How to build, train and test a feed-forward backpropagation network in the PDPyFlow software system[1]

This document assumes you have the PDPyFlow system installed in a directory called PDP on a linux or mac computer, and that you are working from within the PDP directory. The system depends on Python 3.5.2 and Tensorflow 0.12, which must also be installed. Please contact the author at aten@icloud.com or contact pdplab-support@stanford.edu for information on how to install the software.

This tutorial will take you through the process of creating your own feed-forward network capable of training through back propagation using the PDPyFlow software system. Along the way, you will gain some deeper understanding of different classes and functions that underly the FFBP interface. To make instructions clearer, this tutorial will demonstrate the creation of a new network by the example of 8-3-8 network, also known as auto-encoding network. This network will be trained to map an activation pattern of 8 input units to an identical pattern on 8 output units through a distributed hidden representation of only 3 hidden units.

Creating, training, and testing a network is done in three stages:
1. Preliminary stage
    1.1.   Prepare data
    1.2.   Import required tools
2. Construction stage
    2.1.   Create and configure network Layers
    2.2.   Connect the layers into a Network
    2.3.   Configure the network for training and testing
3. Running stage

Each of these steps is simplified by the FFBP API and the tutorial will explain some of the relevant details. However, if your curiosity exceeds the material presented in this tutorial, you are welcome to explore the source code. Moreover, FFBP relies on other fully open-source libraries that can be accessed through the links provided in some of the relevant sections of this tutorial or by a simple web search.

In order to create and edit the code you will need a simple text editor. You can use your own preferred text editor or use a standard UNIX editor such as **nano**. The example network in this tutorial will be created with the **nano** text editor since all users are guaranteed to have access to it.

In order to create a text file, first go to the directory where you want this file to reside. Once in the directory, use the nano command followed by the file name that you want to assign to the file (also see `$ man nano` for more information on usage):

---

[1] Correspondence: This tutorial was written by Alexandr Ten, who contributed to the development of **PDPyflow**. If you have any questions or feedback regarding the code, feel free to contact him at: aten8@icloud.com

```
user PDP $ nano filename.extension
```

This will create and open (but not save) a new text file or simply open an existing file where you can enter your code. If you want to save the file, first exit the editor by pressing $control + X$, then press Y or y if you want to save the contents of the file, and press $Enter$ to perform the command. You can open that file again by typing the same command.

An alternative way to create a file is by using standard commands like $ mkdir and $ touch. If you want to create your own network, it is a good idea to make a separate directory inside the FFBP directory and store your files there. This can be done as follows:

```
user PDP $ mkdir FFBP/dirname
user PDP $ touch FFBP/dirname/mynet.py
user PDP $ touch FFBP/dirname/mydata.txt
```

Of course, you can choose your own names for the directory and the files in it, but keep in mind that you are working with a command line interface and pretty much everything needs to be typed manually. In this tutorial we will name our files net838.py and net838_data.txt.

1. Preliminary Stage

1.1. Prepare data

Data preparation can be approached in different ways. One way is to create a separate text file that will later be read from the main script. That way we don't need to generate the same data every time we run a network. The text file that stores data must follow a strict structure that the object that we will build around it will understand. When such data file is loaded, it's read and processed, line by line, going from top to bottom. The first row and the first column are reserved for labels. The first row contains descriptive labels of the entries in their respective columns and is optional (however, if you choose not to include the labels row, leave a blank line as the first line in the document, since this line will be ignored when the file is read). Columns are separated by commas. The entries in the first column will be interpreted as labels of the pattern pairs contained in the same row. The next column is interpreted as an input pattern. Values of individual input units need to be separated by spaces. The same applies to the next column, which contains the output pattern associated with the pattern that preceded it. Thus, to create the data file for the 838 network we will create or open an existing file named net838_data.txt and then create a data for the network:

```
user PDP $ nano FFBP/dirname/net838_data.txt
```

