neural network, trained it using gradient descent, and used it on the provided toy datasets and MNIST handwritten digits datasets.

We choose to use the softmax formulation as described by Bishop for our loss function. Please note that although this is a different loss function than we were asked to use in our assignment description, there was a follow-up discussion on Piazza (see note @504 if not familiar) which clarified that softmax is actually the correct formulation for 1-of-K classification.

The likelihood according to softmax is:

$$p(\mathbf{t} \mid \mathbf{X}, \mathbf{w}) = \prod_{k=1}^{K} \prod_{n=1}^{N} y_k(\mathbf{x}_n, \mathbf{w})^{t_{nk}} \tag{1}$$

And taking the negative log likelihood we have our unregularized loss function:

$$l(\mathbf{w}) = -\sum_{k=1}^{K} \sum_{n=1}^{N} t_{nk} \ln y_k(\mathbf{x}_n, \mathbf{w}) \tag{2}$$

We note that $\mathbf{w}$ can be considered a vector that represents all of the weights of the neural network, but it is preferable to think of the weights as being organized into two matrices, which we denote $W^{(1)}$ and $W^{(2)}$ and explain later in the context of forward propagation. The matrix representation, however, is useful for including regularization in our final cost function, which is formulated via the Frobenius norm:

$$J(w) = l(w) + \lambda(||W^{(1)}||_F^2 + ||W^{(2)}||_F^2) \tag{3}$$

Gradient Calculation

Having defined our cost function, we are now able to describe how the gradients $\nabla_{W^{(1)}} J(\mathbf{w})$ and $\nabla_{W^{(1)}} J(\mathbf{w})$ may be calculated analytically. Before this derivation, however, we note for clarity that each of these terms are matrices whose elements are simply the partial derivative of the cost function with respect to that element itself (a scalar).

In order to calculate the gradients, we will use error backpropagation, for which we will follow Bishop's nice explanation and follow these sequence of steps:

1. Forward propagation
2. Evaluate δ for the output units
3. Backpropagation of the δ's for each hidden unit
4. Evaluate derivatives with $\frac{\partial E_n}{\partial w_{ji}}$

1. Forward propagation

With the weights represented as matrices, we can vectorize the computation of the unit activations, for example for the first layer:

$$A^{(1)} = W^{(1)} X_{aug} \tag{4}$$

Where X_{aug} is a $(D+1) \times N$ augmented matrix for the input data, for the purpose of including the bias input unit in the vectorized computation. If we consider the original input data to be X, of dimension $D \times N$, where D is the dimensionality of each sample input and N is the number of sample inputs, then we form X_{aug} by augmenting X with a $1 \times N$ vector of 1s ($X_{aug} = \begin{bmatrix} 1_{N\times 1} & | & X^T \end{bmatrix}^T$). $W^{(1)}$ is then a $M \times (D+1)$ matrix that contains all of the weights from every input to every hidden unit, except the hidden bias unit, and $A^{(1)}$ is a $M \times N$ matrix where each column vector is individually the weights for all of the unit activations, given one sample input. The "output" of each unit is computed simply by applying the sigmoid function $g()$ element-wise to the matrix $A^{(1)}$:

$$Z_{ij}^{(1)} = g(A_{ij}^{(1)}) \tag{5}$$

Where as requested in the assignment, we use the logistic sigmoid function as our sigmoid function:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

To forward propagate through the second layer to the output units, we similarly use $A^{(2)} = W^{(2)} Z_{aug}^{(1)}$, where $Z^{(1)}$ has been augmented with a vector of 1s to include the bias unit, and so we have $W^{(2)}$ of dimension $K \times (M+1)$. At this point we also note that in order to do our 1-of-K classification, we have to transform the output from the form they were given in the assigned datasets, from a scalar k to $\mathbf{e}_k$ where $\mathbf{e}_k$ is a vector of length $K = k_{max}$ and contains a 0 for every element except k, for example if $k = 3$ then we form $e_3 = [0\ 0\ 1]$. Thus we consider $\mathbf{t}$ to be an $K \times N$ matrix. Applying $y_{ij} = g(A_{ij}^{(2)})$, we have our outputs and have completed forward propagation.

