Keras -- MLPs on MNIST

In [0]:
```
# if you keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow" use this c
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
```

```
Using TensorFlow backend.
```

In [0]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4
# https://stackoverflow.com/a/14434334
# this function is used to update the plots for each epoch and error
def plt_dynamic(x, vy, ty, ax, colors=['b']):
    ax.plot(x, vy, 'b', label="Validation Loss")
    ax.plot(x, ty, 'r', label="Train Loss")
    plt.legend()
    plt.grid()
    fig.canvas.draw()
```

In [0]:
```
# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz (https://s3.amazona
ws.com/img-datasets/mnist.npz)
11493376/11490434 [==============================] - 1s 0us/step
```

In [0]:
```
print("Number of training examples :", X_train.shape[0], "and each image is of shape (%d, %
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d, %d
```

```
Number of training examples : 60000 and each image is of shape (28, 28)
Number of training examples : 10000 and each image is of shape (28, 28)
```

In [0]:
```
# if you observe the input shape its 2 dimensional vector
# for each image we have a (28*28) vector
# we will convert the (28*28) vector into single dimensional vector of 1 * 784

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1]*X_train.shape[2])
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1]*X_test.shape[2])
```

In [0]:
```
# after converting the input images from 3d to 2d vectors

print("Number of training examples :", X_train.shape[0], "and each image is of shape (%d)"%
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d)"%(
```

```
Number of training examples : 60000 and each image is of shape (784)
Number of training examples : 10000 and each image is of shape (784)
```

In [0]:
```
# if we observe the above matrix each cell is having a value between 0-255
# before we move to apply machine learning algorithms lets try to normalize the data
# X => (X - Xmin)/(Xmax-Xmin) = X/255

X_train = X_train/255
X_test = X_test/255
```

In [0]:
```
# here we are having a class number for each image
print("Class label of first image :", y_train[0])

# lets convert this into a 10 dimensional vector
# ex: consider an image is 5 convert it into 5 => [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
# this conversion needed for MLPs

Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)

print("After converting the output into a vector : ",Y_train[0])
```

```
Class label of first image : 5
After converting the output into a vector :  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

In [0]:
```
# https://keras.io/getting-started/sequential-model-guide/

# The Sequential model is a linear stack of layers.
# you can create a Sequential model by passing a list of layer instances to the constructor

# model = Sequential([
#     Dense(32, input_shape=(784,)),
#     Activation('relu'),
#     Dense(10),
#     Activation('softmax'),
# ])

# You can also simply add layers via the .add() method:

# model = Sequential()
# model.add(Dense(32, input_dim=784))
# model.add(Activation('relu'))

###

# https://keras.io/layers/core/

# keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_unif
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regula
# kernel_constraint=None, bias_constraint=None)

# Dense implements the operation: output = activation(dot(input, kernel) + bias) where
# activation is the element-wise activation function passed as the activation argument,
# kernel is a weights matrix created by the layer, and
# bias is a bias vector created by the layer (only applicable if use_bias is True).

# output = activation(dot(input, kernel) + bias)  => y = activation(WT. X + b)

####
```

```
# https://keras.io/activations/

# Activations can either be used through an Activation layer, or through the activation arg

# from keras.layers import Activation, Dense

# model.add(Dense(64))
# model.add(Activation('tanh'))

# This is equivalent to:
# model.add(Dense(64, activation='tanh'))

# there are many activation functions ar available ex: tanh, relu, softmax


from keras.models import Sequential
from keras.layers import Dense, Activation
from keras import initializers
```

In [0]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

MLP + ReLU + ADAM + BN + Dropout

In [0]:

```
%matplotlib inline
```

In [0]:
```
from keras.layers.normalization import BatchNormalization
from keras.layers import Dropout
```

In [0]:
```
models = {}
histories = {}
```

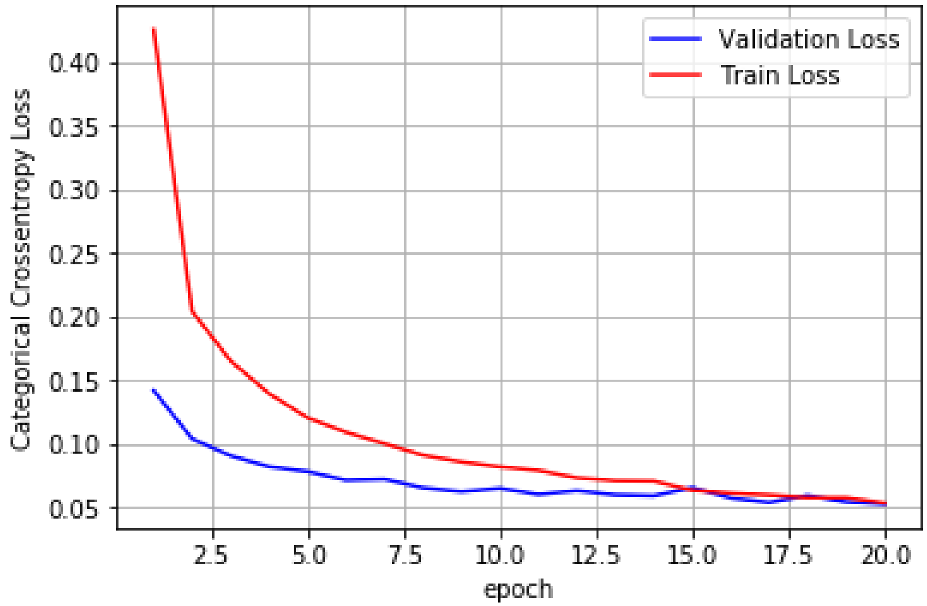
In [0]:
```
nb_epoch=20

import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(128, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['748-512-128-10'] = model_relu
histories['748-512-128-10'] = history
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
 - 8s - loss: 0.4254 - acc: 0.8715 - val_loss: 0.1421 - val_acc: 0.9559
Epoch 2/20
 - 5s - loss: 0.2042 - acc: 0.9391 - val_loss: 0.1042 - val_acc: 0.9669
Epoch 3/20
 - 5s - loss: 0.1655 - acc: 0.9500 - val_loss: 0.0909 - val_acc: 0.9707
Epoch 4/20
 - 5s - loss: 0.1397 - acc: 0.9584 - val_loss: 0.0823 - val_acc: 0.9741
Epoch 5/20
 - 5s - loss: 0.1207 - acc: 0.9627 - val_loss: 0.0785 - val_acc: 0.9763
Epoch 6/20
 - 5s - loss: 0.1094 - acc: 0.9666 - val_loss: 0.0717 - val_acc: 0.9773
Epoch 7/20
```

```
 - 5s - loss: 0.1004 - acc: 0.9690 - val_loss: 0.0725 - val_acc: 0.9778
Epoch 8/20
 - 5s - loss: 0.0913 - acc: 0.9720 - val_loss: 0.0658 - val_acc: 0.9787
Epoch 9/20
 - 5s - loss: 0.0861 - acc: 0.9733 - val_loss: 0.0627 - val_acc: 0.9804
Epoch 10/20
 - 5s - loss: 0.0822 - acc: 0.9743 - val_loss: 0.0653 - val_acc: 0.9790
Epoch 11/20
 - 5s - loss: 0.0794 - acc: 0.9756 - val_loss: 0.0605 - val_acc: 0.9817
Epoch 12/20
 - 5s - loss: 0.0736 - acc: 0.9764 - val_loss: 0.0635 - val_acc: 0.9813
Epoch 13/20
 - 5s - loss: 0.0713 - acc: 0.9769 - val_loss: 0.0602 - val_acc: 0.9813
Epoch 14/20
 - 5s - loss: 0.0710 - acc: 0.9778 - val_loss: 0.0595 - val_acc: 0.9825
Epoch 15/20
 - 5s - loss: 0.0638 - acc: 0.9797 - val_loss: 0.0659 - val_acc: 0.9803
Epoch 16/20
 - 5s - loss: 0.0615 - acc: 0.9802 - val_loss: 0.0577 - val_acc: 0.9836
Epoch 17/20
 - 5s - loss: 0.0601 - acc: 0.9808 - val_loss: 0.0543 - val_acc: 0.9829
Epoch 18/20
 - 5s - loss: 0.0580 - acc: 0.9812 - val_loss: 0.0596 - val_acc: 0.9827
Epoch 19/20
 - 5s - loss: 0.0580 - acc: 0.9817 - val_loss: 0.0546 - val_acc: 0.9846
Epoch 20/20
 - 5s - loss: 0.0538 - acc: 0.9828 - val_loss: 0.0530 - val_acc: 0.9851
```

In [0]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.05295374449652736
Test accuracy: 0.9851
```



In [69]:
```
w_after = model_relu.get_weights()
print(type(w_after))
print(len(w_after))
for w_i in w_after:
  print(w_i.shape)
```

```
<class 'list'>
14
(784, 512)
(512,)
(512,)
(512,)
(512,)
(512,)
(512, 128)
(128,)
(128,)
(128,)
(128,)
(128,)
(128, 10)
(10,)
```

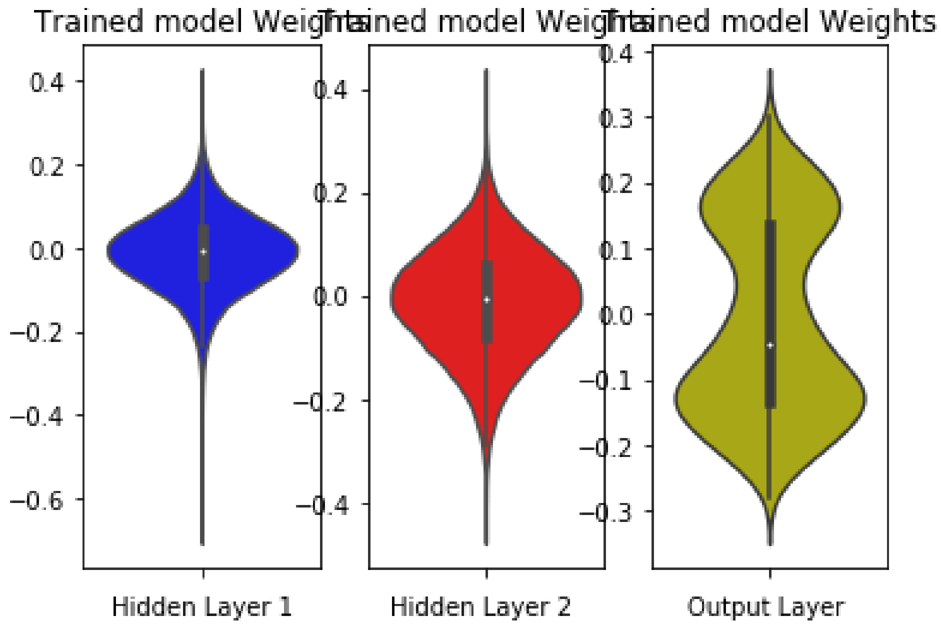
w_after is of length 14:

- indices 0, 1 => hidden_layer 1
- indices 2, 3 => BN 1
- indices 4, 5 => Dropout 1
- indices 6, 7 => Hidden layer 2
- indices 8. 9 => BN 2

- indices 10, 11 => Dropout 2
- indices 12, 13 => Output layer

So taking 0, 6, 12 indices as our main weights to plot

In [70]:
```
w_after = model_relu.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



The above model is best model with the architecture 784-512-128-10 (which is used in video lectures). The model uses BatchNormalisation and Dropout with Relu activation function and Adam optimizer. The model gives loss of 0.05295 and accuracy of 98.51 % which is pretty good.

