10-601: Homework 3

Due: 9 October 2014 11:59pm (Autolab)

TAs: Henry Gifford, Jin Sun

You should submit all code you modify to autolab. Please only modify files if it is indicated that you may modify them. Consult the README for more information.

For this part of the assignment **you may not work in groups**. You may ask clarifying questions on Piazza. However, under no circumstances should you reveal any part of the answer publicly on Piazza or any other public website. The intention of this policy is to facilitate learning, not circumvent it. Any incidents of plagiarism will be handled in accordance with CMU's Policy on Academic Integrity.

1 Logistic Regression (40 points) - Harry

In this section you will implement the two class Logistic Regression classifier and evaluate its performance on digit recognition.

1.1 The Dataset

The dataset we are using for this assignment is a subset of the MNIST handwritten digit database [2], which is a set of 70,000 28×28 handwritten digits from a mixture of high school students and government Census Bureau employees. Your goal will be to write a logistic regression classifier to distinguish between a collection of 4s and 7s, of which you can see some examples in Figure 1.

The data is given to you in the form of a *design matrix* X and a vector y of labels indicating the class. There are two design matrices, one for training and one for evaluation. The design matrix is of size $m \times n$, where m is the number of examples and n is the number of features. This should be familiar to you from Homework 2, where you used a document-term matrix. In this case we will treat each pixel as a feature, giving us $m = 28 \times 28 = 784$.

Figure 1: A set of 64 digits from the MNIST dataset.

1.2 Training Logistic Regression

Given a set of training points $x_1, x_2, ..., x_m$ and a set of labels $y_1, ..., y_m$ we want to estimate the parameters of the model w. We can do this by maximizing the log likelihood function, as we did in MLE. We derived the cost function for logistic regression in class, so we will not repeat the derivation here.

1.2.1 Cost function

First, define the sigmoid logistic function as follows.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

Then the cost and gradient are

$$J(w) = \frac{\lambda}{2}||w||_2^2 - \sum_{i=1}^{m} y_i \log S\left(w^\top x_i\right) + (1 - y_i) \log\left(1 - S\left(w^\top x_i\right)\right) \tag{2}$$

$$\nabla J(w) = \lambda w - \sum_{i=1}^{m} \left(y_i - S\left(w^\top x_i\right)\right) x_i \tag{3}$$

Note that the cost function now has this extra term $\frac{\lambda}{2}||w||_2^2$. This is called a *regularization* term. Regularization forces the parameters of the model to be pushed towards zero by penalizing large w values. This helps to prevent overfitting and also makes the objective function strictly convex, which means that there is a unique solution.

Note For this assignment please regularize the intercept term too. i.e. $w^{(0)}$ should also be regularized. Many resources about Logistic Regression on the web do not regularize the intercept term, so be aware if you see different objective functions.

(20 points) Implement the cost function and compute the cost and gradient for logistic regression. The cost function is located in `costLR.m`. You can run `run_logit.m` to check whether your gradients match the cost. The script should pass the gradient checker and then stop.

One small technicality to be aware of is that in the cost functions below, we assume that we have absorbed the intercept term for the model into the x_i's for convenience. Remember that our model of the log odds is $\log\left(\frac{p}{1-p}\right) = w^{(0)} + w^{(1)}x^{(1)} + ... + w^{(n)}x^{(n)}$. In order to keep the notation clean and make the implementation easier, we assume that each x_i has been augmented with an extra 1 at the beginning, i.e. $x_i' = [1; x_i]$.

1.2.2 Optimization

For models such as linear regression we were able to find a closed form solution for the parameters of the model. Unfortunately, for many machine learning models, including Logistic Regression, no such closed form solutions exist. Therefore we will use a gradient based method to find our parameters.

It is convention that we frame optimization problems as minimization problems, but notice that

$$\max_w f(w) = \min_w -f(w)$$

so we will minimize the negative log likelihood function, or cost function, which we denote as J.

Recall from class that the update rule for gradient descent is

$$w_{i+1} = w_i - \alpha d^i \nabla J(w_i) \tag{4}$$

where α specifies the learning rate, or how large a step we wish to take. d is a decay term that we use to ensure that the step sizes we make will gradually get smaller, so long as we converge. Since we know that the cost function for logistic regression is convex (bowl shaped), this method will give us a way to find w^*, the solution.

(10 points) Implement gradient descent in `minimize.m`.

(5 points) Use your minimizer to complete `trainLR.m`.

