The output should look something like

```
tensor([[0.3380, 0.3845, 0.3217],
        [0.8337, 0.9050, 0.2650],
        [0.2979, 0.7141, 0.9069],
        [0.1449, 0.1132, 0.1375],
        [0.4675, 0.3947, 0.1426]])
```

Let's get started with the assignment.

Instructions

Part 1 - Datasets and Dataloaders (10 points)

In this section we will download the MNIST dataset using PyTorch's own API.

Helpful Resources:

- https://pytorch.org/docs/stable/torchvision/datasets.html#mnist (https://pytorch.org/docs/stable/torchvision/datasets.html#mnist)
- https://pytorch.org/docs/stable/torchvision/transforms.html (https://pytorch.org/docs/stable/torchvision/transforms.html)
- https://pytorch.org/tutorials/beginner/data_loading_tutorial.html (https://pytorch.org/tutorials/beginner/data_loading_tutorial.html)

The `torchvision` package consists of popular datasets, model architectures, and common image transformations for computer vision. We are particularly concerned with `torchvision.datasets` and `torchvision.transforms`. Check out the API for these modules in the links provided above.

Create a directory named `hw8_data` with the following command.

In [2]:

```
!mkdir hw8_data
```

```
mkdir: hw8_data: File exists
```

Now use `torch.datasets.MNIST` to load the Train and Test data into `hw8_data`.

- **Use the directory you created above as the `root` directory for your datasets**
- **Populate the `transformations` variable with any transformations you would like to perform on your data.** (Hint: You will need to do at least one)
- **Pass your `transformations` variable to `torch.datasets.MNIST`. This allows you to perform arbitrary transformations to your data at loading time.**

In [3]:

```
import torchvision
from torchvision import datasets, transforms

## YOUR CODE HERE ##
transformations = torchvision.transforms.Compose([transforms.ToTensor()])
mnist_train = torchvision.datasets.MNIST(root='./hw8_data', train=True, transfor
m=transformations, download=True)
mnist_test = torchvision.datasets.MNIST(root='./hw8_data', train=False, transfor
m=transformations, download=True)
```

Check that your torch datasets have been successfully downloaded into your data directory by running the next two cells.

- Each will output some metadata about your dataset.
- Check that the training set has 60000 datapoints and a `Root Location: hw8_data`
- Check that the testing (**also validation in our case**) set has 10000 datapoints and `Root Location: hw8_data`

Notice that these datasets implement the python `__len__` and `__getitem__` functions. Each element in the dataset should be a 2-tuple. What does yours look like?

In [4]:

```
print(len(mnist_train))
print(len(mnist_train[0]))
mnist_train
```

```
60000
2
```

Out[4]:

```
Dataset MNIST
    Number of datapoints: 60000
    Split: train
    Root Location: ./hw8_data
    Transforms (if any): Compose(
                             ToTensor()
                         )
    Target Transforms (if any): None
```

In [5]:

```
print(len(mnist_test))
print(len(mnist_test[0]))
mnist_test
```

```
10000
2
```

Out[5]:

```
Dataset MNIST
    Number of datapoints: 10000
    Split: test
    Root Location: ./hw8_data
    Transforms (if any): Compose(
                             ToTensor()
                         )
    Target Transforms (if any): None
```

Any file in our dataset will now be read at runtime, and the specified transformations we need on it will be applied when we need it..

We could iterate through these directly using a loop, but this is not idiomatic. PyTorch provides us with this abstraction in the form of `DataLoaders` . The module of interest is `torch.utils.data.DataLoader` .

`DataLoader` allows us to do lots of useful things

- Group our data into batches
- Shuffle our data
- Load the data in parallel using `multiprocessing` workers

Use `DataLoader` to create a loader for the training set and one for the testing set

- **Use a `batch_size` of 32 to start, you may change it if you wish.**
- **Set the `shuffle` parameter to `True` .**

In [6]:

```
from torch.utils.data import DataLoader

batch_size = 32

## YOUR CODE HERE ##
train_loader = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, s
huffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shu
ffle=True, num_workers=4)
```

The following function is adapted from `show_landmarks_batch` at [https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#iterating-through-the-dataset (https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#iterating-through-the-dataset)](https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#iterating-through-the-dataset) .

Run the following cell to see that your loader provides a random `batch_size` number of data points.