Note that the white spaces $between\ columns$ are not necessary, and we've included them just for clearer presentation. Once we've done preparing the data we exit nano and save the changes.

```
GNU nano 2.5.3          File: net838_data.txt          Modified


inp_label,  input,          output,
p1,      1 0 0 0 0 0 0 0,  1 0 0 0 0 0 0 0,
p2,      0 1 0 0 0 0 0 0,  0 1 0 0 0 0 0 0,
p3,      0 0 1 0 0 0 0 0,  0 0 1 0 0 0 0 0,
p4,      0 0 0 1 0 0 0 0,  0 0 0 1 0 0 0 0,
p5,      0 0 0 0 1 0 0 0,  0 0 0 0 1 0 0 0,
p6,      0 0 0 0 0 1 0 0,  0 0 0 0 0 1 0 0,
p7,      0 0 0 0 0 0 1 0,  0 0 0 0 0 0 1 0,
p8,      0 0 0 0 0 0 0 1,  0 0 0 0 0 0 0 1,
```

1.2. Import required modules

When data is prepared we can begin coding the main script. For this we've created a new file called net838.py by typing $ nano FFBP/dirname/net838.py. The first thing we want to do is import all the required modules that will allow us to create the network:

It is not necessary to import everything at once in the beginning, but it's a commonly followed 'pythonean' practice to do so. The first module imported is the code module which will enable us to

```
GNU nano 2.5.3          File: net838.py          Modified


import code
import tensorflow as tf

import utilities.activation_functions as actf
import utilities.evaluation_functions as evalf
import utilities.error_functions as errf

from utilities.model import model

from FFBP.classes.DataSet import DataSet
from FFBP.classes.Layer import Layer
from FFBP.classes.Network import Network
from PDPATH import PDPATH
```

interact with the network when we run the entire script after we've created it. Next, the tensorflow module will be used to create the necessary tensorflow objects and variables. The first three modules from the utilities package contain some useful functions that we are going to configure some of our network's settings with. The model function imported from the model module arranges the layers into a dict that will be used to initialize an instance of the Network object. From the constructors package we import DataSet, Layer, and Network classes which define how these objects are created and structured, and how they behave. Finally, we import the PDPATH function which simply returns the absolute path to the PDP directory, regardless of where this directory is in the file system.

Those of you who are new to python will benefit from noting the intuitive syntax behind import conventions. If you import a module using `import modulename` or `import modulename as mn` syntax, you will need to access the required objects through the name scope of the module. For example, later in the code we will need to refer to one of the activation functions for which we defined the name scope as "actf": `actf.sigmoid`. From any module we can also import individual objects by following up the import expression with a `from` - statement. By importing the DataSet class from `FFBP.classes.DataSet` module we can use it without referring to the module form which it came. For more information, see: https://docs.python.org/3.3/reference/import.html.

Now that the necessary tools are available, we can create a DataSet object, which will allow the network to draw pattern pair batches of specific sizes from it as well as permute the pattern pair order if needed. In order to create a DataSet object we need to point it to the data file that we've saved earlier. And since we will be training and testing on the same data, the `testSet` object will be constructed identically to the `trainSet`:

You may want to use different data sets for training and testing. Accordingly, you would need to create two separate files.

```
path = PDPATH() + '/FFBP/dirname/net838_data.txt'
trainSet = DataSet(path)
testSet = DataSet(path)
```

2. Constructing the Network

2.1. Create and configure network layers

We begin by creating the layers. Our model has three of them, however, the input layer is not considered a Layer object since it behaves differently from hidden and output layers. Rather, it is seen as a placeholder into which we will feed different data from the data set when the network is run. A similar placeholder needs to be created for the target patterns:

Tensorflow has a useful operation to implement such placeholders. The above commands create two tensorflow placeholders. Each of these will be fed numeric arrays that will be converted to tensorflow tensors. The first argument in both cases is `tf.float32` which specifies the type of the

```
PHinp = tf.placeholder(tf.float32, shape=[None,8], name='input')
PHout = tf.placeholder(tf.float32, shape=[None,8], name='target')
```

resulting tensors. The tensor shape is given as a list or a tuple with two elements (the number of rows, the number of columns). If `None` is given instead of an integer, the value for corresponding dimension will be open ended. This is useful when we want to feed batches of different sizes (e.g. for training and testing) without having to change the sizes of the placeholders. The last argument is just a string representing the name of the output tensor. For more information on placeholders and other approaches to feeding data into a graph consult the following links: https://www.tensorflow.org/api_docs/python/io_ops/placeholders#placeholder and https://www.tensorflow.org/how_tos/reading_data/.

