Implementing an Interface to Tensorflow

1 Introduction

This document describes many of the concepts, techniques and tools needed to fulfill the requirements of one or more project modules involving neural networks running in Tensorflow. It serves as an important complement to the lecture materials and code that cover many of these same topics. The relevant code files are: tflowtools.py, tutor1.py, tutor2.py, tutor3.py and mnist_basics.py.

You will build a system that essentially serves as an interface to Tensorflow. The system will accept many user-specified characteristics of a neural network, the data set, and the training/testing regime. It will then configure and run the network, and show various intermediate and final results on the screen. What follows are **important requirements** for the system that you develop.

Your system must be object-oriented and must provide the same type of modularity and building-block composition as the classes Gann (General Artificial Neural Network) and Gannmodule in file tutor3.py. You are free to build the system from scratch or to use tutor3.py as a starting point. Either way, your system must accept any of a wide range of user-specified scenarios, including a variety of network architectures, and **without requiring a recompilation of the system**. Systems that require rebuidling for each scenario will lose a substantial number of points during grading.

2 The Network Scenario

A network scenario consists of:

1. A dataset serving as the basis for a classification task.
2. A specification of the network's architecture (a.k.a. configuration): the layers of neurons, their sizes and activation functions, the cost function, etc.
3. A scheme for training and testing the network on the dataset.
4. A specification of the network behaviors to be visualized.

All four aspects are described below, followed by a simple example and a complete list of the scenario parameters that your system will need to accomodate.

2.1 The Dataset

Each dataset consists of cases, each of which has a set of features along with a class (a.k.a. label). As described below, datasets often reside in large data files but can also be generated by special functions, which enable the production of a wide range of pre-specified cases, such as all 20-element bit vectors labelled by whether or not they are symmetric about the center, or all 4 x 4 bit arrays classified by whether they contain a horizontal line or a vertical line.

2.2 The Network Architecture

In general, a dataset defines a classification task, which will give hints as to the proper neural network configuration. Unfortunately, there is no magic formula for precisely determining the proper artificial neural network (ANN) for a particular task , so trial-and-error comes into play. Some of the important degrees of freedom are:

1. The number of hidden layers.
2. The number of nodes in each hidden layer. Different layers will typically have different sizes.
3. The activation functions used in the hidden layers and output layer.
4. The learning rate.
5. The error function (a.k.a. loss function) for backpropagation.
6. The initial range of values for network weights.

These and other factors determine the structure and general behavior of the network. Choosing them often constitutes the hardest part of problem solving with deep learning.

2.3 The Training and Testing Scheme

This (very standard) scheme consists of several steps:

1. Separate the data into training cases, validation cases and test cases, where all cases are assumed to consist of *features* and *labels*, i.e., the correct classification.
2. Repeatedly pass the training features through the ANN to produce an output value, which yields an error term when compared to the correct classification.
3. Use error terms as the basis of backpropagation to modify weights in the network, thus *learning* to correctly classify the training cases.
4. Intermittently during training, perform a validation test by turning backpropagation learning off and running the complete set of validation cases through the network one time while recording the average error over those cases.
5. When the total training error has been sufficiently reduced by learning, turn backpropagation off.
6. Run each test case through the ANN one time. Record the total error on the test cases and use that as an indicator of the trained ANNs ability to **generalize** to handle new cases (i.e., those that it has not explicitly trained on).

2.4 Visualization

As described below in more detail, your system will need to provide the following behavioral data in graphic form:

1. A plot of progression of training-set and validation-set error (as a function of the training steps, where each step involves the processing of one minibatch).
2. A display of the weights and biases for any user-chosen areas of the network, for example, the weights between layers 1 and 2 and the biases for layer 2. Typically, these are only shown at the end of the run, but displaying them intermittently is also useful (though not mandatory for this assignment).
3. The sets of corresponding activation levels for user-chosen layers that result from a post-training **mapping** run (as described below).
4. Dendrograms (also described below), which graphically display relationships between input patterns and the hidden-layer activations that they invoke.

Users of your system must be able to turn these options on and off, as well as specify more details about them, such as **which hidden layer** to use as the basis of a dendrogram, or **which weights and biases** to display at the end of a run, etc.

2.5 A Simple Example

As an example, the network in Figure 1 was designed to classify images from the MNIST data set (in which features are 28 x 28 pixel-intensity matrices, while labels are the digits 0-9). As shown, the ANN should have an input layer consisting of 28 x 28 = 784 nodes, followed (downstream) by one or more hidden layers, and ending with an output layer housing 10 nodes, one for each possible digit. This large dataset is originally partitioned into training, validation and test subsets.

When features are passed through the network, the output-layer node with the highest activation constitutes the network's *answer*. Each time that answer disagrees with the correct classification, an *error* occurs. During the training phase, errors combine with gradients (i.e. derivatives of the error function with respect to the individual weights of the network) as part of the backpropagation algorithm for modifying the weights. During the validation and testing phases, no learning (i.e. weight modification) occurs, but the errors give an indication of the network's evolving ability to generalize from the training cases.

2.6 The Complete List of Scenario Parameters

What follows is the minimum set of scenario-defining input parameters that your system must accept and then use to build and run the ANN. Although these parameters can certainly be entered via an elaborate graphical user interface (GUI), that is not a requirement for the assignment. It is fine to enter them as arguments to a single, high-level function or to enter them via a scenario-defining script. If entered via a GUI, any combination of menus, submenus, windows, sub-windows, etc. is fine, but if entered by more primitive means (i.e, function arguments or a script), then all data should be entered as the arguments to a single function or as a single script. In general, the specification of a scenario should be very straightforward and thus facilitate a wide variety of user-specified runs during a short 10-15 minute demonstration period.

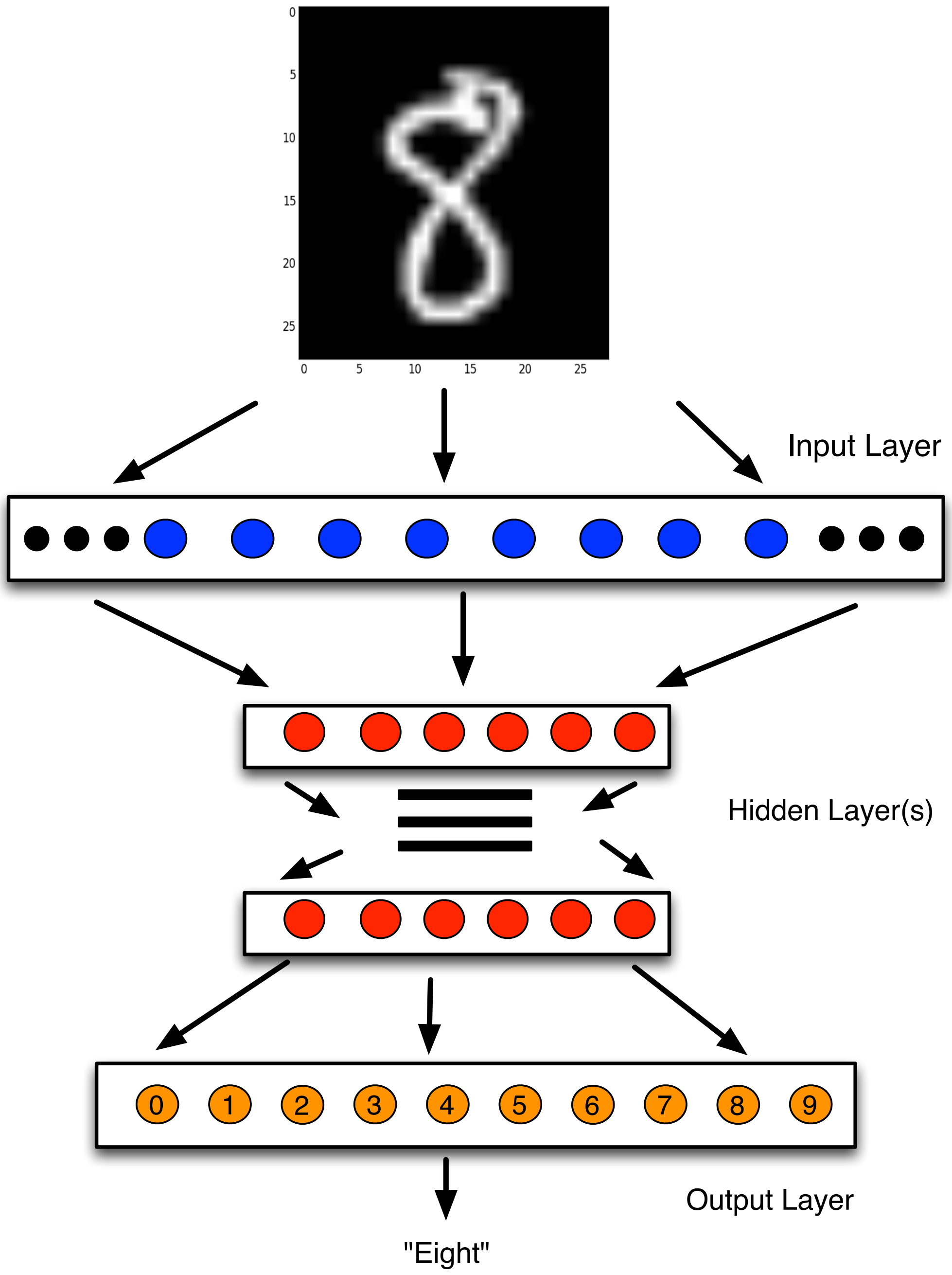


Figure 1: General framework for a neural-network classifier for the MNIST dataset.