In [7]:

```
import matplotlib.pyplot as plt
from torchvision import utils
%matplotlib inline

def show_mnist_batch(sample_batched):
    """Show images for a batch of samples."""
    images_batch = sample_batched[0]
    batch_size = len(images_batch)
    im_size = images_batch.size(2)

    grid = utils.make_grid(images_batch)
    plt.imshow(grid.numpy().transpose((1, 2, 0)))
    plt.title('Batch from DataLoader')

# Displays the first batch of images
for i, batch in enumerate(train_loader):
    if i==1:
        break
    show_mnist_batch(batch)
```

In [8]:

```
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

class OneLayerModel(nn.Module):
    def __init__(self):
        super(OneLayerModel, self).__init__()
        self.fc = nn.Linear(28*28, 10)

    def forward(self, x):
        ## YOUR CODE HERE ##
        x = x.view(-1, 784)
        return self.fc(x)
```

Part 3 - Training and Validation (45 points)

In this section we will learn how to use the concepts we've learned about so far to train the model we built, and validate how well it does.We also want to monitor how well our training is going while it is happening.

For this we can use a package called `tensorboardX` . You will need to install this package using `pip` or `Anaconda` , based on your dev environment. Additionally, we'll want to use a logging module called `tensorboardX.SummaryWriter` . You can consult the API here https://tensorboardx.readthedocs.io/en/latest/tutorial.html (https://tensorboardx.readthedocs.io/en/latest/tutorial.html). Run the next cell to ensure that all is working well.

In [10]:

```
# Try uncommenting these commands if you're facing issues here
# !pip3 install -U protobuf
# !pip3 install -U tensorflow
# !pip3 install -U tensorboardX

%load_ext tensorboard.notebook
from tensorboardX import SummaryWriter
```

We have provided the code to use `tensorboard` just before calling your `train` function. You don't have to change the top-level log directory, but you can create multiple runs (different parameters or versions of your code) just by creating subdirectories for these within your top-level directory.

Now use the information provided above to do the following:

- **Instantiate a `OneLayerModel` with the appropriate input/output parameters.**
- **Define a cross-entropy loss function.**
- **Define a stochastic gradient descent optimizer based for you model's parameters. Start with a learning rate of 0.001, and adjust as necessary. You can start with the vanilla `optim.SGD` optimizer, and change it if you wish.**
- **Create a `SummaryWriter` object that will be responsible for logging our training progress into a directory called `logs/expt1` (Or whatever you wish your top-level directory to be called).**

In [11]:

```
## YOUR CODE HERE ##
model = OneLayerModel()
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
writer = SummaryWriter('logs/expt1')
```

We've finally come to the point where we need to write our training set up. We're going to use both our training and testing (validation) sets for this. Note that traditionally, you would separate part of your training data into validation data in order to get an unbiased estimate of how your model performs, but here we'll just pretend that our testing data is our validation data.

Training a model with batches of data broadly involves the following steps:

1. **One `epoch` is defined as a full pass of your dataset through your model. We choose the number of epochs we wish to train our model for.**
2. **In each epoch, set your model to train mode.**
3. **you feed your model `batch_size` examples at a time, and receive `batch_size` number of outputs until you've gotten through your entire dataset.**
4. **Calculate the loss function for those outputs given the labels for that batch.**
5. **Now calculate the gradients for each model parameter.** (Hint: Your loss function object can do this for you)
6. **Update your model parameters** (Hint: The optimizer comes in here)
7. **Set the gradients in your model to zero for the next batch.**
8. **After each epoch, set your model to evaluation mode.**
9. **Now evaluate your model on the validation data. Log the total loss and accuracy over the validation data.** (Note: PyTorch does automatic gradient calculations in the background through its `Autograd` mechanism https://pytorch.org/docs/stable/notes/autograd.html (https://pytorch.org/docs/stable/notes/autograd.html). Make sure to do evaluation in a context where this is turned off!)

Complete the `train()` function below. Try to make it as general as possible, so that it can be used for improved versions of you model. Feel free to define as many helper functions as needed. Make sure that you do the following:

- **Log the *training loss* and *training accuracy* on each batch for every epoch, such that it will show up on `tensorboard`.**
- **Log the loss on the validation set and the accuracy on the validation set every epoch**

You will need to produce the plots for these.

You may also want to add some print statements in your training function to report progress in this notebook.