The hidden layer and the output layer are created as Layer objects because unlike the input layer they both have incoming tensors, incoming connections, and activation functions associated with them. Layer objects expect many arguments for initialization:
First, we specify the input. Our hidden layer takes the output of the input_layer placeholder, which is

```
hid = Layer(input = PHinp,
            size = 3,
            act = actf.sigmoid,
            layer_name = 'hidden',
            layer_type = 'hidden')
hid.init_wrange([-1, 1, 1])
```

a tensor of shape ?x8. As one of the preconditions of our auto-encoding problem, the hidden layer size is 3 units. Next we define the activation function for the layer. In our example we use the sigmoid function from the activation_functions module. Functions from this module are tensorflow operations, which means that they take tensors as their inputs and return tensors as their outputs. The output tensor will be captured in the Layer.act attribute when we run the network. The layer_name is given by a string and could be a string version of the name of its variable or any other string. Important thing to keep in mind is that this string is used by the viewer to annotate the layers with their corresponding names. Finally, layer_type should be given either the 'hidden' or 'output' string argument. This is optional, but if layer_type is identified as anything other than 'output' you won't be able to visualize the target vector for this layer later, if it exists.

After creating a Layer object we can proceed in two different ways about constraining the initialization of incoming weights connecting the Layer to its sender. In this tutorial we've opted for one of these alternatives and in the XOR.py program you saw the other approach. Here, right after creating the hidden layer, we use the init_wrange() method to set the lower and the upper bounds of weight range. This method expects one of three possible inputs. If you just give it a 0, all weights will be initialized at 0. If you give it a list or a tuple of two numbers, the weights will be initialized randomly and uniformly between the values in the list or the tuple. The first value will be the lower limit, and the second value will be the upper limit. You can optionally include a third item in the input container and it will serve as a random number generator seed. Using a particular seed value, you will be able to run several sessions, perhaps using different hyperparameters, starting from the same pseudo-randomly-generated state. Note that we need to configure the weights for each layer separately, unlike in the XOR.py program. However, this provides some extra flexibility if we wanted a set of weights to a specific layer to be in some specific range.

Similarly to the above, we instantiate our output layer. Take note of the differences between this and the hidden layer initialization:
As you can see the input tensor is the activation attribute of the hidden Layer object. Note also, that you don't have to use the same activation function for each layer. You can use any function from the

```
output = Layer(input = hid,
               size = 8,
               act = actf.sigmoid,
               layer_name = 'output',
               layer_type = 'output')
output.init_wrange([-1, 1, 1])
```

activation_functions module or create your own (just make sure it outputs a tensorflow tensor), if you want to have layers with different activation functions. This module is there for convenience and accessibility purposes and uses tensorflow activation functions directly without adding any utility to

underlying computation. You can always use tensorflow functions directly, e.g. act = tf.nn.softmax (see https://www.tensorflow.org/api_docs/python/nn/activation_functions_). Currently seven choices are provided in the activation_functions module (listed in Table 1). Also see: https://en.wikipedia.org/wiki/Activation_function for a more comprehensive review of activation functions.

Table 1. Activation functions from PDP/utilities/activation_functions.py

name	equation	derivative	range
linear	$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
sigmoid	$f(x) = \dfrac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
tanh	$f(x) = \dfrac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
softmax	$f(\vec{x})_i = \dfrac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} ; i = 1, \ldots, K$	$\dfrac{\partial f(\vec{x})_i}{\partial x_j} = f(\vec{x})_i (\delta_{ij} - f(\vec{x})_j)$	$(0, 1)$
softplus	$f(x) = \ln(1 + e^{x})$	$f'(x) = \dfrac{1}{1 + e^{-x}}$	$(0, \infty)$
relu	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$

2.2 Several layers feeding into one layer

You might want to explore architectures in which a given layer receives activation from two or more sending layers. Imagine we added another hidden layer to our 8-3-8 model, not sequentially to the existing one, but in parallel as in Figure 1.