1. Network Dimensions - the number of layers in the network along with the size of each layer.
2. Hidden Activation Function - that function to be used for all hidden layers (i.e., all layers except the input and output).
3. Output Activation Function - this is often different from the hidden-layer activation function; for example, it is common to use *softmax* for classification problems, but only in the final layer.
4. Cost Function - (a.k.a. loss function) defines the quantity to be minimized, such as mean-squared error or cross-entropy.
5. Learning Rate - the same rate can be applied to each set of weights and biases throughout the network and throughout the training phase. More complex schemes, for example those that reduce the learning rate throughout training, are possible but not required.
6. Initial Weight Range - two real numbers, an upper and lower bound, to be used when randomly initializing all weights (including those from bias nodes) in the network. Optionally, this parameter may take a value such as the word *scaled*, indicating that the weight range will be calculated dynamically, based on the total number of neurons in the upstream layer of each weight matrix.
7. Data Source - specfied as either:
   - a data file along with any functions needed to read that particular file.
   - a function name (such as one of those in tflowtools.py) along with the parameters to that function, such as the number and length of data vectors to be generated, the density range (i.e. upper and lower bound on the fraction of 1's in the feature vectors), etc.
8. Case Fraction - Some data sources (such as MNIST) are very large, so it makes sense to only use a fraction of the total set for the combination of training, validation and testing. This should default to 1.0, but much lower values can come in handy for huge data files.
9. Validation Fraction - the fraction of data cases to be used for validation testing.
10. Test Fraction - the fraction of the data cases to be used for standard testing.
11. Minibatch Size - the number of training cases in a minibatch
12. Map Batch Size - the number of training cases to be used for a *map test* (described below). A value of zero indicates that no map test will be performed.
13. Steps - the total number of minibatches to be run through the system during training.
14. Map Layers - the layers to be visualized during the map test.
15. Map Dendrograms - list of the layers whose activation patterns (during the map test) will be used to produce dendrograms, one per specified layer. See below for more details on dendrograms.
16. Display Weights - list of the weight arrays to be visualized at the end of the run.
17. Display Biases - list of the bias vectors to be visualized at the end of the run.

2.7 Partitioning the Data

For the following description, assume that *data set* refers to that fraction (i.e., the *case fraction*) of the complete data collection to be used for the combination of training, validation and testing. Then, for any data set, the user can specify the *testing fraction* (TeF) and *validation fraction* (VaF), both of which are values between 0 and 1. These direct the system to divide the data set of total size S into three subsets:

1. The training set, of size S * (1 - (TeF + VaF)).
2. The validation set, of size S * VaF
3. The test set, of size S * TeF

Randomly partition the data set into these three subsets, as specified by VaF, TeF and 1 - (VaF + TeF).

Runs of a TensorFlow session that include gradient-descent learning will only be performed on the training set, while the validation and test sets will be processed without learning. Each mini-batch of training cases that run through the network will invoke a forward phase and a backward (learning) phase. At the end of the forward phase, the average (per case) error for the mini-batch should be recorded (and later plotted – see below).

Standard testing occurs at the very end of training. Simply run each test case through the ANN once and record the percentage of incorrect classifications.

Validation testing works the same way: run the entire validation set through the network one time and record the error percentage. However, unlike standard testing, validation testing is **interleaved** with training as specified by another user-specified parameter, **vint** (validation interval): after every vint mini-batches of training cases are run, the system turns off learning and processes the entire validation set one time, recording the error percentage.

2.8 Scenario Results

The mandatory information generated by each run of your system is:

1. A plot of the progression of the training-set error from start to end of training. Each data point is the average (per case) error for a single mini-batch.
2. A plot of the progression of the validation-set error from start to end of training.
3. A listing of the error percentage for the test set, as evaluated after training has finished.

The plots of training-set and validation-set error must be on the same graph. This allows you to easily detect whether the network has been overtrained, and where overtraining begins.

In addition, the scenario specification may call for additional visualizations, with the base requirement consisting of these possibilities:

1. Mapping - This involves taking a small sample of data cases (e.g. 10-20 examples) and running them through the network, with learning turned off. The activation levels of a user-chosen set of layers are then displayed for each case. For any given layer, a comparison of the different activation vectors (across all mapped cases) can then serve as the basis for a *dendrogram*, a convenient graphic indicator of the network's general ability to partition data into relevant groups.
2. Weight and Bias Viewing - These are simple graphic views of the weights and/or biases associated with the connections between any user-chosen pairs of layers.

These additional (but still mandatory) features are described in more detail later in this document. The combination of visualized weights, biases and mappings (along with dendrograms) supports thorough investigations into the behavior of a neural network, thus allowing the user to break open the *black box* and reduce some of the mystery associated with this powerful, but not particularly transparent, AI tool.

3 Data Sets

At the demonstration session, you will be required to use some or all of the data sets described below. Some are generated dynamically by calls to functions in tflowtools.py, while others are provided as simple text files. Those generated by functions are large lists consisting of pairs: feature vector, target vector or label. Data files typically house one case per row, with items separated by commas, and the feature vector consisting of all but the last element of a row, while that last element constiitutes the class label. It is safe to assume this format for most data files but to then write a special reader function for any file that violates that assumption.

Accompanying each dataset below is a quantitative definition of *reasonably good.* It indicates the percentage of correct answers that your network should achieve on the **training set** when training has completed. Some datasets are harder than others, so the numbers vary. For each case, it is mandatory that your training set is a randomly-selected group of cases that constitute 80% or more of the total number of cases in the data set. For example, you might use 10% for validation, 10% for testing and 80% for training.

It is important to note that during training, you will probably want to use one of the standard error/loss functions such as mean-square error or cross-entropy. These give useful gradients for modifying the network's weights and gradually reducing error. However, when training has finished and you want to compute the number of training cases that the network correctly classifies, you should switch to a simpler error function: one that merely tests whether or not the output node with the highest activation level corresponds to the target value. Tensorflow's primitive function **in_top_k(predictions,targets,k)** (with k = 1) is very helpful in this regard. You can use it for checking the network's accuracy on the training, validation and test sets, but you do not want to use it during actual training, since it may not provide useful gradients for learning.

3.1 Parity

The function **gen_all_parity_cases** in tflowtools.py generates all bit vectors of a specified length and then packs them into cases consisting of a vector along with a target of either 1 or 0, indicating the parity of the vector: 1 (0) means that the vector contains an odd (even) number of 1's.

Alternatively, if the *double* flag is True (the default), targets will be either [1,0] indicating even or [0,1] denoting odd.

REASONABLY GOOD: 95% (using the set of all 10-bit vectors as the dataset and using a random 80% of them for training)

3.2 Autoencoder

Cases in these data sets consist of binary feature vectors whose targets are identical to the feature vectors. In tflowtools.py, two functions generate autoencoder data sets:

- **gen_all_one_hot_cases** produces all one-hot feature vectors of a specified length, e.g. all 8 one-hot vectors of length 8.
- **gen_dense_autoencoder_cases** produces vectors of a specified length with a specified density range. For example, a density range of (0.4, 0.7) entails that all vectors will consist of anywhere from 40% to 70% 1's.

Both of these functions produce cases consisting of the feature vector and its copy (the target).

REASONABLY GOOD: You will not be asked to run a performance test on an autoencoder at the demo session, but you may choose an autoencoder as the network that you explain in detail.

3.3 Bit Counter

This data consists of bit vectors whose class is simply the count of the number of 1's in the vector. To produce these dat sets, simply call the function **gen_vector_count_cases** in tflowtools.py.

REASONABLY GOOD: 97.5% (on a set of 500 randomly-generated cases of length 15)

3.4 Segment Counter

These data cases consist of a bit vector as the features and a number (represented as a one-hot vector) denoting the number of 1's segments in the vector. For example the feature vector 111110001010000111 is classified it as a "4", since there are 4 groups of 1's(separated by 0's) in the vector. To generate these segment-vector cases, use the function **gen_segmented_vector_cases** in tflowtools.py. This function takes the following arguments:

- **size**, length (in bits) of each input/feature vector,
- **count**, the number of cases to produce,
- **minsegs**, the minimum number of segments in a vector,
- **maxsegs**, the maximum number of segments in a vector,
- **poptargs**, a flag indicating whether or not one-hot vectors (also known as population-coded vectors) are being used as targets. The default is True.