1.3 Testing Logistic Regression

Once you have trained the model, you can then use it to make predictions. You will now implement `predictLR` which will generate the most likely classes for a given x_i.

Notation	Meaning	Type
m	number of training examples	scalar
n	number of features	scalar
x_i	ith augmented training data point (one digit example)	$(n+1) \times 1$
X	design matrix (all training examples)	$m \times (n+1)$
y_i	ith training label (is the digit a 7?)	$\{0, 1\}$
Y or y	all training labels	$m \times 1$
w	parameter vector	$(n+1) \times 1$
S	sigmoid function, $S(t) := (1 + e^{-t})^{-1}$	$\dim(t) \to \dim(t)$
J	cost (loss) function	$\mathbb{R}^n \to \mathbb{R}$
∇J	gradient of J (vector of derivatives in each dimension)	$\mathbb{R}^n \to \mathbb{R}^n$
H_J	hessian of J (matrix of second order derivatives)	$\mathbb{R}^n \to \mathbb{R}^{n \times n}$
α	(parameter) gradient descent learning rate	scalar
d	(parameter) decay constant for α to decrease by every iteration	scalar
λ	(parameter) regularization strength	scalar

Table 1: Summary of notation used for Logistic Regression

(5 points) Implement `predictLR.m`. `predictLR.m` should predict the classes for each data point and return them.

1.3.1 Putting it together

You should now be able to run `run_logit.m` to completion and see how well your model performs. You should get about 96% accuracy.

2 Neural Networks (60 points) - Harry

Now we are going to implement a simple *Feedforward Neural Network*. The feedforward neural network is the most basic type of neural network, but they are surprisingly powerful and the state of the art in many machine learning classification problems were achieved using the same basic architecture that you are about to implement.

You have three tasks in this section.

1. Implement a neural network with a single hidden layer and test it out on a simple toy dataset, known as the Banana Dataset, as shown in Figure 2.
2. Use your neural network implementation to perform handwritten digit recognition.
3. Modify your neural network to perform regression.

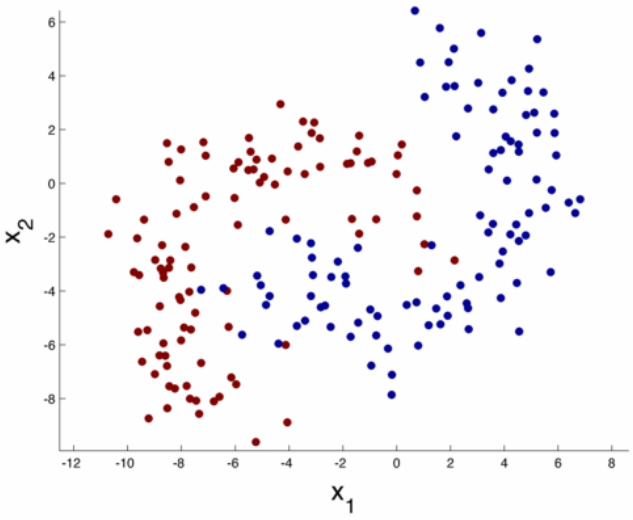

Figure 2: Banana dataset. Created from two intertwined banana shaped distributions. Red is class 1 and blue is class 0. It is not possible to classify this dataset well using a 2D linear classifier.

2.1 Implement the Neural Network

Figure 3 shows an example configuration of the neural network we will use to learn the banana dataset. The two inputs x_1 and x_2 correspond to the two axes in the figure. The final input labeled 1 is a bias term so we

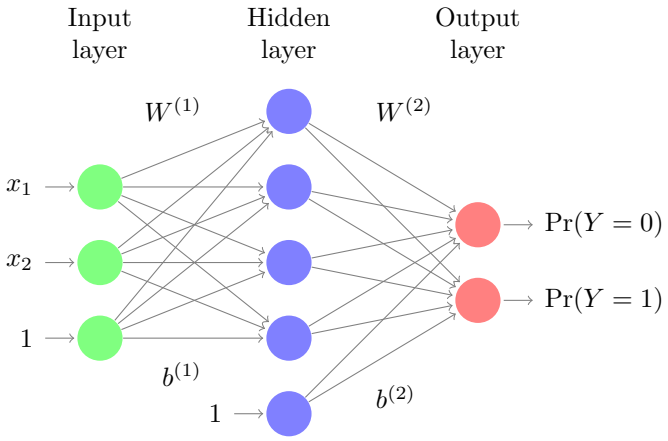

Figure 3: Schematic of a Feedforward Neural Network used for the Banana Dataset.

can have a decision boundary that does not have to go through the origin. We will use the same input and outputs, and will vary the number of units in the hidden layer with the `hidden_layer` parameter.

