Digit Classification with Multi-Layer Neural Net

Brian Wilcox and Brian Preskitt
bpwilcox@ucsd.edu and bpreskitt@ucsd.edu

Abstract

In this assignment, we built a small neural network to classify handwritten digits, obtained from the MNIST database. We experimented with 1 and 2 hidden layers between the inputs (785 image pixels) and the outputs (10 classification probabilities) – the hidden layers used hyperbolic tangent and sigmoid activation functions, while the output layer used softmax activation to discern classes. To increase speed of convergence, we employed some standard techniques such as momentum, stochastic gradient descent, and preprocessing on input data. We observed that with momentum, stochastic descent, and two relatively small hidden layers, we managed to achieve a classification accuracy of 95% on the test dataset, which is a strong improvement from the 87% accuracy achieved with logistic regression alone.

1 Basic Classification

(a) Data is read in during the testscript (`PA2_Tests.m`) in these lines:

```
%% This folder has the loading code used to access the MNIST data
addpath('mnistHelper');

%% Load the data, take only the first 20000 training points and 2000 test points
images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
tsimages = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
tslabels= loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');
```

(b) We observed that the pixel values were already mapped to the range $[0, 1]$ after reading in the data, but we did preprocess by subtracting the mean in the function `preprocess.m`:

```
function [tri, tsi] = preprocess(tri, tsi)
  % Subtract mean from training and testing data.
  tri = tri - mean(tri, 2);
  tsi = tsi - mean(tsi, 2);
```

(c) The sigmoid is implemented for our base case test (see part (e)). In particular, consider the lines in PA2_Tests.m:

```
weight_opts.layers = [256];
weight_opts.funs = {'sigmoid','softmax'};
weight_opts.distr = 'zeros';
W = initw(weight_opts,d,ncl);
```

(d) The gradient was calculated according to the backwards propagation technique derived in the individual assignment. The gradient was then estimated according to

$$\frac{\partial E}{\partial w_{ij}} \approx \frac{E(w + \epsilon E_{ij}) - E(w - \epsilon E_{ij})}{2\epsilon},$$

where E_{ij} is the matrix whose only nonzero entry is $(E_{ij})_{ij} = 1$. We used a value of $\epsilon = 10^{-3}$ and calculated $||G - G_{\text{est}}||_\infty < 6 \times 10^{-6}$, where here we define $||M||_\infty = \max_{i,j} |M_{ij}|$ (and $G = \nabla E$, with G_{est} the finite difference estimate). The results are listed below:

```
>> gradest
 Error between grads, layer 2:
 ans =     5.5845e-06
 Error between grads, layer 1:
 ans =     5.4589e-07
```

Indeed, the error is $\approx \epsilon^2$ as would be expected, so we can safely conclude that our method for calculating the gradient is accurate. The full code is found in `gradest.m` and `gradestprep.m`.

(e) We use the update rule (derived in the individual assignment) and train our classifier on the MNIST data. The test of the basic case is contained in lines 1-99 of `PA2_Tests.m`. Our test included 19000 training images, 1000 holdout images, and 2000 test images. The neural netwrok uses 1 hidden layer with 256 nodes using the sigmoid activation function.

We used a modified stopping criterion – instead of stopping immediately once the holdout error increased, we kept a running minimum of the cost function (and classification error) on the holdout set. If the minimum cost or error ever failed to decrease by a factor of 10^{-4} within 8 iterations, the test was terminated early. This stopping criterion was robust in the case of stochastic gradient descent, where the cost was likely to increase between single iterations – our method catches when the training has provably ceased to be effective, instead of when it coincidentally produces no improvement.

The results of this training are summarized in figure 1, where we see the percentage classification error on the training and test sets plotted agaainst training iterations. The convergence here is very slow, because we are not using any of the convergence acceleration techniques such as momentum or stochastic gradient descent; therefore, we end with a classification accuracy of 83.7% on the training data and 80.3% on the test data. These results *do not yet* out-perform the simple logistic regression. This is because the training process did not terminate early – indeed, it went 1024 iterations without converging, so we can chalk up the poor performance to slow convergence (by comparison, the logistic regression used momentum, so that the model was able to reach its optimum in a practical amount of time). After adding in stochastic gradient descent (which makes the tests run much faster) and momentum (which tends to speed up convergence), we are able to obtain much stronger results, which show that the *method* is correct, but the un-accelerated training method is too slow to be practical in obtaining a good model.

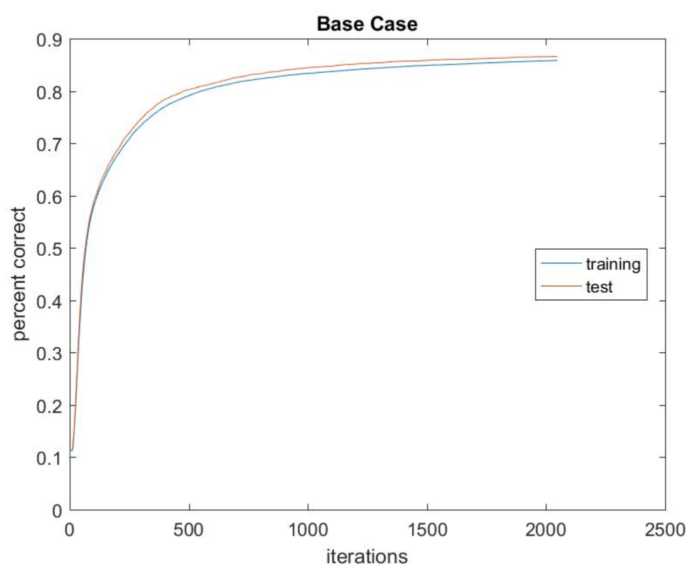

Figure 1: Classification Accuracy for 1 layer with no acceleration

2 Tricks of the Trade

In this section, we employed various 'Tricks of the Trade' in order to observe their performance on the classification task. Table 1 below shows a summary of the tricks implemented in this report.

Table 1: Tricks of the Trade Descriptions

Trick	Description
Shuffling	Stochastic Gradient Descent using 'minibatches'
Change Sigmoid	Hyperbolic tangent for sigmoid activation function
Weight Initialization	Initialize distribution with zero mean and std of 1/sqrt(fan-in)
Momentum	Add momentum term to update rule

Table 2: Results of applying tricks

Modifications	Iterations	Training Accuracy	Test Accuracy
base	2048	85.84%	86.57%
+ shuffle	2048	86.37%	86.89%
+ tanh	1105	90.61%	90.96%
+ random normal weights	990	90.7%	91.29%
+ momentum	898	95.03%	94.36%
double units	461	92.29%	92.04%
half units	990	95.54%	95.12%
two hidden layers	944	97.18%	96.18%

These changes were implemented in the order above and applied to the training data. Sections 2.1-2.4 will discuss each trick.

2.1 Shuffling

Figure 2 shows the classification error over the training process when stochastic gradient descent was employed. You can see that there appears to be a lot of noise on the curve, but the results still appear to converge to $\approx 90\%$ – this can be accounted for by considering that only random subsets of the training data were used in the training process, but "on average," the randomness of the shuffling process includes the entire dataset. Therefore, the updates to the weights reflect a *noisy* version of the true gradient, but over several iterations, the correct update rule has been enforced.