Weight distributions are compared before hyper-parameter tuning where we discuss the differences between all the models.

Model with 2 hidden layers. Architecture: 784-256-128-10

In [0]:
```
import warnings
warnings.filterwarnings('ignore')

temp_model = Sequential()
temp_model.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_initializer=i
temp_model.add(BatchNormalization())
temp_model.add(Dropout(0.5))
temp_model.add(Dense(128, activation='relu', kernel_initializer=initializers.he_normal(seed
temp_model.add(BatchNormalization())
temp_model.add(Dropout(0.5))
temp_model.add(Dense(output_dim, activation='softmax'))

temp_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

temp_history = temp_model.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ver

models['784-256-128-10'] = temp_model
histories['784-256-128-10'] = temp_history
```

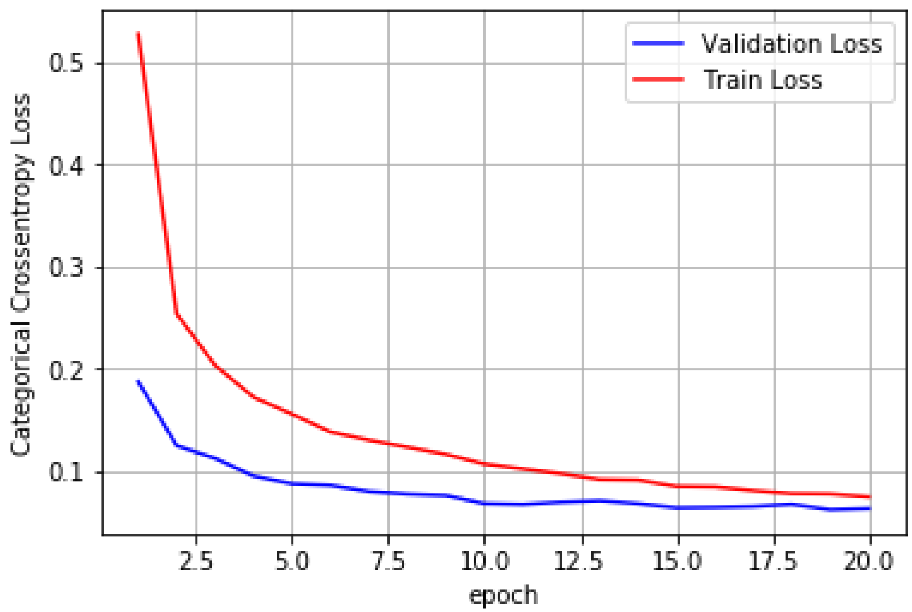
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.5272 - acc: 0.839
1 - val_loss: 0.1871 - val_acc: 0.9414
Epoch 2/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.2536 - acc: 0.924
5 - val_loss: 0.1246 - val_acc: 0.9612
Epoch 3/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.2031 - acc: 0.939
0 - val_loss: 0.1122 - val_acc: 0.9652
Epoch 4/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1721 - acc: 0.948
1 - val_loss: 0.0949 - val_acc: 0.9696
Epoch 5/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.1554 - acc: 0.952
```

```
5 - val_loss: 0.0873 - val_acc: 0.9734
Epoch 6/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.1382 - acc: 0.958
8 - val_loss: 0.0858 - val_acc: 0.9739
Epoch 7/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.1301 - acc: 0.960
0 - val_loss: 0.0798 - val_acc: 0.9757
Epoch 8/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.1231 - acc: 0.962
6 - val_loss: 0.0772 - val_acc: 0.9765
Epoch 9/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.1161 - acc: 0.964
2 - val_loss: 0.0759 - val_acc: 0.9769
Epoch 10/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.1067 - acc: 0.967
1 - val_loss: 0.0679 - val_acc: 0.9793
Epoch 11/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.1019 - acc: 0.968
4 - val_loss: 0.0671 - val_acc: 0.9785
Epoch 12/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0972 - acc: 0.969
4 - val_loss: 0.0693 - val_acc: 0.9788
Epoch 13/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0916 - acc: 0.971
2 - val_loss: 0.0707 - val_acc: 0.9793
Epoch 14/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0910 - acc: 0.971
5 - val_loss: 0.0679 - val_acc: 0.9795
Epoch 15/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0847 - acc: 0.973
9 - val_loss: 0.0638 - val_acc: 0.9818
Epoch 16/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0843 - acc: 0.973
4 - val_loss: 0.0642 - val_acc: 0.9818
Epoch 17/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0806 - acc: 0.974
```

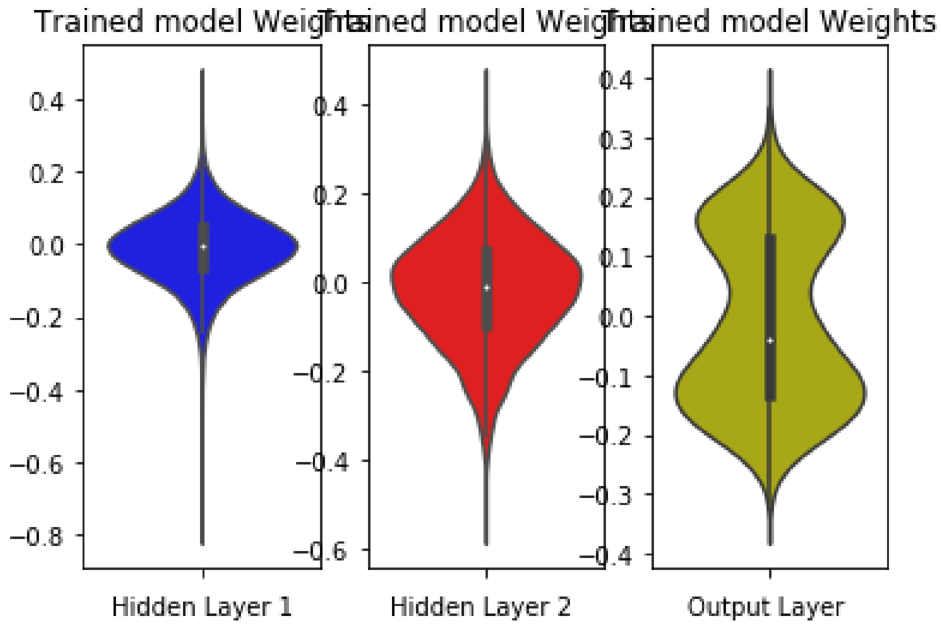
```
9 - val_loss: 0.0650 - val_acc: 0.9808
Epoch 18/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0777 - acc: 0.975
1 - val_loss: 0.0672 - val_acc: 0.9804
Epoch 19/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0774 - acc: 0.975
6 - val_loss: 0.0618 - val_acc: 0.9818
Epoch 20/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0745 - acc: 0.976
3 - val_loss: 0.0630 - val_acc: 0.9811
```

In [0]:
```
score = temp_model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = temp_history.history['val_loss']
ty = temp_history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.06300898878761219
Test accuracy: 0.9811
```



In [71]:
```
w_after = models['784-256-128-10'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



This model gives more loss and less accuracy when compared to previous model. So let us increase the nodes in both hidden layers which may increase our accuracy.