In [12]:

```
def train(model, train_loader, test_loader, loss_func, opt, num_epochs, writer):
    batch = 0
    for epoch in range(num_epochs):
        model.train()
        for i, (images, classes) in enumerate(train_loader):
            batch+=1
            data, target = Variable(images), Variable(classes)
            opt.zero_grad()
            output = model(data)
            pred = output.data.max(1)[1]
            accu = (float(pred.eq(target.data).sum())/float(batch_size))*100.0
            l = loss_func(output, target)
            writer.add_scalar('train_accuracy', accu, batch)
            writer.add_scalar('train_loss', l.item(), batch)
            l.backward()
            opt.step()

        model.eval()
        # Test Loss
        total_l = 0
        counter = 0
        test_accuracy_sum = 0.0
        for i, (images, classes) in enumerate(test_loader):
            data, target = Variable(images), Variable(classes)
            output = model(data)
            total_l += loss_func(output, target).item()
            prediction = output.data.max(1)[1]
            accuracy = (float(prediction.eq(target.data).sum())/float(batch_size
))*100.0
            counter += 1
            test_accuracy_sum = test_accuracy_sum + accuracy
        test_accuracy_ave = test_accuracy_sum/float(counter)
        writer.add_scalar('test_accuracy', test_accuracy_ave, epoch+1)
        writer.add_scalar('test_loss', total_l, epoch+1)
        print(epoch, total_l, test_accuracy_ave)
```

Finally call `train` with the relevant parameters. Run the tensorboard command on your top-level logs directory to monitor training. If there is logging data from a previous run, just delete the directory for the run, and reinstantiate the `SummaryWriter` for that run. (You may want to reinstantiate the model itself if you want to clear the model parameters too).

Note : This function may take a while to complete if you're training for many epochs on a cpu. This is where it comes in handy to be running on Google Colab, or just have a GPU on hand.

In [13]:

```
#%tensorboard --logdir=logs
train(model, train_loader, test_loader, loss, optimizer, 15, writer)
```

```
0 149.9664268195629 88.16892971246007
1 124.64519420266151 89.48682108626198
2 114.79251365363598 90.0758785942492
3 108.66677984595299 90.46525559105432
4 104.81157589703798 90.67492012779553
5 101.96907778829336 90.97444089456869
6 99.64868049323559 91.12420127795527
7 98.09817900508642 91.27396166134186
8 96.89715529233217 91.39376996805112
9 95.92379009723663 91.51357827476038
10 94.64994954317808 91.5535143769968
11 93.83166018873453 91.61341853035144
12 92.94159860908985 91.56349840255591
13 92.25527238845825 91.63338658146965
14 91.96004275232553 91.67332268370608
```

Final Validation Loss: 91.96

Final Validation Accuracy: 91.67%

What is familiar about a 1-layer neural network with cross-entopy loss? Have you seen this before?

Answer: It is like SVM in terms of updating its parameters by SGD during back propagation. However, neural network can have more than 2 labels, which can predict more complex models. It is also like Logistic Regression in terms of using Cross Entropy Loss as the loss function. And the classification is by choosing the label with the highest "prediction score".

Part 4 - Two Layer Neural Net (20 points)

The thing that makes neural networks really powerful is that they are able to do complex function approximation. As we saw earlier, we can organize the computation done in neural networks into units called *layers*. In a general neural network, there is an *input layer*, and an *output layer*. These may be the same layer as they were in our previous example. When they are not the same, there are intermediate layers known as *hidden layers*. These layers receive input from other layers and send their output to other layers.

We have been dealing with a certain type of neural network known as a **fully connected** network. For our purposes, this just means that the output of the layer is just the dot product of its input `x`, its weights `w` plus a bias term `b`, all wrapped in a non-linear *activation function* `F`.

`y = F(w^T x + b).`

These non-linear activation functions are very important but where in our last neural network did we apply such a function? Implicitly we applied what's known as a **softmax activation** in order to compute cross-entropy loss https://en.wikipedia.org/wiki/Softmax_function (https://en.wikipedia.org/wiki/Softmax_function).

We'll now try to create a neural network with one hidden layer. This means that we have to come up with an activation function for the output of that hidden layer. A famous, simple but powerful activation function is the **Rectified Linear Unit (ReLU)** function defined nas `ReLU(x) = max(x,0)`. We will use this on the output of the hidden layer.

`torch.nn` has a module known as `nn.Sequential` that allows us to chain together other modules. This module implements a `forward()` function that automatically handles input-output connections etc. Check out the API at https://pytorch.org/docs/stable/nn.html#sequential (https://pytorch.org/docs/stable/nn.html#sequential).