2.1.1 Training

We will first consider the cost and gradient for a single training datapoint pair $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^k$. For the Banana Dataset, $n = 2$ and $k = 2$.

We will use the squared error loss, similar to that used in Linear Regression. That is we want to minimize $\frac{1}{2}||a^{(3)} - y||_2^2$ for each training example, where $a^{(3)}$ is the activation from the Neural Network on a single example, which is described in detail below. We will also regularize the weights, just as we did in Logistic Regression, although we will not regularize the bias terms.

Forward propagation First we have to compute the response of the network to individual data examples. We let $W^{(i)}$ represent the weights on the edges between the i and $i+1$ layer. $b^{(i)}$ are the corresponding biases. Then we get the activations as follows

$$z^{(2)} = W^{(1)}x + b^{(1)} \tag{5}$$
$$a^{(2)} = S(z^{(2)}) \tag{6}$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{7}$$
$$a^{(3)} = S(z^{(3)}) \tag{8}$$

We can then compute the cost for a single training example as

$$J(W, b, x, y) = \frac{1}{2}||a^{(3)} - y||_2^2 \tag{9}$$

Backpropagation Now that we have computed the output activation for an input we need to update the weights of the network to adjust for error in cost. This is where we use the back propagation algorithm discussed in class. We will not re-derive the algorithm here, but we summarize the algorithm given in [3] as follows.

1. Letting $*$ denote element wise multiplication, we define the weighted error averages as

$$\delta^{(3)} = (a^{(3)} - y) * S(z^{(3)}) * (1 - S(z^{(3)})) \tag{10}$$
$$\delta^{(2)} = (W^{(2)})^\top \delta^{(3)} * S(z^{(2)}) * (1 - S(z^{(2)})) \tag{11}$$

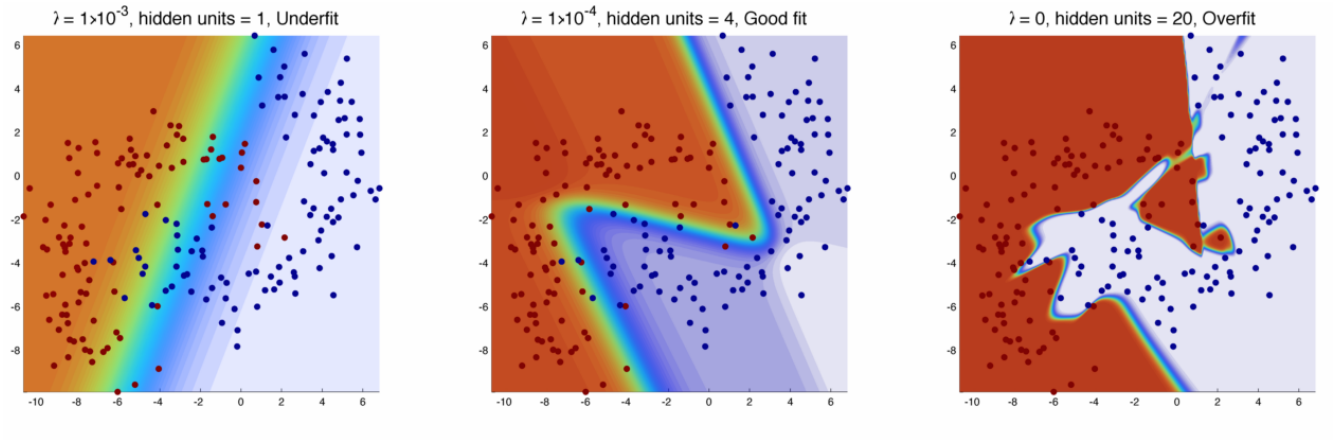

Figure 4: Three neural networks with different numbers of hidden units and different λ settings and the effect on learning.

