

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:

```
1 %% This folder has the loading code used to access the MNIST data
2 addpath('mnistHelper');
3
4 %% Load the data, take only the first 20000 training points and 2000 test points
5 images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
6 labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
7 tsimages = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
8 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`:

```
1 function [tri, tsi] = preprocess(tri, tsi)
2 % Subtract mean from training and testing data.
3 tri = tri - mean(tri, 2);
4 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:

```
1 weight_opts.layers = [256];
2 weight_opts.funs = {'sigmoid', 'softmax'};
3 weight_opts.distr = 'zeros';
4 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:

```
1 >> gradest
2 Error between grads, layer 2:
3 ans = 5.5845e-06
4 Error between grads, layer 1:
5 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 `gratest.m` and `gratestprep.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 network 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 against 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.

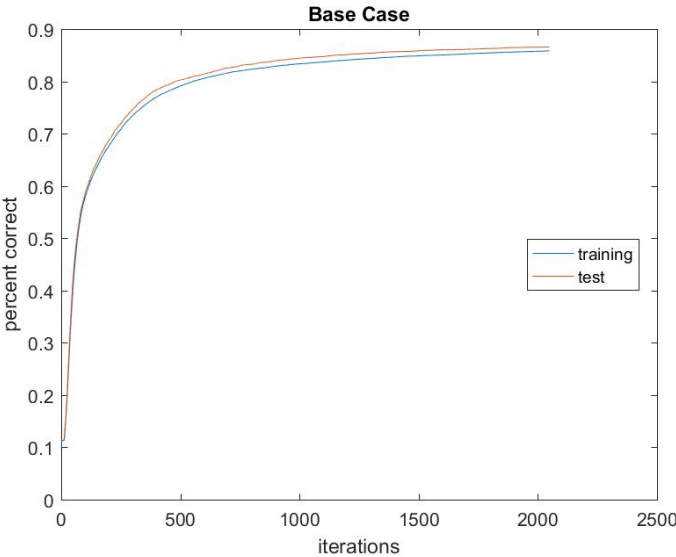


Figure 1: Classification Accuracy for 1 layer with no acceleration

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 $1/\sqrt{\text{fan-in}}$
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.

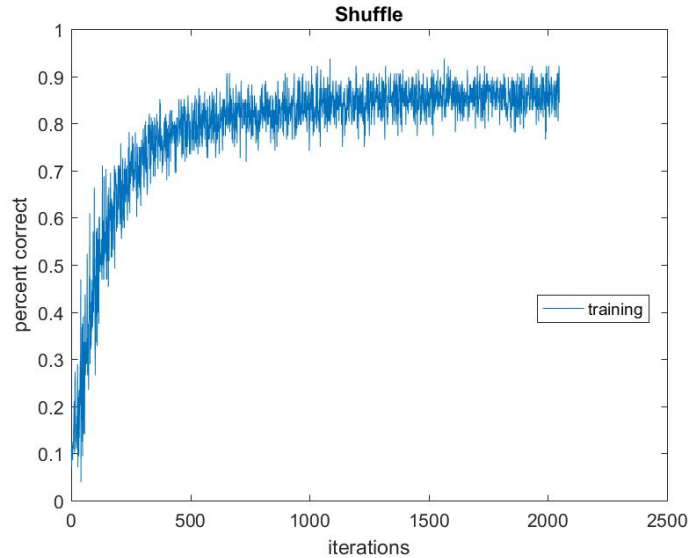


Figure 2: Accuracy with stochastic shuffling (batch-size of 128)

2.2 Changing Activation Function

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

```

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

```

For back propagation, we find the derivative:

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

We vectorize the computation as we compute the δ_j :

```

1  b = 2/3;
2  a = 1.7159;
3  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.

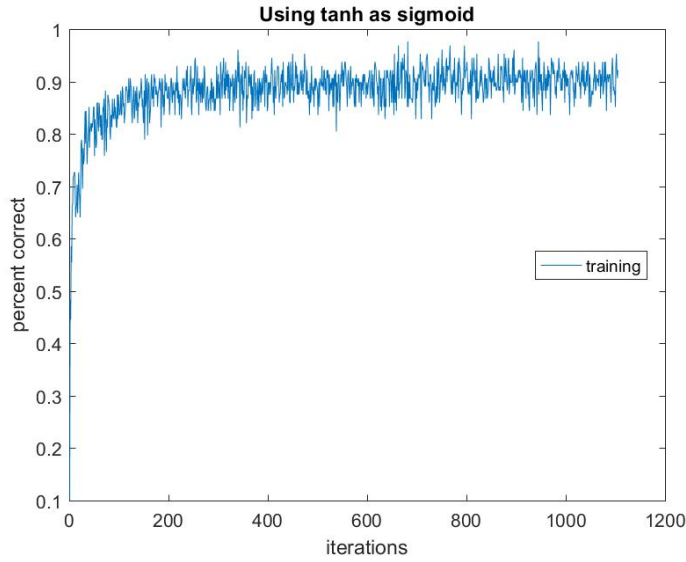


Figure 3: Accuracy replacing sigmoid with tanh activation function

2.3 Weight Initialization

In this stage, we initialize the weights according to normal distribution of mean 0 and variance $1/(\text{fan-in})$, 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.

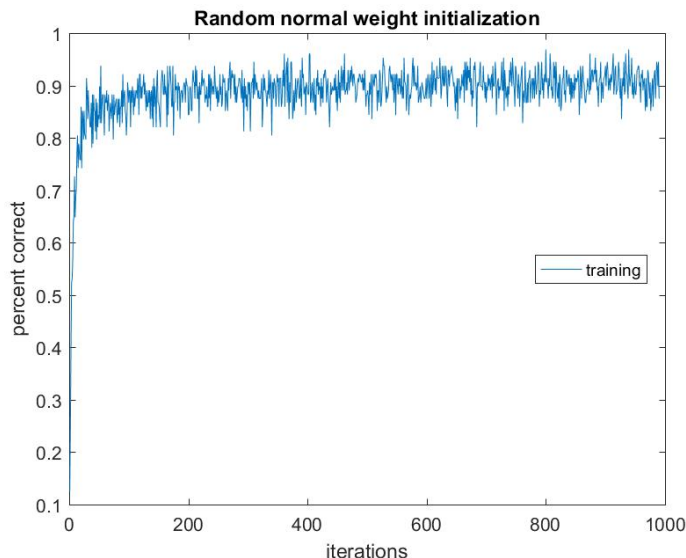


Figure 4: Accuracy with weights initialized by random normal distribution

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!

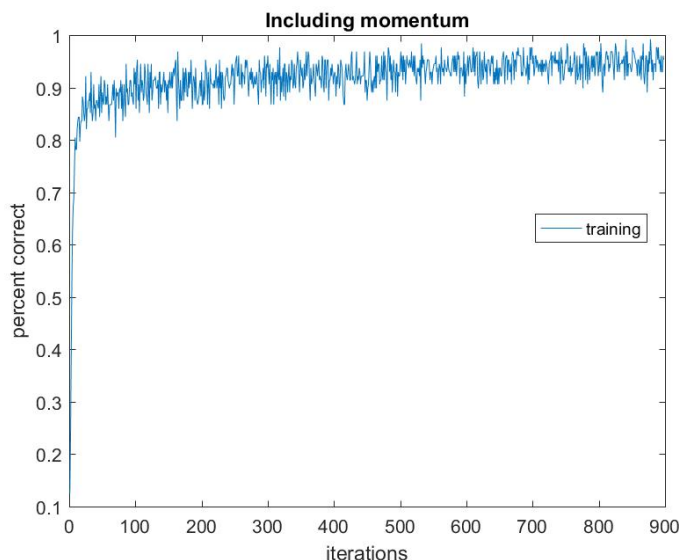


Figure 5: Accuracy adding momentum term (mom = 0.9)

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 initial 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.

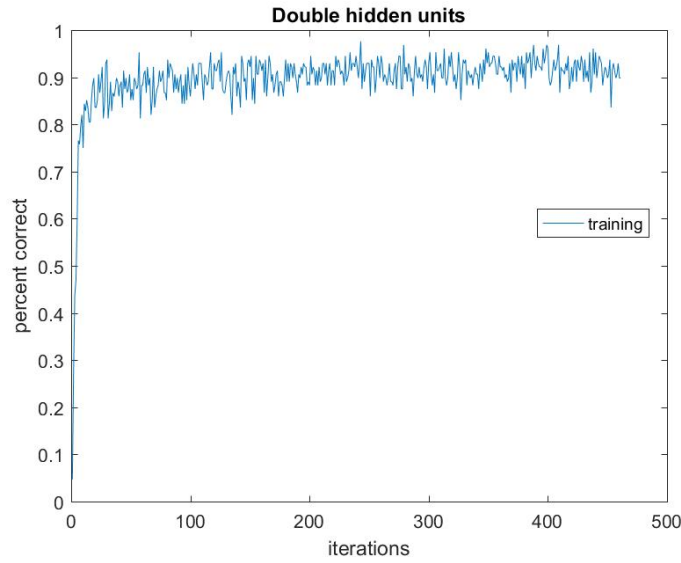


Figure 6: Accuracy with double number of hidden units - 512

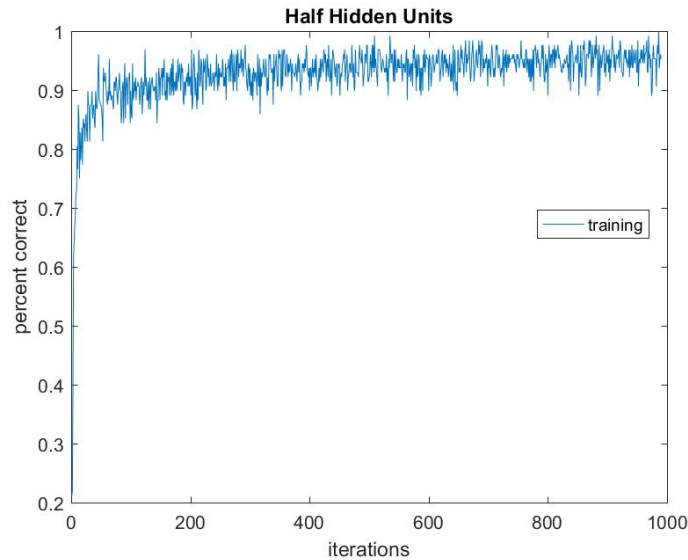


Figure 7: Accuracy with half number of hidden units - 128

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.

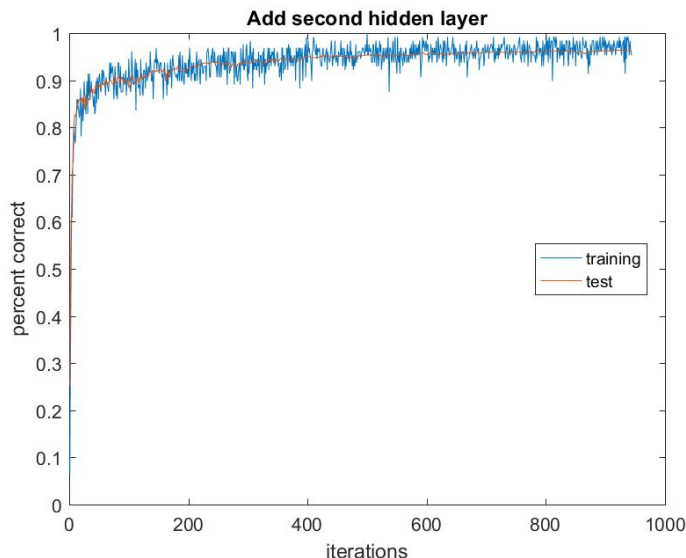


Figure 8: Accuracy with two hidden layers of same size (both 204 units)

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 backpropagation. 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 developed the initial architecture for storing the net model in a single data structure. He helped update the testing procedure to accommodate 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

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