For example, the call **gen_segmented_vector_cases(25,10,0,5)** will produce 10 cases, each employing a vector consisting of 25 bits and housing anywhere from zero to five segments. The returned cases are in the normal format: a pair of vectors, the features and the one-hot target. Note that a vector with zero segments will have a one-hot coding of [1,0,....] while a vector with one segment has [0,1,0,....]. Do not worry about the fact that this function may occasionally produce duplicate cases.

REASONABLY GOOD: 95% (on a set of 1000 randomly-generated cases of length 25 with 0 - 8 segments in each case)

3.5 MNIST

This is a classic machine-learning benchmark consisting of all 10 digit classes (0 - 9) represented by 28 x 28 pixel arrays. The original data files contain tens of thousands of cases in a rather complex format, but by importing the file mnist_basics.py, you gain access to several simple functions for generating standard data sets (consisting of one-dimensional feature vectors and integer class labels), such as **load_all_flat_cases**. For more important details on installing and using the MNIST files and accessors, read tflow-mnist.pdf.

For this dataset, it is fine to use only a random subset (S) containing about 10%, of the complete dataset. Then divide S into training, validation and test sets.

For MNIST, you will probably want to scale all pixel values to numbers in the range [0,1] before feeding them into the input layer of your network.

REASONABLY GOOD: 95% (on the training fraction of subset S)

3.6 Popular Datasets from UC Irvine

All of the following datasets are available from the UC Irvine Machine-Learning Repository, which provides supporting information about each set. The raw datasets (without explanations) can also be found on the ”Projects” web page for this course.

Some of these datasets have features whose ranges vary dramatically. For example, feature 3 may have values in the range (0,1) whereas those for feature 5 might be in range (0, 100). If these values are fed directly into the input layer of a neural network, feature 5 could easily dominate other features, making classification extremely difficult.

One solution to this problem is to scale all features by their average and standard deviation. Hence, feature j for case c ($f_{c,j}$) is scaled to $I_{c,j}$ (the value of input neuron j for case c) as follows:

$$I_{c,j} \leftarrow \frac{f_{c,j} - \mu_j}{\sigma_j} \tag{1}$$

where μ_j and σ_j are, respectively, the mean and standard deviation of feature j as calculated across all the entire case set.

Alternatively, to insure an input value in the closed range [0,1], you can scale by $f_{j,min}$ and $f_{j,max}$, the minimum and maximum values of feature j across the case set. Then:

$$I_{c,j} \leftarrow \frac{f_{c,j} - f_{j,min}}{f_{j,max} - f_{j,min}} \tag{2}$$

These or other similar functions are worth considering when working with the Irvine data sets.

3.6.1 Wine Quality

This dataset consists of 1599 cases, each of which has 11 real (or integer) features and one of SIX integer classes. If downloading this set from UC Irvine, note that it contains two files, one for red and one for white wine. The file provided on the course webpage is the one for red wine (which involves 6 classes). The white-wine file involves 7 classes. You are only responsible for the handling the red-wine file. Note: In both files, the elements of each case are separted by semicolons, not commas.

REASONABLY GOOD: 95%

3.6.2 Glass

This dataset contains 214 cases, each of which has 9 real-valued features and one of SIX integer classes. Note: For some odd reason, the class labels are 1-7 but with **no examples of class 4**. So a one-hot bit vector of length 6 can be used as a target.

REASONABLY GOOD: 95%

3.6.3 Yeast

There are 1482 cases in this set. Each case has 8 real-valued features and one of 10 integer labels.

REASONABLY GOOD: 90%

3.6.4 Hacker's Choice

Choose any data set (consisting of at least 100 cases) from the UC Irvine Machine-Learning Repository. **You must have at least one such data set available to your system.** As with all of the datasets listed above, it should be possible to run scenarios involving a hacker's-choice dataset at the demo session.

It must be possible to run any of these data sets through your ANN system without the need to recompile your code between datasets.

4 Visualizing Internal States of a Neural Network

Our focus now turns to the internal *representations* used by neural networks to encode the original input vectors and/or to serve as intermediate computational states between the inputs and outputs. These representations are often characterized by the activation patterns of hidden-layer neurons. The two relationships that most interest us in this section are:

1. the correspondence between input patterns and the internal representations that they invoke, and

2. the vector distances between representational patterns, and how those a) reflect semantic similarities between the input vectors, and b) change with learning.

Each of these is discussed below, in detail.

4.1 Visualizing Patterns for Qualitative Comparison

As an example of the correspondence between patterns, Figure 2 shows the behavior of a well-trained autoencoder that accepts an 8-bit, one-hot input vector, has 3 hidden nodes, and produces an 8-bit vector that resembles the input. Notice how each input vector produces a unique pattern in the hidden layer, and how the output patterns are not perfect replicas of the input patterns (but the most active neuron for each case corresponds to the active input neuron of that case).

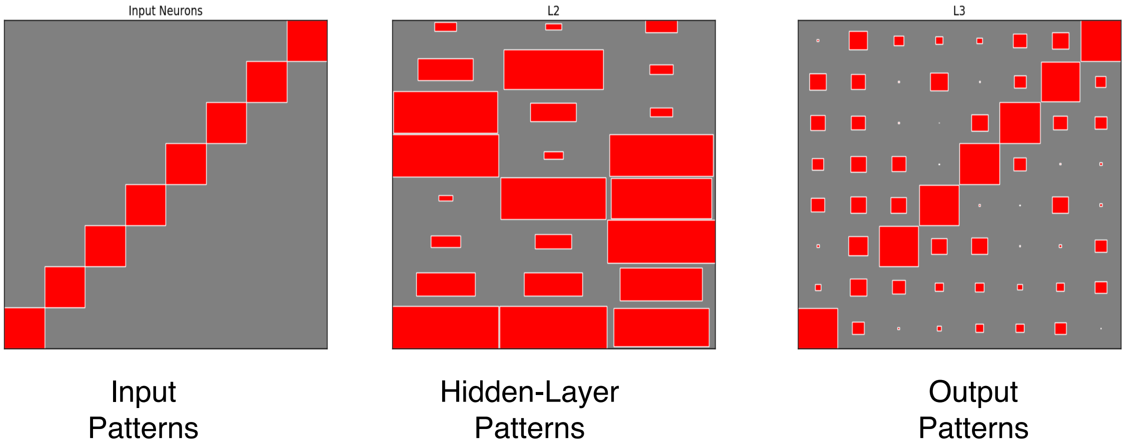


Figure 2: The activation levels of the input, hidden and output layers of a 8-3-8 feed-forward network trained by back-propagation to reproduce the input at the output while compressing to a 3-bit code at the hidden layer. Each of the 8 test cases (which were also the training cases) is represented by corresponding rows in the three matrices. Larger red squares indicate higher activation levels of the neurons.

Conversely, Figure 3 shows activations from a weakly-trained 8-3-8 network. Notice the lack of inter-case differentiation between patterns in the hidden and output layers.

4.1.1 Visualizing Mappings

In this document, the term *mapping* will refer to post-training activity that is designed to clarify the link (or mapping) between input patterns and hidden and/or output patterns. Figures 2 and 3 illustrate typical mappings: they show the results of sending a small set of cases through the network, with learning turned off.

In order to produce visualizations of this type, your code will need to perform the following sequence of activities:

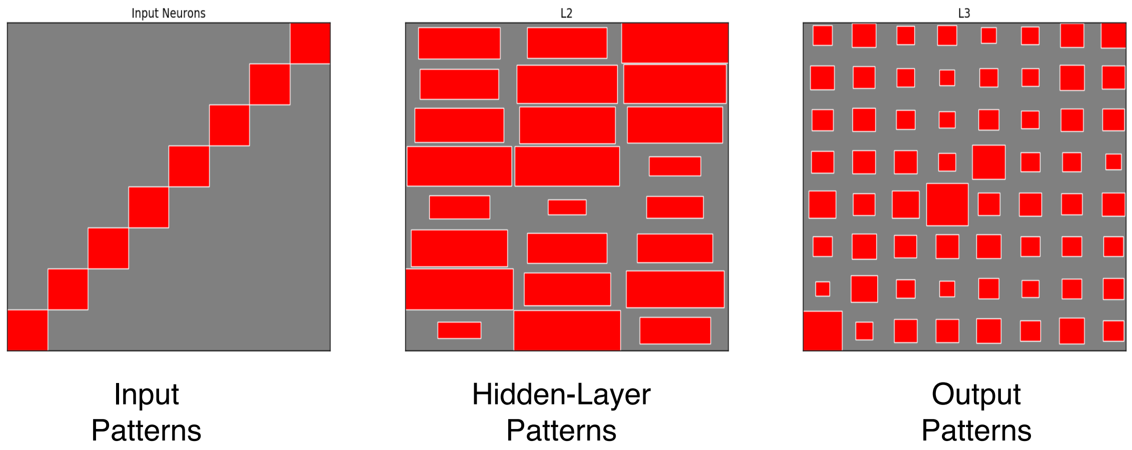


Figure 3: Activation levels at an early phase in training of the same 8-3-8 network shown in Figure 2.