According to this curve, it would appear that the stochastic gradient descent doesn't contribute to the training performance, but what this graph doesn't reveal is the time it takes to run these experiments! In particular, by running with a batchsize of 128 training samples, each iteration was reduced in computational cost by a factor of $(20000)/(128) \approx 150$, and the computing time was reduced from roughly. Notice that this process decreases the training time without significantly hurting the model performance, because the "entropy" from having a large training set is maintained through the shuffling process – each training image is equally likely to be selected by the shuffling process, and the empirical results bear out the claim that the (average) convergence isn't hurt by the subsampling. However, we still see that after 2000 iterations, we still only have $\approx 85\%$ accuracy on the training set. This is expected, since we are only throwing away information for the sake of computational speed.

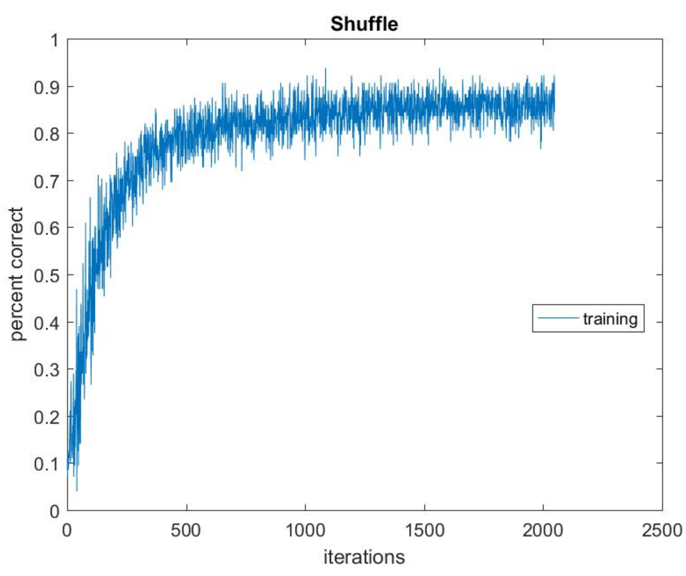

Figure 2: Accuracy with stochastic shuffling (batch-size of 128)

2.2 Changing Activation Function

Specifically, here we use the hyperbolic tangent $g(a_j) = 1.7159tanh(\frac{2}{3}a_j)$. This formulation is preferred because it is symmetric about the origin. In MATLAB, we vectorize the computation as follows

```
b = 2/3;
a = 1.7159;
y = a*tanh(b*A);
```

For back propagation, we find the derivative:

$$
\begin{aligned}
\frac{\partial g}{\partial a_j} &= (1.7159)(\frac{2}{3})(1 - tanh^2(\frac{2}{3}a_j)) \\
&= (1.7159)(\frac{2}{3}) - (1.7159)(\frac{2}{3})tanh^2(\frac{2}{3}a_j) \\
&= (\frac{2}{3})(1.7159 - \frac{g^2}{1.7159})
\end{aligned}
$$

We vectorize the computation as we compute the δ_j :

```
b = 2/3;
a = 1.7159;
delta = (a * b) * (1 - (y.*y) / a^2) .* (W.' * delta);
```

Using this activation function, we were able to achieve improved performance and learning speed on both the training and test data set than when using the standard logistic sigmoid. Figure 3 shows the classification accuracy of the training. An accuracy after 1105 iterations was 90.96%, an improvement of 4% and half the iterations from previous test with the sigmoid and batch shuffling. We can account for this by considering that the tanh function is specifically designed to offer steeper slopes (and faster gradient descent) than the sigmoid function, especially when the data is preprocessed to have 0 mean. In this case, we (unsurprisingly) continue to observe the noisy convergence curve because of the stochastic elements, but the convergence is faster and the model therefore appears more accurate.

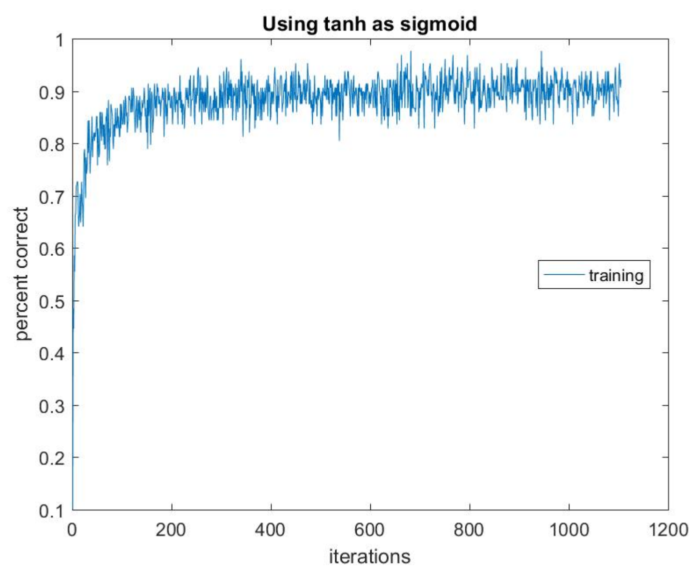

Figure 3: Accuracy replacing sigmoid with tanh activation function

2.3 Weight Initialization

In this stage, we initialize the weights according to normal distribution of mean 0 and variance 1/(fan-in), where fan-in at each layer is the number of inputs to each node. This is meant to optimize where the activation functions are hit, such that we will observe strong gradients that get the learning process moving. Results are summarized in figure 4. Indeed, it appears that the classification accuracy gets “kickstarted” towards its optimum more quickly, so that after 990 iterations the convergence is stronger. We observe a small increase in percent correct classification on the test set.

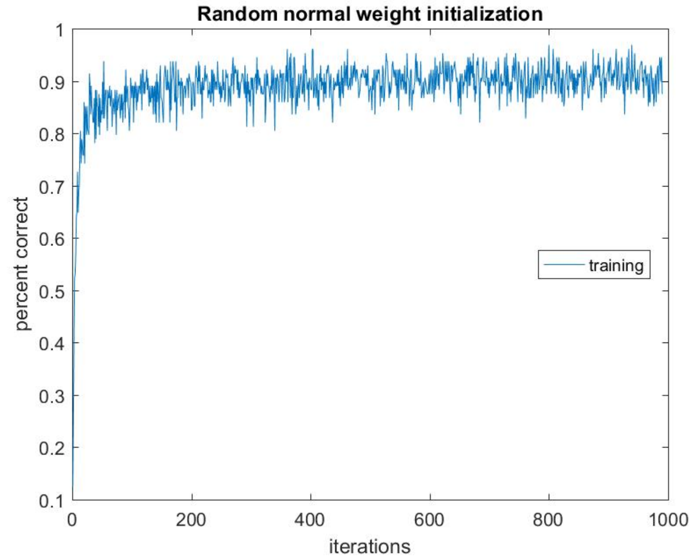

Figure 4: Accuracy with weights initialized by random normal distribution

2.4 Momentum

We used momentum to improve the convergence rate as in the first assignment, using the formulation

$$w^{n+1} = w^n - \eta \nabla E + \mu(w^n - w^{n-1}).$$

This magnificently improved the convergence rate as before, as seen in figure 5. This ultimately achieved an accuracy of 94.4% on the test set, converging in under 1000 iterations!