```

420 4 % type = string defining the activation function type.
421 5 % Returns:
422 6 % y = fcn values after applied to A.
423 7
424 8 % You can do softmax...
425 9 if strcmp(type, 'softmax')
426 10     y = exp(A);
427 11     y = y ./ sum(y, 1);
428 12
429 13 % ...or sigmoid...
430 14 elseif or(strcmp(type, 'sig'), strcmp(type, 'sigmoid'))
431 15     y = 1 ./ (1 + exp(-A));
432 16
433 17 % ...or that special tanh.
434 18 elseif strcmp('tanh', type)
435 19     b = 2/3;
436 20     a = 1.7159;
437 21     y = a*tanh(b*A);
438 22 end
439
440 1 function delta = actgrad(y, W, delta, type)
441 2 % Calculates the gradient of the activation function for the layer at hand.
442 3 % y = results of the activation function of lower layer
443 4 % W = weight matrix of present layer
444 5 % delta = "slope" of previous (higher) layer
445 6 % type = type of activation function
446 7
447 8 % Derivation for sigmoid gradient...
448 9 if or(strcmp(type, 'sig'), strcmp(type, 'sigmoid'))
449 10     delta = (y .* (1-y)) .* (W.' * delta);
450 11     % ...and for tanh...
451 12 elseif strcmp(type, 'tanh')
452 13     b = 2/3;
453 14     a = 1.7159;
454 15     delta = (a * b) * (1 - (y.*y) / a^2) .* (W.' * delta);
455 16 end
456 17 % Truncate to throw away the bias node (which has no gradient)
457 18 delta = delta(2:end, :);
458 19 end
459
460 1 function etap = anneal(eta, ann, annpar, it)
461 2 % This function returns the "annealed" learning parameter etap (eta prime)
462 3 % eta = base learning parameter
463 4 % ann = annealment mode ('exp', 'hyp', or 'none')
464 5 % annpar = parameter controlling annealment strategy
465 6 % it = current iteration number
466 7
467 8 % hyperbolic annealment
468 9 if strcmp(ann, 'hyp')
469 10     etap = eta / (1 + (it - 1) * annpar);
470 11 % exponential annealment
471 12 elseif strcmp(ann, 'exp')
472 13     etap = eta * annpar^(it - 1);
473 14 % no annealment
474 15 else
475 16     etap = eta;
476 17 end
477
478 1 function [G] = backprop(WX, W, x, labs)
479 2 % Does back propagation on the neural net.
480 3 % WX = Intermediate calculations (weighted sums and activation results)
481 4 % W = Neural net model
482 5 % x = input to the net
483 6 % labs = correct labels
484 7
485 8 % Size parameters and basic initializes

```



```

480 9 L = size(W, 1);
481 10 Npts = size(WX{1, 1}, 2);
482 11 G = cell(size(W, 1), 1);
483 12
484 13 % Isolate a vector of activation function names.
485 14 actfuns=cell(L,1);
486 15 for l = 1 : L
487 16     actfuns{l} = W{l, 2};
488 17 end
489 18
490 19 % Calculate delta for output layer...
491 20 a = WX{L, 1};
492 21 y = WX{L, 2};
493 22 delta = loglikegrad(y, labs, actfuns{L});
494 23
495 24 % For each layer, traversing backwards...
496 25 for l = L : -1 : 1
497 26
498 27     % Except for the first layer, "inputs" were the activation results
499 28     % of the previous layer.
500 29     if l > 1
501 30         y = WX{l - 1, 2};
502 31         % For first layer, "inputs" were actual inputs
503 32     else
504 33         y = x;
505 34     end
506 35
507 36     % Gradient of weights is outer product of delta and inputs to layer
508 37     % according to derivation
509 38     G{l} = delta * y';
510 39
511 40     % Calculate gradient of activation function, delta for next layer.
512 41     if l > 1
513 42         delta = actgrad(y, W{l, 1}, delta, actfuns{l-1, :});
514 43     end
515 44 end
516 45 end
517
518 1 function [class, pe] = classifier(W, x, labs)
519 2     % Predicts classes of x data, gives percent wrong.
520 3     % W = net model
521 4     % x = input data
522 5     % labs = correct labels
523 6     % Returns
524 7     % class = class predictions
525 8     % pe = percent error
526 9
527 10    % Straight up, just run a forward prop...
528 11    [WX, loss] = forwardprop(W, x, labs);
529 12    N = length(labs);
530 13    J = size(WX,1);
531 14
532 15    % ...then do a 'one hot' to figure out prediction
533 16    [~, class] = max(WX{J,2}', [], 2);
534 17    pe = 1 - sum(class(:) == labs) / N;
535 18 end
536
537 1 function E = costfun(wx, y, type, labs)
538 2     % Calculates the cost function of the neural net.
539 3     % wx = weighted sums going into the output layer
540 4     % y = results of output layer activations function
541 5     % type = type of activation function
542 6     % labs = data labels
543 7
544 8     % So far... only handles softmax activation :)
545 9     if ~strcmp(type, 'softmax')

```