1. Train the network to a desired level of expertise (or lack thereof). You need not save any data during this training phase but will surely want to monitor the error progression.
2. Declare *grab variables* as any Tensorflow variables that you wish to monitor during the post-training phase.
3. Determine the cases that the network will process in the post-training phase. These may or may not be the same as the training cases.
4. Run the network in mapping mode, which involves only one forward-pass per case. This can either be a test mode in which the errors of each case are calculated and summed or a simple prediction mode in which the output values of the network are the final objective. Either way, you must insure that a) no learning occurs, and b) the values of monitored (grabbed) variables are gathered and stored by your code (after each call to **session.run**). For these post-training situations, one call to session.run should be sufficient, with a map batch containing all the relevant cases.
5. When the mapping run finishes, the gathered rows of data (one row per neural-net layer per case) are sent (normally as a 2-d array) to a plotting routine, such as **hinton_plot** or **display_matrix** in tflowtools.py.

Note that in the file tutor3.py, the default **run** method for a GANN includes a call to **close_current_session**. Thus, any mapping operations will require a call to **reopen_current_session**, which will open a new session and load the current weights and biases into the neural network as a starting point for the new session.

You will probably want to define a new method called **do_mapping**, which will behave similarly to method **do_testing** in tutor3.py, although it need not have **self.error** as its main operator, since **self.predictor** would suffice. It will also need code for gathering and storing the grabbed values. Be aware that the resulting dimensions of the grabbed variables could vary depending upon whether you run all the cases through as a single mini-batch or whether you perform N calls to session.run, where N is the number of cases.

4.2 Quantifying Pattern Comparisons with Dendrograms

An important feature of a well-trained neural network is the ability to form similar internal patterns for input cases that should be producing similar results. For some data sets, this means that syntactically similar input vectors should produce similar hidden and output vectors, but in other cases, the similarities and differences between the input vectors will not be obvious from simple syntactic comparisons; but the human user will know, for example, that the two input patterns 101010 and 001110 are similar if the network's task is to count the number of 1's in the vector. In short, we would expect these two patterns to produce identical (or nearly so) patterns at some level of a neural network trained to perform the one-counting task.

Dendrograms are a standard tool for displaying the similarities and differences of neural network activation patterns in a concise graphical form that resembles a tree. You have probably seen their counterpart, the cladogram, in biology class. Cladograms use the same tree format to display relationships among organisms, with the key characteristic that closely related organisms have short paths between them in the tree, while very different organisms have longer paths.

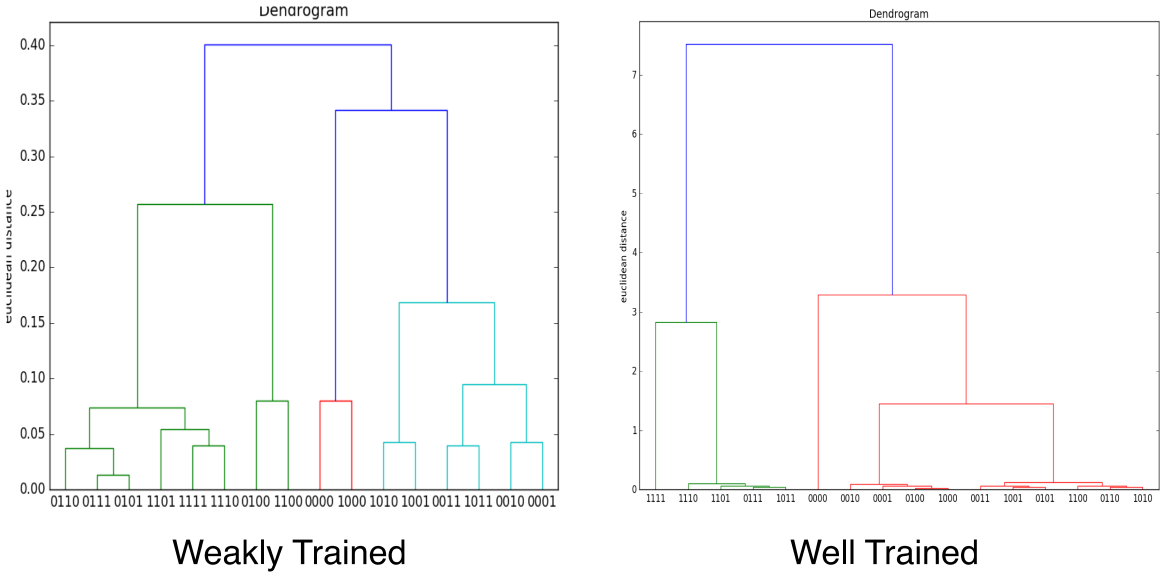


Figure 4: Dendrograms based on the Euclidean distance between hidden-layer activation patterns for all 16 4-bit input vectors in a 4-5-5 network for counting 1's. Distances appear on the y axis. Note the much smaller separation distances between hidden-layer patterns in the weakly-trained network. Labels along the bottom denote the input patterns, while distances up the tree from two inputs to their intersecting branch point indicate the distance between their corresponding hidden-layer vectors.

Dendrograms group cases based on their *signatures*, which are typically embodied in the activation patterns of particular layers of the network, often a (the) hidden layer. Cases are judged as similar when the distance between their signatures is small, where distance metrics may vary with the problem. Typically, when the signatures are activation vectors, the metric is the Manhattan or Euclidean distance.

Figure 4 portrays a pair of dendrograms for a one-counting neural network: for any binary input vector, the output is the number of ones in the input pattern. This visualizes the distances between the hidden-layer patterns for all 16 4-bit input patterns. If inputs X and Y are to be treated similarly by the network, then they will normally appear close to one another on the bottom of the dendrogram, and hence the distance to their nearest common branch point will be short, indicating a small distance between the hidden-layer activations for X and Y. In this example, the dendrogram of the well-trained network clearly shows this property, as input vectors with the same or similar 1-counts appear nearby one another at the bottom of the dendrogram; the weakly-trained network's dendrogram exhibits a much poorer grouping of similar inputs, which is reflected in the inferior classification abilities of that network.

Notice that in the poorly-trained network, the distances between all patterns are quite short, with none exceeding 0.4. Conversely, the well-trained network achieves superior pattern separation, giving distances over 7.0 for the hidden layers corresponding to inputs 0000 and 1111. These distances are possible in a 5-neuron hidden layer, since the activation function for this hidden layer is the ReLU (which has no upper bound on its output), in contrast to the sigmoid (bounded below by 0 and above by 1).

It is important to remember that the dendrogram of a hidden layer is not a definitive indicator of performance, since weights from the hidden to output layer can still further separate or condense activation patterns. But in most cases, the dendrogram gives a concise portrayal of a network's ability to recognize the *differences that make a difference* among its input patterns. They indicate whether or not the network has generalized and discriminated properly.

In this assignment, your dendrograms will be based on the collections of activation vectors produced by a mapping. Hence, the activation vectors for any layer can serve as the *signatures* for the inputs in the map batch.

4.2.1 Generating Dendrograms

The file tflowtools.py provides the function **dendrogram**, which handles all the graphics of dendrogram drawing. All you have to do is send it two lists: the features and labels, where, in this case, each feature is a vector of values (i.e., hidden-layer activations) and the label is a STRING corresponding to each feature. Within this code, the features constitute the signatures of each label. It is these signatures that determine the distance between labels on the dendrogram.

For example, consider the following results of 4 2-bit cases run through a network (with 3 hidden-layer nodes):

Input Pattern	Hidden-Layer Pattern
0,0	0.5, 0.4, 1.2
0,1	0.3, 0.1, 0.8
1,0	0.9, 0.0, 0.3
1,1	0.6, 0.6, 0.4

The following call will build and display the dendrogram of Figure 5:

dendrogram([[0.5, 0.4, 1.2], [0.3, 0.1, 0.8] , [0.9, 0.0, 0.3], [0.6, 0.6, 0.4]] , ['00','01','10','11'])

You can use the function **bits_to_str** in tflowtools.py to convert an input vector, such as [0,1] into its corresponding string: '01'.