Just like you did with the single layer model, define a class `TwoLayerModel`, a neural network with ReLU activation for the hidden layer. `nn.Sequential` may come in handy.

In [14]:

```
class TwoLayerModel(nn.Module):
    ## YOUR CODE HERE ##
    def __init__(self):
        super(TwoLayerModel, self).__init__()
        self.fc1 = nn.Linear(28*28, 100)
        self.fc2 = nn.Linear(100, 10)
    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Once again use the information provided above to do the following:

- **Instantiate a `TwoLayerModel` with the appropriate input/output/hidden layer parameters.**
- **Define a cross-entropy loss function again.**
- **Define a stochastic gradient descent optimizer based for you model's parameters. Start with a learning rate of 0.001, and adjust as necessary. You can start with the vanilla `optim.SGD` optimizer, and change it if you wish.**
- **Create a `SummaryWriter` object that will be responsible for logging our training progress into a directory called `logs/expt2` (Or whatever you wish your top-level directory to be called, just make sure the subdirectory is different from your previous SummaryWriter).**

In [15]:

```
## YOUR CODE HERE ##
model2 = TwoLayerModel()
loss2 = nn.CrossEntropyLoss()
optimizer2 = torch.optim.SGD(model2.parameters(), lr=0.01)
writer2 = SummaryWriter('logs/expt2')
```

Call `train` on your two layer neural network.

In [16]:

```
#%tensorboard --logdir=logs
train(model2, train_loader, test_loader, loss2, optimizer2, 15, writer2)
```

```
0 127.88707558810711 89.1673322683706
1 102.67858019471169 90.45527156549521
2 91.92709394544363 91.59345047923323
3 85.22648024559021 92.17252396166134
4 79.41518040373921 92.78154952076677
5 75.32206980511546 93.06110223642173
6 71.49682911112905 93.31070287539936
7 66.69918876513839 93.70007987220447
8 63.446708146482706 93.95966453674122
9 60.40530047006905 94.35902555910543
10 57.65646004118025 94.47883386581469
11 55.294440193101764 94.75838658146965
12 52.73280991613865 95.0179712460064
13 50.228772055357695 95.1976837060703
14 48.86544347368181 95.25758785942492
```

Final Validation Loss: *48.87*

Final Validation Accuracy: *95.26%*

Did your accuracy on the validation set improve with multiple layers? Why do you think this is ?

Answer: Yes, improved by about 4%. As the number of layers increase, the model is able to predict non-linear and more complex relations. For example, MNIST data require more than one layer to predict, since there are 10 labels with different features that need more complex model.

Part 5 - What is being learned at each layer? (10 points)

So what exactly are these weights that our network is learning at each layer? By conveniently picking our layer dimensions as perfect square numbers, we can try to visualize the weights learned at each layer as square images. Use the following function to do so for *all interesting layers* across your models. Feel free to modify the function as you wish.

At the very least, you must generate:

1. **The ten 28x28 weight images learned by your one layer model.**
2. **The 256 28x28 weight images learned by the hidden layer in your two-layer model.**

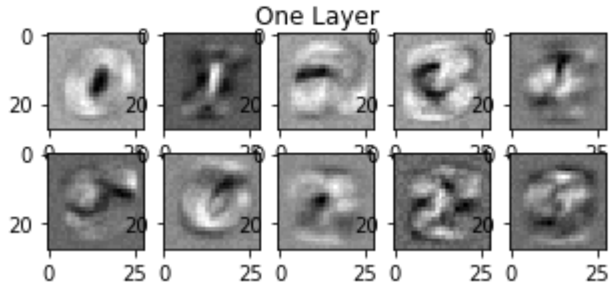
In [17]:

```
def visualize_layer_weights(model, layer_idx, num_images, image_dim, title):
    # Find number of rows and columns based on number of images
    for d in range(1,num_images):
        f = num_images/d
        if int(f)==f:
            dim1 = int(min(f,d))
            dim2 = int(max(f,d))
        if d > f:
            break
    # Plot weights as square images
    fig, ax  = plt.subplots(dim1, dim2)