```

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

547 function [WX, loss, pe, class] = forwardprop(W, x, labs)
548 % Performs forward propagation on the neural net defined by W.
549 % x is a matrix containing the input vectors
550 % labs is a vector of the labels
551 % Returns:
552 % WX, intermediate computations of weight maps and activation
553 % functions.
554 % loss = loss function (log likelihood, etc.) on the output nodes
555 % pe = percent error (classification error)
556 % class = classes (by "one-hot encoding")
557
558 % Get some size parameters.
559 % Initialize the cell structure of WX.
560 L = size(W, 1);
561 WX = cell(size(W));
562 N = size(x, 2);
563
564 % 'y' is the input to the linear map at each level;
565 % on the first layer, this is the actual inputs.
566 y = x;
567
568 % For each layer...
569 for l = 1 : L
570     % Apply the weight map
571     WX{l, 1} = W{l, 1} * y;
572     % Apply the activation. If not on the output layer, add a bias node.
573     if l < L
574         y = [ones(1, N); actfun(WX{l, 1}, W{l, 2})];
575     else
576         y = actfun(WX{l, 1}, W{l, 2});
577     end
578
579     % Store the results of activation function (for use elsewhere)
580     WX{l, 2} = y;
581 end
582
583 % Calculate the loss after all is said and done.
584 loss = costfun(WX{L, 1}, WX{L, 2}, W{L, 2}, labs);
585
586 % Usual classification procedure with percent error.
587 % (class = argmax of net results)
588 M = length(labs);
589 J = size(WX, 1);
590 [~, class] = max(WX{J, 2}', [], 2);
591 pe = 1 - sum(class(:) == labs) / M;
592 end

593 % Do the calculations to get a candidate for gradient...
594 gradestprep;
595
596 % Define a perturbation
597 pert = 1e-3;
598
599 % Grab all size parameters, appropriate matrices for testing
600 % second layer.
601 L = size(W, 2);
602 [m, n] = size(W{L, 1});
603 wx = WX{L, 1};
604 y = WX{L-1, 2};

```

```

600 13 w = W{L, 1};
601 14 af = W{L, 2};
602 15
603 16 % For each entry in the weight matrix...
604 17 for i = 1 : m
605 18     for j = 1 : n
606 19         % Perturb the weights...
607 20         wxp = wx;
608 21         wxp(i, :) = wxp(i, :) + pert * y(j, :);
609 22         wxm = wx;
610 23         wxm(i, :) = wxm(i, :) - pert * y(j, :);
611 24         wp = w;
612 25         wp(i, j) = wp(i, j) + pert;
613 26         wm = w;
614 27         wm(i, j) = wm(i, j) - pert;
615 28         % ...calculate the resulting loss...
616 29         yp = actfun(wxp, af);
617 30         ym = actfun(wxm, af);
618 31         lp = costfun(wxp, yp, af, labs);
619 32         lm = costfun(wxm, ym, af, labs);
620 33         % ...estimate derivative.
621 34         g{L}(i, j) = (lp - lm) / (2 * pert);
622 35     end
623 36 end
624 37
625 38 % Display the results
626 39 disp('Error between grads, layer 2:');
627 40 max(abs(g{L} - G{L})(:))
628 41
629 42 % Clean data to test the first layer.
630 43 [m, n] = size(W{1, 1});
631 44 wx = WX{1, 1};
632 45 y = dats;
633 46 af1 = W{1, 2};
634 47 af2 = W{2, 2};
635 48 w = W{2, 1};
636 49
637 50 % Repeat the previous process...
638 51 for i = 1 : m
639 52     for j = 1 : n
640 53         wxp = wx;
641 54         wxp(i, :) = wxp(i, :) + pert * y(j, :);
642 55         wxm = wx;
643 56         wxm(i, :) = wxm(i, :) - pert * y(j, :);
644 57         yp = [ones(1, N); actfun(wxp, af1)];
645 58         ym = [ones(1, N); actfun(wxm, af1)];
646 59         ap = w * yp;
647 60         am = w * ym;
648 61         yp = actfun(ap, af2);
649 62         ym = actfun(am, af2);
650 63         lp = costfun(ap, yp, af2, labs);
651 64         lm = costfun(am, ym, af2, labs);
652 65         g{1}(i, j) = (lp - lm) / (2 * pert);
653 66     end
654 67 end
655 68
656 69 % Display results.
657 70 disp('Error between grads, layer 1:');
658 71 max(abs(g{1} - G{1})(:))
659 72
660 1 %% Generates test cases and plots for Programming Assignment 1
661 2 clear all
662 3
663 4 %% This folder has the loading code used to access the MNIST data
664 5 addpath('mnistHelper');
665 6

```

```

660 7 %% Load the data, take only the first 20000 training points and 2000 test points
661 8 images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
662 9 labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
663 10 sim = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
664 11 slb = loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');
665 12
666 13 % Cut down train and test data
667 14 trkeep = 1000;
668 15 tskeep = 500;
669 16 images = images(:, 1:trkeep);
670 17 labels = labels(1:trkeep);
671 18 sim = sim(:, 1:tskeep);
672 19 slb = slb(1:tskeep);
673 20
674 21 % Throw out all but some digits!
675 22 digs = 0:9;
676 23 inds = logical(any(labels == digs, 2));
677 24 labs = labels(inds);
678 25 dats = images(:, inds);
679 26 [~, labs] = max((labs == digs), [], 2);
680 27 N = numel(labs);
681 28 dats = [ones(1, N); dats];
682 29
683 30 % Size parameters...
684 31 d = size(dats, 1); % Dimension of input data
685 32 ncl = max(labs); % Number of classes (distinct outputs)
686 33 jhid = 16; % Number of hidden nodes.x
687 34
688 35 % Initialize the net model values...
689 36 W = cell(2, 2);
690 37 W{1, 1} = randn(jhid, d);
691 38 W{1, 2} = 'sigmoid';
692 39 W{2, 1} = randn(ncl, jhid + 1);
693 40 W{2, 2} = 'softmax';
694 41
695 42 % Calculate the gradient!
696 43 [WX, loss] = forwardprop(W, dats, labs);
697 44 G = backprop(WX, W, dats, labs);
698 45
699 1 function [tri, trl, sim, slb, N, Nt, hoi, hol] = importimages(images, labels, sim, slb, trkeep,
700 2 tskeep)
701 3
702 4 images = images(:, 1:trkeep);
703 5 labels = labels(1:trkeep);
704 6 sim = sim(:, 1:tskeep);
705 7 slb = slb(1:tskeep);
706 8
707 9 %% Throw out all but some digits!
708 10 % digs = [2, 3];
709 11 digs = 0:9;
710 12 inds = logical(any(labels == digs, 2));
711 13 labs = labels(inds);
712 14 dats = images(:, inds);
713 15 [~, labs] = max((labs == digs), [], 2);
714 16 N = numel(labs);
715 17 dats = [ones(1, N); dats];
716 18 tri = dats;
717 19 trl = labs;
718 20
719 21 % Same for test data
720 22 tst = logical(any(slb == digs, 2));
721 23 sim = sim(:, tst);
722 24 slb = slb(tst);
723 25 [~, slb] = max(slb == digs, [], 2);
724 26 Nt = numel(slb);

```