Note that if you send multiple copies of the same feature-label pair to the dendrogram plotter, it will display multiple copies on the diagram. In that case, you will want to filter for uniqueness ahead of time. In other situations, such as those involving large vectors of real numbers (such as those from the wine data set), a wiser choice for a case label might simply be its class (e.g. one of the 7 wine types). In general, the labels of

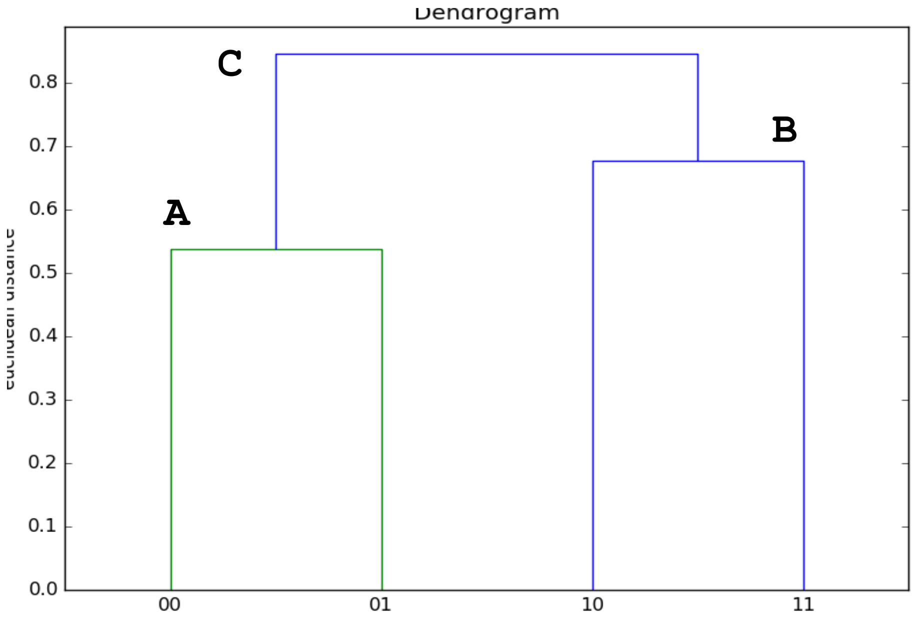


Figure 5: Simple dendrogram for the 4 vectors described in the text. The bar marked "A" at a height just above 0.5 indicates that the distance between the hidden-layer vectors for inputs 00 and 01 is approximately 0.55, while the bar at mark "B" denotes a distance near 0.7 for the hidden-layer vectors for inputs 10 and 11. The bar marked "C" signals that 0.85 is approximately the average distance between any hidden-layer vector in the cluster (00,01) and one in the cluster (10,11).

the dendrogram should quickly indicate whether or not the signature layer treats *similar* inputs similarly, where the definition of *similar* depends upon the classification task.

The details of building dendrograms (when given features and labels) are beyond the scope of this document. Briefly, it involves finding clusters of nearby vectors, then larger clusters of nearby clusters, then still larger clusters until the whole vector set is one big cluster. Then, the history of the clustering process – i.e. what clustered with what, and when – provides the basis for the dendrogram tree, with vectors that clustered early appearing nearby at the bottom of the tree.

The details that you will have to worry about are the Python and Tensorflow operations required to gather hidden-layer values from an active neural network. The data gathered in the first section of this (second) assignment should be readily reusable for dendrogram viewing, and, in fact, the basic procedure is nearly identical to that above, but instead of sending an array of activation values to **hinton_plot**, you will want to send a list of hidden-layer activation vectors (as *features*) and a list of the corresponding strings as (*labels*) to the **dendrogram** function in tflowtools.py.

5 Detailed Analysis of a Functioning Network

This section employs a combination of weight vectors and activation patterns to create a detailed explanation of **how** a well-trained neural network performs a particular task. For each hidden layer and the output layer of a fully-trained network, the incoming weights give strong indications of the types of upstream signals that stimulate a particular neuron. Activation-pattern data gathered via the **do_mapping** method provides additional support. Together, this information enables a comprehensive account of the inner workings of the network and its manifest input-output behavior.

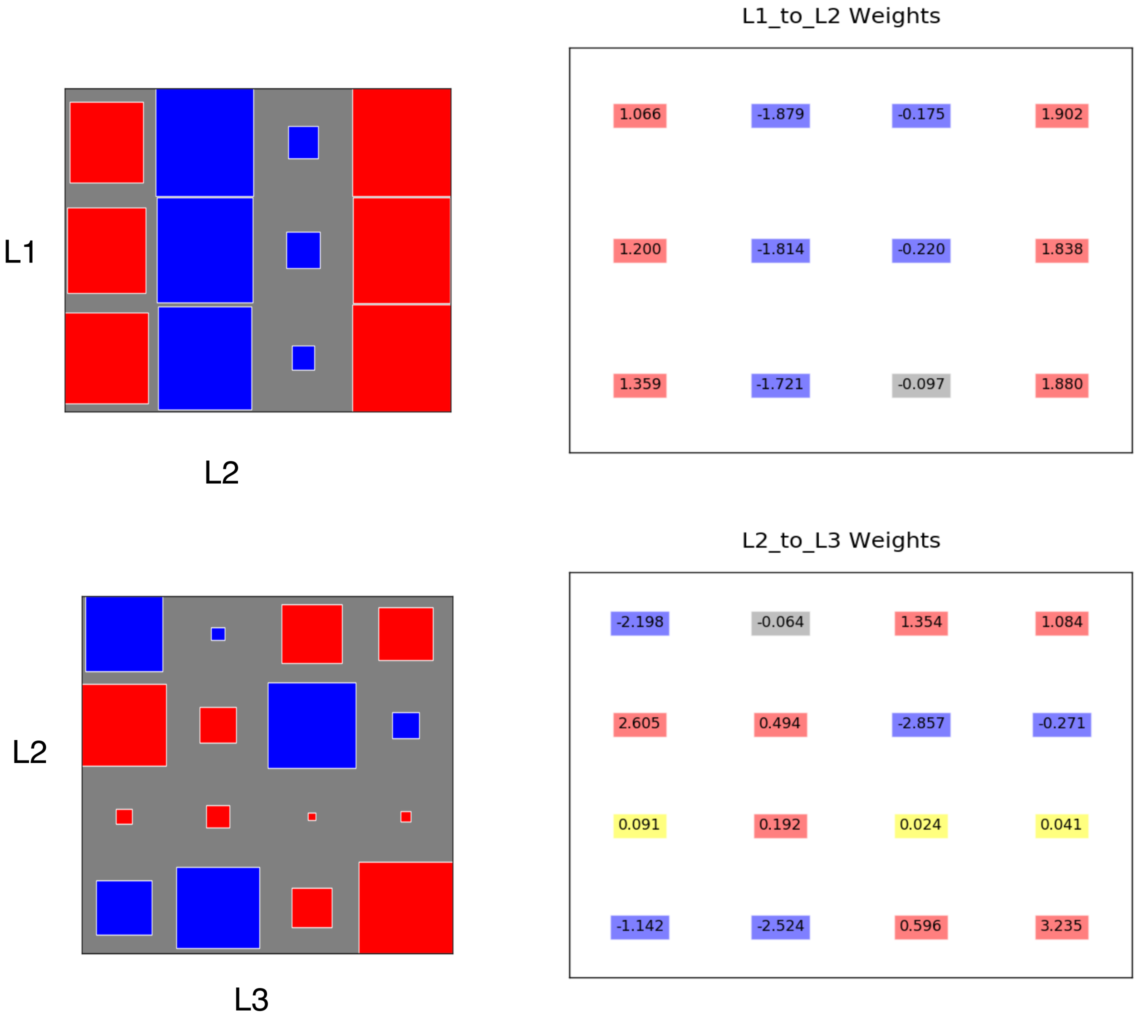
As an example, consider a neural network designed to count the number of 1's in a binary input vector of length 3. This requires 3 input neurons and 4 output neurons (to cover counts of 0, 1, 2 and 3). In this example, we use a single hidden layer with 4 nodes. After training on all 8 3-bit cases (using one-hot target vectors) for a few hundred epochs, the network learns to perfectly classify them into one of the 4 categories; the weights of that network appear in Figure 6.

The diagrams of Figure 6 are generated by the functions **hinton_plot** (qualitative) and **display_matrix** (quantitative) in tflowtools.py. Although not shown, these two functions can also be used to display the bias values associated with each layer. However, note that bias vectors are only one dimensional and thus should be wrapped in another level of brackets and converted to a 2-dimensional numpy array prior to being sent to either of these two display functions. If v is such a bias vector, then the following command will do the trick:

$$a = \text{numpy.array}([\text{v}])$$

Weights and biases, combined with the activation functions, tell most of the story of a network. Your job is to analyze that information and present the salient aspects in explaining network functionality. Many of the smaller weights and biases will have little significance and be omitted completely from that explanation. This filtering of irrelevant details occurs in Figure 7, which only retains the larger weights from Figure 6 but does include 7 of the 8 biases.

Given the (slightly) simplified rendering of the network in Figure 7, a logical analysis of behavior becomes feasible. In this account, we already know that the network was trained so that the nth output neuron



L1_to_L2 Weights

1.066	-1.879	-0.175	1.902
1.200	-1.814	-0.220	1.838
1.359	-1.721	-0.097	1.880

L2_to_L3 Weights

-2.198	-0.064	1.354	1.084
2.605	0.494	-2.857	-0.271
0.091	0.192	0.024	0.041
-1.142	-2.524	0.596	3.235

Figure 6: Two different views, one more qualitative (left) and the other more quantitative (right), of the weights in a fully-trained bit-counting ANN with layer sizes (3,4,4) (input, hidden, output). The hidden layer employs an ReLU activation function, while the output layer applies softmax to each neuron's sum of weighted inputs. For both the qualitative and quantitative display, each row denotes the weights on the output connections of a single upstream neuron, while each column houses the weights on the incoming connections of a single downstream neuron. Both displays use the same color coding: red for positive and blue for negative, but the quantitative diagrams use gray to signal small negative values and yellow for small positive values. In the qualitiative diagrams, box size reflects a weight's absolute value.