Model with 2 hidden layers. Architecture: 784-512-256-10

In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10'] = model_relu
histories['784-512-256-10'] = history
```

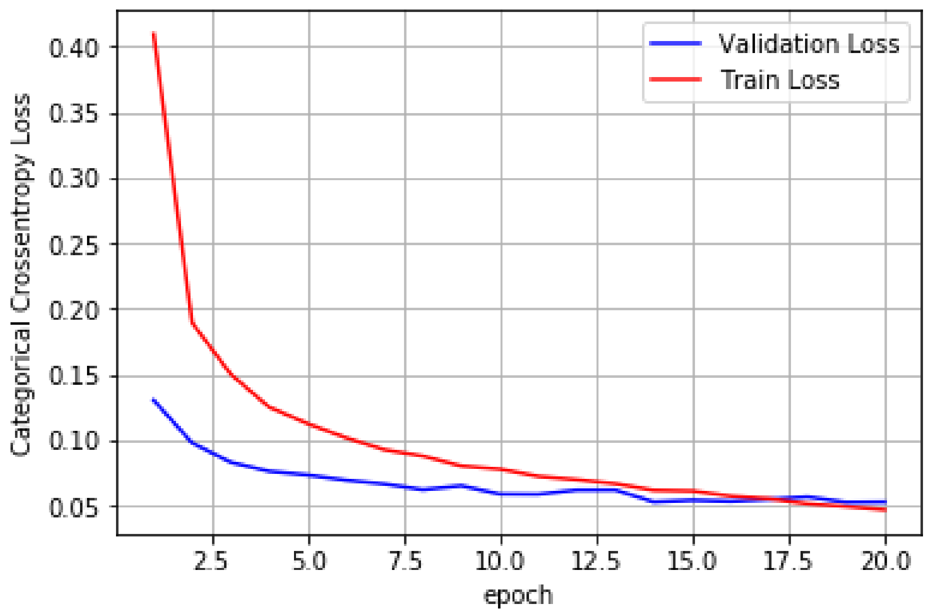
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.4092 - acc: 0.87
61 - val_loss: 0.1306 - val_acc: 0.9593
Epoch 2/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.1895 - acc: 0.941
8 - val_loss: 0.0981 - val_acc: 0.9697
Epoch 3/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1506 - acc: 0.953
7 - val_loss: 0.0834 - val_acc: 0.9735
Epoch 4/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1254 - acc: 0.960
9 - val_loss: 0.0765 - val_acc: 0.9766
Epoch 5/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.1128 - acc: 0.964
2 - val_loss: 0.0737 - val_acc: 0.9776
```

```
Epoch 6/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1021 - acc: 0.967
7 - val_loss: 0.0698 - val_acc: 0.9789
Epoch 7/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0929 - acc: 0.970
9 - val_loss: 0.0669 - val_acc: 0.9793
Epoch 8/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0881 - acc: 0.971
8 - val_loss: 0.0626 - val_acc: 0.9797
Epoch 9/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0806 - acc: 0.974
3 - val_loss: 0.0655 - val_acc: 0.9798
Epoch 10/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0781 - acc: 0.975
4 - val_loss: 0.0592 - val_acc: 0.9808
Epoch 11/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0727 - acc: 0.976
3 - val_loss: 0.0590 - val_acc: 0.9826
Epoch 12/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0702 - acc: 0.976
9 - val_loss: 0.0623 - val_acc: 0.9804
Epoch 13/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0671 - acc: 0.978
6 - val_loss: 0.0623 - val_acc: 0.9807
Epoch 14/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0621 - acc: 0.980
0 - val_loss: 0.0533 - val_acc: 0.9832
Epoch 15/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0616 - acc: 0.980
6 - val_loss: 0.0545 - val_acc: 0.9827
Epoch 16/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0577 - acc: 0.980
9 - val_loss: 0.0537 - val_acc: 0.9840
Epoch 17/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0557 - acc: 0.981
9 - val_loss: 0.0552 - val_acc: 0.9832
```

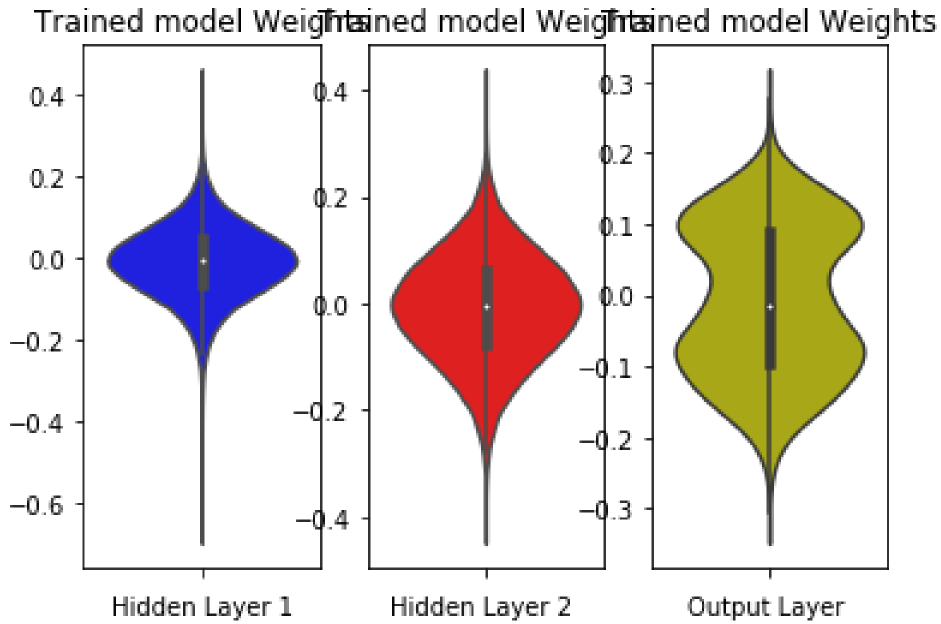
```
Epoch 18/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0519 - acc: 0.983
2 - val_loss: 0.0570 - val_acc: 0.9835
Epoch 19/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0497 - acc: 0.983
3 - val_loss: 0.0530 - val_acc: 0.9848
Epoch 20/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0472 - acc: 0.984
1 - val_loss: 0.0533 - val_acc: 0.9844
```

In [0]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.053256970743143756
Test accuracy: 0.9844
```



In [72]:
```
w_after = models['784-512-256-10'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



This model gives slightly less accuracy than the previous architecture. Let us increase the hidden layers to 3 and see the results for any improvement.

Model with 3 hidden layers. Architecture: 784-512-256-128-10

In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(128, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-128-10'] = model_relu
histories['784-512-256-128-10'] = history
```

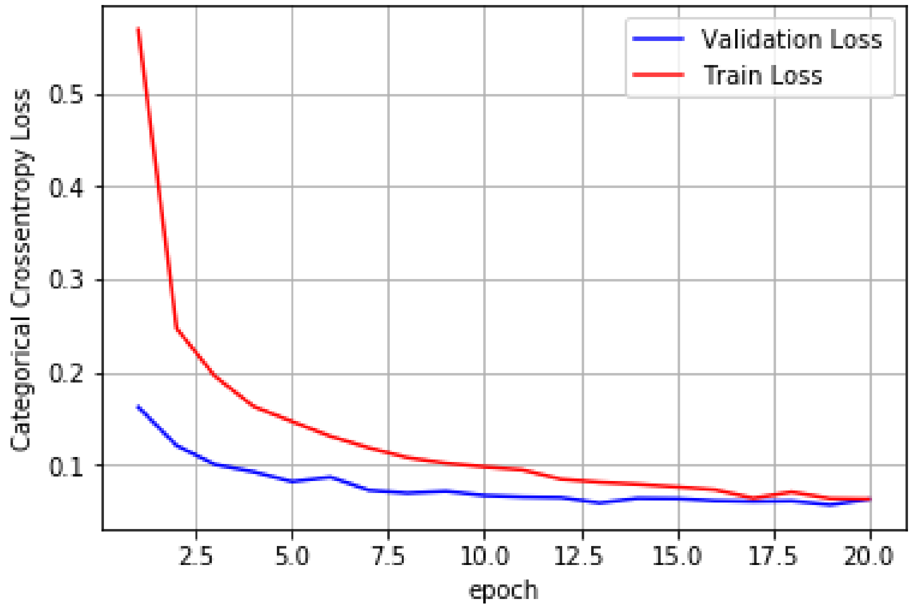
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.5687 - acc: 0.82
76 - val_loss: 0.1627 - val_acc: 0.9482
Epoch 2/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.2471 - acc: 0.92
59 - val_loss: 0.1212 - val_acc: 0.9634
Epoch 3/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.1956 - acc: 0.94
23 - val_loss: 0.1008 - val_acc: 0.9707
Epoch 4/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.1634 - acc: 0.95
15 - val_loss: 0.0931 - val_acc: 0.9707
```

```
Epoch 5/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.1472 - acc: 0.95
61 - val_loss: 0.0825 - val_acc: 0.9739
Epoch 6/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.1313 - acc: 0.96
15 - val_loss: 0.0875 - val_acc: 0.9728
Epoch 7/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.1188 - acc: 0.96
45 - val_loss: 0.0733 - val_acc: 0.9781
Epoch 8/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.1083 - acc: 0.96
71 - val_loss: 0.0701 - val_acc: 0.9781
Epoch 9/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.1023 - acc: 0.96
90 - val_loss: 0.0722 - val_acc: 0.9786
Epoch 10/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.0984 - acc: 0.97
03 - val_loss: 0.0676 - val_acc: 0.9792
Epoch 11/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0951 - acc: 0.97
14 - val_loss: 0.0660 - val_acc: 0.9815
Epoch 12/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0852 - acc: 0.97
43 - val_loss: 0.0654 - val_acc: 0.9811
Epoch 13/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0818 - acc: 0.97
51 - val_loss: 0.0595 - val_acc: 0.9824
Epoch 14/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0797 - acc: 0.97
63 - val_loss: 0.0646 - val_acc: 0.9814
Epoch 15/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0768 - acc: 0.97
66 - val_loss: 0.0642 - val_acc: 0.9820
Epoch 16/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0738 - acc: 0.97
77 - val_loss: 0.0619 - val_acc: 0.9830
```

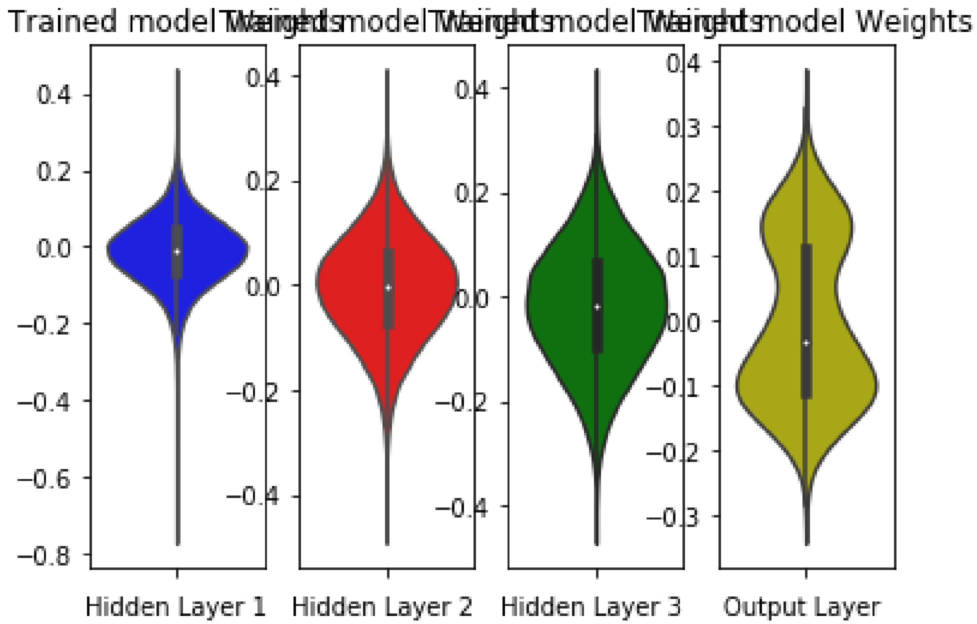
```
Epoch 17/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.0647 - acc: 0.97
97 - val_loss: 0.0611 - val_acc: 0.9832
Epoch 18/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0713 - acc: 0.97
86 - val_loss: 0.0617 - val_acc: 0.9834
Epoch 19/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.0643 - acc: 0.98
01 - val_loss: 0.0575 - val_acc: 0.9846
Epoch 20/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0638 - acc: 0.98
05 - val_loss: 0.0634 - val_acc: 0.9826
```