All we need to do for this computation to work is concatenate (or join) the activations of the two

```
hid1 = Layer(…)
hid1.init_weights(…)
hid2 = Layer(…)
hid2.init_weights(…)

output = Layer(input = [hid1, hid2],
                size = 8,
                act = actf.sigmoid,
                layer_name = 'output',
                layer_type = 'output')
output.init_wrange([-1, 1, 1])
```

hidden layers with the help of tf.concat() function and use the resulting composite as the output layer's

input tensor. This is done internally by the receiving layer object, so all we are left with is passing the list of inputs:

You can see a concrete example of implementing parallel converging layers in the FFBP/EightThings.py program and the actual code of how concatenation is done in the FFBP/classes/Layer.py file in the first few lines of the __init__ method of the Layer class.

Figure 1. FFBP network with two parallel hidden units
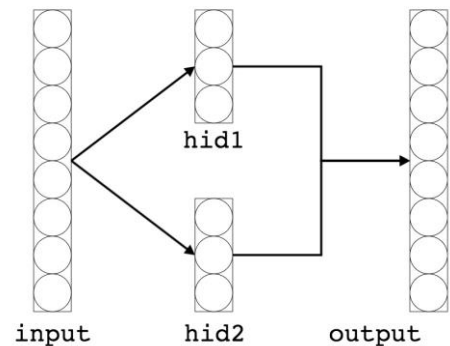
2.3. Connect layers into a network

To construct a network from the created Layers, we first need

```
model838 = model(PHinp, [hid, output], PHout)
```

to build a model of the network. It is simply a python dict with three keys: 'images', 'network', and 'labels'. The first key stores input layers (i.e. input placeholders of the network), The 'network' key stores the Layer objects, and the 'labels' key contains the target placeholder. You can create such dict yourself, but as a convenience tool we use the model function that we've already imported:

```
net838 = Network(model838, name='net838', logdir='dirname')
```

The order in which we pass the arguments matters. All input placeholders must be given first. If there are several input layers, pass them as a list or a tuple containing each input layer. Next, analogously pass all the Layer object arguments. Finally, the targets placeholder is given as the third position argument. When we have the model of out network, we can pass it to the Network constructor:

Optionally, you can name your network with a string. Another optional argument is logdir. If this argument is omitted, the network will store training logs and parameter checkpoints inside the default directory FFBP/logs. If you do specify the logdir parameter (which should be the same as you directory in the FFBP directory), the default path will be changed to FFBP/dirname/logs and everything will be stored there.

2.4. Configure the network for training and testing

```
net838.train_set = trainSet
net838.test_set = testSet
```

To configure our network, we will use various methods that are available for the Network class as well as assign some values to its attributes directly. We will first give it the training and testing data

```
net838.initconfig(loss = errf.squared_error,
                  train_batch_size = 8,
                  learning_rate = .3,
                  momentum = .9,
                  permute = False,
                  ecrit = 0.01,
                  test_func = evalf.tss)
```

sets that we've defined earlier:

Next, we will use the initconfigure method to initialize tensorflow variables that represent network weights and configure various training settings:

If you add an extra argument wrange, and define it as a two- or three-element list, it will overwrite

```
net838.init_weights()
net838.configure(loss, traing_batch_size, learnin_rate, momentum,
permute, test_func)
```

the initialized weights that have been specified before.

Alternatively, we could perform configuration and initialization separately:
The keyword arguments are rather self-explanatory. We first define the loss function to be the squared_error function from the error_functions module that we imported as errf. Likewise, we configure the test function to compute tss with the help of the corresponding function from the evaluation_functions module (imported as evalf). Although functions for error and performance evaluation calculations can be set independently, they perform identical computations. That is, the squared error function from error functions module is the same as the tss function from the evaluation

Table 2. Error (loss) and evaluation functions from PDP/utilities

name	equation	
squared error	$L_p(t,a) = \sum_i (t_{ip} - a_{ip})^2$	$\dfrac{\partial L_p}{\partial a_{ip}} = 2 \sum_i (t_{ip} - a_{ip})$
cross entropy	$L_p(t,a) = -\sum_i [t_{ip} \log(a_{ip}) + (1 - t_{ip}) \log(1 - a_{ip})]$	$\dfrac{\partial L_p}{\partial a_{ip}} = -\left(\dfrac{t_{ip}}{a_{ip}} + \dfrac{1 - t_{ip}}{1 - a_{ip}} \right)$

L = loss, p = pattern index, t = target value, a = obtained activation value, i = unit in

function module. Keep in mind, however that error function (errf) is used to compute gradients during training, while the evaluation function (evalf) is the one used for testing. Table 2 shows the available error and evaluation functions and their derivatives:

Note that the batch size needs to divide the total number of patterns in the train set, otherwise an error will be raised.