2. The partial derivatives are then

$$\nabla J_{W^{(n)}}(W, b, x, y) = \delta^{(n+1)}(a^{(n)})^\top \tag{12}$$

$$\nabla J_{b^{(n)}}(W, b, x, y) = \delta^{(n+1)} \tag{13}$$

Putting it together Now that we have the cost $J(W, b, x, y)$ and gradient $\nabla J(W, b, x, y)$, for a single example we can compute the cost and gradient for a batch as

$$J(W, b) = \frac{\lambda}{2}(||W^{(1)}||_2^2 + ||W^{(2)}||_2^2) + \frac{1}{2}\sum_{i=1}^{m} ||a_i^{(3)} - y_i||_2^2 \tag{14}$$

$$\nabla J_{W^{(n)}}(W, b) = \lambda W^{(n)} + \sum_{x,y} \nabla J_{W^{(n)}}(W, b, x, y) \tag{15}$$

$$\nabla J_{b^{(n)}}(W, b) = \sum_{(x,y)} \nabla J_{b^{(n)}}(W, b, x, y) \tag{16}$$

(10 points) Implement the function `nnComputeActivations`. `nnComputeActivations` should return $a^{(3)}$. In general, the function should just generate and return the predictions $a^{(L+1)}$ for all the given X, where L is the total number of layers in the Neural Network. In our case, for a Neural Network with just one hidden layer, $L = 2$.

(40 points) Implement the function `costNN` in `./NN/costNN.m`. For the forward pass, we compute the activations for the forward pass, as above and then run backpropagation.

If your code passes the gradient check, then you can likely move on to the next step. Note that passing the gradient check is not a guarantee that your costs and gradients are correct!

2.1.2 Testing

Now that we have trained the parameters of our network, we want to use those parameters to make predictions. We can do this by simply running the forward pass and returning $a^{(3)}$, using the `nnComputeActivations` we just implemented.

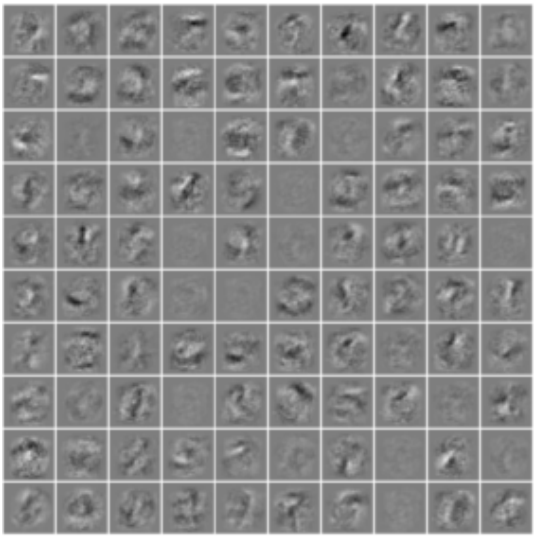

Figure 5: Visualizing the weights W_1 from the input layer to the hidden layer on a neural network with 100 hidden units. Running `digits.m` should generate a similar set of weights.

(0 points) Run `runNNClassification`. You should now be able to see a figure much like that in Figure 4. You may not see exactly the same predictions, since we are optimizing a non-convex cost function, which will have many local minima. This is in fact one of the major downsides to using a neural network, since we have no guarantees about how good our solution will be.

Model Selection You might have been wondering why we have these *hyperparameters* λ and u_i which we have to choose. Why can we not optimize λ and u_i directly?

Firstly, we are not guaranteed to even get a smooth function to optimize over if we optimize the hyperparameters directly.

More importantly, what would we use to evaluate our model? Notice that we can increase the number of hidden units arbitrarily high and decrease λ to zero and always get 0 training error. Indeed the rightmost figure gets perfect classification accuracy on the training set, but do you think it is the best model?

How we actually choose these hyperparameters is a process known as model selection, which we will come to later in the course.

2.2 Applying the Neural Network to Classifying MNIST

Now we will use the Neural Network trainer you just wrote to classify handwritten digits. Specifically, we will let each pixel of a digit be an input unit in our network and we will let the ten classes be the output neurons in the last layer. We can vary the number of hidden units, but currently that number is set to 100.

(5 points) Implement the function `nnPredictClassification`. `nnPredictCl(theta, X, output_size , input_size)` should return a $1 \times$ `size(X, 1)` vector of class predictions.

(0 points) Run `runDigits.m` This script will train and evaluate your Neural Network against 4000 examples from the MNIST dataset. Our implementation achieves around 94% accuracy. Your accuracy should be within a percent or two of this number. If it is not, check your code. Our implementation runs in less than 30 seconds on Octave, but if you have not vectorized your code this stage may take a long time to run. Now you plenty of time to vectorize it.