2. Evaluate δ_k for the output units

We want to calculate $\delta_{nk} = \frac{\partial J_n}{\partial a_k}$. We know this won't be a function of the regularization term, as that does not depend on a_k. With our softmax formulation, we can expand Equation 2 to be:

$$l(\mathbf{w}) = -\sum_{k} t_k \big[a_k - \ln\big(\sum_{k'} \exp(a_{k'})\big)\big] \tag{7}$$

$$l(\mathbf{w}) = (-\sum t_k a_k) + \ln\big(\sum_{k'} \exp(a_{k'})\big) \sum t_k \quad (8)$$

And we nicely end up with:

$$\delta_{nk} = \frac{\partial J_n}{\partial a_k} = y_{nk} - t_{nk} \quad (9)$$

Which is simply vectorized by element-wise subtraction of $K \times N$ matrices: $\delta = \mathbf{y} - \mathbf{t}$.

3. Backpropagation of the δ's for each hidden unit

Having calculated the δ's for the output unit, we have only one step of backpropagation to perform, since we have only a 2-layer neural network. We first provide the backpropagation formula, unvectorized:

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k \quad (10)$$

where $g'(z) = g(z)(1 - g(z))$ signifies the gradient of our sigmoid function. We may start vectorizing this computation so that δ is an $M \times 1$ vector and we have:

$$\delta_{hidden} = g'(A^{(1)}) * W_{\text{no bias}}^{(2)^T} \delta_{outputs} \quad (11)$$

In the above, $*$ is used to signify element-wise multiplication, and $W_{\text{no bias}}^{(2)^T}$ is a transposed matrix of $W^{(2)}$ except without the vector that corresponds to the bias units. The above equation holds for calculating δ_{hidden} for one sample at a time, but also equivalently holds for batch computation N at a time, just by using $\delta_{outputs}$ as a $K \times N$ matrix from step 2, with $W_{\text{no bias}}^{(2)^T}$ of dimension $M \times K$, and $A^{(1)}$ of dimension $M \times N$.

4. Evaluate derivatives with $\frac{\partial E_n}{\partial w_{ji}}$

Finally, we use both δ's together with our inputs, $\mathbf{x}$, and our hidden unit outputs, $Z^{(1)}$, to calculate the gradients $\nabla_{W^{(1)}} J(\mathbf{w})$ and $\nabla_{W^{(1)}} J(\mathbf{w})$.

Following Bishop, we know that for an unregularized neural network we can compute the derivative with respect to any element via:

$$\frac{\partial J_n}{\partial w_{ji}} = \delta_j z_i \quad (12)$$

In the above, δ represents the δ at the output of the weight, and z represents the activation at the input of the weight. The above only calculates the gradient for one sample, and for one element of the matrix, but to vectorize we formulate as:

$$\nabla_{W^{(1)}} J_{unregularized}(\mathbf{w}) = \delta_{outputs} Z^{(1)^T} \quad (13)$$

$$\nabla_{W^{(2)}} J_{unregularized}(\mathbf{w}) = \delta_{hidden} X^T \quad (14)$$

where we have explicitly calculated for the gradients with respect to each of the matrices. Finally, to account for the regularization we note that due to the Frobenius norm, the derivative with respect to each element w_{ij} is simply $2\lambda w_{ij}$. We have our final gradients:

$$\nabla_{W^{(1)}} J(\mathbf{w}) = \delta_{outputs} Z^{(1)^T} + 2\lambda W^{(1)} \quad (15)$$

$$\nabla_{W^{(2)}} J(\mathbf{w}) = \delta_{hidden} X^T + 2\lambda W^{(2)} \quad (16)$$

Implementing 2-Layer Neural Network

We implemented a 2-layer neural network, and generalized it so that we can specify the number of hidden nodes as just a parameter. Many of the details of this implementation were already mentioned in the previous section so as to be able to explain the gradient calculation in its vectorized form.