In [0]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.06344375926086214
Test accuracy: 0.9826
```



In [73]:
```
w_after = models['784-512-256-128-10'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
h3_w = w_after[12].flatten().reshape(-1,1)
out_w = w_after[18].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 4, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 4, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 4, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1, 4, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



Having 3 hidden layers decreased our accuracy a lot. Let us see a model with 5 hidden layers and finally choose best architecture to work further and increase the accuracy by increasing epochs

Model with 5 hidden layers. Architecture: 784-512-128-64-32-16-10

In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(128, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(64, activation='relu', kernel_initializer=initializers.he_normal(seed=
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(32, activation='relu', kernel_initializer=initializers.he_normal(seed=
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(16, activation='relu', kernel_initializer=initializers.he_normal(seed=
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-128-64-32-16-10'] = model_relu
histories['784-512-128-64-32-16-10'] = history
```

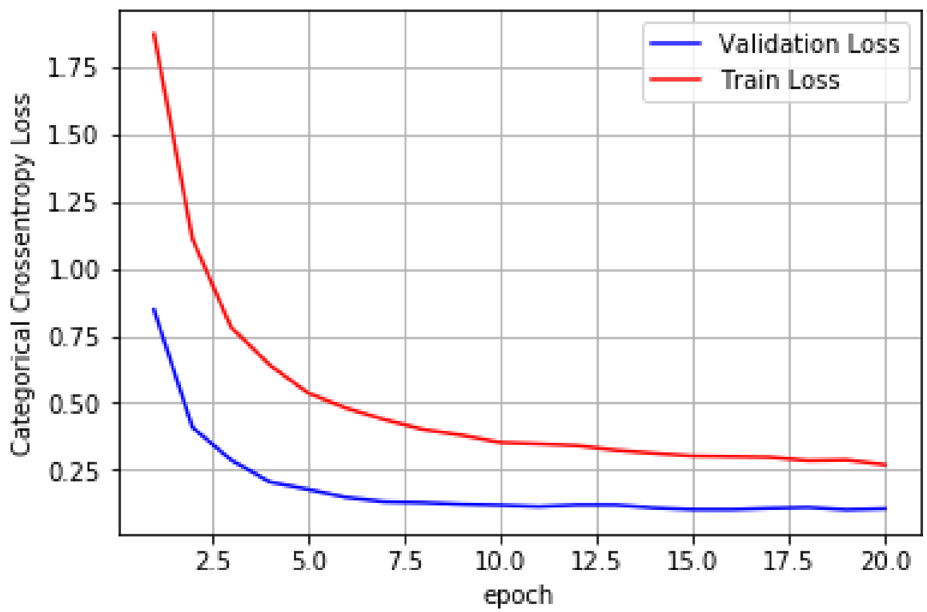
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 189us/step - loss: 1.8746 - acc: 0.3
506 - val_loss: 0.8461 - val_acc: 0.7740
Epoch 2/20
60000/60000 [==============================] - 9s 149us/step - loss: 1.1108 - acc: 0.61
65 - val_loss: 0.4079 - val_acc: 0.8689
```

```
Epoch 3/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.7822 - acc: 0.73
77 - val_loss: 0.2863 - val_acc: 0.9303
Epoch 4/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.6412 - acc: 0.79
62 - val_loss: 0.2044 - val_acc: 0.9531
Epoch 5/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.5363 - acc: 0.84
06 - val_loss: 0.1763 - val_acc: 0.9566
Epoch 6/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.4789 - acc: 0.86
15 - val_loss: 0.1462 - val_acc: 0.9638
Epoch 7/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.4361 - acc: 0.87
82 - val_loss: 0.1300 - val_acc: 0.9692
Epoch 8/20
60000/60000 [==============================] - 9s 148us/step - loss: 0.3998 - acc: 0.89
05 - val_loss: 0.1268 - val_acc: 0.9714
Epoch 9/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.3785 - acc: 0.89
78 - val_loss: 0.1207 - val_acc: 0.9720
Epoch 10/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.3513 - acc: 0.90
41 - val_loss: 0.1172 - val_acc: 0.9726
Epoch 11/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.3462 - acc: 0.90
68 - val_loss: 0.1119 - val_acc: 0.9737
Epoch 12/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.3394 - acc: 0.90
93 - val_loss: 0.1181 - val_acc: 0.9734
Epoch 13/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.3224 - acc: 0.91
19 - val_loss: 0.1180 - val_acc: 0.9741
Epoch 14/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.3108 - acc: 0.91
62 - val_loss: 0.1071 - val_acc: 0.9756
```

```
Epoch 15/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.3009 - acc: 0.91
78 - val_loss: 0.1007 - val_acc: 0.9791
Epoch 16/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.2987 - acc: 0.91
89 - val_loss: 0.1005 - val_acc: 0.9795
Epoch 17/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.2963 - acc: 0.91
92 - val_loss: 0.1057 - val_acc: 0.9784
Epoch 18/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.2836 - acc: 0.92
21 - val_loss: 0.1091 - val_acc: 0.9767
Epoch 19/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.2859 - acc: 0.92
33 - val_loss: 0.0999 - val_acc: 0.9793
Epoch 20/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.2684 - acc: 0.92
49 - val_loss: 0.1040 - val_acc: 0.9783
```

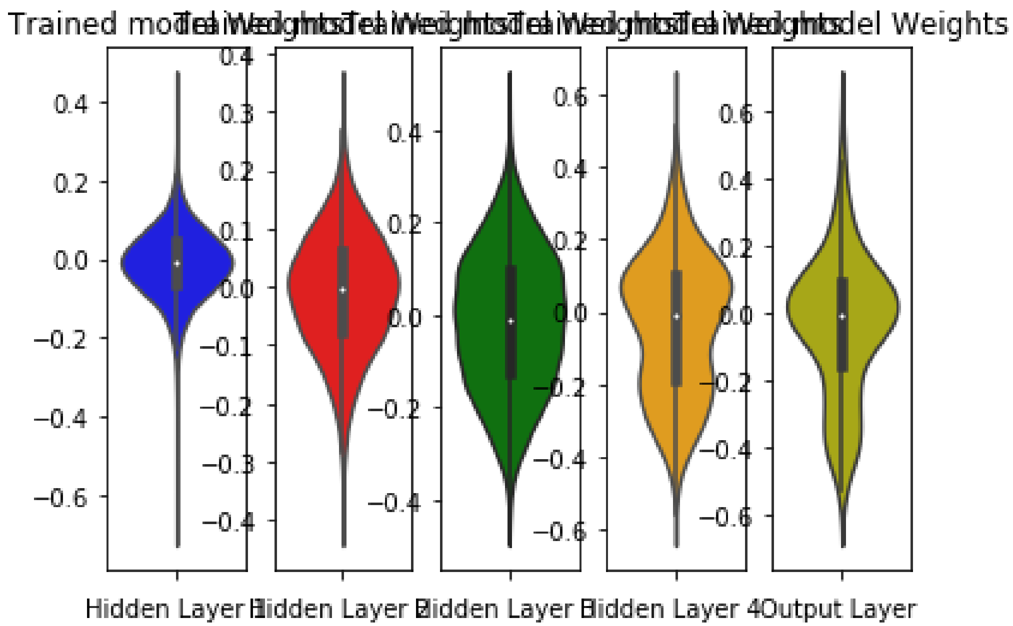
In [0]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.10397892101514153
Test accuracy: 0.9783
```



In [78]:
```
w_after = models['784-512-128-64-32-16-10'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
h3_w = w_after[12].flatten().reshape(-1,1)
h4_w = w_after[18].flatten().reshape(-1,1)
out_w = w_after[24].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 5, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 5, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 5, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1, 5, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='orange')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(1, 5, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
```

```
plt.show()
```



Having 5 hidden layers decreased our accuracy a lot (least accuracy by far). the loss seems to be stagnating and not reducing much after some epochs. having less number of layers seems to be best as our problem is not complex enough to do deeper networks. Taking 2-hidden layers and trying to increase the accuracy of our model.

Model with 2 hidden layers. Architecture: 784-512-256-10 with more epochs

In [0]:
```
nb_epoch = 50
```

In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_50'] = model_relu
histories['784-512-256-10_50'] = history
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 7s 121us/step - loss: 0.4028 - acc: 0.8785
- val_loss: 0.1324 - val_acc: 0.9589
Epoch 2/50
60000/60000 [==============================] - 5s 86us/step - loss: 0.1933 - acc: 0.9410
- val_loss: 0.1044 - val_acc: 0.9679
Epoch 3/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.1490 - acc: 0.9535
- val_loss: 0.0875 - val_acc: 0.9726
Epoch 4/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.1281 - acc: 0.9610
- val_loss: 0.0757 - val_acc: 0.9756
Epoch 5/50
60000/60000 [==============================] - 5s 86us/step - loss: 0.1116 - acc: 0.9649
```

```
- val_loss: 0.0733 - val_acc: 0.9763
Epoch 6/50
60000/60000 [==============================] - 5s 87us/step - loss: 0.1004 - acc: 0.9688
- val_loss: 0.0741 - val_acc: 0.9765
Epoch 7/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0968 - acc: 0.9699
- val_loss: 0.0648 - val_acc: 0.9789
Epoch 8/50
60000/60000 [==============================] - 5s 87us/step - loss: 0.0882 - acc: 0.9721
- val_loss: 0.0634 - val_acc: 0.9814
Epoch 9/50
60000/60000 [==============================] - 5s 86us/step - loss: 0.0824 - acc: 0.9737
- val_loss: 0.0626 - val_acc: 0.9808
Epoch 10/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0755 - acc: 0.9760
- val_loss: 0.0571 - val_acc: 0.9826
Epoch 11/50
60000/60000 [==============================] - 5s 85us/step - loss: 0.0722 - acc: 0.9769
- val_loss: 0.0579 - val_acc: 0.9812
Epoch 12/50
60000/60000 [==============================] - 5s 87us/step - loss: 0.0663 - acc: 0.9790
- val_loss: 0.0620 - val_acc: 0.9811
Epoch 13/50
60000/60000 [==============================] - 5s 85us/step - loss: 0.0646 - acc: 0.9790
- val_loss: 0.0618 - val_acc: 0.9803
Epoch 14/50
60000/60000 [==============================] - 5s 86us/step - loss: 0.0610 - acc: 0.9795
- val_loss: 0.0568 - val_acc: 0.9828
Epoch 15/50
60000/60000 [==============================] - 5s 85us/step - loss: 0.0605 - acc: 0.9811
- val_loss: 0.0545 - val_acc: 0.9827
Epoch 16/50
60000/60000 [==============================] - 5s 85us/step - loss: 0.0587 - acc: 0.9809
- val_loss: 0.0561 - val_acc: 0.9834
Epoch 17/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0571 - acc: 0.9817
```