Notation	Meaning	Type
$\|\| \cdot \|\|_2$	ℓ_2 norm of a vector or Frobenius Norm of a matrix (ℓ_2 of the vector obtained by unrolling the matrix)	$\mathbb{R}^{a \times b} \to \mathbb{R}$
m	number of training examples	scalar
n	number of features	scalar
k	number of classes (classification) or number of output dimensions (regression)	scalar
x_i	ith training data point (one digit example)	$n \times 1$
X	design matrix (all training examples)	$m \times n$
y_i	ith training label (indicator) or training value	$k \times 1$
Y or y	all training labels	$k \times m$
u_i	number of units in i^{th} layer	natural
$W^{(i)}$	weights of edges between layer i and $i+1$	$u_{i+1} \times u_i$
$b^{(i)}$	bias as input to layer $i+1$	$u_{i+1} \times 1$
$a^{(i)}$	activation of layer i to an input x	$u_i \times 1$
$z^{(i)}$	pre-activation of layer i to an input x. i.e. the input to the activation function. Note $z^{(1)} = x$.	$u_i \times 1$
S	sigmoid function, $S(t) := (1 + e^{-t})^{-1}$	$\dim(t) \to \dim(t)$
J	cost (loss) function	$\mathbb{R}^n \to \mathbb{R}$
∇J	gradient of J (vector of derivatives in each dimension)	$\mathbb{R}^n \to \mathbb{R}^n$
α	(parameter) gradient descent learning rate	scalar
λ	(parameter) regularization strength	scalar

Table 2: Summary of notation used for Neural Networks

Weights Once `runDigits` runs to completion you should see your learned weights for the hidden layer of the neural network, as displayed in Figure 5. If your code is correct you should see weights similar to these.

For many problems, these weights can give you interesting insight into the structure of the problem you are trying to learn. The weights can often be used to make better feature representations for other machine learning algorithms. For example, when you train a neural network on patches taken from images you will often learn very similar features to those that have been long known by the computer vision community to be useful for vision tasks and are known to be similar to those used by the brain.

3 Extra Credit (up to 5 points) - Harry, Jin

Over the past five years Neural Networks have seen major increases in popularity, largely due to the fact that they work so well on a great many tasks. One reason for their popularity is that they are incredibly flexible. You have the chance to improve the performance of your neural network by exploiting this flexibility.

3.1 Rules

The rules from the previous assignments still apply to this section, but in particular we want to draw your attention to the following rules:

- You must implement a Neural Network of some sort! For example, you may not use Naive Bayes or Decision Trees to solve this problem.
- You must write all code yourself. You cannot use external libraries for any part of this problem. All extra files submitted will be ignored. If you have questions about what you may use please ask on Piazza.
- The MNIST dataset provided is exactly the same as it is in previous sections. Do not use any other data beyond the training and test set we give you to train your Neural Network. Specifically, do **not** download the MNIST dataset from the web and train on it. Using more data to get a higher classification accuracy is not the point of the exercise and will be considered cheating.

- You must store all the weights so that TAs can repeat your experiments. Non-repeatable experiments will be regarded as cheating.
- You only have 10 chances for submission. Please submit your best experiment for the last submission. We will not look at submission histories to find out the highest score.

3.2 Code Structure

You should implement your training algorithm (typically the forward propagation and back propagation) in 'train_ann.m' and testing algorithm (using trained weights to predict labels) in 'test_ann.m'. In your training algorithm, you need to store your initial and final weights into two mat files. In the simple example below, two weight matrices "W_{ih}" and "W_{ho}" are stored into "weights.mat":

```
save('weights.mat','Wih','Who');
```

We strongly recommend you to run your training algorithm on your local machine, and store all the weights begin and after the training process. Your test code should load the trained weights and do classification task on the test set. This prevents

Be sure your "test_ann.m" runs fast enough. It is always good to vectorize your code in Matlab. Please refer to "hints for fast code" post on Piazza.

3.3 Grading Policy

After the deadline, all submissions will be re-evaluated and final scores will be posted on the scoreboard. The grading policies are as follows:

- Students ranking from 1 to 5 will get 5 extra points on this assignment.
- Students ranking from 6 to 10 will get 3 extra points on this assignment.
- Students participating this competition with **a valid neural network model** and **a better-than-random-guess classification accuracy (10%)** will get 1 extra point on this assignment.

3.4 Possible Ideas

There are plenty of things you can do to improve the performance of your neural network, but here are some possible starting points.