In addition to the details from the previous section, we also note that a key step in completing the implementation of the neural network was to implement training via gradient descent. Essentially, this was no different than simple 1-D gradient descent, with Equation 1 as our cost function, and Equations 15 and 16 as our gradients. We expanded our gradient descent implementation from HW1 to be able to accept a list of parameters as inputs to the function, so that we could descend our cost function by adjusting the weights of both of the matrices. Another important implementation detail was that we maintained the ability to plot the objective function value vs. the number of gradient descent iterations. This feature is critical for being able to tune the gradient descent parameters (step size, maximum number of function calls).

Once a network was trained on the training data with gradient descent, then classifying new input data was as simple as performing forward propagation on the input dataset. Misclassification rates are also computed and allow us to search for optimal λ and M using cross-validation.

Stochastic Gradient Descent

The implementation of stochastic gradient descent (SGD) only requires a small modification to the existing code. In particular if we define

$$E_n(\mathbf{w}) = -\sum_k t_{n,k} \log(y_k(\mathbf{x}_n, \mathbf{w})) \quad (17)$$

we can write the objective function as

$$l(w) = \sum_n \left[E_n(w) + \frac{\lambda}{N}(||W^{(1)}||_F^2 + ||W^{(2)}||_F^2) \right] \quad (18)$$

$$= \sum_n l_n(w) \quad (19)$$

Computing the gradient of $l_n(w)$ is fundamentally no different than in the standard case. The only difference is that

we only forward/backward propagate for a single sample x_n. Thus the computation is much faster than the full gradient computation which takes N times as long, where N is the number of training examples. The other thing we need to define is the learning rate η_n. The only requirement is that the learning rate satisfy the Robbins-Munro condition. That is $\sum_n \eta_n = \infty, \sum_n \eta_n^2 < \infty$. Following the discussion on Piazza we choose

$$\eta_n = \frac{\alpha}{\beta + n^\gamma} \tag{20}$$

where $\gamma > 0.5$ ensures the Robbins-Munro condition is satisfied. The actual SGD algorithm simply updates the weights W as

$$W^{(j+1)} = W^{(j)} - \eta_j \frac{\partial l_{n_j}}{\partial W}|_{W^{(j)}} \tag{21}$$

here $n_j \in \{1, \ldots, N\}$ is the index of the training point used in the j^{th} step of the SGD algorithm. To choose n_j we randomly shuffle the values $\{1, \ldots, N\}$ and loop through them until convergence. The convergence criterion for SGD is also slightly different. The standard convergence criterion requires evaluating the full cost function $l(w)$ at each iteration. Since the whole point of SGD is to avoid doing this, we propose an alternate convergence criterion. In particular we converge when $||W^{(j+1)} - W^{(j)}||$ is sufficiently small.

Testing the Neural Network Code

We applied our neural network implementation with both batch gradient descent and stochastic gradient to the toy datasets. As can be shown in the figures (1 through 4), we were able to achieve essentially perfect fits to "toy multiclass 1" and good fits to "toy multiclass 2" considering that the data is not easily separable – our classification error rate on the validation dataset was in the range of 6 percent with a sufficient number of hidden nodes.

To find the number of hidden nodes required for batch gradient descent, we performed cross-validation by varying the number of hidden nodes over the range [1,20] and plotted the classification error rates for each of the training, test, and validation datasets (Figure 5). The shape of the curve reveals that good fits can be achieved with M=5 hidden units, although slightly better fits can be achieved with increasing the number of hidden units through 20. We attempted to apply regularization to the neural networks trained on the toy datasets, but since the unregularized neural networks already achieved a high level of performance, there was not a noticeable increase in the performance of regularized neural networks. For stochastic gradient descent, in practice it was necessary to use a relatively large step size η_n to get rapid convergence. This makes sense since we expect that the gradient of $l_n(w)$ is about $1/N$

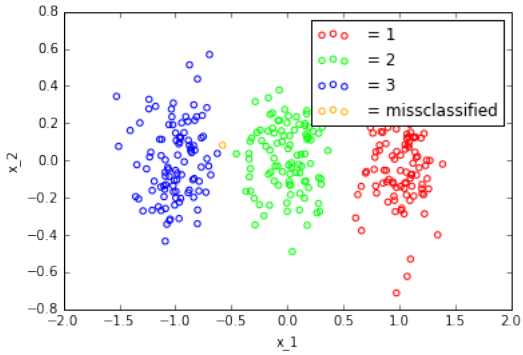

Figure 1. Classification performance of a batch-gradient-descent-trained neural network on the training data itself for "toy multiclass 1". M = 30 hidden nodes were used, with $\lambda = 0$, step size of 5e-5, and max 3,000 iterations. Shown classification error rate is 0.033 percent.