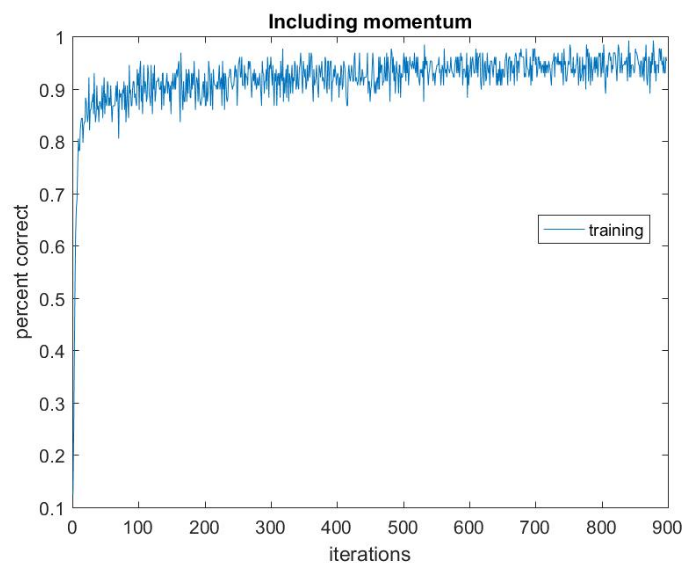

Figure 5: Accuracy adding momentum term (mom = 0.9)

3 Adjusting Topology

In this section, we are going to keep all of the 'Tricks of the Trade' from section 2, but this time we'll explore the affects of changing the topology of our neural network. We do this by changing the number of hidden units or by adding another hidden layer to the network.

3.1 Changing Number of Hidden Units

We chose 256 hidden units as our intitial size for the hidden layer. By doubling and cutting in half the number of nodes, we show in figure 6 and figure 7 its affects. Apparently, with our current architecture, doubling the amount of hidden units to

512 greatly reduced the number of iterations to converge but at the cost of a lower training and test accuracy. Conversely, reducing the hidden units to 128 increased the performance in error but with an increase in iterations (990) than before the network change.

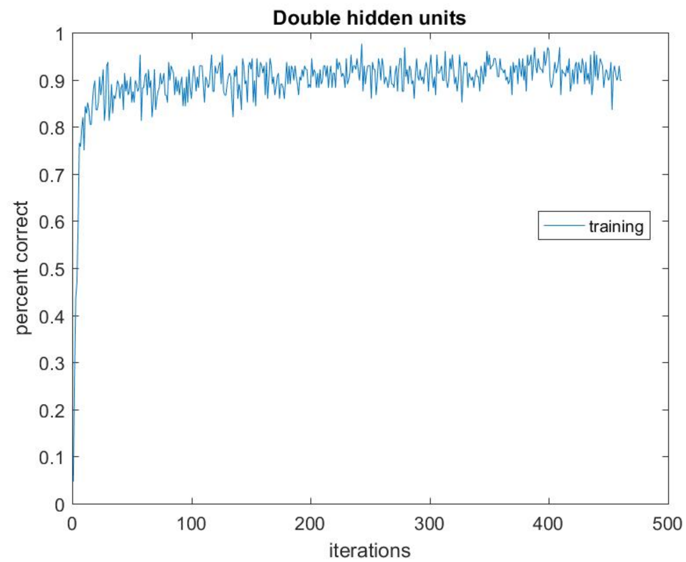

Figure 6: Accuracy with double number of hidden units - 512

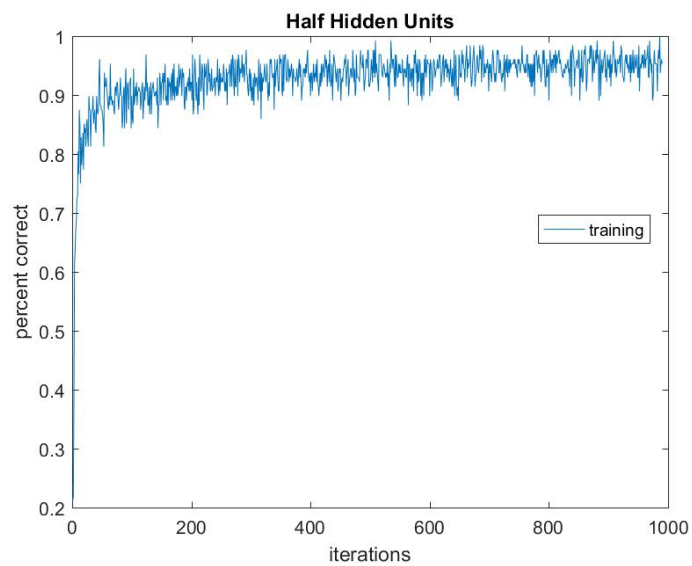

Figure 7: Accuracy with half number of hidden units - 128

In some additional tests, we saw that as the number of units became too small, the performance degraded. At larger units, the performance also declined, which suggests appropriate tuning of the hidden layer units between the number of input and output layer nodes.

3.2 Adding a Second Hidden Layer

The single layer network architecture employed thus far has shown reasonable results in classification error, down to 5%error. Here we compare this to an architecture with two hidden layers, both of the same size. For the sake of comparison, we used 204 units in each layer, corresponding to approximately the same number of weight parameters as our single layer with 256 units.

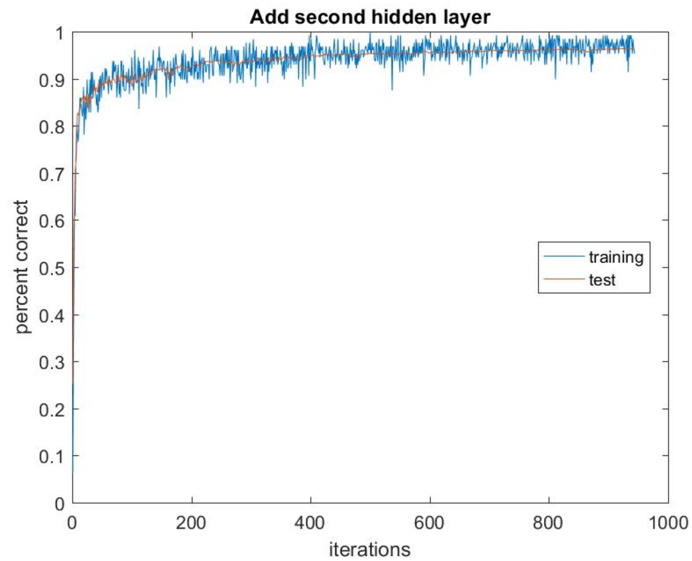

Figure 8: Accuracy with two hidden layers of same size (both 204 units)

As seen in figure 8, there is an improvement in classification, receiving the best performance of all the tests in this report, 96.14%, with only a small increase in the number of iterations (898 vs 944). This suggests that a topology with multiple hidden layers may be more optimal for this task of digit classification.