When we construct, configure, and initialize the network, several inconspicuous processes occur behind the scenes. Essentially, we are coding the flow of information through a computational process. A level deeper than the relatively intuitive interface, the code relies on a tensorflow Graph object that contains all created tensorflow Operations (units of computation or functions) and Tensors (units of data that flow through the graph). A complete graph can be run in a tensorflow Session which is capable of reserving computational resources for launching the graph. So what we've really done so far is created a functional tensorflow graph and reserved resources for its execution. Here is a recommended tensorflow page that introduces some of the concepts mentioned so far: https://www.tensorflow.org/get_started/basic_usage.

Add the following line at the end of your code to enable interactive usage (see 3.3 Interactive Methods on page 9): code.interact(local = locals()); then save and close the file.

3. Running the Network

The network we've created can be trained and tested with or without visualization depending on user preference. Each of these capabilities can be accessed either interactively through the use of the python shell, or autonomously by scripting the appropriate commands, or through a combination of both. Regardless of how training / testing is approached, the Network relies on two basic methods

that are not intended for direct use, but are as accessible as any other Network method. These are Network._train() and Network._test() methods. It is useful to be familiar with these methods, so they will be briefly described in the following section.

Although visualization methods rely on data generated by these training and testing methods, it is an independent process that can be called outside any particular run, provided that some appropriate data exists in a known and accessible location. We will therefore demonstrate visualization in a separate section.

3.1 Network._train() Method

Network._train() requires three positional arguments: *num_epochs*, *dataset*, *batch_size*. It also takes two keyword arguments with default values: *ecrit* = 0.01, *permute* = False. Unsurprisingly, *num_epochs* is responsible for setting the number of training iterations the network will go through. One epoch accomplishes a complete sweep through the data set, though learning experience which the network can get in one epoch can vary. An epoch can be comprised of one or more steps depending on the batch size. The bigger the batch size, the less steps it will take the network to complete an epoch. More precisely, the number of steps is equal to the total number of training pairs divided by batch_size Therefore, batch size cannot be greater than the number of training pairs and must divide it into equal parts. Every training step yields a batch-wise error measure that is accumulated in the epoch-wise error. This epoch-wise or total error is compared to a critical value. The *ecrit* parameter sets this criterion so that the network breaks from the training loop whenever the obtained loss becomes less than or equal to *ecrit*. Finally, if the permute parameter is set to True, the network will train in ptrain mode. If you want to perform stochastic gradient descent, specify a train_batch_size that is less than the number of patterns in the training set, and divides evenly into that number, and then set permute to True. This will then cause the network to run minibatches of train_batch_size patterns (selected at random without replacement from the training data), updating the weights after each minibatch.

3.2 Network._test() Method

Network._test() method is designed to do two things: (1) evaluate the error at the current state of the network and (2) make a snapshot of the particular state and store it for later analyses. The method takes two positional arguments (*dataset* and *evalfunc*) and two keyword arguments with a default values (*snapshot* = False and *checkpoint* = False). Therefore, it needs to know which data set it will be tested on and which evaluation function it is to use for this test. It will also check whether the *snapshot* parameter is anything other than False or 0. In this case, the test method will try to take a snapshot of the network state. The checkpoint parameter gates the activity of tensorflow's saver function that saves all session variable values (i.e. weights and biases) at a particular time. Saved checkpoint files which are essentially network states can be restored later.

As you can see, direct calls to the _train() and _test()methods require entering all of these parameters every time you want run some training or test your network, so it might not be ideal for interactive use. If we configure the network the way we did our example network, it will be much simpler to train and test the network interactively.