```

720 27 sim = [ones(1, Nt); sim];
721 28
722 29 % Develop a hold out set
723 30 hopct = 0.05;
724 31 hoind = round((1 - hopct) * N);
725 32 hoi = dats(:, hoind : end);
726 33 hol = labs(hoind : end);
727 34 tri = dats(:, 1 : (hoind - 1));
728 35 trl = labs(1 : (hoind - 1));
729 36
730 37 end
731
731 1 function W = initw(weight_opts, nin, nout)
732 2     %% Initialize the net model based on certain parameters.
733 3     %% weight_opts contains the sizes of the intermediate layers,
734 4     %% as well as the types of activation functions used
735 5     %% and the distribution from which initial weights shall be drawn.
736 6     %% nin = number of input layer values (dimension of input data)
737 7     %% nout = number of potential classes
738 8
739 9     % Work the "meta-dimension" into the layer sizes.
740 10    layers = [nin, weight_opts.layers, nout];
741 11    funs = weight_opts.funs;
742 12    distr = weight_opts.distr;
743 13
744 14    % If you're using normally distributed weights... go for it
745 15    if strcmp(distr, 'randnorm')
746 16        for i=1:length(layers)-1
747 17            if i ~= 1
748 18                W{i,1} = normrnd(0,1/sqrt(layers(i)), layers(i+1), layers(i) + 1);
749 19            else
750 20                W{i,1} = normrnd(0,1/sqrt(layers(i)), layers(i+1), layers(i));
751 21            end
752 22            W{i,2} = funs{i};
753 23
754 24        end
755 25        % Or initialize to zeros
756 26    elseif strcmp(distr, 'zeros')
757 27        for i=1:length(layers)-1
758 28            if i ~= 1
759 29                W{i,1} = zeros(layers(i+1), layers(i) + 1);
760 30            else
761 31                W{i,1} = normrnd(0,1/sqrt(layers(i)), layers(i+1), layers(i));
762 32            end
763 33            W{i,2} = funs{i};
764 34
765 35        end
766 36        % Or distributed randomly...
767 37    elseif strcmp(distr, 'uniform')
768 38        for i=1:length(layers)-1
769 39            if i ~= 1
770 40                W{i,1} = rand(layers(i+1), layers(i) + 1);
771 41            else
772 42                W{i,1} = normrnd(0,1/sqrt(layers(i)), layers(i+1), layers(i));
773 43            end
774 44            W{i,2} = funs{i};
775 45        end
776 46    end
777 47 end
778
778 1 function upd = l1upd(W)
779 2     % Returns the gradient of the L1 norm of w
780 3
781 4     N= size(W,1);
782 5     upd = cell(N,1);
783 6

```

```

780 7   for i=1:N
781 8
782 9   upd{i} = (W{i,1} > 0) - (W{i,1} < 0);
783 10
784 11   end
785
786 1 function upd = l2upd(W)
787 2   % Returns the gradient of the L2 norm of w
788 3
789 4   N = size(W,1);
790 5   upd = cell(N,1);
791 6
792 7   for i = 1:N
793 8
794 9   upd{i} = 2 * W{i,1};
795 10
796 11   end
797
798 1 %% Generates test cases and plots for Programming Assignment 1
799 2 clear all
800 3 close all
801 4 %% This folder has the loading code used to access the MNIST data
802 5 addpath('mnistHelper');
803 6
804 7 %% Load the data, take only the first 20000 training points and 2000 test points
805 8 images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
806 9 labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
807 10 sim = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
808 11 slb = loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');
809 12
810 13 % Cut down to 20000 train, 2000 test
811 14 trkeep = numel(labels);
812 15 tskeep = numel(slb);
813 16 % trkeep = 20000;
814 17 % tskeep = 2000;
815 18 images = images(:, 1:trkeep);
816 19 labels = labels(1:trkeep);
817 20 sim = sim(:, 1:tskeep);
818 21 slb = slb(1:tskeep);
819 22 [images, sim] = preprocess(images, sim);
820 23
821 24 % Throw out all but some digits!
822 25 % digs = [2, 3];
823 26 digs = 0:9;
824 27 inds = logical(any(labels == digs, 2));
825 28 labs = labels(inds);
826 29 dats = images(:, inds);
827 30 [~, labs] = max((labs == digs), [], 2);
828 31 N = numel(labs);
829 32 dats = [ones(1, N); dats];
830 33 tri = dats;
831 34 trl = labs;
832 35
833 36 % Same for test data
834 37 tst = logical(any(slb == digs, 2));
835 38 sim = sim(:, tst);
836 39 slb = slb(tst);
837 40 [~, slb] = max(slb == digs, [], 2);
838 41 Nt = numel(slb);
839 42 sim = [ones(1, Nt); sim];
840 43
841 44 % Develop a hold out set
842 45 hopct = 0.05;
843 46 hoind = round((1 - hopct) * N);
844 47 hoi = dats(:, hoind : end);
845 48 hol = labs(hoind : end);

```

```

840 49 tri = dats(:, 1 : (hoind - 1));
841 50 trl = labs(1 : (hoind - 1));
842 51
843 52 d = size(tri, 1);
844 53 ncl = max(trl);
845 54
846 55 weight_opts.layers = [256];
847 56 weight_opts.funs = {'sigmoid', 'softmax'};
848 57 weight_opts.distr = 'zeros';
849 58
850 59 W = initw(weight_opts, d, ncl);
851 60
852 61 batchsize = 128;
853 62
853 63 % Set the data and parameters for the experiment
854 64 p.tri = tri;
855 65 p.trl = trl;
856 66 p.hoi = hoi;
857 67 p.hol = hol;
858 68 p.homin = 0.03;
859 69 % p.tsi = sim;
860 70 % p.tsl = slb;
861 71 p.eta = 0.1;
862 72 p.shuffle = 1;
863 73 p.ann = 'hyp';
864 74 p.annpar = 1 / 256;
865 75 p.maxit = 2048;
866 76 p.esi = 3;
867 77 p.mom = 0.9;
868 78 p.l1 = 0;
869 79 p.l2 = 0.0005;
870 80 p.batchsize = batchsize;
871 81 p.hodelmin = 0.0001;
872 82 p.winit = W;
873 83 p = netregdefs(p);
874 84
875 85 x = p.tri;
876 86 labs = p.trl;
877 87
878 88 [w, out] = netreg(p);
879 89 iters = 1:length(out.trc);
880 90
881 91 [~, tpe] = classifier(w, x, labs);
882 92 [~, spe] = classifier(w, sim, slb);
883 93
884 1 function delta = loglikegrad(y, labs, actfun)
885 2     %% Calculate the gradient of the loglikelihood at the end
886 3     %% y = activation results of last layer
887 4     %% labs = correct labels of data
888 5     %% actfun = activation function type for last layer
889 6
890 7     actfun = deblank(actfun);
891 8     % Right now... only using softmax at end.
892 9     if or(strcmp(actfun, 'soft'), strcmp(actfun, 'softmax'))
893 10         % Calculate the gradient according to derivation.
894 11         N = size(y, 2);
895 12         index = sub2ind(size(y), labs', 1:N);
896 13         C = size(y, 1);
897 14         t = (labs == (1 : C))';
898 15         delta = y - t;
899 16         return;
900 17     end
901 18
902 1 function makeplots(iters, out, titlep, filename)
903 2
904 3     figure

```