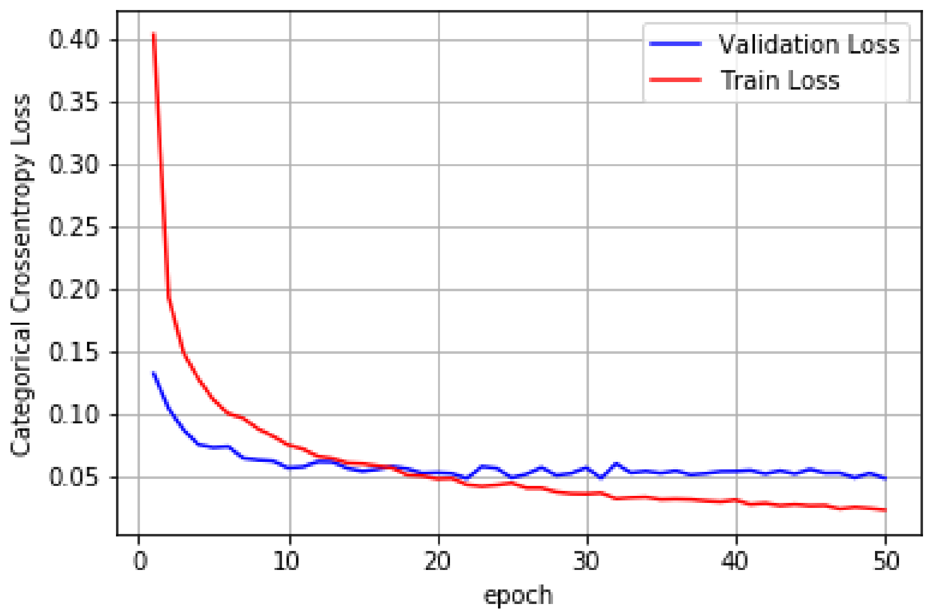
```
- val_loss: 0.0582 - val_acc: 0.9830
Epoch 18/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0515 - acc: 0.9832
- val_loss: 0.0565 - val_acc: 0.9836
Epoch 19/50
60000/60000 [==============================] - 5s 86us/step - loss: 0.0508 - acc: 0.9830
- val_loss: 0.0529 - val_acc: 0.9833
Epoch 20/50
60000/60000 [==============================] - 5s 85us/step - loss: 0.0484 - acc: 0.9839
- val_loss: 0.0533 - val_acc: 0.9841
Epoch 21/50
60000/60000 [==============================] - 5s 86us/step - loss: 0.0488 - acc: 0.9842
- val_loss: 0.0527 - val_acc: 0.9848
Epoch 22/50
60000/60000 [==============================] - 5s 87us/step - loss: 0.0437 - acc: 0.9852
- val_loss: 0.0487 - val_acc: 0.9858
Epoch 23/50
60000/60000 [==============================] - 5s 85us/step - loss: 0.0424 - acc: 0.9861
- val_loss: 0.0582 - val_acc: 0.9827
Epoch 24/50
60000/60000 [==============================] - 5s 87us/step - loss: 0.0435 - acc: 0.9858
- val_loss: 0.0568 - val_acc: 0.9843
Epoch 25/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0451 - acc: 0.9849
- val_loss: 0.0492 - val_acc: 0.9852
Epoch 26/50
60000/60000 [==============================] - 6s 100us/step - loss: 0.0411 - acc: 0.9864
- val_loss: 0.0522 - val_acc: 0.9836
Epoch 27/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0411 - acc: 0.9863
- val_loss: 0.0575 - val_acc: 0.9843
Epoch 28/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0379 - acc: 0.9873
- val_loss: 0.0512 - val_acc: 0.9855
Epoch 29/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0367 - acc: 0.9878
```

```
- val_loss: 0.0528 - val_acc: 0.9847
Epoch 30/50
60000/60000 [==============================] - 5s 90us/step - loss: 0.0364 - acc: 0.9879
- val_loss: 0.0574 - val_acc: 0.9837
Epoch 31/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0371 - acc: 0.9876
- val_loss: 0.0489 - val_acc: 0.9860
Epoch 32/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0325 - acc: 0.9895
- val_loss: 0.0605 - val_acc: 0.9834
Epoch 33/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0334 - acc: 0.9884
- val_loss: 0.0532 - val_acc: 0.9849
Epoch 34/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0338 - acc: 0.9891
- val_loss: 0.0545 - val_acc: 0.9841
Epoch 35/50
60000/60000 [==============================] - 5s 91us/step - loss: 0.0319 - acc: 0.9893
- val_loss: 0.0529 - val_acc: 0.9852
Epoch 36/50
60000/60000 [==============================] - 5s 90us/step - loss: 0.0323 - acc: 0.9892
- val_loss: 0.0549 - val_acc: 0.9850
Epoch 37/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0319 - acc: 0.9893
- val_loss: 0.0516 - val_acc: 0.9856
Epoch 38/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0308 - acc: 0.9894
- val_loss: 0.0527 - val_acc: 0.9862
Epoch 39/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0301 - acc: 0.9903
- val_loss: 0.0543 - val_acc: 0.9853
Epoch 40/50
60000/60000 [==============================] - 5s 88us/step - loss: 0.0315 - acc: 0.9897
- val_loss: 0.0544 - val_acc: 0.9844
Epoch 41/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0280 - acc: 0.9909
```

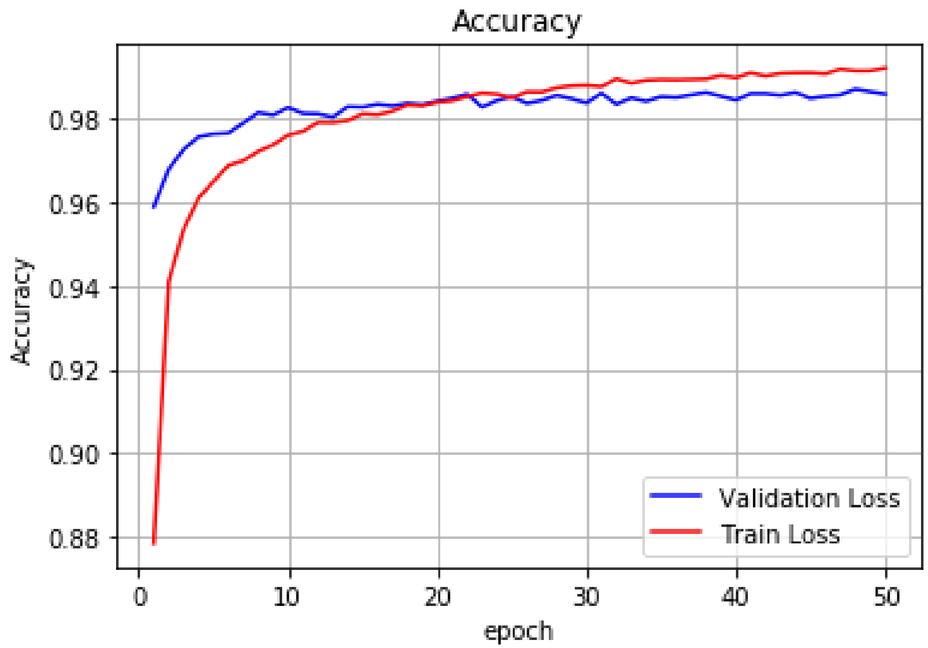
```
- val_loss: 0.0552 - val_acc: 0.9859
Epoch 42/50
60000/60000 [==============================] - 5s 91us/step - loss: 0.0289 - acc: 0.9901
- val_loss: 0.0524 - val_acc: 0.9859
Epoch 43/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0271 - acc: 0.9908
- val_loss: 0.0549 - val_acc: 0.9855
Epoch 44/50
60000/60000 [==============================] - 5s 87us/step - loss: 0.0280 - acc: 0.9909
- val_loss: 0.0524 - val_acc: 0.9862
Epoch 45/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0270 - acc: 0.9909
- val_loss: 0.0561 - val_acc: 0.9848
Epoch 46/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0272 - acc: 0.9907
- val_loss: 0.0530 - val_acc: 0.9853
Epoch 47/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0244 - acc: 0.9918
- val_loss: 0.0531 - val_acc: 0.9856
Epoch 48/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0258 - acc: 0.9914
- val_loss: 0.0493 - val_acc: 0.9870
Epoch 49/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0249 - acc: 0.9914
- val_loss: 0.0529 - val_acc: 0.9864
Epoch 50/50
60000/60000 [==============================] - 5s 89us/step - loss: 0.0236 - acc: 0.9920
- val_loss: 0.0490 - val_acc: 0.9858
```