3.3 Interactive Methods

The first thing to include in our main script that will allow to interact with our network is to use the

```
code.interact(local = locals())
```

code module that we've imported in the beginning:
This function launches an interactive console which emulates the behavior of the interactive python interpreter and loads variables in the local scope of the script to the scope of the console. At this point we can consider our main script complete. If we save and quit script and run it in the terminal

by using the `$ python` command and passing to it the path to our script we will be set to interact with the network. Note that if you created your.py file in a subdirectory, you will need to precede to put

```
user PDP $ python3.5 FFBP/net838.py
Python 3.5.2 |Continuum Analytics, Inc.| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

the subdirectory name in the path to the file:
The three angular brackets are the console's command prompt. The prompt indicates that the control is given to the user and that the console is waiting for some input from them. Let's go ahead and use

```
>>> net838.test()
[net838] Initial test...
[net838] Error = 15.319734573364258
```

the interactive test method to test our yet untrained network:
Internally, the $_{test()}$ method was called. However, we didn't need to enter any arguments, because we've already configured them in the script. The output is the single error value. As you might imagine, more has happened that it seems. Since we didn't change the scope setting, the network took a snapshot of all 9 attributes (see Table 1) and stored them into a snapshot file which was created inside the logging directory. We will talk more about the logging directory (or logdir for short) later when we discuss visualization.

```
>>> net838.train(50)
[net838] Training: |###############| 100.0%
[net838] Done training for 50/50 epochs (0.051 seconds)
```

Now let's train the network for 50 epochs with the interactive train method:
A text-based progress bar appears and fills up as the network completes each iteration of the loop inside the $_{test()}$ method. When the training loop terminates, the network reports the number of epochs it trained for on the most recent call, the total number of epochs, and the time it took to finish the training loop.

```
>>> net838.tnt(500,50,0)
[net838] Now in train and test mode...
[net838] epoch 0: 15.319734573364258
[net838] epoch 50: 5.697778701782227
[net838] epoch 100: 5.017020225524902
[net838] epoch 150: 5.009669303894043
[net838] epoch 200: 5.006933689117432

...
```

```
...
[net838] epoch 250: 5.005369186401367
[net838] epoch 300: 5.004334449768066
[net838] epoch 350: 5.003580093383789
[net838] epoch 400: 5.0029826164245605
[net838] epoch 450: 5.002461910247803
[net838] Final error (epoch 500): 5.00193977355957
[net838] Process terminated.
```

There is another useful method that combines the two described above. It will make the network test and train intermittently for some maximum number of epochs or until ecrit is reached. We call this method tnt:

What we did is told the network to test once, then train for 500 epochs, stopping every 50th epoch to test again. The argument of 0 is passed to the *checkpoints* parameter and stops the network from saving tf.checkpoints. Any other positive integer would set the interval at which the network would

```
>>> net838.tnt(500,50,0,
              train_set = some_set, test_set = some_other_set)
```

save the checkpoints, but note that this process compromises speed quite a bit. If you want to use different data sets than those configured with Network.initconfig(), you can add two corresponding keyword arguments. e.g:

3.4 Visualization

The train() and test() methods you saw above were designed to function with minimal input. However,

