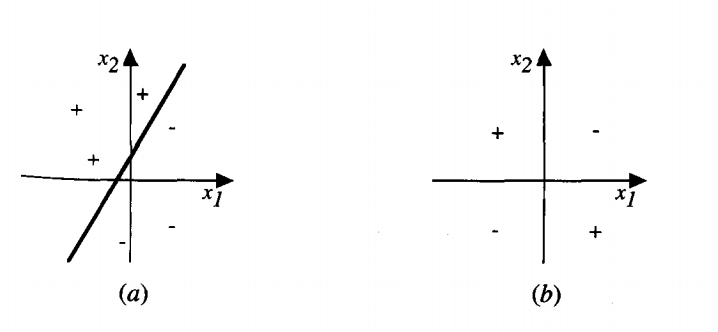
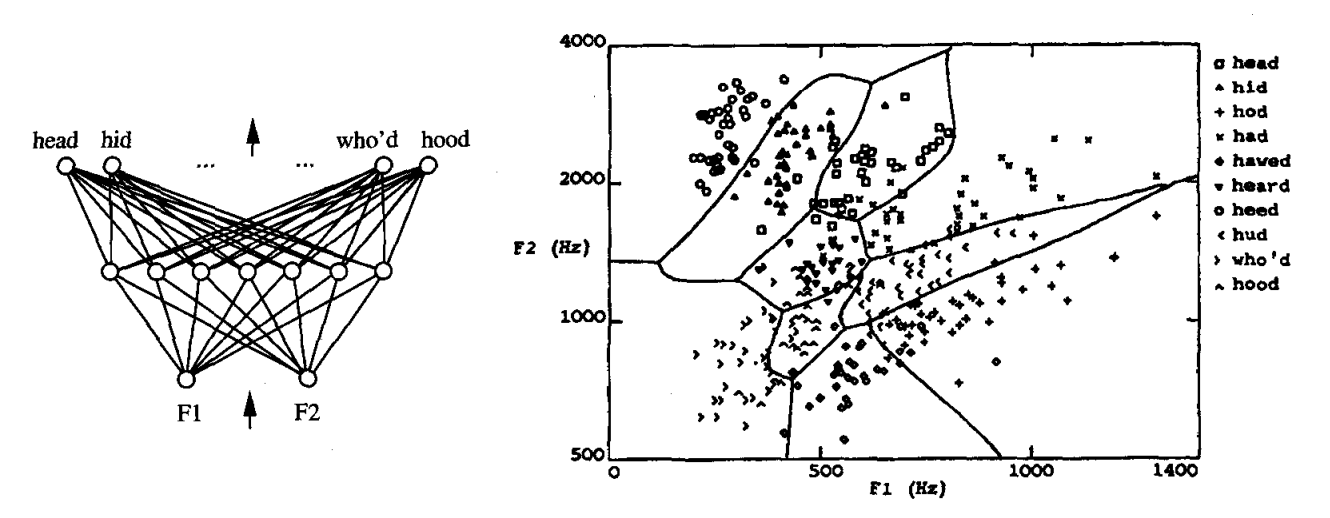
**Problem One**

**A.** No, a single-layer perceptron cannot represent non-linear decision surfaces. If a set of training examples cannot be accurately classified by a straight line, also known as being linearly separable, then the data set cannot be represented by a single-layer perceptron. As such, non-linear decision surfaces require multi-layered perceptrons to correctly classify the data because a single-layer perceptron would never converge. In the below images, found on page 87 of the article, this notion is clearly illustrated. In image (a), the data points can clearly be classified using a straight line. In image (b), however, a straight line cannot be drawn to divide the two classes. In turn, a single-layer perceptron cannot classify the data set.



**B.** No, linear activation functions also can only represent linear functions. Even layered linear functions can only produce linear decision boundaries. Meaning, a multi-layer perceptron with linear functions would not be able to represent a nonlinear decision boundary.

**C.** Yes, there is a substantial advantage to using sigmoid activation functions over linear activation functions in multi-layer perceptrons. As discussed in part B, multi-layer perceptrons with linear activation functions can only model linear boundaries. With this, it essentially has the same capacity as a single-layer perceptron. On the other hand, sigmoid activation functions enable a multi-layer perceptron to model non-linear boundaries. An example of a multi-layered surface and its ensuing classification, as provided by the text is below. It is evident that this enables perceptrons to be effectively applied to a far larger breadth of data sets.



D. **NEED TO DO THIS:** would not be monotonical, steepness of threshold?

**Problem Two**

**A.** An RBM is a neural network that takes a set of inputted parameters, and then learn a probability distribution. This neural network consists of nodes that form a bipartite graph. Because it is restricted, every pair of nodes between the group of hidden units and group of visible units must have a connection between them. Moreover, there cannot be connections within a given group. This is contrary to the nature of an Unrestricted Boltzmann Machine, which permits connections within the group of hidden units. This set of restrictions enforced by RBMs enables increasingly efficient training algorithms.