- Add extra layers to the Neural Network. While this doesn't increase the representative power of the neural network it does allow you to represent more complex models with less hidden units and thus less parameters.
- Add a better form of regularization to the neural network. What happens if you use ℓ_1 instead of ℓ_2. Hint: [4]
- Another form of implicit regularization is to randomly drop half of the hidden units during training on each data point. This helps the hidden units to all learn useful information without relying heavily on other units. This is commonly known as *Dropout*. See [1] for more information.
- Play with the activation functions. Try tanh or rectified linear units. Do they improve your performance?
- Many other options... Check the verbose guide in the handout folder.

3.5 Hints

3.5.1 Training

In previous parts of the assignment you were given the values of hyperparameters of the Neural Network model, such as λ. Your Neural Network model may not work well with the default hyperparameters and so you will need to choose them. One simple method is to divide your data into three sets: a training set, a validation set, a test set. You can use any sizes for three sets as long as they are reasonable (e.g. 60%, 20%, 20%). Use the training and validation sets to select your hyper-parameters and then use the test set at the end to evaluate your choice of hyper-parameters.

Alternatively, combine the training set and the validation set and do *k-fold cross-validation*, where k is the number of hyper parameter selections you wish to test. Make sure to have balanced numbers of instances for each class in every set.

If the training accuracy is much higher than validation accuracy, the model is overfitting; if the training accuracy and validation accuracy are both very low, the model is underfitting;

3.5.2 Optimization

Batch Gradient Descent vs Stochastic Gradient Descent Should I use batch gradient descent?

+ Batch gradient descent usually converges faster than stochastic gradient descent.
+ Batch gradient descent can exploit vectorized code more effectively, giving it a major performance advantage on small to medium sized datasets.
+ Can use second order derivative information to converge more quickly to the solution.
– Can take a long time to run on very large datasets.
– It is more difficult to exploit cost functions which have randomness built in.

In practice, most people use *mini-batches*, where we run batch gradient descent, but on a fixed number of samples from the dataset. Then we get that 1-batch gradient descent is simply stochastic gradient descent and n-batch gradient descent is batch gradient descent, where n is the total number of training samples.

Momentum Recall that the cost function for a Neural Network is non-convex. This means that we can have many different local minima. One way to escape from a bad minimum is adding a momentum term into weight updates. The momentum term is $\alpha * \Delta W(n-1)$ in Equation 17, where n denotes the number of iterations. By adding this term to the update rule, the weights will have some chance to escape from minimum. You can set initial momentum to zero.

$$\Delta W(n) = \nabla_W J(W, b) + \alpha * \Delta W(n-1) \tag{17}$$

The intuition behind this approach is the same as this term in physics systems. In Figure 6, assume weights grow positively during training, without the momentum term, the neural net will converge to point A. If we add the momentum term, the weights may jump over the barrier and converge to a better minimum at point B.

Have fun and good luck!

4 What to hand in

For logistic regression and neural networks, you should put all required files into a folder called "submission", and compress your theoretical question pdf and "submission" folder into a single tgz file called HW3.tgz.

```
tar cvf HW3.tgz submission hw3.pdf
```

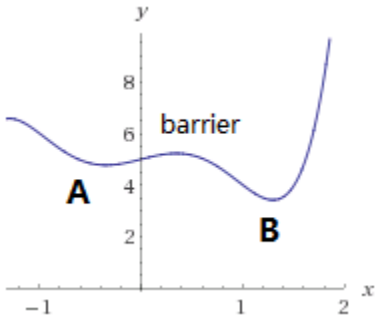

Figure 6: Non-convex function with two local minima.

For the extra credit problem, you should submit another tgz file containing all four files required to a separate link on Autolab.

```
tar cvf HW3extra.tgz ini_weights.mat weights.mat train_ann.m test_ann.m
```

File	Points
./LogisticRegression/costLR.m	20
./LogisticRegression/minimize.m	10
./LogisticRegression/trainLR.m	5
./LogisticRegression/predictLR.m	5
./NN/nnComputeActivations.m	15
./NN/costNN.m	40
./NN/nnPredictClassification.m	5
./train_ann.m ./test_ann.m ./ini_weights.mat ./weights.mat	(5)
Total	100(+5)

References

[1] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR abs/1207.0580* (2012).

[2] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (Nov 1998), 2278–2324.

[3] Ng, A. Sparse autoencoder. CS294A Lecture notes, January 2011.

[4] Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B 58* (1994), 267–288.