```
>>> net838.train(100,1)
[net838] Training: |###############| 100.0%
[net838] Done training for 100/650 epochs (0.082 seconds)
```

the behavior of both methods can be modified by changing the default value of the vis keyword argument from False to True or 1. 'Vis' stands for visualization, and if set to True, the two methods
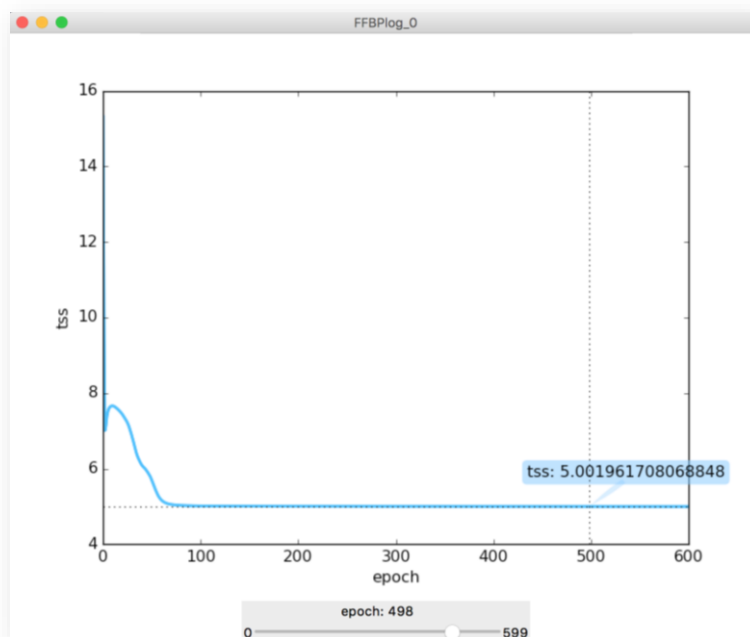
will create their respective visualizations. So, for example, if we wanted to train our 838 network for another 100 epochs and show us how the loss value has changed over time since epoch 0 we would type:

As with the previous demonstration the output shows us the progress bar, epochs, and time, but this time, a new window has appeared:

In it, you can see the line plot of the error (in our case tss) across time. The slide widget below the plot allows you to slide back and forth through the timeline to see the value of tss more precisely. If we click on any point of the slide except the slider pin, the position of the pin will change one unit in the direction of where we've clicked. Note that the '>>>' prompt has not disappeared from the shell.

```
>>> net838.test(1)
[net838] Test after epoch 600:
[net838] Error = 4.999884128570557
[FFBP Viewer] Initializing viewer for FFBPlog_0 ...
```

If you execute the same command again, the network will train for another 100 epochs and the graph will be updated accordingly.

If we want to inspect the state of the network rather than its performance, we can use the test method with visualization parameter set to True:

Network state visualization is also interactive, which is controlled through the Controls panel at the bottom of the window. The visualization shows you arrays of values that were tested at any particular moment. In the example below we can see depictions of the hidden layer and the output layer. Each layer's visualization can be divided into two parts: the forward propagation (FP; on the left) and the back propagation (BP; right) part. The FP side shows you the values of each individual weight to each individual unit in the layer, as we as bias, net input and activation values of each unit. If the
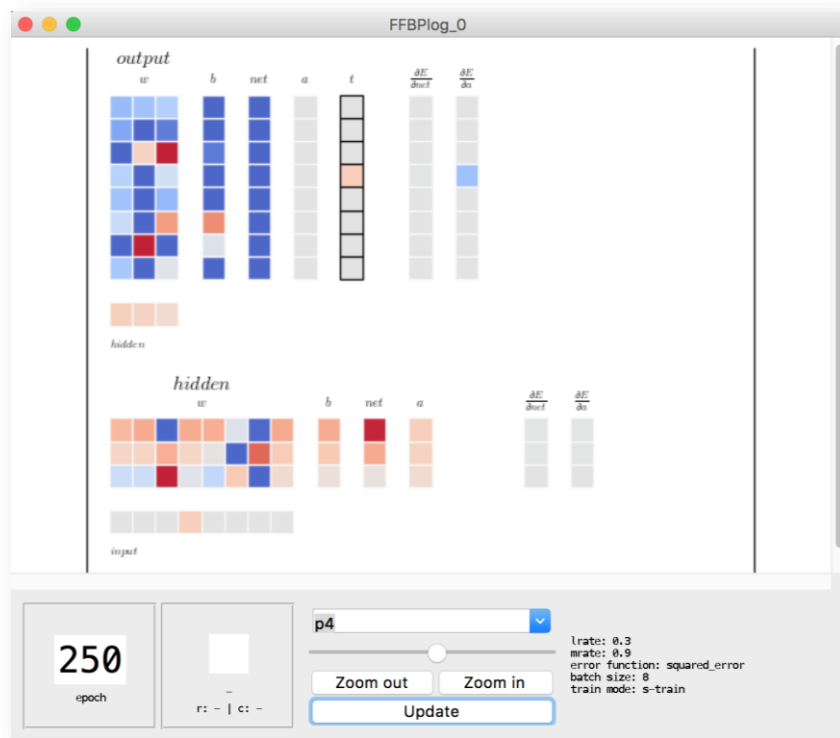
layer has a target vector associated with it, you will also see it next to the activation vector. Below each weight array, you can see the input vector to the layer. The BP side contains arrays with values of gradients with respect to net input and activation. You can inspect the precise value of each tile by clicking on it. As you do, the chosen tile will be represented at the left-hand side of the controls panel.

In addition to inspecting tile values, the control panel allows you to discretely move through time and observe the state of the network at each epoch that it was tested on. You can use the slider widget to navigate the particular time slice you want to see and then click the update button to change the image. Since the state of the network is different in response to different inputs, you can look at any input on a given epoch, by choosing one from the drop-down menu and hitting the update button.

3.5 FFBP Viewer and Reader

The above visualization is a result of reading a logged snapshot-log file that was created when the network was tested for the first time and appended on subsequent tests. The snapshot-log file is stored in the session directory which is created every time a new Network object is instantiated. The directory is stamped by a single index and contains whatever contents the user has saved during the session. If you would like to view a previously stored session, you can use the FFBP_viewer.py. To do this, go back to the terminal prompt and run the viewer. You will first be asked to provide the name of the user directory if you have created one inside FFBP. If you have not, the program will look for snapshots in the default directory. Assuming you have created a separate directory for your files, you need to enter the name of that directory first. Then you will see another prompt asking you