**B.** A deep belief network is a neural network that consists of multiple layers of hidden units. As depicted through a graph, nodes in one layer must have connections with nodes in other layers, but cannot have interconnected nodes in the same given layer. The nature of deep belief networks greatly resembles the bipartite nature of RBMs discussed in part A. Given this, a deep belief network can be formed by essentially stacking RBMs such that one layer’s group of hidden units acts as the subsequent layer’s group of visible units.

**Problem Three**

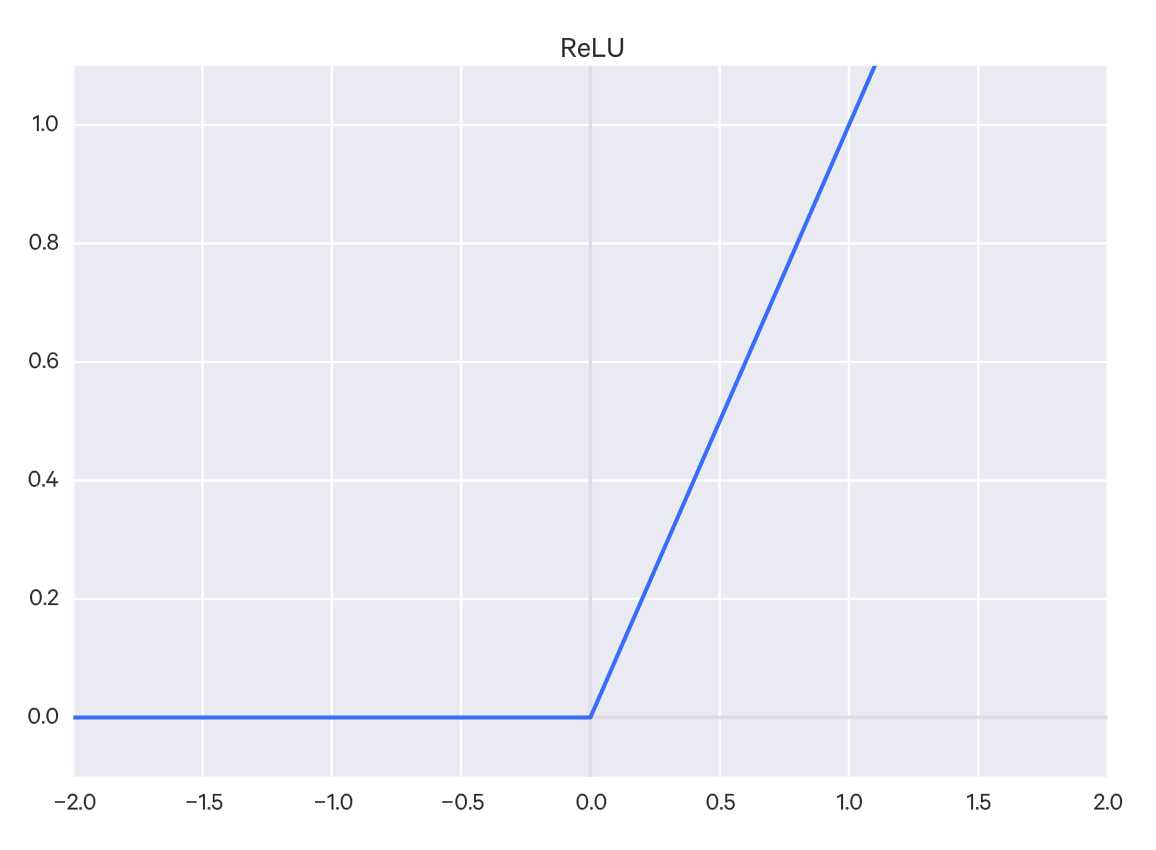
**A.** A kernel machine is a shallow architecture, meaning it only has one hidden layer like SVMs. This is contrary to a deep neural network, which consists of 2 or more hidden layers. With this, the article states that a kernel machine possesses one layer of template matchers analyzing the training set, and then one layer of coefficients that can be trained through linear combination. Here, the first layer falls under the realm of unsupervised training, while the second layer falls under the realm of supervised training. However, there are two distinct limitations with kernel machines. One is that they are innately inefficient due the vast number of required computational elements. The other limitation of kernel machines that they are constrained to localized bounds. Deeper architectures are able to abstract to higher levels and combine lower level elements together. With this, they can generalize beyond local bounds such as direct neighbors, which betters their ability to handle increasingly complex tasks (unlike kernel machines).