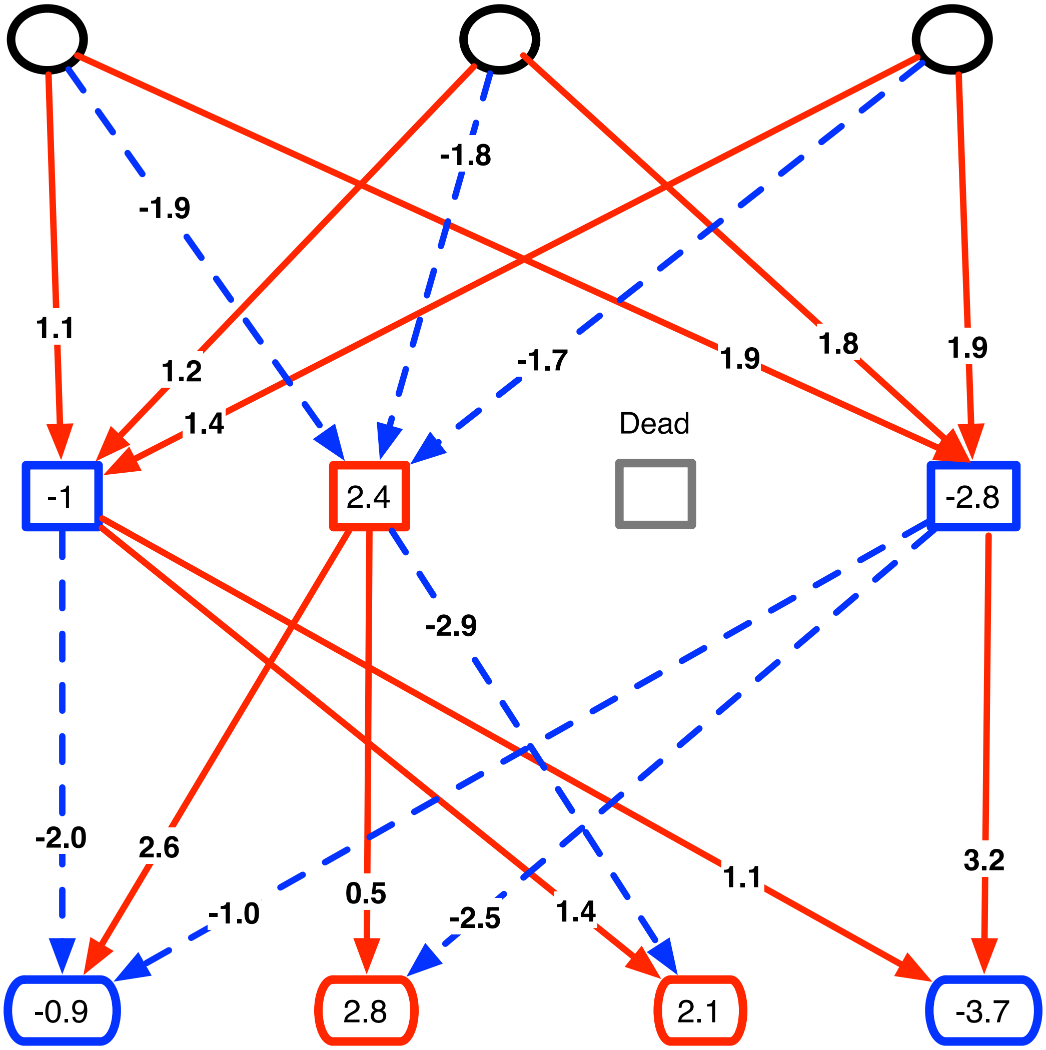


Figure 7: Important weights and biases for the same 3-4-4 bit-counter network whose weights appear in Figure 6. Red boxes or ovals denote positive biases – these nodes require less input to fire – while blue boxes/ovals signify negative biases – these nodes require more input to fire. Biases appear inside the box/oval. Solid red lines represent positive weights, while dashed blue lines denote negative weights. The third neuron of the hidden layer had only weak weights and never fired.

should produce a 1 when the bit sum is n (for n = 0-3), while all other output neurons should yield a 0. So it is no surprise that the fully-trained network uses output neurons 0,1,2, and 3 to detect the corresponding sums. However, the logic behind those decisions cannot be predicted ahead of time; there are many possible logical circuits that learning may produce. Only by investigating the weights and biases can we discern that underlying logic.

For example, in Figure 7, we see that the first hidden node (H1) has a negative bias and therefore requires some input in order to fire the ReLU (i.e., achieve a non-zero output value). Since all three incoming weights exceed 1, any one of those 3 lines would suffice to fire H1. Hence a sum of 1, 2 or 3 would do the job. Node H4 is trickier to analyze, though only slightly. Clearly, one active input would not be enough to overcome the -2.8 bias, but any 2 inputs could. H2, on the other hand, would fire by default due to its high bias. However, all incoming weights are negative, so 1's in the input vector will inhibit H2; but it takes at least 2 1's to fully squelch it. So H2 fires on sums of 0 and 1.

The logic often becomes more convoluted near the output layer, because now we need to explain why the nth output node should have the highest activation (over all output nodes) on a sum of n. In these cases, we need to consider the sums that each hidden node detects along with the weights coming out of the hidden nodes. For example, the first output node, O1, has a weight vector and bias indicating that it must have an active H2 (and inactive H1) in order to fire. Active H2 indicates a sum of 0 or 1, but inactive H1 entails that the sum cannot be 1, 2 or 3. Thus, the sum most be 0. The only other output node stimulated by H2 (and a sum of 0) is O2, but a sum of zero provides the following sums of weighted inputs to O1 and O2:

O1 input: 2.4 * 2.6 - 0.9 = 5.34

O2 input: 2.4 * 0.4 + 2.8 = 3.76

where 2.4 is the output of H2 on a sum of zero, and H2 is the only hidden node that fires on a sum of zero. Thus, O1 fires harder than O2 on a zero, while neither O3 nor O4 is even stimulated by a sum of 0 (via H2).

Conversely, on a sum of 1, H2 has less output (between 0.5 and 0.7) while H1 provides inhibition to O1. Together, these insure that O2 beats O1 on a sum of 1; and that sum only mildly affects O3 and O4. So O2 is the sum-of-1 detector.

As indicated earlier, the solutions found by neural networks are hardly ever unique. In fact, many problems admit an infinite number of possible weight arrays and bias vectors that yield perfect classification. Figure 9 shows another set of weights from a different learned solution for the 3-bit counter problem, using a 3-4-4 architecture, as before. In this case (as shown in Figure 12), all 4 hidden nodes detect 2 or 3 sums, and none of the weights are tiny. All of this complicates our analysis of the output layer. As shown in Figure 10, the network contains more salient connections, though the hidden nodes do exhibit a pattern similar to that of Figure 7: all salient incoming weights to a neuron have the same sign.

To supplement the analysis based on weights and biases, a set of activation patterns (generated by **do_mapping**) gives further insights into the behavior of the network. For example, notice that the second, third and fifth rows of the hidden matrix in Figure 11 are nearly identical, as are the fourth, sixth and seventh rows. The former three correspond to a bit count of 1, while the latter 3 involve a bit count of 2. A dendrogram based on hidden-level activation patterns would also reveal that the input vectors [1,0,0], [0,1,0] and [0,0,1] are being handled similarly, as are [1,1,0], [1,0,1], and [0,1,1].