4 Summary

In this report, we explored multi-layer neural networks and 'Tricks of the Trade' to improve classification performance on the MNIST database of handwritten digits. Starting with a base case of a single hidden layer of 256 units, we incrementally added methods (shown in Table 1) to improve the network's performance. Finally, we compared our network architecture with changes to its topology.

The base network began with a classification accuracy of 80.30% on the testing data set. Shuffling a batch-size of 128 examples greatly reduced the computation time as well as improved the error with enough iterations. Replacing the activation function with the hyperbolic tangent also improved the performance along with initializing the weights according to a normal distribution with a standard deviation based on the number of input units to the next layer. The last trick, adding a momentum term, provided the fastest convergence with 898 iterations and best performance, 94.35%, before adjusting the network topology.

Exploring changes to network topology showed that our performance improves with a lower number of hidden layer units (128), yet suffers at double the number of units from the base case. Adding an additional hidden layer, however, with 204 units per layer, improved classification accuracy even further to 96.18% with relatively small additional iterations required. This was the best performance found in this report.

5 Contributions

Brian Wilcox was responsible for the code of the activation functions and the gradient calculation for backpropogation. He also wrote the classifier function to calculate the accuracy of the data as well as the functionality to create and initialize weights for an arbitrary number of hidden networks and units with desired distribution. Wilcox wrote and implemented the scripts to generate the test cases and plots used in this report. He contributed to minor modifications and debugging of Preskitt's code. He and Preskitt both contributed to the various segments of this report.

Brian Preskitt devloped the initial architechture for storing the net model in a single data structure. He helped update the testing procedure to accomodate the new model. He worked on developing vectorized computations for forward and backward propagations. Preskitt and Wilcox both contributed to miscellaneous code, narrowing down test parameters for final results, and writing the report.

6 Appendix : Code

```
function y = actfun(A, type)
  % Applies the given activation function to the supplied data.
  % A = matrix of inputs to the activation function.
```

```
  % type = string defining the activation function type.
  % Returns:
  % y = fcn values after applied to A.

  % You can do softmax...
  if strcmp(type, 'softmax')
    y = exp(A);
    y = y ./ sum(y, 1);

    % ...or sigmoid...
  elseif or(strcmp(type, 'sig'), strcmp(type, 'sigmoid'))
    y = 1 ./ (1 + exp(-A));

    % ...or that special tanh.
  elseif strcmp('tanh',type)
    b = 2/3;
    a = 1.7159;
    y = a*tanh(b*A);
  end

function delta = actgrad(y, W, delta, type)
  % Calculates the gradient of the activation function for the layer at hand.
  % y = results of the activation function of lower layer
  % W = weight matrix of present layer
  % delta = "slope" of previous (higher) layer
  % type = type of activation function

  % Derivation for sigmoid gradient...
  if or(strcmp(type,'sig'), strcmp(type, 'sigmoid'))
    delta = (y .* (1-y)) .* (W.' * delta);
    % ...and for tanh...
  elseif strcmp(type,'tanh')
    b = 2/3;
    a = 1.7159;
    delta = (a * b) * (1 - (y.*y) / a^2) .* (W.' * delta);
  end
  % Truncate to throw away the bias node (which has no gradient)
  delta = delta(2:end, :);
end

function etap = anneal(eta, ann, annpar, it)
  % This function returns the "annealed" learning parameter etap (eta prime)
  % eta     = base learning parameter
  % ann     = annealment mode ('exp', 'hyp', or 'none')
  % annpar  = parameter controlling annealment strategy
  % it      = current iteration number

  % hyperbolic annealment
  if strcmp(ann, 'hyp')
    etap = eta / (1 + (it - 1) * annpar);
  % exponential annealment
  elseif strcmp(ann, 'exp')
    etap = eta * annpar^(it - 1);
  % no annealment
  else
    etap = eta;
  end

function [G] = backprop(WX, W, x, labs)
  % Does back propagation on the neural net.
  % WX = Intermediate calculations (weighted sums and activation results)
  % W = Neural net model
  % x = input to the net
  % labs = correct labels

  % Size parameters and basic initializes
```

```
  L = size(W, 1);
  Npts = size(WX{1, 1}, 2);
  G = cell(size(W, 1), 1);

  % Isolate a vector of activation function names.
  actfuns=cell(L,1);
  for l = 1 : L
    actfuns{l} = W{l, 2};
  end

  % Calculate delta for output layer...
  a = WX{L, 1};
  y = WX{L, 2};
  delta = loglikegrad(y, labs, actfuns{L});

  % For each layer, traversing backwards...
  for l = L : -1 : 1

    % Except for the first layer, "inputs" were the activation results
    % of the previous layer.
    if l > 1
      y = WX{l - 1, 2};
      % For first layer, "inputs" were actual inputs
    else
      y = x;
    end

    % Gradient of weights is outer product of delta and inputs to layer
    % according to derivation
    G{l} = delta * y';

    % Calculate gradient of activation function, delta for next layer.
    if l > 1
      delta = actgrad(y, W{l, 1}, delta, actfuns{l-1, :});
    end
  end
end
```

```
function [class, pe] = classifier(W, x, labs)
  % Predicts classes of x data, gives percent wrong.
  % W = net model
  % x = input data
  % labs = correct labels
  % Returns
  % class = class predictions
  % pe = percent error

  % Straight up, just run a forward prop...
  [WX, loss] = forwardprop(W, x, labs);
  N = length(labs);
  J = size(WX,1);

  % ...then do a 'one hot' to figure out prediction
  [~, class] = max(WX{J,2}', [], 2);
  pe = 1 - sum(class(:) == labs) / N;
end
```

```
function E = costfun(wx, y, type, labs)
  % Calculates the cost function of the neural net.
  % wx = weighted sums going into the output layer
  % y = results of output layer activations function
  % type = type of activation function
  % labs = data labels

  % So far... only handles softmax activation :)
  if ~strcmp(type, 'softmax')
```

```
      return
    else
      % Just sum the logs of the properly labeled softmax values!
      N = size(wx, 2);
      index = sub2ind(size(wx), labs', 1:N);
      E = -sum(log(y(index)));
    end
```

```
function [WX, loss, pe, class] = forwardprop(W, x, labs)
  % Performs forward propagation on the neural net defined by W.
  % x is a matrix containing the input vectors
  % labs is a vector of the labels
  % Returns:
  % WX, intermediate computations of weight maps and activation
  % functions.
  % loss = loss function (log likelihood, etc.) on the output nodes
  % pe = percent error (classification error)
  % class = classes (by "one-hot encoding")

  % Get some size parameters.
  % Initialize the cell structure of WX.
  L = size(W, 1);
  WX = cell(size(W));
  N = size(x, 2);

  % 'y' is the input to the linear map at each level;
  % on the first layer, this is the actual inputs.
  y = x;

  % For each layer...
  for l = 1 : L
    % Apply the weight map
    WX{l, 1} = W{l, 1} * y;
    % Apply the activation.  If not on the output layer, add a bias node.
    if l < L
      y = [ones(1, N); actfun(WX{l, 1}, W{l, 2})];
    else
      y = actfun(WX{l, 1}, W{l, 2});
    end

    % Store the results of activation function (for use elsewhere)
    WX{l, 2} = y;
  end
  % Calculate the loss after all is said and done.
  loss = costfun(WX{L, 1}, WX{L, 2}, W{L, 2}, labs);

  % Usual classification procedure with percent error.
  % (class = argmax of net results)
  M = length(labs);
  J = size(WX,1);
  [~, class] = max(WX{J,2}', [], 2);
  pe = 1 - sum(class(:) == labs) / M;
end
```

```
% Do the calculations to get a candidate for gradient...
gradestprep;

% Define a perturbation
pert = 1e-3;

% Grab all size parameters, appropriate matrices for testing
% second layer.
L = size(W, 2);
[m, n] = size(W{L, 1});
wx = WX{L, 1};
y = WX{L-1, 2};
```

```
w = W{L, 1};
af = W{L, 2};

% For each entry in the weight matrix...
for i = 1 : m
  for j = 1 : n
    % Perturb the weights...
    wxp = wx;
    wxp(i, :) = wxp(i, :) + pert * y(j, :);
    wxm = wx;
    wxm(i, :) = wxm(i, :) - pert * y(j, :);
    wp = w;
    wp(i, j) = wp(i, j) + pert;
    wm = w;
    wm(i, j) = wm(i, j) - pert;
    % ...calculate the resulting loss...
    yp = actfun(wxp, af);
    ym = actfun(wxm, af);
    lp = costfun(wxp, yp, af, labs);
    lm = costfun(wxm, ym, af, labs);
    % ...estimate derivative.
    g{L}(i, j) = (lp - lm) / (2 * pert);
  end
end

% Display the results
disp('Error between grads, layer 2:');
max(abs(g{L} - G{L})(:))

% Glean data to test the first layer.
[m, n] = size(W{1, 1});
wx = WX{1, 1};
y = dats;
af1 = W{1, 2};
af2 = W{2, 2};
w = W{2, 1};

% Repeat the previous process...
for i = 1 : m
  for j = 1 : n
    wxp = wx;
    wxp(i, :) = wxp(i, :) + pert * y(j, :);
    wxm = wx;
    wxm(i, :) = wxm(i, :) - pert * y(j, :);
    yp = [ones(1, N); actfun(wxp, af1)];
    ym = [ones(1, N); actfun(wxm, af1)];
    ap = w * yp;
    am = w * ym;
    yp = actfun(ap, af2);
    ym = actfun(am, af2);
    lp = costfun(ap, yp, af2, labs);
    lm = costfun(am, ym, af2, labs);
    g{1}(i, j) = (lp - lm) / (2 * pert);
  end
end

% Display results.
disp('Error between grads, layer 1:');
max(abs(g{1} - G{1})(:))
```

```
%% Generates test cases and plots for Programming Assignment 1
clear all

%% This folder has the loading code used to access the MNIST data
addpath('mnistHelper');