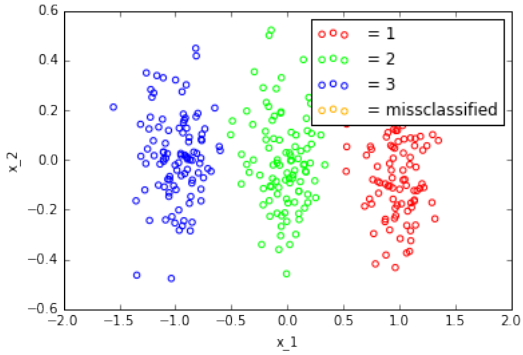

Figure 2. Classification performance of a batch-gradient-descent-trained neural network on the validation data for "toy multiclass 1". M = 30 hidden nodes were used, with $\lambda = 0$, step size of 5e-5, and max 3,000 iterations. Shown classification error rate is 0.033 percent.

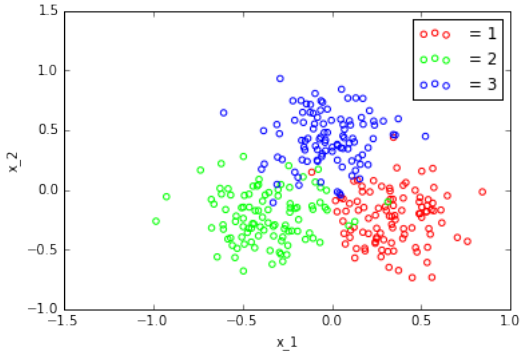

Figure 3. Classification performance of a batch-gradient-descent-trained neural network on the training data itself for "toy multiclass 2". M = 30 hidden nodes were used, with $\lambda = 0$, step size of 5e-5, and max 3,000 iterations. Shown classification error rate is 0.

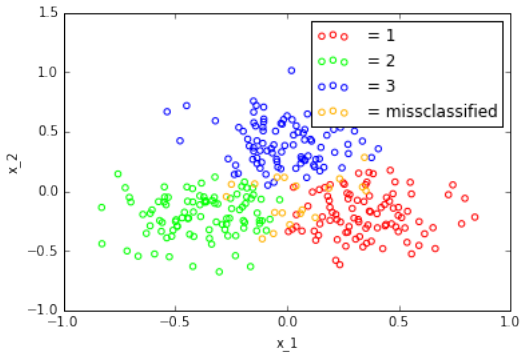

Figure 4. Classification performance of a batch-gradient-descent-trained neural network on the validation data for "toy multiclass 2" . M = 30 hidden nodes were used, with $\lambda = 0$, step size of 5e-5, and max 3,000 iterations. Shown classification error rate is 6.0 percent.

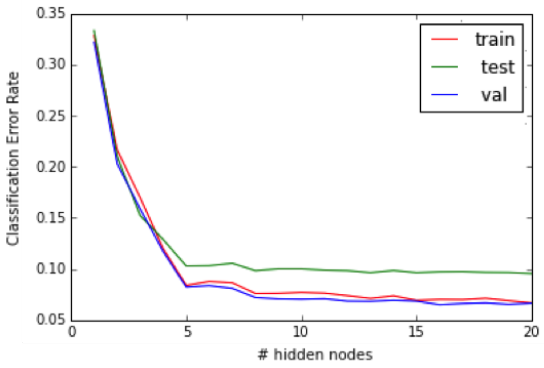

Figure 5. The classification error rate on the train, test and validation datasets for different numbers of hidden nodes. Performed with batch gradient descent, step size - 5e-5, max 3,000 iterations. Results shown are the average of ten trials.