In [0]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

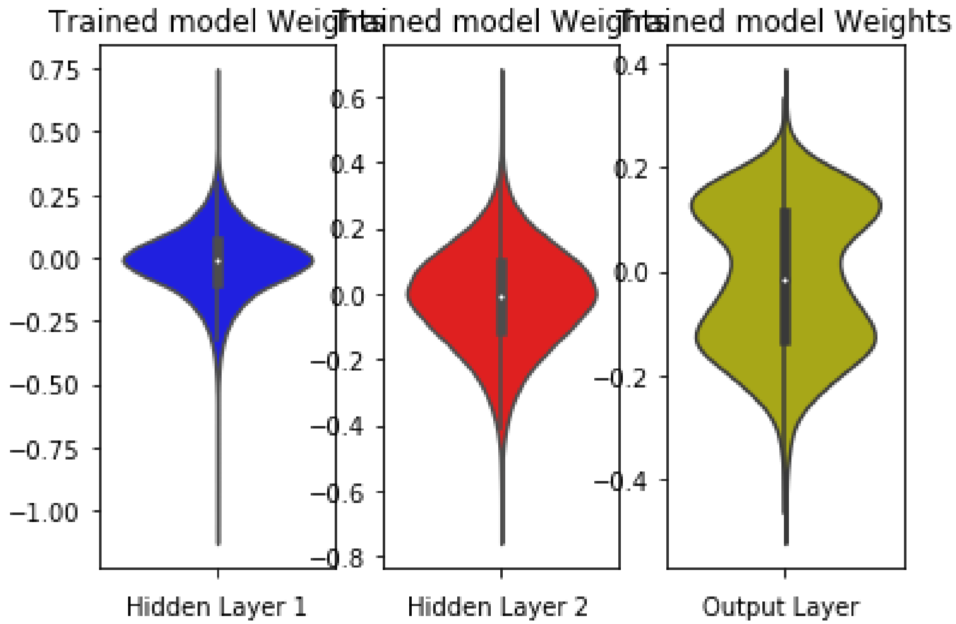
```
Test score: 0.04899904340120793
Test accuracy: 0.9858
```



In [0]:
```
fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Accuracy')
ax.set_title('Accuracy')

x = list(range(1,nb_epoch+1))

vy = history.history['val_acc']
ty = history.history['acc']
plt_dynamic(x, vy, ty, ax)
```



In [81]:
```
w_after = models['784-512-256-10_50'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



This model gives slightly more accuracy. The model's validation accuracy is not increasing much from epoch 30 but Train accuracy is increasing which might indicate overfitting. So this model architecture's best accuracy seems to be around 98.58 %. Now we try for 3 hidden layers.

Model with 3 hidden layers. Architecture: 784-512-256-128-10 with more epochs

In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(128, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-128-10_50'] = model_relu
histories['784-512-256-128-10_50'] = history
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 9s 154us/step - loss: 0.5805 - acc: 0.8245
- val_loss: 0.1593 - val_acc: 0.9519
Epoch 2/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.2457 - acc: 0.9274
- val_loss: 0.1116 - val_acc: 0.9651
Epoch 3/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.1938 - acc: 0.9415
- val_loss: 0.0982 - val_acc: 0.9687
Epoch 4/50
60000/60000 [==============================] - 7s 116us/step - loss: 0.1657 - acc: 0.9516
```

```
- val_loss: 0.0865 - val_acc: 0.9733
Epoch 5/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.1412 - acc: 0.9580
- val_loss: 0.0807 - val_acc: 0.9752
Epoch 6/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.1299 - acc: 0.9616
- val_loss: 0.0776 - val_acc: 0.9766
Epoch 7/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.1179 - acc: 0.9653
- val_loss: 0.0733 - val_acc: 0.9790
Epoch 8/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.1103 - acc: 0.9666
- val_loss: 0.0736 - val_acc: 0.9792
Epoch 9/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.1028 - acc: 0.9695
- val_loss: 0.0727 - val_acc: 0.9788
Epoch 10/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0989 - acc: 0.9701
- val_loss: 0.0670 - val_acc: 0.9797
Epoch 11/50
60000/60000 [==============================] - 7s 115us/step - loss: 0.0950 - acc: 0.9715
- val_loss: 0.0627 - val_acc: 0.9808
Epoch 12/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0867 - acc: 0.9732
- val_loss: 0.0693 - val_acc: 0.9803
Epoch 13/50
60000/60000 [==============================] - 7s 112us/step - loss: 0.0832 - acc: 0.9749
- val_loss: 0.0625 - val_acc: 0.9827
Epoch 14/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0792 - acc: 0.9763
- val_loss: 0.0627 - val_acc: 0.9829
Epoch 15/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0744 - acc: 0.9772
- val_loss: 0.0589 - val_acc: 0.9838
Epoch 16/50
60000/60000 [==============================] - 7s 115us/step - loss: 0.0757 - acc: 0.9776
```

```
- val_loss: 0.0673 - val_acc: 0.9814
Epoch 17/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0705 - acc: 0.9787
- val_loss: 0.0601 - val_acc: 0.9831
Epoch 18/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0708 - acc: 0.9788
- val_loss: 0.0571 - val_acc: 0.9846
Epoch 19/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0651 - acc: 0.9806
- val_loss: 0.0566 - val_acc: 0.9834
Epoch 20/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0654 - acc: 0.9799
- val_loss: 0.0597 - val_acc: 0.9842
Epoch 21/50
60000/60000 [==============================] - 7s 112us/step - loss: 0.0614 - acc: 0.9812
- val_loss: 0.0572 - val_acc: 0.9843
Epoch 22/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0606 - acc: 0.9813
- val_loss: 0.0507 - val_acc: 0.9852
Epoch 23/50
60000/60000 [==============================] - 7s 112us/step - loss: 0.0561 - acc: 0.9828
- val_loss: 0.0669 - val_acc: 0.9820
Epoch 24/50
60000/60000 [==============================] - 7s 111us/step - loss: 0.0572 - acc: 0.9823
- val_loss: 0.0574 - val_acc: 0.9841
Epoch 25/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0544 - acc: 0.9832
- val_loss: 0.0567 - val_acc: 0.9846
Epoch 26/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0516 - acc: 0.9835
- val_loss: 0.0550 - val_acc: 0.9847
Epoch 27/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0536 - acc: 0.9838
- val_loss: 0.0564 - val_acc: 0.9860
Epoch 28/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0505 - acc: 0.9844
```

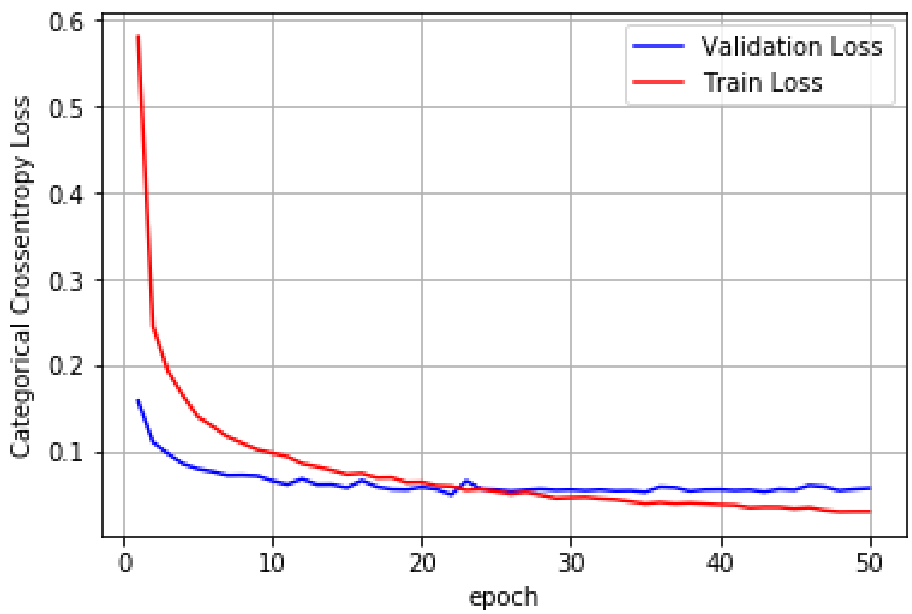
```
- val_loss: 0.0575 - val_acc: 0.9844
Epoch 29/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0467 - acc: 0.9854
- val_loss: 0.0562 - val_acc: 0.9850
Epoch 30/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0474 - acc: 0.9851
- val_loss: 0.0567 - val_acc: 0.9849
Epoch 31/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0480 - acc: 0.9852
- val_loss: 0.0557 - val_acc: 0.9849
Epoch 32/50
60000/60000 [==============================] - 7s 112us/step - loss: 0.0464 - acc: 0.9859
- val_loss: 0.0566 - val_acc: 0.9849
Epoch 33/50
60000/60000 [==============================] - 7s 116us/step - loss: 0.0453 - acc: 0.9859
- val_loss: 0.0552 - val_acc: 0.9855
Epoch 34/50
60000/60000 [==============================] - 7s 115us/step - loss: 0.0429 - acc: 0.9865
- val_loss: 0.0555 - val_acc: 0.9849
Epoch 35/50
60000/60000 [==============================] - 7s 111us/step - loss: 0.0404 - acc: 0.9876
- val_loss: 0.0538 - val_acc: 0.9859
Epoch 36/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0419 - acc: 0.9872
- val_loss: 0.0600 - val_acc: 0.9841
Epoch 37/50
60000/60000 [==============================] - 7s 115us/step - loss: 0.0405 - acc: 0.9876
- val_loss: 0.0592 - val_acc: 0.9848
Epoch 38/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0412 - acc: 0.9872
- val_loss: 0.0549 - val_acc: 0.9857
Epoch 39/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0403 - acc: 0.9878
- val_loss: 0.0566 - val_acc: 0.9855
Epoch 40/50
60000/60000 [==============================] - 7s 114us/step - loss: 0.0393 - acc: 0.9879
```

```
- val_loss: 0.0569 - val_acc: 0.9852
Epoch 41/50
60000/60000 [==============================] - 7s 112us/step - loss: 0.0388 - acc: 0.9880
- val_loss: 0.0558 - val_acc: 0.9853
Epoch 42/50
60000/60000 [==============================] - 7s 112us/step - loss: 0.0359 - acc: 0.9888
- val_loss: 0.0564 - val_acc: 0.9863
Epoch 43/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0366 - acc: 0.9887
- val_loss: 0.0541 - val_acc: 0.9856
Epoch 44/50
60000/60000 [==============================] - 7s 116us/step - loss: 0.0365 - acc: 0.9887
- val_loss: 0.0573 - val_acc: 0.9859
Epoch 45/50
60000/60000 [==============================] - 7s 116us/step - loss: 0.0347 - acc: 0.9890
- val_loss: 0.0562 - val_acc: 0.9860
Epoch 46/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0359 - acc: 0.9887
- val_loss: 0.0617 - val_acc: 0.9837
Epoch 47/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0328 - acc: 0.9896
- val_loss: 0.0602 - val_acc: 0.9861
Epoch 48/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0311 - acc: 0.9901
- val_loss: 0.0555 - val_acc: 0.9864
Epoch 49/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0313 - acc: 0.9899
- val_loss: 0.0571 - val_acc: 0.9858
Epoch 50/50
60000/60000 [==============================] - 7s 113us/step - loss: 0.0314 - acc: 0.9897
- val_loss: 0.0584 - val_acc: 0.9861
```