```
user PDP $ python3.5 FFBP/FFBP_viewer.py
[FFBP Viewer] Provide user directory (if any), or press 'enter' to use default directory: dirname
[FFBP Viewer] Enter name of log directory OR corresponding index: 0
[FFBP Viewer] Initializing viewer for FFBPlog_0 ...
[FFBP Viewer] Would you like to proceed?
[y/n] -> 
```

to provide the name of log directory itself or simply its index. When you enter the name or the index, the familiar visualization panel will appear. The program will ask you if you want to proceed to view another file. If you do, enter 'y' and you will be prompted to give another index:

Another program, FFBP_reader.py, allows the reader to extract data from a log. You access the log in the same was as with the FFBP_viewer program. In this case, however, the data is simply held on a 'reader' object that can return the values of logged variables to the user. Each variable associated with each layer of the network is stored in a list with an array for the values of the variable at each test time point. For example, if the network was tested a total of 13 times, once prior to the start of training and one final time after the criterion was reached, there will be 13 entries in the list of arrays for each variable. These variables are the same ones that are displayed in the viewer. One can place the values desidered valiable in an array for analysis within python or one can save them to a csv file for analysis with another program.

To see a summary of the contents type 'reader.summary()' at the command prompt. One might see, for example, that the hidden layer in our 8-3-8 encoder has a list of 13 entries for a variable called 'act' in an 8x3 array. The first dimension corresponds to the test item, while the second dimension corresponds to the unit within the hidden layer activation pattern for that item. One can put the list of hidden acts in an array with a command like

 'hacts = reader.main['hidden'].act

(the quoted string is the name you gave the layer when you defined it). To access the array for a given test time point, you can simply enter 'hacts[index]', where index ranges from 0 to (in this case) 13-1, or 12. To store the act array from the test with index 0 in a csv file called hacts.csv, you would use a command like

reader.scsv('hacts.csv','hidden',0,'act')

The file will be saved in the same directory as the log file that the data was taken from. You can append additional data to the same file if you re-use the same file name; otherwise a new file will be created.

Conclusion

We have walked through the steps necessary for running a functional feed-forward neural network. We began by creating a data file and data objects for training and testing our network. We then constructed the actual network and configured it for further interaction. After initializing the network, we were able to train and test it using the interactive methods as well as visualize the results with the help of a couple of visualization tools. By now, you should be able to build your own feed-forward network and explore its progress as you train it. Although the range of possible architectures is rather limited with the provided tool set, this tutorial didn't go through every possible variation and more can be understood and achieved by exploring the source code.

You may have already noticed that there is a net838.py file inside the FFBP directory. If you follow the instructions of this tutorial you should end up with a copy of the exact same program inside your own directory. You can compare how the two programs behave to make sure you haven't missed anything. Another thing you can do is use the net838.py file as a template that you can then modify to create your own network.