```

900 4
901 5 plot(iters,1-out.tre)
902 6 hold on
903 7 if isfield(out,'tse')
904 8     plot(iters,1-out.tse)
905 9
906 10 end
907 11
908 12 xlabel('iterations')
909 13 ylabel('percent correct')
910 14 title(titlep)
911 15 legend('training','test','Location','Best')
912 16
913 17 saveas(gcf,filename,'jpg')
914 18
915 19
916 20
917 21
918 22 end
919
920 1 function opars = netregdefs(params)
921 2     % Enforce default values for test parameters.
922 3
923 4     opars = params;
924 5     d = size(params.tri, 1);
925 6     if ~isfield(params, 'eta')
926 7         opars.eta = 0.1;
927 8     end
928 9     if ~isfield(params, 'l1')
929 10         opars.l1 = 0;
930 11     end
931 12     if ~isfield(params, 'l2')
932 13         opars.l2 = 0;
933 14     end
934 15     if ~isfield(params, 'hoi')
935 16         opars.hoi = [];
936 17         opars.hol = [];
937 18         opars.homin = -1;
938 19     elseif ~isfield(params, 'homin')
939 20         opars.homin = 0.01;
940 21     end
941 22     if ~isfield(params, 'tsi')
942 23         opars.tsi = [];
943 24         opars.tsl = [];
944 25         opars.tsmin = -1;
945 26     elseif ~isfield(params, 'tsmin')
946 27         opars.tsmin = 0.01;
947 28     end
948 29     if ~isfield(params, 'batchsize')
949 30         opars.batchsize = d;
950 31     end
951 32     if ~isfield(params, 'winit')
952 33         if ~isfield(params, 'ncl')
953 34             opars.ncl = max(params.trl);
954 35         end
955 36         opars.winit = initw(p.netpars, d, opars.ncl);
956 37     else
957 38         opars.ncl = size(params.winit, 2);
958 39     end
959 40     if ~isfield(params, 'maxit')
960 41         opars.maxit = 16;
961 42     end
962 43     if ~isfield(params, 'ann')
963 44         opars.ann = 'none';
964 45     end
965 46     if ~isfield(params, 'annpar')

```

```

960 47     opars.annpar = 0;
961 48 end
962 49 if ~isfield(params, 'mom')
963 50     opars.mom = 0;
964 51 end
965 52 if ~isfield(params, 'esi')
966 53     opars.esi = inf;
967 54 end
968 55 if ~isfield(params, 'delwmin')
969 56     opars.delwmin = -1;
970 57 end
971 58 if ~isfield(params, 'trmin')
972 59     opars.trmin = -1;
973 60 end
974 61 if ~isfield(params, 'trdelmin')
975 62     opars.trdelmin = -1;
976 63 end
977 64 if ~isfield(params, 'shuffle')
978 65     opars.shuffle = 0;
979 66 end
980 1 function [w, out] = netreg(p)
981 2     % Expected (or defaulted) fields of 'params'
982 3     % tri = training images (d x N, data in columns)
983 4     % trl = training labels
984 5     % eta = learning parameter (def=0.1)
985 6     % l1 = L1-penalty parameter (def=0)
986 7     % l2 = L2-penalty parameter (def=0)
987 8     % hoi = hold out images (d x N_h) (def=[])
988 9     % hol = hold out labels (def=[])
989 10    % tsi, tsl = test images and labels (d x N_t) (def=[], [])
990 11    % repute = Reporting rate (integer, reports per epoch) (def=1)
991 12    % winit = initial guess for weight vector (def=0)
992 13    % maxit = maximum number of iterations (def=16)
993 14    % ann = Learning parameter annealment strategy ('exp', 'hyp', 'none')
994 15    % (def='none')
995 16    % annpar = parameter for annealment strategy (def=0)
996 17    % mom = momentum parameter (def=0)
997 18    % esi = 'Early Stopping iterations,' consecutive it's with no improvement
998 19    % (def=3)
999 20
1000 21
1001 22    %% ===== %%
1002 23    %% ===== %%
1003 24
1004 25 hocost= Inf;
1005 26
1006 27
1007 28 % Initialize some timing stuff.
1008 29 tic;
1009 30 ft = 0;
1010 31 bt = 0;
1011 32 mt = 0;
1012 33
1013 34 % Enforce default parameters
1014 35 p = netregdefs(p);
1015 36
1016 37 % Organize stop conditions for the iterative process
1017 38 stpmin = [p.trmin, p.tsmn, p.homin, -p.maxit, -p.esi, p.trdelmin];
1018 39
1019 40 %% Initialize a bunch of iterative process data.
1020 41 N = size(p.tri, 2); % Number of training examples
1021 42 it = 1; % Iteration number
1022 43 w = p.winit; % Initial net model
1023 44 L = size(w, 1); % Number of layers
1024 45 C = size(w{L, 1}, 2); % Number of classes

```

```

1020 46 delw = {}; % change in weights (for momentum)
1021 47 stpdat = inf(size(stpmin)); % Stop conditions
1022 48 trc = []; % Array to record training cost and error
1023 49 tre = [];
1024 50 esits = -inf; % Useless iteration counts
1025 51 mhc = inf; % Min appropriate value of holdout cost
1026 52 mhe = inf; % " " " " " classification error
1027 53
1028 54 % If we are using batchsizes, make a shuffle
1029 55 if p.batchsize < N
1030 56     shuffle = randperm(N);
1031 57 end
1032 58
1033 59 % If we are going to look at test and holdout data, initialize their
1034 60 % arrays
1035 61 if ~isempty(p.tsi)
1036 62     tsc = [];
1037 63     tse = [];
1038 64     Nt = length(p.tsl);
1039 65 end
1040 66 if ~isempty(p.hoi)
1041 67     hoc = [];
1042 68     hoe = [];
1043 69     Nh = length(p.hol);
1044 70     if p.batchsize < N
1045 71         hofreq = round(Nh / p.batchsize);
1046 72     else
1047 73         hofreq = 1;
1048 74     end
1049 75 end
1050 76
1051 77 %% Run grad descent operations until you hit a stop condition!
1052 78 while all(stpdat > stpmin)
1053 79     eta = anneal(p.eta, p.ann, p.annpar, it);
1054 80     %% If we are using batches, select the images to be used
1055 81     %% for this batch.
1056 82     if p.batchsize < N
1057 83         inds = shuffle(1 : p.batchsize);
1058 84         tdat = p.tri(:, inds);
1059 85         tlab = p.trl(inds);
1060 86         %% Else, use all training data!
1061 87     else
1062 88         tdat = p.tri;
1063 89         tlab = p.trl;
1064 90     end
1065 91     %% Do a forwardprop for this iteration
1066 92     t = toc;
1067 93     [curwx, trcost, trcerr] = forwardprop(w, tdat, tlab);
1068 94     ft = ft + (toc - t);
1069 95
1070 96     %% Do a backprop to calculate the gradient.
1071 97     t = toc;
1072 98     upd = backprop(curwx, w, tdat, tlab);
1073 99     bt = bt + (toc - t);
1074 100
1075 101     %% Add in regularization terms
1076 102     if p.l1 ~= 0
1077 103         llw = llupd(w);
1078 104         for l = 1 : L
1079 105             upd{l} = upd{l} + p.l1 * llw{l} / p.batchsize;
1080 106         end
1081 107     end
1082 108 end
1083 109 if p.l2 ~= 0
1084 110     llw = llupd(w);

```

```

1080     for l = 1 : L
1081         upd{1} = upd{1} + p.l2 * l2u{1} / p.batchsize;
1082     end
1083 end
1084
1085 %% Normalize and scale the update by the learning parameter.
1086 for l = 1 : L
1087     upd{1} = -1 / p.batchsize * eta * upd{1};
1088 end
1089
1090 %% Add in momentum term
1091 if p.mom ~= 0
1092     if ~isempty(delw)
1093         t = toc;
1094         for l = 1 : L
1095             upd{1} = upd{1} + p.mom * delw{1};
1096         end
1097         mt = mt + (toc - t);
1098     end
1099 end
1100
1101 %% Record change (for future momentum)
1102 delw = upd;
1103
1104 %% Add in the update!
1105 t = toc;
1106 for l = 1 : L
1107     w{1, 1} = w{1, 1} + upd{1};
1108 end
1109 mt = mt + (toc - t);
1110
1111 %% Calculate current cost
1112 trcost = trcost / N;
1113 %% Calculate relative change in cost
1114 if it > 1
1115     dlcost = (trcost - pvcost) / pvcost;
1116 else
1117     dlcost = -1;
1118 end
1119 pvcost = trcost;
1120 %% Record in 'trc'
1121 trc = [trc, trcost];
1122 tre = [tre, trcerr];
1123
1124 %% Calculate stop condition markers
1125 %% If we have holdout data...
1126 if ~isempty(p.hoi)
1127     %% (Only activate every so often — calculations are expensive)
1128     if mod(it, hofreq) == 1
1129         %
1130         if it > 1
1131             pvcost = hocost;
1132         end
1133
1134         % Calculate cost and class err on holdout data
1135         t = toc;
1136         [~, hocost, hocerr] = forwardprop(w, p.hoi, p.hol);
1137         ft = ft + (toc - t);
1138
1139         % Check to see if we actually got better... if not,
1140         % count this as a useless iteration.
1141         hocost = hocost / Nh;
1142         % 'mhc' and 'mhe' are running minima of cost and error.
1143         % If either of them is unimproved after several iterations,
1144         % we terminate.

```