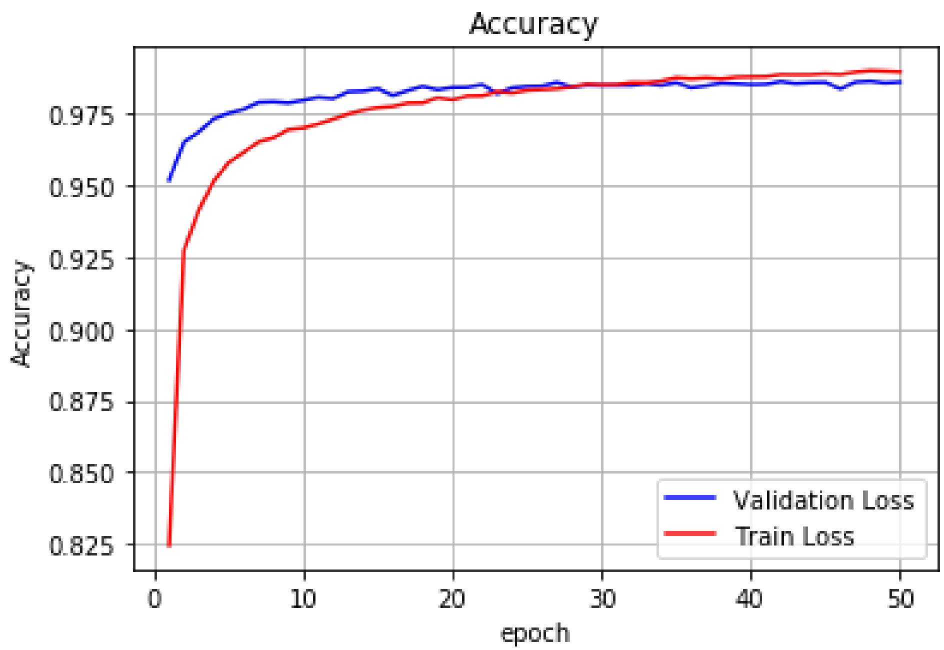
In [0]:

```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

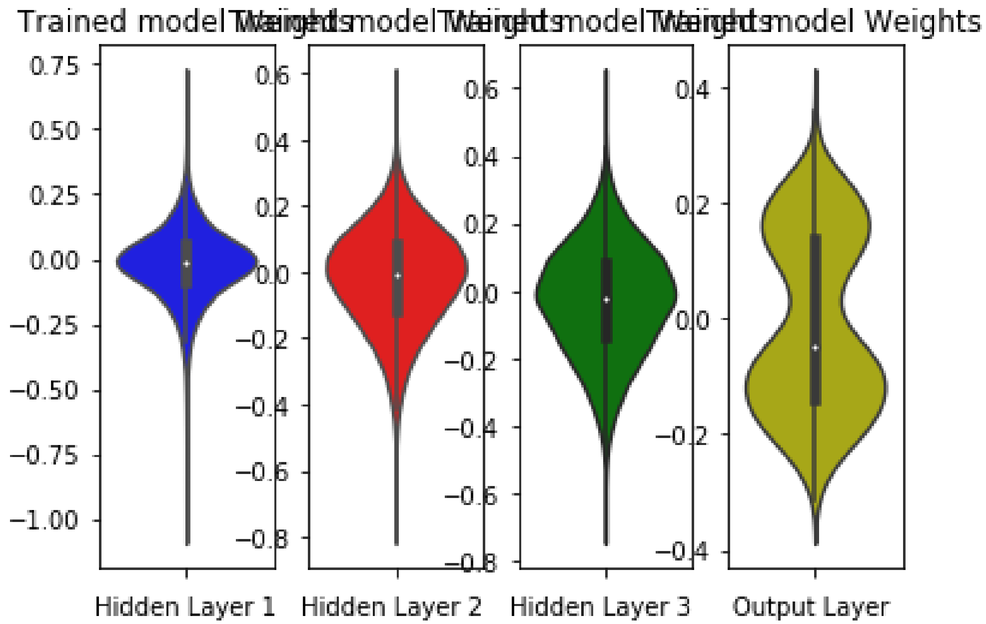
```
Test score: 0.058391568849549366
Test accuracy: 0.9861
```



In [0]:
```
fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Accuracy')
ax.set_title('Accuracy')

x = list(range(1,nb_epoch+1))

vy = history.history['val_acc']
ty = history.history['acc']
plt_dynamic(x, vy, ty, ax)
```



In [80]:
```
w_after = models['784-512-256-128-10_50'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
h3_w = w_after[12].flatten().reshape(-1,1)
out_w = w_after[18].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 4, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 4, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 4, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1, 4, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```

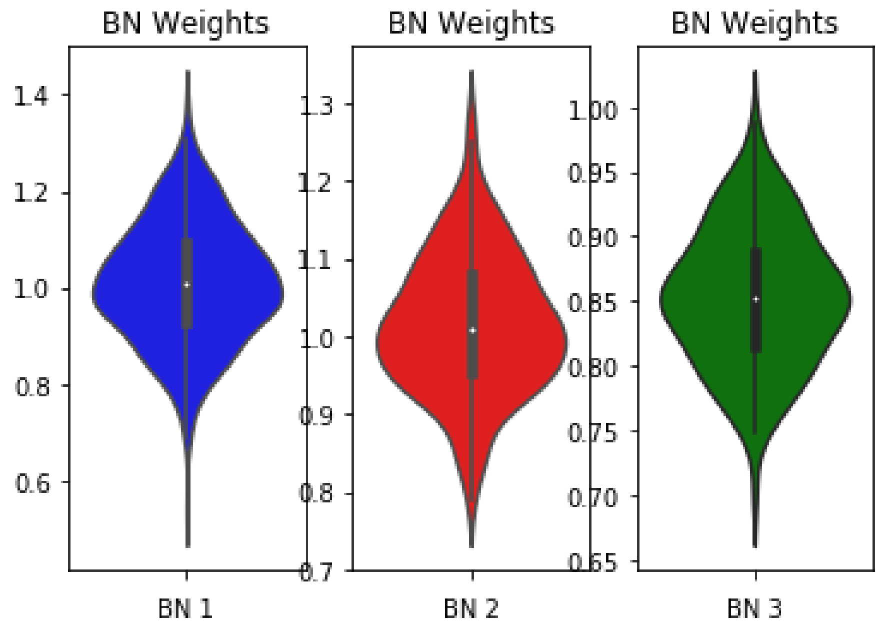


This model did give good accuracy and loss. But let us take models with 2 hidden layers and do some hyper-parameter tuning on those models to get optimal parameters. Not taking 3 hidden layers as tuning becomes difficult

All the weight distributions pretty much look same for all plots (Except for 5 hidden layers model). All weights of hidden layers are distributed around 0 and are similar to normally distributed except for last output layer.
This is my analogy of ditributions of weights in output layer. The distribution have two peaks one with negative values and one with positive values. As the output layer has softmax activation and all our outputs are vectors with one 1 and remaining zeros (ex: [0, 1, 0, 0, 0, ...]). The weights have to be such that the values of X.W should have some negative values so the e^(X.W) is less value and some weights should be positive so e^(X.W) is high to get a single confident value in the ouputs.

But in one of comments mentioned that weights of output layer are distributed around 1 after Batch normalization. Plotting weights of BN to make sure if any of them are mis-interpreted as output layer weights.

In [83]:
```
w_after = models['784-512-256-128-10_50'].get_weights()

h1_w = w_after[2].flatten().reshape(-1,1)
h2_w = w_after[8].flatten().reshape(-1,1)
h3_w = w_after[14].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weigths after training")
plt.subplot(1, 3, 1)
plt.title("BN Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('BN 1')

plt.subplot(1, 3, 2)
plt.title("BN Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('BN 2')

plt.subplot(1, 3, 3)
plt.title("BN Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('BN 3')
plt.show()
```



So Batch normalization weights are distributed around 1 and hidden layer weights are distributed around 0 which makes sense as our hidden layer weights are initially distributed as gaussian. And Weights for batch normalization are used to scale the input data and add some bias to them. As the weights indicate scaling they are distributed around 1.

Hyper-parameter tuning with hyperas

After hyper-parameter tuning section we compare models without batch normalization and without dropout and also changing other parameters like optimizer, activation function etc..

The below code is taken from the blog written by Shashank Ramesh
URL: [https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b (https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b)](https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b)

In [0]:
```
from sklearn.model_selection import train_test_split
```

In [0]:
```
!pip install hyperas

from hyperopt import Trials, STATUS_OK, tpe
from hyperas import optim
from hyperas.distributions import choice, uniform
```

In [0]:
```
def data():
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, rand
    X_train = X_train.reshape(48000, 784)
    X_val = X_val.reshape(12000, 784)
    X_train = X_train.astype('float32')
    X_val = X_val.astype('float32')
    X_train /= 255
    X_val /= 255
    nb_classes = 10
    Y_train = np_utils.to_categorical(y_train, nb_classes)
    Y_val = np_utils.to_categorical(y_val, nb_classes)
    return X_train, Y_train, X_val, Y_val
```

Only tuning the required parameters which are number of nuerons in hidden layers, Dropout rate, learning rate of Adam and batch size for training. Not tuning activation functions and optimizers as 'Relu' and 'Adam' are better choices than others.