```

```
%% Load the data, take only the first 20000 training points and 2000 test points
images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
sim = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
slb = loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');

% Cut down train and test data
trkeep = 1000;
tskeep = 500;
images = images(:, 1:trkeep);
labels = labels(1:trkeep);
sim = sim(:, 1:tskeep);
slb = slb(1:tskeep);

% Throw out all but some digits!
digs = 0:9;
inds = logical(any(labels == digs, 2));
labs = labels(inds);
dats = images(:, inds);
[~, labs] = max((labs == digs), [], 2);
N = numel(labs);
dats = [ones(1, N); dats];

% Size parameters...
d = size(dats, 1);              % Dimension of input data
ncl = max(labs);                % Number of classes (distinct outputs)
jhid = 16;                      % Number of hidden nodes.x

% Initialize the net model values...
W = cell(2, 2);
W{1, 1} = randn(jhid, d);
W{1, 2} = 'sigmoid';
W{2, 1} = randn(ncl, jhid + 1);
W{2, 2} = 'softmax';

% Calculate the gradient!
[WX, loss] = forwardprop(W, dats, labs);
G = backprop(WX, W, dats, labs);
```

```
function [tri,trl,sim,slb,N,Nt,hoi,hol] = importimages(images, labels,sim,slb, trkeep,
    tskeep)


images = images(:, 1:trkeep);
labels = labels(1:trkeep);
sim = sim(:, 1:tskeep);
slb = slb(1:tskeep);

%% Throw out all but some digits!
% digs = [2, 3];
digs = 0:9;
inds = logical(any(labels == digs, 2));
labs = labels(inds);
dats = images(:, inds);
[~, labs] = max((labs == digs), [], 2);
N = numel(labs);
dats = [ones(1, N); dats];
tri = dats;
trl = labs;

% Same for test data
tst = logical(any(slb == digs, 2));
sim = sim(:, tst);
slb = slb(tst);
[~, slb] = max(slb == digs, [], 2);
Nt = numel(slb);
```

```
sim = [ones(1, Nt); sim];

% Develop a hold out set
hopct = 0.05;
hoind = round((1 - hopct) * N);
hoi = dats(:, hoind : end);
hol = labs(hoind : end);
tri = dats(:, 1 : (hoind - 1));
trl = labs(1 : (hoind - 1));

end
```

```
function W = initw(weight_opts,nin,nout)
  %% Initialize the net model based on certain parameters.
  %% weight_opts contains the sizes of the intermediate layers,
  %%   as well as the types of activation functions used
  %%   and the distribution from which initial weights shall be drawn.
  %% nin = number of input layer values (dimension of input data)
  %% nout = number of potential classes

  % Work the "meta-dimension" into the layer sizes.
  layers = [nin, weight_opts.layers, nout];
  funs = weight_opts.funs;
  distr = weight_opts.distr;

  % If you're using normally distributed weights... go for it
  if strcmp(distr,'randnorm')
    for i=1:length(layers)-1
      if i ~= 1
        W{i,1} = normrnd(0,1/sqrt(layers(i)),layers(i+1),layers(i) + 1);
      else
        W{i,1} = normrnd(0,1/sqrt(layers(i)),layers(i+1),layers(i));
      end
      W{i,2} = funs{i};

    end
    % Or initialize to zeros
  elseif strcmp(distr,'zeros')
    for i=1:length(layers)-1
      if i ~= 1
        W{i,1} = zeros(layers(i+1),layers(i) + 1);
      else
        W{i,1} = normrnd(0,1/sqrt(layers(i)),layers(i+1),layers(i));
      end
      W{i,2} = funs{i};

    end
    % Or distributed randomly...
  elseif strcmp(distr,'uniform')
    for i=1:length(layers)-1
      if i ~= 1
        W{i,1} = rand(layers(i+1),layers(i) + 1);
      else
        W{i,1} = normrnd(0,1/sqrt(layers(i)),layers(i+1),layers(i));
      end
      W{i,2} = funs{i};
    end
  end
end
```

```
function upd = l1upd(W)
  % Returns the gradient of the L1 norm of w

  N= size(W,1);
  upd = cell(N,1);