```

1140     if or(hocost > mhc * (1 - p.hodelmin), hocerr > mhe * (1 - p.hodelmin))
1141         esits = esits + 1;
1142     else
1143         esits = 0;
1144     end
1145     if hocost < mhc * (1 - p.hodelmin)
1146         mhc = hocost;
1147     end
1148     if hocerr < mhe * (1 - p.hodelmin)
1149         mhe = hocerr;
1150     end
1151
1152     % Append results to our record arrays...
1153     hoc = [hoc, hocost];
1154     hoe = [hoe, hocerr];
1155     %% if it > 1
1156     %%     dlcost = (hocost - pvcost) / pvcost;
1157     %% else
1158     %%     dlcost = -1;
1159     %% end
1160     %% %% This is "consecutive it's with no improvement"
1161     %% if p.esi > 0
1162     %%     if dlcost > -p.hodelmin
1163     %%         esits = esits + 1;
1164     %%     else
1165     %%         esits = 0;
1166     %%     end
1167     %% end
1168 end
1169 else
1170     hocost = 1;
1171 end
1172 %% If we have test data, calculate and record current cost
1173 if ~isempty(p.tsi)
1174     [~, tscost, tscerr] = forwardprop(w, p.tsi, p.tsl);
1175     tscost = tscost / Nt;
1176     tsc = [tsc, tscost];
1177     tse = [tse, tscerr];
1178 else
1179     tscost = 1;
1180 end
1181
1182 % Finalize stop parameters and increase iteration count!
1183 % trcost;
1184 stpdat = [trcost, tscost, hocost, -it, -esits, abs(dlcost)];
1185 it = it + 1;
1186
1187 % If we're using batches, get a new shuffle.
1188 if p.batchsize < N
1189     shuffle = randperm(N);
1190 end
1191 end
1192
1193 stpdat > stpmin;
1194
1195 % Store our results in the 'out' structure.
1196 if ~isempty(p.tsi)
1197     out.tsc = tsc;
1198     out.tse = tse;
1199 end
1200 if ~isempty(p.hoi)
1201     out.hoc = hoc;
1202     out.hoe = hoe;
1203 end
1204 out.trc = trc;

```

```

1200 out.tre = tre;
1201 out.it = it - 1;
1202 out.ft = ft;
1203 out.bt = bt;
1204 out.mt = mt;
1205 out.tt = toc;
1206
1207 1 %% Test cases and plots for PA2
1208 2
1209 3 clear all
1210 4 close all
1211 5 %% Problem 3) Base case - 1 hidden layer, include momentum, regularization,
1212 6 %annealing
1213 7
1214 8
1215 9 testcomp = struct;
1216 10 k=0;
1217 11
1218 12 %Import data
1219 13
1220 14 %% Generates test cases and plots for Programming Assignment 1
1221 15
1222 16
1223 17 %% This folder has the loading code used to access the MNIST data
1224 18 addpath('mnistHelper');
1225 19
1226 20 %% Load the data, take only the first 20000 training points and 2000 test points
1227 21 images = loadMNISTImages('mnistdata/train-images.idx3-ubyte');
1228 22 labels = loadMNISTLabels('mnistdata/train-labels.idx1-ubyte');
1229 23 tsimages = loadMNISTImages('mnistdata/t10k-images.idx3-ubyte');
1230 24 tslabels= loadMNISTLabels('mnistdata/t10k-labels.idx1-ubyte');
1231 25
1232 26 %% Pre-process
1233 27 % images = images/255;
1234 28 images = images -mean(images);
1235 29 % sim = sim/255;
1236 30 tsimages = tsimages-mean(tsimages);
1237 31
1238 32 %% Cuttin Data
1239 33 % trkeep = numel(labels);
1240 34 % tskeep = numel(tslabels);
1241 35 trkeep = 20000;
1242 36 tskeep = 2000;
1243 37 [tri,trl,sim,slb,N,Nt,hoi,hol] = importimages(images, labels,tsimages,tslabels, trkeep,
1244 38 tskeep);
1245 39
1246 40 %% Weight initialization (zeros for part Problem 3)
1247 41 d = size(tri, 1);
1248 42 ncl = max(trl);
1249 43
1250 44 weight_opts.layers = [256];
1251 45 weight_opts.funs = {'sigmoid','softmax'};
1252 46 weight_opts.distr = 'zeros';
1253 47 W = initw(weight_opts,d,ncl);
1254 48
1255 49 %% Set the data and parameters for the experiment
1256 50 batchsize = N;
1257 51 p.batchsize = batchsize;
1258 52 p.tri = tri;
1259 53 p.trl = trl;
1260 54 p.hoi = hoi;
1261 55 p.hol = hol;
1262 56 p.tsi = sim;
1263 57 p.tsl = slb;
1264 58 p.hodelmin = 0.0001;
1265 59 p.homin = 0.03;

```

```

1260 59 p.eta = 0.1;
1261 60 p.shuffle = 0;
1262 61 p.ann = 'hyp';
1263 62 T = 256;
1264 63 p.annpar = 1 / T;
1265 64 % p.trmin = 0.01;
1266 65 p.maxit = 1024;
1267 66 p.esi = 3;
1268 67 p.mom = 0;
1269 68 p.l1 = 0;
1270 69 p.l2 = 0.0005;
1271 70 p.reprate = round(N / batchsize);
1272 71 % p.trdelmin = 0.0001;
1273 72 p.winit = W;
1274 73
1275 74 p = netregdefs(p);
1276 75
1277 76 x = p.tri;
1278 77 labs = p.trl;
1279 78
1280 79 %% Run Algorithm and Generate Plots
1281 80
1282 81
1283 82 [w, out] = netreg(p);
1284 83 [classtrain, tre] = classifier(w, p.tri, labs);
1285 84 [classtest, tse] = classifier(w, sim, slb);
1286 85
1287 86 %%
1288 87 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1289 88 iters = 1:length(out.trc);
1290 89
1291 90 titlep = 'Base Case';
1292 91 filename = 'base_f';
1293 92 makeplots(iters, out, titlep, filename)
1294 93
1295 94 k=k+1;
1296 95 testcomp(k).test = filename;
1297 96 testcomp(k).numits = length(out.trc);
1298 97 testcomp(k).accuracy_train = 1-min(tre);
1299 98 testcomp(k).accuracy_test = 1-min(tse);
1300 99
1301 100 %% Check gradient comparison
1302 101 % gradestprep;
1303 102 % gradest;
1304 103
1305 104 %% Problem 4) Tricks of trade
1306 105
1307 106 % trkeep = 20000;
1308 107 % tskeep = 2000;
1309 108 % [tri, trl, sim, slb, N, Nt, hoi, hol] = importimages(images, labels, tsimages, tslabels, trkeep,
1310 109 tskeep);
1311 110 trkeep = numel(labels);
1312 111 tskeep = numel(tslabels);
1313 112 % trkeep = 20000;
1314 113 % tskeep = 2000;
1315 114 [tri, trl, sim, slb, N, Nt, hoi, hol] = importimages(images, labels, tsimages, tslabels, trkeep,
1316 115 tskeep);
1317 116
1318 117 %% Base case for small set
1319 118
1320 119 % batchsize = N;
1321 120 % p.batchsize = batchsize;
1322 121 % % p.tri = tri;
1323 122 % % p.trl = trl;
1324 123 % p.hoi = hoi;

```