**B.** As discussed in the prior section, deep architectures have multiple layers of hidden variables that are trainable, as demonstrated through multi-layer neural networks. Through this structure, deep architectures provide greater training flexibility, with multiple sources of inputs and outputs. With this layered flexibility, deep architectures are more efficient across high dimensional data. Moreover, they require fewer computational resources and human labor for coding purposes. This addresses shallow architectures’ first concern, which is being computationally expensive. The second concern is that shallow architectures struggle to represent broad groups of functions due to their limited layers. Deep architectures are able to handle more complex representations, as each layer’s output can serve as the subsequent layer’s input. With this, there are several intermediate results that are then combined and abstracted to be the higher-level solution. Because shallow architectures do not have this layered flexibility as a result of their innate structure, deep architectures provide a solution to this limitation.

**Problem Four**

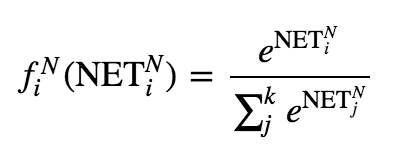
**A.** The formula for the ReLU Activation Function: *f(x) = max(0, x)*

Below is a plot that shows the shape of the ReLU function.



Source for above formula and graph: <http://frnsys.com/ai_notes/machine_learning/neural_nets.html>

B. The formula for the softmax activation function is:



Where the neurons are being summed in the denominator.

Source: <http://frnsys.com/ai_notes/machine_learning/neural_nets.html>

There are several benefits to using the softmax algorithm as opposed to standard normalization. The softmax algorithm minimizes cross entropy between actual and predicted values. Moreover, it reduces the impact of outliers without having to explicitly remove them from the raw dataset. It ensures that all normalized values fall between 0 and 1, limiting the range, and smoothening the normalized dataset. Further, the output will always equal zero. Essentially, softmax provides greater stability.

Source: <http://stackoverflow.com/questions/17187507/why-use-softmax-as-opposed-to-standard-normalization>, <https://en.wikipedia.org/wiki/Softmax_function>

**Problem Five**

**A.** The architecture of the neural network has a total of three layers. The input data consists of 6 features: pclass, sex, age, sibsp (number of siblings/spouses aboard), parch (number of parents or children aboard), and fare. This 6-element vector is processed, such that two irrelevant columns are disposed of, and sex is transformed into a binary form. As a result, the input shape is [‘None’, 6] with 6 nodes. Two layers use the reLU activation function, and have 32 nodes. The output layer uses softmax and has two output nodes.

\*\*NEED TO ADD MORE \*\*

**B.** Training Step 2 is the first step where the system shows a non-zero number for data accuracy.

As shown by the below output, the accuracy is .3937, a non-zero number.

Training Step: 2 | total loss: 0.62361

| Adam | epoch: 000 | loss: 0.62361 - acc: 0.3937 -- iter: 0032/1309

As shown by the below output, the final accuracy of the training data is .7575.

Training Step: 820 | total loss: 0.52037

| Adam | epoch: 010 | loss: 0.52037 - acc: 0.7575 -- iter: 1309/1309

**C.** 20 epochs: As shown by the below output, the final accuracy of the training data is .7740.

Training Step: 1640 | total loss: 0.48996

| Adam | epoch: 020 | loss: 0.48996 - acc: 0.7740 -- iter: 1309/1309

100 epochs: As shown by the below output, the final accuracy of the training data is 0.8101.

Training Step: 8200 | total loss: 0.42296

| Adam | epoch: 100 | loss: 0.42296 - acc: 0.8101 -- iter: 1309/1309

**\*\* ADD EXPLANATION HERE\*\***

**D.** 10 epochs:

DiCaprio Surviving Rate: 0.130324989557

Winslet Surviving Rate: 0.890094101429

20 epochs:

DiCaprio Surviving Rate: 0.116550065577

Winslet Surviving Rate: 0.856163084507

100 epochs:

DiCaprio Surviving Rate: 0.167878061533

Winslet Surviving Rate: 0.996512591839

**Problem Six**

**A.** Implemented in *spiral\_classifier.py,* can be called using the following command:

*python spiral\_classifier.py spiral\_data.csv 0*

**B.** I decided to use all 400 data points to train my neural net, and not test the resulting neural network. I chose to do this because effective neural networks require substantial training data in order to have strong predictive capabilities. With that, we were provided with a relatively small dataset for this homework. I prioritized substantially training my neural network over creating a rather small testing set and compromising the neural network’s accuracy. Moreover, because the testing set would have to be quite small to retain a large training set, I did not think the testing would be very reflective of the neural network’s accuracy. Even with cross validation, splitting this data set would either prioritize testing or training’s accuracy (whichever split had more data points), and would be compromised in some regard. That being said, I thought it would be smartest to not split the data set at all, create a well-trained neural network, and not provide skewed testing results based off of a small sample size.

**C.**

**D.**

Training Step: 400 | total loss: 0.32452

| Adam | epoch: 010 | loss: 0.32452 - acc: 0.8993 -- iter: 400/400

--