```

```
  for i=1:N

  upd{i} = (W{i,1} > 0) - (W{i,1} < 0);

  end
```

```
function upd = l2upd(W)
  % Returns the gradient of the L2 norm of w

  N = size(W,1);
  upd = cell(N,1);

  for i = 1:N

  upd{i} = 2 * W{i,1};

  end
```

```
%% Generates test cases and plots for Programming Assignment 1
clear all
close all
%% This folder has the loading code used to access the MNIST data
addpath('mnistHelper');

%% Load the data, take only the first 20000 training points and 2000 test points
images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
sim = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
slb = loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');

% Cut down to 20000 train, 2000 test
trkeep = numel(labels);
tskeep = numel(slb);
% trkeep = 20000;
% tskeep = 2000;
images = images(:, 1:trkeep);
labels = labels(1:trkeep);
sim = sim(:, 1:tskeep);
slb = slb(1:tskeep);
[images, sim] = preprocess(images, sim);

% Throw out all but some digits!
% digs = [2, 3];
digs = 0:9;
inds = logical(any(labels == digs, 2));
labs = labels(inds);
dats = images(:, inds);
[~, labs] = max((labs == digs), [], 2);
N = numel(labs);
dats = [ones(1, N); dats];
tri = dats;
trl = labs;

% Same for test data
tst = logical(any(slb == digs, 2));
sim = sim(:, tst);
slb = slb(tst);
[~, slb] = max(slb == digs, [], 2);
Nt = numel(slb);
sim = [ones(1, Nt); sim];

% Develop a hold out set
hopct = 0.05;
hoind = round((1 - hopct) * N);
hoi = dats(:, hoind : end);
hol = labs(hoind : end);
```

```
tri = dats(:, 1 : (hoind - 1));
trl = labs(1 : (hoind - 1));

d = size(tri, 1);
ncl = max(trl);

weight_opts.layers = [256];
weight_opts.funs = {'sigmoid', 'softmax'};
weight_opts.distr = 'zeros';

W = initw(weight_opts,d,ncl);

batchsize = 128;

% Set the data and parameters for the experiment
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
p.homin = 0.03;
% p.tsi = sim;
% p.tsl = slb;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
p.annpar = 1 / 256;
p.maxit = 2048;
p.esi = 3;
p.mom = 0.9;
p.l1 = 0;
p.l2 = 0.0005;
p.batchsize = batchsize;
p.hodelmin = 0.0001;
p.winit = W;
p = netregdefs(p);

x = p.tri;
labs = p.trl;

[w, out] = netreg(p);
iters = 1:length(out.trc);

[~, tpe] = classifier(w, x, labs);
[~, spe] = classifier(w, sim, slb);
```

```
function delta = loglikegrad(y, labs, actfun)
  %% Calculate the gradient of the loglikelihood at the end
  %% y = activation results of last layer
  %% labs = correct labels of data
  %% actfun = activation function type for last layer

  actfun = deblank(actfun);
  % Right now... only using softmax at end.
  if or(strcmp(actfun,'soft'), strcmp(actfun, 'softmax'))
    % Calculate the gradient according to derivation.
    N = size(y, 2);
    index = sub2ind(size(y), labs', 1:N);
    C = size(y, 1);
    t = (labs == (1 : C))';
    delta = y - t;
    return;
  end
```

```
function makeplots(iters,out,titlep,filename)

figure
```

```
plot(iters,1-out.tre)
hold on
if isfield(out,'tse')
    plot(iters,1-out.tse)

end

xlabel('iterations')
ylabel('percent correct')
title(titlep)
legend('training','test','Location','Best')

saveas(gcf,filename,'jpg')




end
```

```
function opars = netregdefs(params)
  % Enforce default values for test parameters.

  opars = params;
  d = size(params.tri, 1);
  if ~isfield(params, 'eta')
    opars.eta = 0.1;
  end
  if ~isfield(params, 'l1')
    opars.l1 = 0;
  end
  if ~isfield(params, 'l2')
    opars.l2 = 0;
  end
  if ~isfield(params, 'hoi')
    opars.hoi = [];
    opars.hol = [];
    opars.homin = -1;
  elseif ~isfield(params, 'homin')
    opars.homin = 0.01;
  end
  if ~isfield(params, 'tsi')
    opars.tsi = [];
    opars.tsl = [];
    opars.tsmin = -1;
  elseif ~isfield(params, 'tsmin')
    opars.tsmin = 0.01;
  end
  if ~isfield(params, 'batchsize')
    opars.batchsize = d;
  end
  if ~isfield(params, 'winit')
    if ~isfield(params, 'ncl')
      opars.ncl = max(params.trl);
    end
    opars.winit = initw(p.netpars, d, opars.ncl);
  else
    opars.ncl = size(params.winit, 2);
  end
  if ~isfield(params, 'maxit')
    opars.maxit = 16;
  end
  if ~isfield(params, 'ann')
    opars.ann = 'none';
  end
  if ~isfield(params, 'annpar')
```

```matlab
    opars.annpar = 0;
  end
  if ~isfield(params, 'mom')
    opars.mom = 0;
  end
  if ~isfield(params, 'esi')
    opars.esi = inf;
  end
  if ~isfield(params, 'delwmin')
    opars.delwmin = -1;
  end
  if ~isfield(params, 'trmin')
    opars.trmin = -1;
  end
  if ~isfield(params, 'trdelmin')
    opars.trdelmin = -1;
  end
  if ~isfield(params, 'shuffle')
    opars.shuffle = 0;
  end
```

```matlab
function [w, out] = netreg(p)
  % Expected (or defaulted) fields of 'params'
  % tri = training images (d x N, data in columns)
  % trl = training labels
  % eta = learning parameter (def=0.1)
  % l1  = L1-penalty parameter (def=0)
  % l2  = L2-penalty parameter (def=0)
  % hoi = hold out images (d x N_h) (def=[])
  % hol = hold out labels           (def=[])
  % tsi, tsl = test images and labels (d x N_t) (def=[], [])
  % reprate = Reporting rate (integer, reports per epoch) (def=1)
  % winit = initial guess for weight vector (def=0)
  % maxit = maximum number of iterations    (def=16)
  % ann = Learning parameter annealment strategy ('exp', 'hyp', 'none')
  %       (def='none')
  % annpar = parameter for annealment strategy (def=0)
  % mom = momentum parameter (def=0)
  % esi = 'Early Stopping iterations,' consecutive it's with no improvement
  %       (def=3)


  %% ============================================================ %%
  %% ============================================================ %%

hocost= Inf;


  % Initialize some timing stuff.
  tic;
  ft = 0;
  bt = 0;
  mt = 0;

  % Enforce default parameters
  p = netregdefs(p);

  % Organize stop conditions for the iterative process
  stpmin = [p.trmin, p.tsmin, p.homin, -p.maxit, -p.esi, p.trdelmin];

  %% Initialize a bunch of iterative process data.
  N = size(p.tri, 2);  % Number of training examples
  it = 1;              % Iteration number
  w = p.winit;         % Initial net model
  L = size(w, 1);      % Number of layers
  C = size(w{L, 1}, 2);          % Number of classes
```

```
    delw = {};                      % change in weights (for momentum)
    stpdat = inf(size(stpmin));     % Stop conditions
    trc = [];                       % Array to record training cost and error
    tre = [];
    esits = -inf;                   % Useless iteration counts
    mhc = inf;                      % Min appropriate value of holdout cost
    mhe = inf;                      % " " " " " classification error


    % If we are using batchsizes, make a shuffle
    if p.batchsize < N
      shuffle = randperm(N);
    end

    % If we are going to look at test and holdout data, initialize their
    % arrays
    if ~isempty(p.tsi)
      tsc = [];
      tse = [];
      Nt = length(p.tsl);
    end
    if ~isempty(p.hoi)
      hoc = [];
      hoe = [];
      Nh = length(p.hol);
      if p.batchsize < N
        hofreq = round(Nh / p.batchsize);
      else
        hofreq = 1;
      end
    end

    %% Run grad descent operations until you hit a stop condition!
    while all(stpdat > stpmin)
      eta = anneal(p.eta, p.ann, p.annpar, it);
      %% If we are using batches, select the images to be used
      %% for this batch.
      if p.batchsize < N
        inds = shuffle(1 : p.batchsize);
        tdat = p.tri(:, inds);
        tlabs = p.trl(inds);
        %% Else, use all training data!
      else
        tdat = p.tri;
        tlabs = p.trl;
      end
      %% Do a forwardprop for this iteration
      t = toc;
      [curwx, trcost, trcerr] = forwardprop(w, tdat, tlabs);
      ft = ft + (toc - t);

      %% Do a backprop to calculate the gradient.
      t = toc;
      upd = backprop(curwx, w, tdat, tlabs);
      bt = bt + (toc - t);

      %% Add in regularization terms
      if p.l1 ~= 0
        l1u = l1upd(w);
        for l = 1 : L
          upd{l} = upd{l} + p.l1 * l1u{l} / p.batchsize;
        end
      end
      if p.l2 ~= 0
        l2u = l2upd(w);
```

```
      for l = 1 : L
        upd{l} = upd{l} + p.l2 * l2u{l} / p.batchsize;
      end
    end

    %% Normalize and scale the update by the learning parameter.
    for l = 1 : L
      upd{l} = -1 / p.batchsize * eta * upd{l};
    end

    %% Add in momentum term
    if p.mom ~= 0
      if ~isempty(delw)
        t = toc;
        for l = 1 : L
          upd{l} = upd{l} + p.mom * delw{l};
        end
        mt = mt + (toc - t);
      end
    end

    %% Record change (for future momentum)
    delw = upd;

    %% Add in the update!
    t = toc;
    for l = 1 : L
      w{l, 1} = w{l, 1} + upd{l};
    end
    mt = mt + (toc - t);

    %% Calculate current cost
    trcost = trcost / N;
    %% Calculate relative change in cost
    if it > 1
      dlcost = (trcost - pvcost) / pvcost;
    else
      dlcost = -1;
    end
    pvcost = trcost;
    %% Record in 'trc'
    trc = [trc, trcost];
    tre = [tre, trcerr];

    %% Calculate stop condition markers
    %% If we have holdout data...
    if ~isempty(p.hoi)
      %% (Only activate every so often -- calculations are expensive)
      if mod(it, hofreq) == 1
        %
        if it > 1
          pvcost = hocost;
        end

        % Calculate cost and class err on holdout data
        t = toc;
        [~, hocost, hocerr] = forwardprop(w, p.hoi, p.hol);
        ft = ft + (toc - t);

        % Check to see if we actually got better... if not,
        % count this as a useless iteration.
        hocost = hocost / Nh;
        % 'mhc' and 'mhe' are running minima of cost and error.
        % If either of them is unimproved after several iterations,
        % we terminate.
```

```
        if or(hocost > mhc * (1 - p.hodelmin), hocerr > mhe * (1 - p.hodelmin))
          esits = esits + 1;
        else
          esits = 0;
        end
        if hocost < mhc * (1 - p.hodelmin)
          mhc = hocost;
        end
        if hocerr < mhe * (1 - p.hodelmin)
          mhe = hocerr;
        end

        % Append results to our record arrays...
        hoc = [hoc, hocost];
        hoe = [hoe, hocerr];
        %% if it > 1
        %%   dlcost = (hocost - pvcost) / pvcost;
        %% else
        %%   dlcost = -1;
        %% end
        %%%% This is "consecutive it's with no improvement"
        %% if p.esi > 0
        %%   if dlcost > -p.hodelmin
        %%     esits = esits + 1;
        %%   else
        %%     esits = 0;
        %%   end
        %% end
      end
    else
      hocost = 1;
    end
    %% If we have test data, calculate and record current cost
    if ~isempty(p.tsi)
      [~, tscost, tscerr] = forwardprop(w, p.tsi, p.tsl);
      tscost = tscost / Nt;
      tsc = [tsc, tscost];
      tse = [tse, tscerr];
    else
      tscost = 1;
    end

    % Finalize stop parameters and increase iteration count!
    % trcost;
    stpdat = [trcost, tscost, hocost, -it, -esits, abs(dlcost)];
    it = it + 1;

    % If we're using batches, get a new shuffle.
    if p.batchsize < N
      shuffle = randperm(N);
    end
  end

  stpdat > stpmin;

  % Store our results in the 'out' structure.
  if ~isempty(p.tsi)
    out.tsc = tsc;
    out.tse = tse;
  end
  if ~isempty(p.hoi)
    out.hoc = hoc;
    out.hoe = hoe;
  end
  out.trc = trc;
```

```
    out.tre = tre;
    out.it = it - 1;
    out.ft = ft;
    out.bt = bt;
    out.mt = mt;
    out.tt = toc;
```

```
%% Test cases and plots for PA2

clear all
close all
%% Problem 3) Base case - 1 hidden layer, include momentum, regularization,
%annealing


testcomp = struct;
k=0;

%Import data

%% Generates test cases and plots for Programming Assignment 1


%% This folder has the loading code used to access the MNIST data
addpath('mnistHelper');

%% Load the data, take only the first 20000 training points and 2000 test points
images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
tsimages = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
tslabels= loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');

%% Pre-process
% images = images/255;
images = images -mean(images);
% sim = sim/255;
tsimages = tsimages-mean(tsimages);

% %% Cuttin Data
% trkeep = numel(labels);
% tskeep = numel(tslabels);
trkeep = 20000;
tskeep = 2000;
[tri,trl,sim,slb,N,Nt,hoi,hol] = importimages(images, labels,tsimages,tslabels, trkeep,
    tskeep);

%% Weight initialization (zeros for part Problem 3)
d = size(tri, 1);
ncl = max(trl);

weight_opts.layers = [256];
weight_opts.funs = {'sigmoid','softmax'};
weight_opts.distr = 'zeros';
W = initw(weight_opts,d,ncl);

%% Set the data and parameters for the experiment
batchsize = N;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
p.tsi = sim;
p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
```

```
p.eta = 0.1;
p.shuffle = 0;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 1024;
p.esi = 3;
p.mom = 0;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);

x = p.tri;
labs = p.trl;

%% Run Algorithm and Generate Plots


[w, out] = netreg(p);
[classtrain, tre] = classifier(w, p.tri, labs);
[classtest, tse] = classifier(w, sim, slb);

%%
% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
iters = 1:length(out.trc);

titlep = 'Base Case';
filename = 'base_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);

%% Check gradient comparison
% gradestprep;
% gradest;

%% Problem 4) Tricks of trade

% trkeep = 20000;
% tskeep = 2000;
% [tri,trl,sim,slb,N,Nt,hoi,hol] = importimages(images, labels,tsimages,tslabels, trkeep, tskeep);
trkeep = numel(labels);
tskeep = numel(tslabels);
% trkeep = 20000;
% tskeep = 2000;
[tri,trl,sim,slb,N,Nt,hoi,hol] = importimages(images, labels,tsimages,tslabels, trkeep, tskeep);

%% Base case for small set

% batchsize = N;
% p.batchsize = batchsize;
% % p.tri = tri;
% % p.trl = trl;
% p.hoi = hoi;
```

```
% p.hol = hol;
% p.tsi = [];
% p.tsl = [];
% p.reprate = round(N / batchsize);
%

% x = p.tri;
% labs = p.trl;
% [w, out] = netreg(p);
% [classtrain, tre] = classifier(w, x, labs);
% [classtest, tse] = classifier(w, sim, slb);

%% a) Shuffle
clear p

batchsize = 128;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
% p.tsi = sim;
% p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
p.mom = 0;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);

%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain, tre] = classifier(w, x, labs);
[classtest, tse] = classifier(w, sim, slb);

%%
% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
iters = 1:length(out.trc);

titlep = 'Shuffle';
filename = 'shuffle_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);

%% c) Use tanh sigmoid

clear W
```

```
weight_opts.layers = [256];
weight_opts.funs = {'tanh','softmax'};
weight_opts.distr = 'zeros';
W = initw(weight_opts,d,ncl);


%%
clear p

batchsize = 128;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
% p.tsi = sim;
% p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
p.mom = 0;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);
%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain, tre] = classifier(w, x, labs);
[classtest, tse] = classifier(w, sim, slb);

iters = 1:length(out.trc);

% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
titlep = 'Using tanh as sigmoid';
filename = 'sig_tanh_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);

%% d) Random normal weight distrubiton with mean and std 1/sqrt(fan-in)
clear W
weight_opts.layers = [256];
weight_opts.funs = {'tanh','softmax'};
weight_opts.distr = 'randnorm';
W = initw(weight_opts,d,ncl);
%%

clear p

batchsize = 128;
```

```
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
% p.tsi = sim;
% p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
p.mom = 0;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);

%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain, tre] = classifier(w, x, labs);
[classtest, tse] = classifier(w, sim, slb);

iters = 1:length(out.trc);
% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
titlep = 'Random normal weight initialization';
filename = 'rnweights_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);
%% e) use momentum

clear p

batchsize = 128;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
% p.tsi = sim;
% p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
```

```
p.mom = 0.9;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);

%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain, tre] = classifier(w, x, labs);
[classtest, tse] = classifier(w, sim, slb);

iters = 1:length(out.trc);
% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
titlep = 'Including momentum';
filename = 'momentum_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);
%% 5) Network Topology



%% a) Double number of hidden units

clear W
weight_opts.layers = [512];
weight_opts.funs = {'tanh','softmax'};
weight_opts.distr = 'randnorm';
W = initw(weight_opts,d,ncl);
%%
clear p

batchsize = 128;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
% p.tsi = sim;
% p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
p.mom = 0.9;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