```

1320 % p.hol = hol;
1321 % p.tsi = [];
1322 % p.tsl = [];
1323 % p.reprate = round(N / batchsize);
1324 %
1325
1326 % x = p.tri;
1327 % labs = p.trl;
1328 % [w, out] = netreg(p);
1329 % [classtrain, tre] = classifier(w, x, labs);
1330 % [classtest, tse] = classifier(w, sim, slb);
1331
1332 %% a) Shuffle
1333 clear p
1334
1335 batchsize = 128;
1336 p.batchsize = batchsize;
1337 p.tri = tri;
1338 p.trl = trl;
1339 p.hoi = hoi;
1340 p.hol = hol;
1341 % p.tsi = sim;
1342 % p.tsl = slb;
1343 p.hodelmin = 0.0001;
1344 p.homin = 0.03;
1345 p.eta = 0.1;
1346 p.shuffle = 1;
1347 p.ann = 'hyp';
1348 T = 256;
1349 p.annpar = 1 / T;
1350 % p.trmin = 0.01;
1351 p.maxit = 2048;
1352 p.esi = 8;
1353 p.mom = 0;
1354 p.l1 = 0;
1355 p.l2 = 0.0005;
1356 p.reprate = round(N / batchsize);
1357 % p.trdelmin = 0.0001;
1358 p.winit = W;
1359
1360 p = netregdefs(p);
1361
1362 %% Run Algorithm and Classify
1363
1364 [w, out] = netreg(p);
1365 [classtrain, tre] = classifier(w, x, labs);
1366 [classtest, tse] = classifier(w, sim, slb);
1367
1368 %%
1369 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1370 iters = 1:length(out.trc);
1371
1372 titlep = 'Shuffle';
1373 filename = 'shuffle_f';
1374 makeplots(iters, out, titlep, filename)
1375
1376 k=k+1;
1377 testcomp(k).test = filename;
1378 testcomp(k).numits = length(out.trc);
1379 testcomp(k).accuracy_train = 1-min(tre);
1380 testcomp(k).accuracy_test = 1-min(tse);
1381
1382 %% c) Use tanh sigmoid
1383
1384 clear W

```

```

1380 187 weight_opts.layers = [256];
1381 188 weight_opts.funs = {'tanh', 'softmax'};
1382 189 weight_opts.distr = 'zeros';
1383 190 W = initw(weight_opts,d,ncl);
1384 191
1385 192
1386 193 %%
1387 194 clear p
1388 195
1389 196 batchsize = 128;
1390 197 p.batchsize = batchsize;
1391 198 p.tri = tri;
1392 199 p.trl = trl;
1393 200 p.hoi = hoi;
1394 201 p.hol = hol;
1395 202 % p.tsi = sim;
1396 203 % p.tsl = slb;
1397 204 p.hodelmin = 0.0001;
1398 205 p.homin = 0.03;
1399 206 p.eta = 0.1;
1400 207 p.shuffle = 1;
1401 208 p.ann = 'hyp';
1402 209 T = 256;
1403 210 p.annpar = 1 / T;
1404 211 % p.trmin = 0.01;
1405 212 p.maxit = 2048;
1406 213 p.esi = 8;
1407 214 p.mom = 0;
1408 215 p.l1 = 0;
1409 216 p.l2 = 0.0005;
1410 217 p.reprate = round(N / batchsize);
1411 218 % p.trdelmin = 0.0001;
1412 219 p.winit = W;
1413 220
1414 221 p = netregdefs(p);
1415 222 %% Run Algorithm and Classify
1416 223
1417 224 [w, out] = netreg(p);
1418 225 [classtrain, tre] = classifier(w, x, labs);
1419 226 [classtest, tse] = classifier(w, sim, slb);
1420 227
1421 228 iters = 1:length(out.trc);
1422 229
1423 230 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1424 231 titlep = 'Using tanh as sigmoid';
1425 232 filename = 'sig-tanh-f';
1426 233 makeplots(iters,out,titlep,filename)
1427 234
1428 235 k=k+1;
1429 236 testcomp(k).test = filename;
1430 237 testcomp(k).numits = length(out.trc);
1431 238 testcomp(k).accuracy_train = 1-min(tre);
1432 239 testcomp(k).accuracy_test = 1-min(tse);
1433 240
1434 241 %% d) Random normal weight distrubiton with mean and std 1/sqrt(fan-in)
1435 242 clear W
1436 243 weight_opts.layers = [256];
1437 244 weight_opts.funs = {'tanh', 'softmax'};
1438 245 weight_opts.distr = 'randnorm';
1439 246 W = initw(weight_opts,d,ncl);
1440 247 %%
1441 248
1442 249 clear p
1443 250
1444 251 batchsize = 128;

```

```

1440 252 p.batchsize = batchsize;
1441 253 p.tri = tri;
1442 254 p.trl = trl;
1443 255 p.hoi = hoi;
1444 256 p.hol = hol;
1445 257 % p.tsi = sim;
1446 258 % p.tsl = slb;
1447 259 p.hodelmin = 0.0001;
1448 260 p.homin = 0.03;
1449 261 p.eta = 0.1;
1450 262 p.shuffle = 1;
1451 263 p.ann = 'hyp';
1452 264 T = 256;
1453 265 p.annpar = 1 / T;
1454 266 % p.trmin = 0.01;
1455 267 p.maxit = 2048;
1456 268 p.esi = 8;
1457 269 p.mom = 0;
1458 270 p.l1 = 0;
1459 271 p.l2 = 0.0005;
1460 272 p.reprate = round(N / batchsize);
1461 273 % p.trdelmin = 0.0001;
1462 274 p.winit = W;
1463 275
1464 276 p = netregdefs(p);
1465 277
1466 278 %% Run Algorithm and Classify
1467 279
1468 280 [w, out] = netreg(p);
1469 281 [classtrain, tre] = classifier(w, x, labs);
1470 282 [classtest, tse] = classifier(w, sim, slb);
1471 283
1472 284 iters = 1:length(out.trc);
1473 285 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1474 286 titlep = 'Random normal weight initialization';
1475 287 filename = 'rnweights_f';
1476 288 makeplots(iters, out, titlep, filename)
1477 289
1478 290 k=k+1;
1479 291 testcomp(k).test = filename;
1480 292 testcomp(k).numits = length(out.trc);
1481 293 testcomp(k).accuracy_train = 1-min(tre);
1482 294 testcomp(k).accuracy_test = 1-min(tse);
1483 295 %% e) use momentum
1484 296
1485 297 clear p
1486 298
1487 299 batchsize = 128;
1488 300 p.batchsize = batchsize;
1489 301 p.tri = tri;
1490 302 p.trl = trl;
1491 303 p.hoi = hoi;
1492 304 p.hol = hol;
1493 305 % p.tsi = sim;
1494 306 % p.tsl = slb;
1495 307 p.hodelmin = 0.0001;
1496 308 p.homin = 0.03;
1497 309 p.eta = 0.1;
1498 310 p.shuffle = 1;
1499 311 p.ann = 'hyp';
1500 312 T = 256;
1501 313 p.annpar = 1 / T;
1502 314 % p.trmin = 0.01;
1503 315 p.maxit = 2048;
1504 316 p.esi = 8;

```