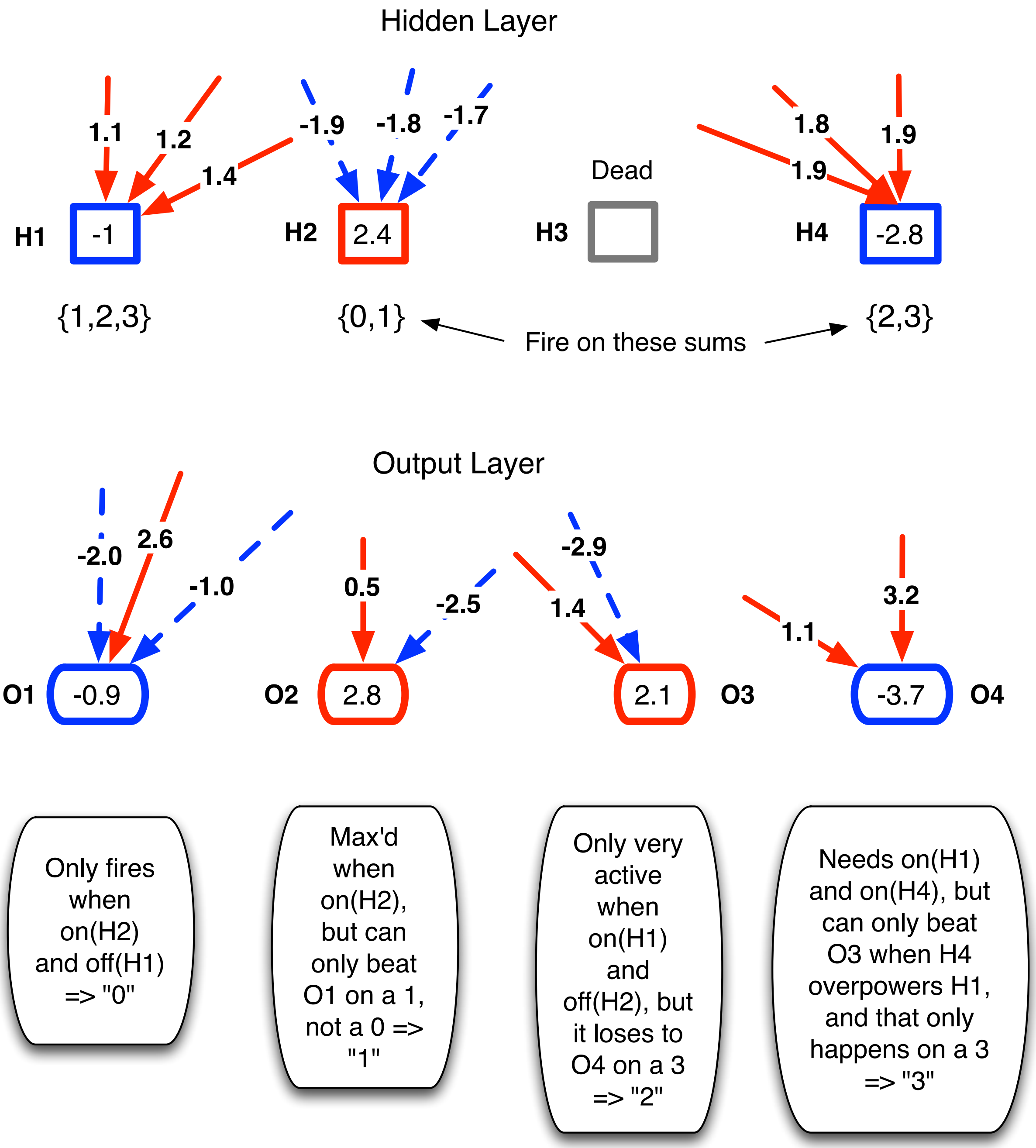
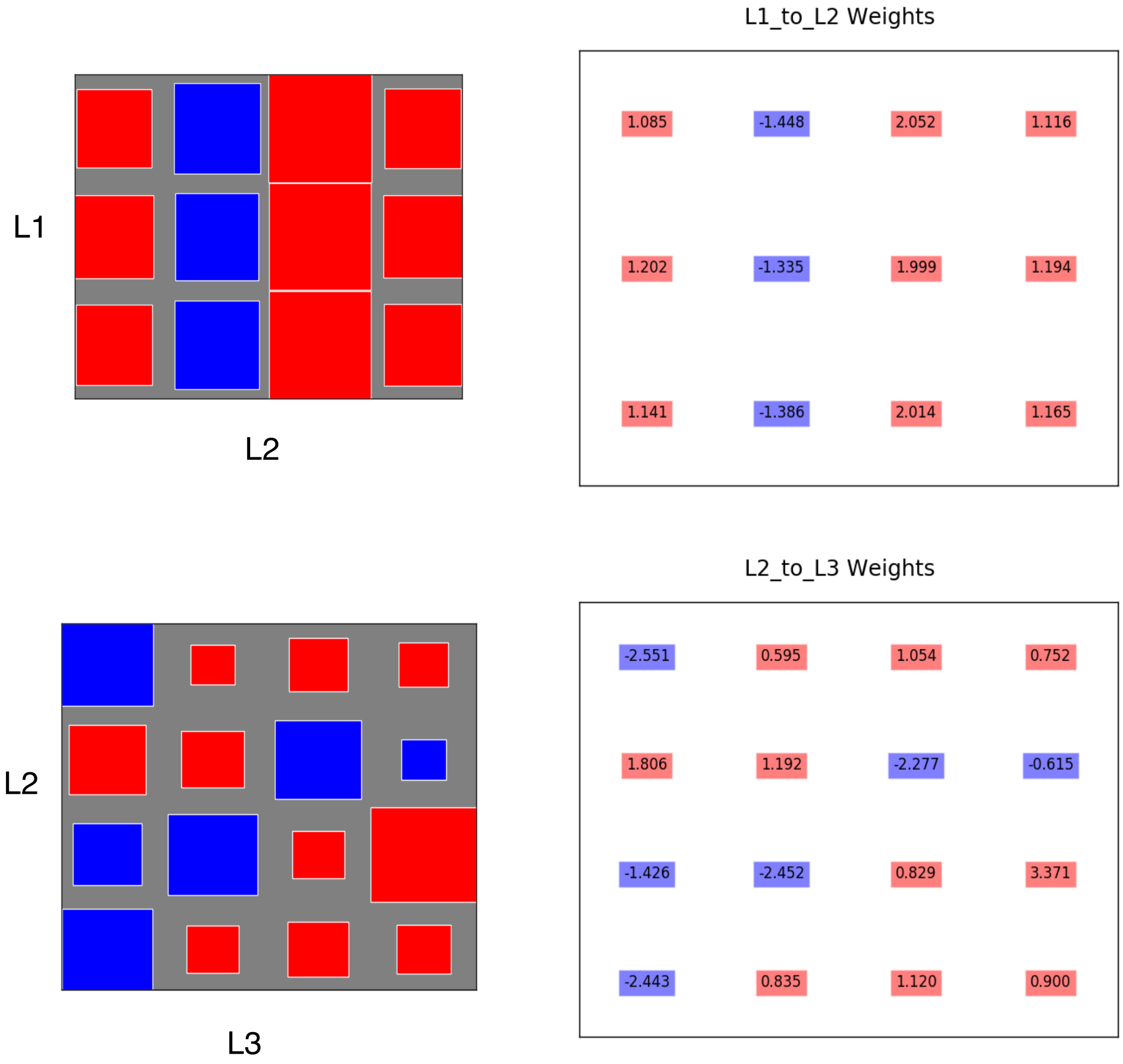

Hidden Layer
1.1
1.2
1.4
-1.9
-1.8
-1.7
1.8
1.9
1.9
Dead
H1
-1
H2
2.4
H3
H4
-2.8
{1,2,3}
{0,1}
Fire on these sums
{2,3}
Output Layer
-2.0
2.6
-1.0
0.5
-2.5
-2.9
1.4
1.1
3.2
O1
-0.9
O2
2.8
2.1
O3
-3.7
O4
Only fires when on(H2) and off(H1) => "0"
Max'd when on(H2), but can only beat O1 on a 1, not a 0 => "1"
Only very active when on(H1) and off(H2), but it loses to O4 on a 3 => "2"
Needs on(H1) and on(H4), but can only beat O3 when H4 overpowers H1, and that only happens on a 3 => "3"

Figure 8: Brief logical analysis of the network from Figures 6 and 7. Under each hidden node, the numbers in curly brackets indicate the bit sums that cause the corresponding node's ReLU to fire (i.e. output a non-zero value) . Those values can then be used to infer the sums that stimulate output neurons, and to what degree a particular sum (e.g. 2) would stimulate one output neurons versus another. Remember that the predicted class corresponds to the output neuron with the highest activation value, where the network is trained to activate the nth output neuron when the sum is n.



L1_to_L2 Weights

1.085	-1.448	2.052	1.116
1.202	-1.335	1.999	1.194
1.141	-1.386	2.014	1.165

L2_to_L3 Weights

-2.551	0.595	1.054	0.752
1.806	1.192	-2.277	-0.615
-1.426	-2.452	0.829	3.371
-2.443	0.835	1.120	0.900

Figure 9: Weights for a second neural network (of dimensions 3-4-4) also trained as a bit counter. In this instance, more weights seem vital to the proper functioning of the network.