```

```
p = netregdefs(p);


%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain , tre] = classifier(w, x, labs);
[classtest , tse] = classifier(w, sim, slb);
iters = 1:length(out.trc);

% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
titlep = 'Double hidden units';
filename = 'double_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);
%% a.2) Half number of hidden units

clear W
weight_opts.layers = [128];
weight_opts.funs = {'tanh','softmax'};
weight_opts.distr = 'randnorm';
W = initw(weight_opts,d,ncl);


%%

clear p

batchsize = 128;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
% p.tsi = sim;
% p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
p.mom = 0.9;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);
%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain , tre] = classifier(w, x, labs);
[classtest , tse] = classifier(w, sim, slb);
iters = 1:length(out.trc);
```

```
% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
titlep = 'Half Hidden Units';
filename = 'half_f';
makeplots(iters,out,titlep,filename)

k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1-min(tre);
testcomp(k).accuracy_test = 1-min(tse);
%% b) Add another hidden layer

% trkeep = 20000;
% tskeep = 2000;
% [tri,trl,sim,slb,N,Nt,hoi,hol] = importimages(images, labels,tsimages,tslabels, trkeep,
    tskeep);

%%
clear W
weight_opts.layers = [204,204];
weight_opts.funs = {'tanh','tanh','softmax'};
weight_opts.distr = 'randnorm';
W = initw(weight_opts,d,ncl);


%%
clear p

batchsize = 128;
p.batchsize = batchsize;
p.tri = tri;
p.trl = trl;
p.hoi = hoi;
p.hol = hol;
p.tsi = sim;
p.tsl = slb;
p.hodelmin = 0.0001;
p.homin = 0.03;
p.eta = 0.1;
p.shuffle = 1;
p.ann = 'hyp';
T = 256;
p.annpar = 1 / T;
% p.trmin = 0.01;
p.maxit = 2048;
p.esi = 8;
p.mom = 0.9;
p.l1 = 0;
p.l2 = 0.0005;
p.reprate = round(N / batchsize);
% p.trdelmin = 0.0001;
p.winit = W;

p = netregdefs(p);
%% Run Algorithm and Classify

[w, out] = netreg(p);
[classtrain, tre] = classifier(w, x, labs);
[classtest, tse] = classifier(w, sim, slb);
iters = 1:length(out.trc);
% epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
titlep = 'Add second hidden layer';
filename = 'twohidlayers_f';
makeplots(iters,out,titlep,filename)