```

1500 317 p.mom = 0.9;
1501 318 p.l1 = 0;
1502 319 p.l2 = 0.0005;
1503 320 p.reprate = round(N / batchsize);
1504 321 % p.trdelmin = 0.0001;
1505 322 p.winit = W;
1506 323
1507 324 p = netregdefs(p);
1508 325
1509 326 %% Run Algorithm and Classify
1510 327
1511 328 [w, out] = netreg(p);
1512 329 [classtrain, tre] = classifier(w, x, labs);
1513 330 [classtest, tse] = classifier(w, sim, slb);
1514 331
1515 332 iters = 1:length(out.trc);
1516 333 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1517 334 titlep = 'Including momentum';
1518 335 filename = 'momentum_f';
1519 336 makeplots(iters, out, titlep, filename)
1520 337
1521 338 k=k+1;
1522 339 testcomp(k).test = filename;
1523 340 testcomp(k).numits = length(out.trc);
1524 341 testcomp(k).accuracy_train = 1-min(tre);
1525 342 testcomp(k).accuracy_test = 1-min(tse);
1526 343 %% 5) Network Topology
1527 344
1528 345
1529 346
1530 347 %% a) Double number of hidden units
1531 348
1532 349 clear W
1533 350 weight_opts.layers = [512];
1534 351 weight_opts.funs = {'tanh', 'softmax'};
1535 352 weight_opts.distr = 'randnorm';
1536 353 W = initw(weight_opts, d, ncl);
1537 354 %%
1538 355 clear p
1539 356
1540 357 batchsize = 128;
1541 358 p.batchsize = batchsize;
1542 359 p.tri = tri;
1543 360 p.trl = trl;
1544 361 p.hoi = hoi;
1545 362 p.hol = hol;
1546 363 % p.tsi = sim;
1547 364 % p.tsl = slb;
1548 365 p.hodelmin = 0.0001;
1549 366 p.homin = 0.03;
1550 367 p.eta = 0.1;
1551 368 p.shuffle = 1;
1552 369 p.ann = 'hyp';
1553 370 T = 256;
1554 371 p.annpar = 1 / T;
1555 372 % p.trmin = 0.01;
1556 373 p.maxit = 2048;
1557 374 p.esi = 8;
1558 375 p.mom = 0.9;
1559 376 p.l1 = 0;
1560 377 p.l2 = 0.0005;
1561 378 p.reprate = round(N / batchsize);
1562 379 % p.trdelmin = 0.0001;
1563 380 p.winit = W;
1564 381

```

```

1560 p = netregdefs(p);
1561
1562
1563 %% Run Algorithm and Classify
1564
1565 [w, out] = netreg(p);
1566 [classtrain, tre] = classifier(w, x, labs);
1567 [classtest, tse] = classifier(w, sim, slb);
1568 iters = 1:length(out.trc);
1569
1570 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1571 titlep = 'Double hidden units';
1572 filename = 'double_f';
1573 makeplots(iters, out, titlep, filename)
1574
1575 k=k+1;
1576 testcomp(k).test = filename;
1577 testcomp(k).numits = length(out.trc);
1578 testcomp(k).accuracy_train = 1-min(tre);
1579 testcomp(k).accuracy_test = 1-min(tse);
1580 %% a.2) Half number of hidden units
1581
1582 clear W
1583 weight_opts.layers = [128];
1584 weight_opts.funs = {'tanh', 'softmax'};
1585 weight_opts.distr = 'randnorm';
1586 W = initw(weight_opts, d, ncl);
1587
1588 %%
1589
1590 clear p
1591
1592 batchsize = 128;
1593 p.batchsize = batchsize;
1594 p.tri = tri;
1595 p.trl = trl;
1596 p.hoi = hoi;
1597 p.hol = hol;
1598 % p.tsi = sim;
1599 % p.tsl = slb;
1600 p.hodelmin = 0.0001;
1601 p.homin = 0.03;
1602 p.eta = 0.1;
1603 p.shuffle = 1;
1604 p.ann = 'hyp';
1605 T = 256;
1606 p.annpar = 1 / T;
1607 % p.trmin = 0.01;
1608 p.maxit = 2048;
1609 p.esi = 8;
1610 p.mom = 0.9;
1611 p.l1 = 0;
1612 p.l2 = 0.0005;
1613 p.reprate = round(N / batchsize);
1614 % p.trdelmin = 0.0001;
1615 p.winit = W;
1616
1617 p = netregdefs(p);
1618 %% Run Algorithm and Classify
1619
1620 [w, out] = netreg(p);
1621 [classtrain, tre] = classifier(w, x, labs);
1622 [classtest, tse] = classifier(w, sim, slb);
1623 iters = 1:length(out.trc);

```

```

1620 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1621 titlep = 'Half Hidden Units';
1622 filename = 'half_f';
1623 makeplots(iters ,out ,titlep ,filename)
1624
1625 k=k+1;
1626 testcomp(k).test = filename;
1627 testcomp(k).numits = length(out.trc);
1628 testcomp(k).accuracy_train = 1-min(tre);
1629 testcomp(k).accuracy_test = 1-min(tse);
1630 %% b) Add another hidden layer
1631
1632 % trkeep = 20000;
1633 % tskeep = 2000;
1634 % [tri ,trl ,sim ,slb ,N,Nt,hoi ,hol] = importimages(images , labels ,tsimages ,tslabels , trkeep ,
1635 % tskeep);
1636
1637 %%
1638 clear W
1639 weight_opts.layers = [204,204];
1640 weight_opts.funs = {'tanh','tanh','softmax'};
1641 weight_opts.distr = 'randnorm';
1642 W = initw(weight_opts,d,ncl);
1643
1644 %%
1645 clear p
1646
1647 batchsize = 128;
1648 p.batchsize = batchsize;
1649 p.tri = tri;
1650 p.trl = trl;
1651 p.hoi = hoi;
1652 p.hol = hol;
1653 p.tsi = sim;
1654 p.tsl = slb;
1655 p.hodelmin = 0.0001;
1656 p.homin = 0.03;
1657 p.eta = 0.1;
1658 p.shuffle = 1;
1659 p.ann = 'hyp';
1660 T = 256;
1661 p.annpar = 1 / T;
1662 % p.trmin = 0.01;
1663 p.maxit = 2048;
1664 p.esi = 8;
1665 p.mom = 0.9;
1666 p.l1 = 0;
1667 p.l2 = 0.0005;
1668 p.reprate = round(N / batchsize);
1669 % p.trdelmin = 0.0001;
1670 p.winit = W;
1671
1672 p = netregdefs(p);
1673 %% Run Algorithm and Classify
1674
1675 [w, out] = netreg(p);
1676 [classtrain , tre] = classifier(w, x, labs);
1677 [classtest , tse] = classifier(w, sim, slb);
1678 iters = 1:length(out.trc);
1679 % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1680 titlep = 'Add second hidden layer';
1681 filename = 'twohidlayers_f';
1682 makeplots(iters ,out ,titlep ,filename)
1683

```

```

1680 k=k+1;
1681 testcomp(k).test = filename;
1682 testcomp(k).numits = length(out.trc);
1683 testcomp(k).accuracy_train = 1-min(tre);
1684 testcomp(k).accuracy_test = 1-min(tse);
1685
1686
1687 save('testresults',testcomp);
1688 %%% b.2 Change hidden layers size
1689 %
1690 % weight_opts.layers = [300,100];
1691 % weight_opts.funs = {'tanh','tanh','softmax'};
1692 % weight_opts.distr = 'randnorm';
1693 % W = initw(weight_opts,d,ncl);
1694 % p.winit = W;
1695 %
1696 %%% Run Algorithm and Classify
1697 % [w, out] = netreg(p);
1698 % [classtrain, tre] = classifier(w, p.tri, labs);
1699 % [classtest, tse] = classifier(w, sim, slb);
1700 % iters = 1:length(out.trc);
1701 %
1702 % % epochs = 1/p.reprate:1/p.reprate:length(out.trc)/p.reprate;
1703 % titlep = 'Change units of both hidden layer';
1704 % filename = 'twohiddif';
1705 % makeplots(iters,out,titlep,filename)
1706 %
1707 % k=k+1;
1708 % testcomp(k).test = filename;
1709 % testcomp(k).numits = length(out.trc);
1710 % testcomp(k).accuracy_train = 1-min(tre);
1711 % testcomp(k).accuracy_test = 1-min(tse);
1712
1713
1714
1715 function [tri, tsi] = preprocess(tri, tsi)
1716 % Subtract mean from training and testing data.
1717 tri = tri - mean(tri, 2);
1718 tsi = tsi - mean(tsi, 2);
1719
1720
1721
1722 function rv = stringin(str, strlist)
1723 % Just want to check if 'str' is an element of 'strlist'
1724 rv = 0;
1725 N = size(strlist, 2);
1726 for i = 1 : N
1727     if strcmp(str, deblank(strlist(i, :)))
1728         rv = 1;
1729         return;
1730     end
1731 end
1732 return;
1733 end
1734
1735
1736
1737
1738
1739

```