    # At least 1 inch by 1 inch images
    fig.set_size_inches(dim2, dim1)
    weights = (list(model.parameters())[layer_idx])
    fig.suptitle(title)
    for i in range(dim1):
        for j in range(dim2):
            ax[i][j].imshow(weights[dim2*i+j].reshape(image_dim,image_dim).detac
h().numpy(), cmap='gray')
```

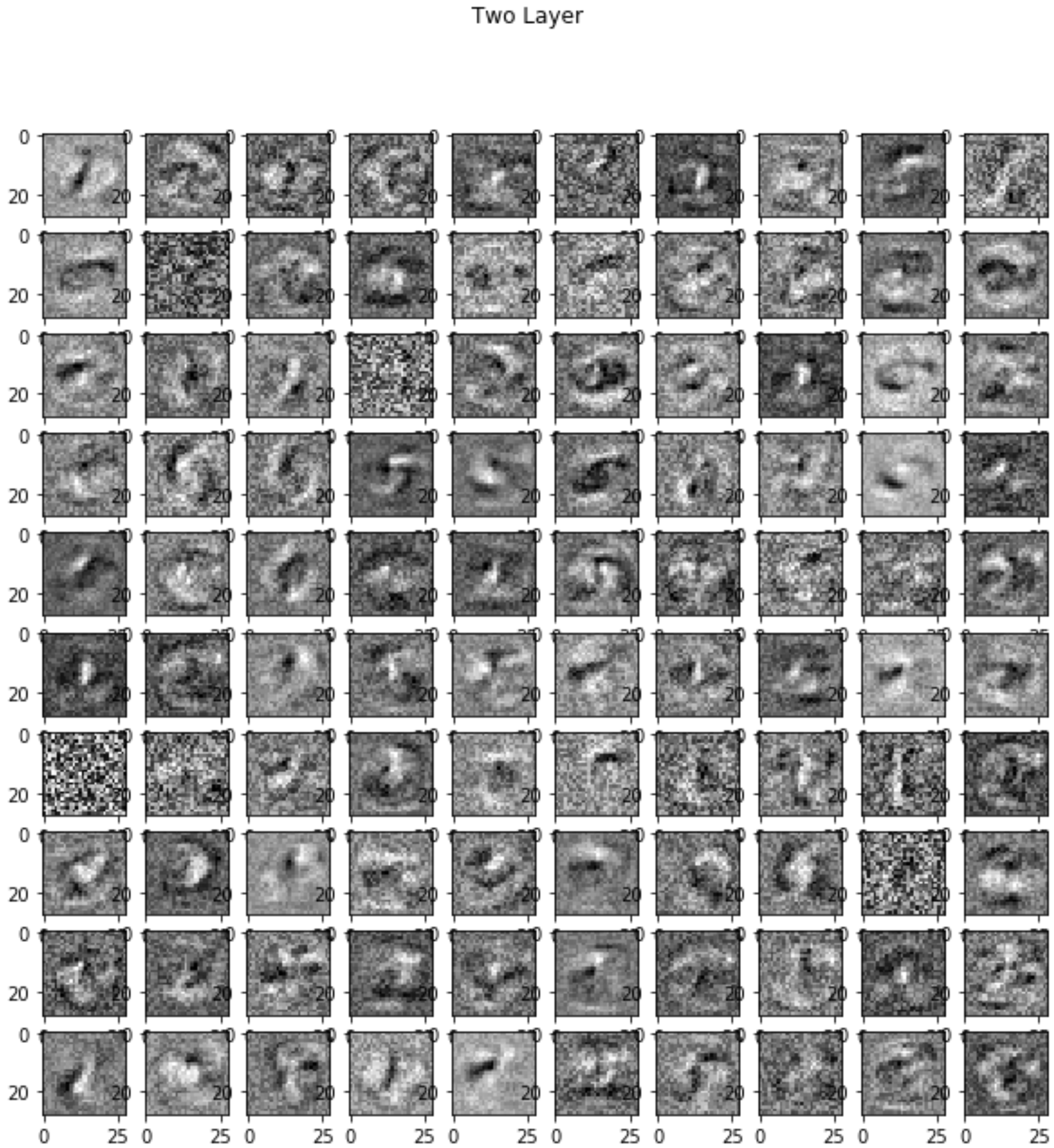
In [20]:

```
visualize_layer_weights(model, 0, 10, 28, 'One Layer')
```



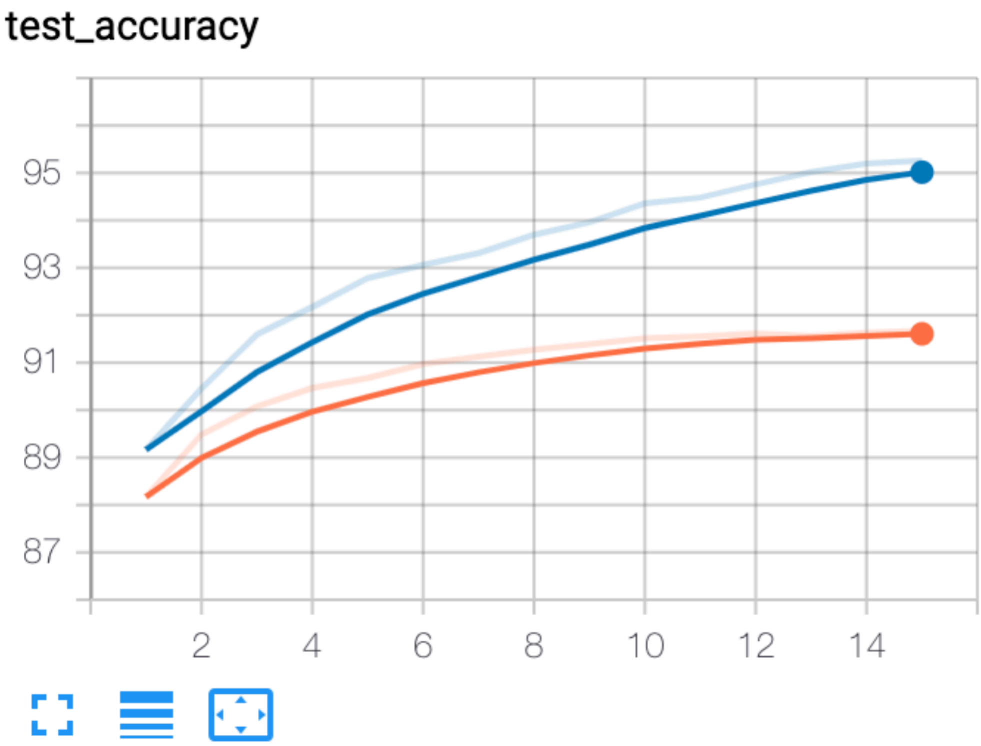
In [24]:

```
visualize_layer_weights(model2, 0, 100, 28, 'Two Layer')
```

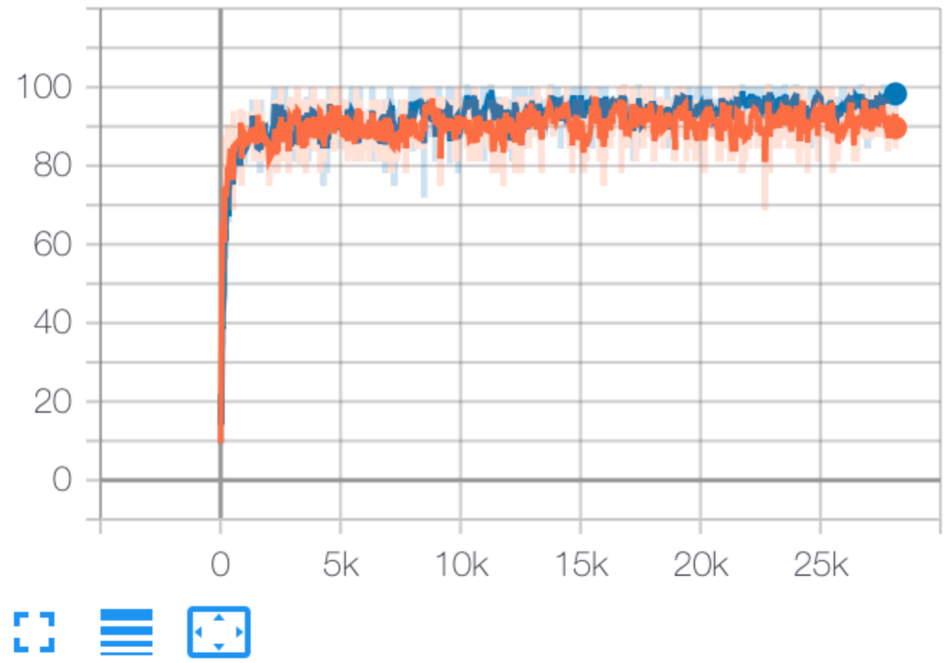


In [ ]:

Blue curve is Two Layer Model, orange curve is One Layer Model.



train_accuracy



train_loss