```

```
k=k+1;
testcomp(k).test = filename;
testcomp(k).numits = length(out.trc);
testcomp(k).accuracy_train = 1−min(tre);
testcomp(k).accuracy_test = 1−min(tse);


save('testresults',testcomp);
% %% b.2 Change hidden layers size
%
% weight_opts.layers = [300,100];
% weight_opts.funs = {'tanh','tanh','softmax'};
% weight_opts.distr = 'randnorm';
% W = initw(weight_opts,d,ncl);
% p.winit = W;
%
% %% Run Algorithm and Classify
% [w, out] = netreg(p);
% % [classtrain , tre] = classifier(w, p.tri , labs);
% % [classtest , tse] = classifier(w, sim, slb);
% iters = 1:length(out.trc);
%
% % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
% titlep = 'Change units of both hidden layer';
% filename = 'twohiddif';
% makeplots(iters,out,titlep,filename)
%
% k=k+1;
% testcomp(k).test = filename;
% testcomp(k).numits = length(out.trc);
% testcomp(k).accuracy_train = 1−min(tre);
% testcomp(k).accuracy_test = 1−min(tse);
```

```
function [tri, tsi] = preprocess(tri, tsi)
  % Subtract mean from training and testing data.
   tri = tri − mean(tri, 2);
   tsi = tsi − mean(tsi, 2);
```

```
function rv = stringin(str, strlist)
  % Just want to check if 'str' is an element of 'strlist'
  rv = 0;
  N = size(strlist, 2);
   for i = 1 : N
     if strcmp(str, deblank(strlist(i, :)))
       rv = 1;
       return;
     end
   end
   return;
end
```