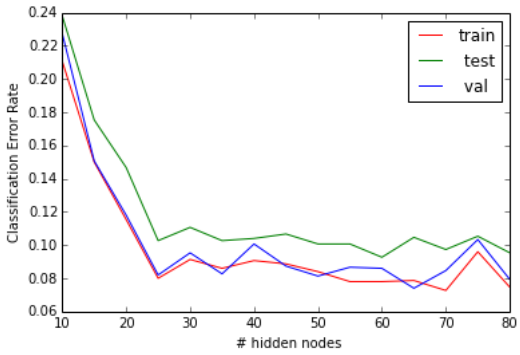

Figure 6. The classification error rate on the train, test and validation datasets for different numbers of hidden nodes

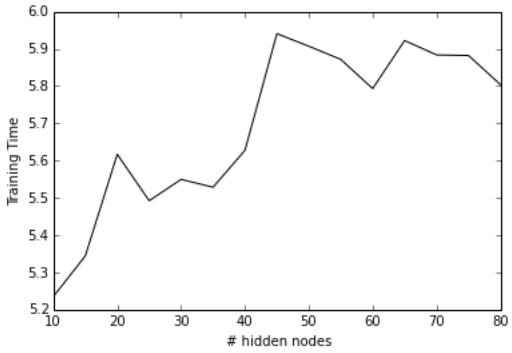

Figure 7. The classification error rate on the train, test and validation datasets for different numbers of hidden nodes

as large as the gradient of $l(w)$. For the toy dataset we used a step size of $N * 20 = 6000$ and $\lambda = 0$. In order to get good performance, we set the maximum number of iterations to $50,000$. As can be seen in Figure 6 the performance of stochastic gradient descent is comparable, just slightly worse, than that of the batch method. Cross validation shows that 60 hidden nodes provides the best performance on the test set. However, as can be seen from the figure, we already get good performance with 20 hidden nodes, and that increasing the number of hidden nodes beyond that doesn't yield large gains. However, as Figure 7 shows, the runtime increases as we increase the number of hidden nodes. This makes sense, since as we increase the number of hidden nodes, we effectively increase the size of the matrices we must multiply together to compute the backpropagation step.

Stochastic Gradient Descent

MNIST Data (Parts 5 and 6)

We applied both batch and stochastic gradient descent to the MNIST dataset.

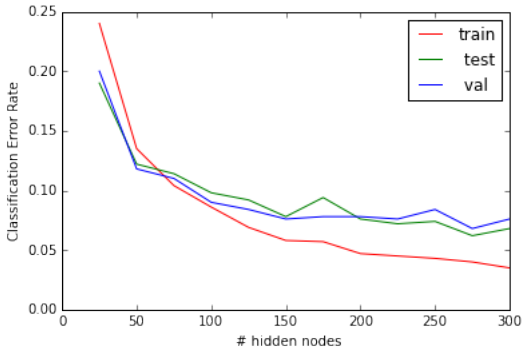

Figure 8. The classification error rate on the train, test and validation datasets for different numbers of hidden nodes for the MNIST dataset. Step size used was 1e-4, max iterations 5,000, and $\lambda = 0$.

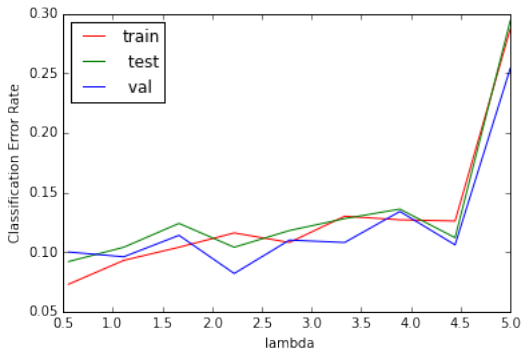

Figure 9. The classification error rate on the train, test and validation datasets for different regularization parameters, λ. Step size used was 1e-4, max iterations 5,000, and $M = 150$