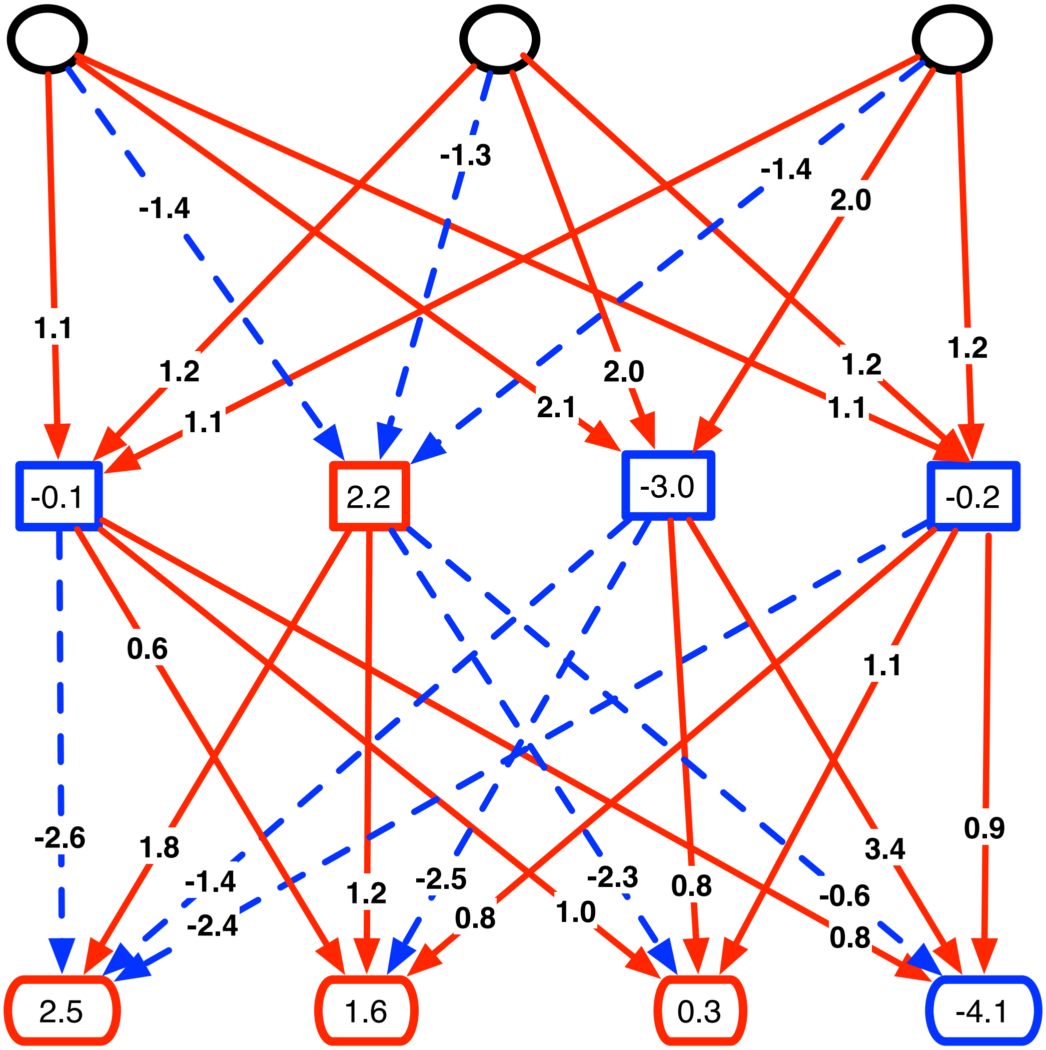


Figure 10: The full network (with all weights and biases) for the second bit counter.

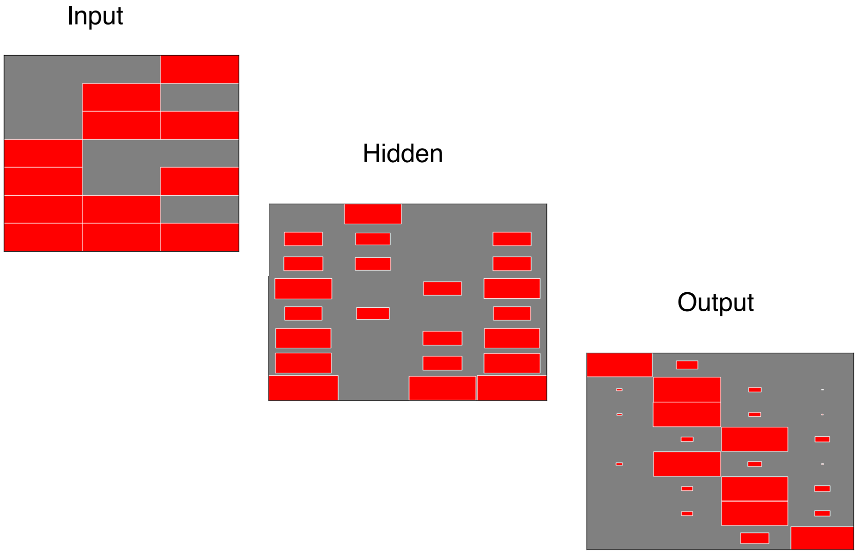
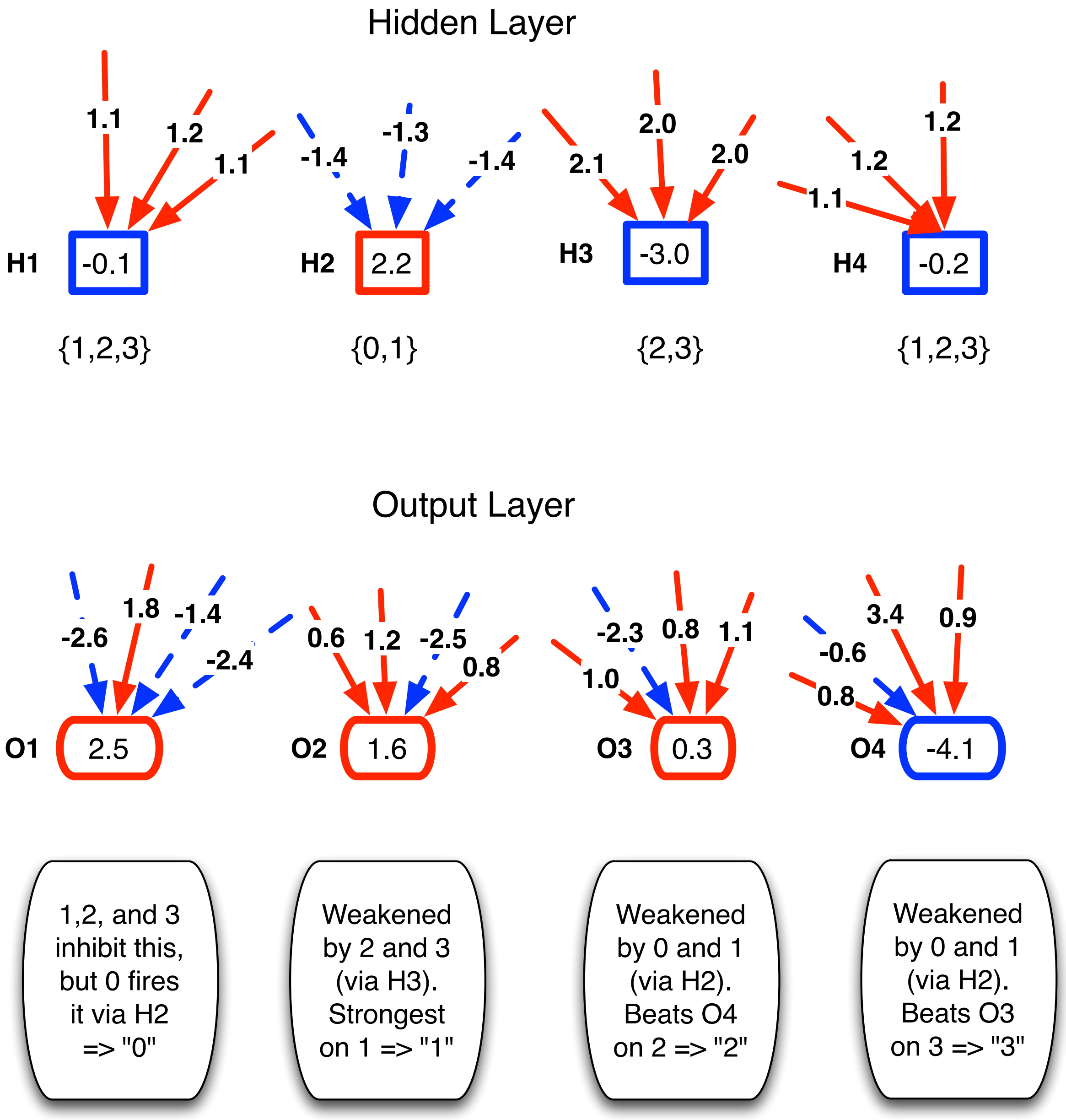


Figure 11: Activation patterns for the second 3-4-4 bit-counter network described in the text and in Figures 9 and 10. Input cases are the 3-bit encodings of the integers 0 to 7, but the input matrix does not include the zero vector: [0,0,0]. However, the hidden and output matrices show the 0-input case in their top row. Hence, the input matrix has 7 rows, while the other two matrices have 8 rows. Note that in all cases, the most active output neuron corresponds to the correct bit count.



1,2, and 3 inhibit this, but 0 fires it via H2 => "0"

Weakened by 2 and 3 (via H3). Strongest on 1 => "1"

Weakened by 0 and 1 (via H2). Beats O4 on 2 => "2"

Weakened by 0 and 1 (via H2). Beats O3 on 3 => "3"

Figure 12: Basic analysis of the second bit-counter network, many of whose details appear in Figures 9, 10, and 11.