In [0]:
```
def model(X_train, Y_train, X_val, Y_val):

    model = Sequential()
    model.add(Dense({{choice([128, 256, 512])}}, input_shape=(784,)))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Dropout({{uniform(0, 1)}}))
    model.add(Dense({{choice([128, 256, 512])}}))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Dropout({{uniform(0, 1)}}))

    model.add(Dense(10))
    model.add(Activation('softmax'))
    optim = keras.optimizers.Adam(lr={{choice([10**-3, 10**-2, 10**-1])}})

    model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer=optim)
    model.fit(X_train, Y_train,
              batch_size={{choice([128,256,512])}},
              nb_epoch=35,
              verbose=2,
              validation_data=(X_val, Y_val))
    score, acc = model.evaluate(X_val, Y_val, verbose=0)
    print('Test accuracy:', acc)
    return {'loss': -acc, 'status': STATUS_OK, 'model': model}
```

Before running the hyper-parameter tuning using hyperas, As I am using google colab I need to save the notebook into drive so that hyperas locates it correctly. Code took from blog written by Nils Schlüter
URL: [https://towardsdatascience.com/keras-hyperparameter-tuning-in-google-colab-using-hyperas-624fa4bbf673 (https://towardsdatascience.com/keras-hyperparameter-tuning-in-google-colab-using-hyperas-624fa4bbf673)](https://towardsdatascience.com/keras-hyperparameter-tuning-in-google-colab-using-hyperas-624fa4bbf673)

In [0]:
```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials

# Authenticate and create the PyDrive client.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

# Copy/download the file
fid = drive.ListFile({'q':"title='Keras_Mnist_ilmnarayana.ipynb'"}).GetList()[0]['id']
f = drive.CreateFile({'id': fid})
f.GetContentFile('Keras_Mnist_ilmnarayana.ipynb')
```

In [0]:
```
import keras
```

In [0]:
```
X_train, Y_train, X_val, Y_val = data()
best_run, best_model = optim.minimize(model=model,
                                      data=data,
                                      algo=tpe.suggest,
                                      max_evals=35,
                                      trials=Trials(),
                                      notebook_name='Keras_Mnist_ilmnarayana')
```

```
>>> Imports:
#coding=utf-8

try:
    from keras.utils import np_utils
except:
    pass

try:
    from keras.datasets import mnist
except:
    pass

try:
    import seaborn as sns
except:
    pass

try:
```

In [0]:
```
print(best_run)
print(best_model)
```

```
{'Dense': 2, 'Dense_1': 2, 'Dropout': 0.6518168887306186, 'Dropout_1': 0.1003113452504375
3, 'batch_size': 2, 'lr': 0}
<keras.engine.sequential.Sequential object at 0x7fd6354d5518>
```

Activation and optimizer are not tuned as they are taken as 'Relu' and 'Adam' respectively which are good enough for performance of the models. Below is the best model after hyper-parameter tuning which is trained on 50 epochs to see the accuracy.

In [0]:
```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=initia
model.add(BatchNormalization())
model.add(Dropout(0.652))
model.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=initia
model.add(BatchNormalization())
model.add(Dropout(0.1))
model.add(Dense(output_dim, activation='softmax'))

optim = keras.optimizers.Adam(lr=10**-3)

model.compile(optimizer=optim, loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(X_train, Y_train, batch_size=128, epochs=nb_epoch, verbose=1, validatio

score, acc = model.evaluate(X_test, Y_test, verbose=0)

models['784-512-512-10_50'] = model
histories['784-512-512-10_50'] = history

print('Test accuracy:', acc)
```

```
WARNING:tensorflow:Large dropout rate: 0.652 (>0.5). In TensorFlow 2.x, dropout() uses
dropout rate instead of keep_prob. Please ensure that this is intended.
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 8s 135us/step - loss: 0.3453 - acc: 0.89
45 - val_loss: 0.1337 - val_acc: 0.9588
Epoch 2/50
60000/60000 [==============================] - 6s 96us/step - loss: 0.1900 - acc: 0.941
3 - val_loss: 0.0947 - val_acc: 0.9693
Epoch 3/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.1511 - acc: 0.953
4 - val_loss: 0.0826 - val_acc: 0.9742
Epoch 4/50
```

```
60000/60000 [==============================] - 6s 96us/step - loss: 0.1326 - acc: 0.958
2 - val_loss: 0.0758 - val_acc: 0.9775
Epoch 5/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.1191 - acc: 0.962
5 - val_loss: 0.0681 - val_acc: 0.9785
Epoch 6/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.1076 - acc: 0.966
4 - val_loss: 0.0678 - val_acc: 0.9786
Epoch 7/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.1028 - acc: 0.967
5 - val_loss: 0.0680 - val_acc: 0.9789
Epoch 8/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0933 - acc: 0.969
7 - val_loss: 0.0605 - val_acc: 0.9805
Epoch 9/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0868 - acc: 0.972
4 - val_loss: 0.0581 - val_acc: 0.9818
Epoch 10/50
60000/60000 [==============================] - 6s 95us/step - loss: 0.0829 - acc: 0.973
6 - val_loss: 0.0592 - val_acc: 0.9821
Epoch 11/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0799 - acc: 0.974
2 - val_loss: 0.0595 - val_acc: 0.9815
Epoch 12/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0786 - acc: 0.974
3 - val_loss: 0.0578 - val_acc: 0.9827
Epoch 13/50
60000/60000 [==============================] - 6s 95us/step - loss: 0.0734 - acc: 0.975
8 - val_loss: 0.0551 - val_acc: 0.9833
Epoch 14/50
60000/60000 [==============================] - 6s 96us/step - loss: 0.0716 - acc: 0.976
8 - val_loss: 0.0557 - val_acc: 0.9829
Epoch 15/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0705 - acc: 0.977
2 - val_loss: 0.0539 - val_acc: 0.9850
Epoch 16/50
```

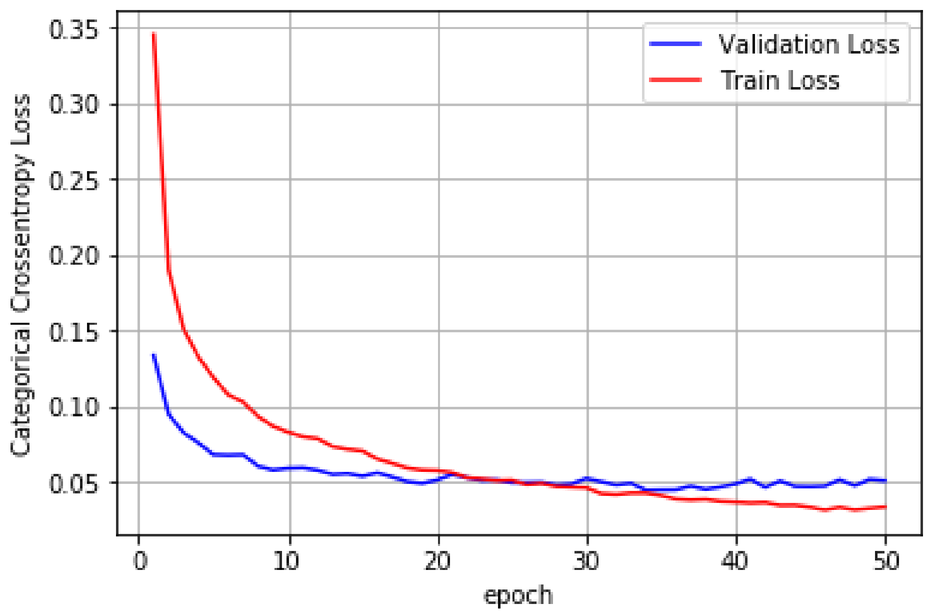
```
60000/60000 [==============================] - 6s 93us/step - loss: 0.0652 - acc: 0.978
8 - val_loss: 0.0562 - val_acc: 0.9828
Epoch 17/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0623 - acc: 0.979
6 - val_loss: 0.0533 - val_acc: 0.9843
Epoch 18/50
60000/60000 [==============================] - 6s 95us/step - loss: 0.0593 - acc: 0.980
5 - val_loss: 0.0504 - val_acc: 0.9837
Epoch 19/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0580 - acc: 0.980
7 - val_loss: 0.0492 - val_acc: 0.9828
Epoch 20/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0577 - acc: 0.981
3 - val_loss: 0.0514 - val_acc: 0.9834
Epoch 21/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0564 - acc: 0.982
3 - val_loss: 0.0550 - val_acc: 0.9837
Epoch 22/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0529 - acc: 0.982
3 - val_loss: 0.0529 - val_acc: 0.9827
Epoch 23/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0521 - acc: 0.982
4 - val_loss: 0.0512 - val_acc: 0.9844
Epoch 24/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0510 - acc: 0.982
7 - val_loss: 0.0515 - val_acc: 0.9856
Epoch 25/50
60000/60000 [==============================] - 5s 91us/step - loss: 0.0513 - acc: 0.983
3 - val_loss: 0.0497 - val_acc: 0.9857
Epoch 26/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0483 - acc: 0.984
2 - val_loss: 0.0498 - val_acc: 0.9845
Epoch 27/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0491 - acc: 0.983
4 - val_loss: 0.0499 - val_acc: 0.9849
Epoch 28/50
```

```
60000/60000 [==============================] - 6s 93us/step - loss: 0.0471 - acc: 0.984
4 - val_loss: 0.0481 - val_acc: 0.9850
Epoch 29/50
60000/60000 [==============================] - 5s 92us/step - loss: 0.0466 - acc: 0.984
0 - val_loss: 0.0487 - val_acc: 0.9868
Epoch 30/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0463 - acc: 0.984
3 - val_loss: 0.0524 - val_acc: 0.9844
Epoch 31/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0424 - acc: 0.986
3 - val_loss: 0.0503 - val_acc: 0.9857
Epoch 32/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0419 - acc: 0.985
8 - val_loss: 0.0481 - val_acc: 0.9864
Epoch 33/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0428 - acc: 0.985
8 - val_loss: 0.0492 - val_acc: 0.9855
Epoch 34/50
60000/60000 [==============================] - 6s 96us/step - loss: 0.0428 - acc: 0.985
3 - val_loss: 0.0447 - val_acc: 0.9866
Epoch 35/50
60000/60000 [==============================] - 6s 95us/step - loss: 0.0412 - acc: 0.985
9 - val_loss: 0.0448 - val_acc: 0.9862
Epoch 36/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0388 - acc: 0.987
4 - val_loss: 0.0449 - val_acc: 0.9861
Epoch 37/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0382 - acc: 0.986
9 - val_loss: 0.0473 - val_acc: 0.9865
Epoch 38/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0386 - acc: 0.987
1 - val_loss: 0.0452 - val_acc: 0.9863
Epoch 39/50
60000/60000 [==============================] - 6s 95us/step - loss: 0.0373 - acc: 0.987
8 - val_loss: 0.0467 - val_acc: 0.9868
Epoch 40/50
```

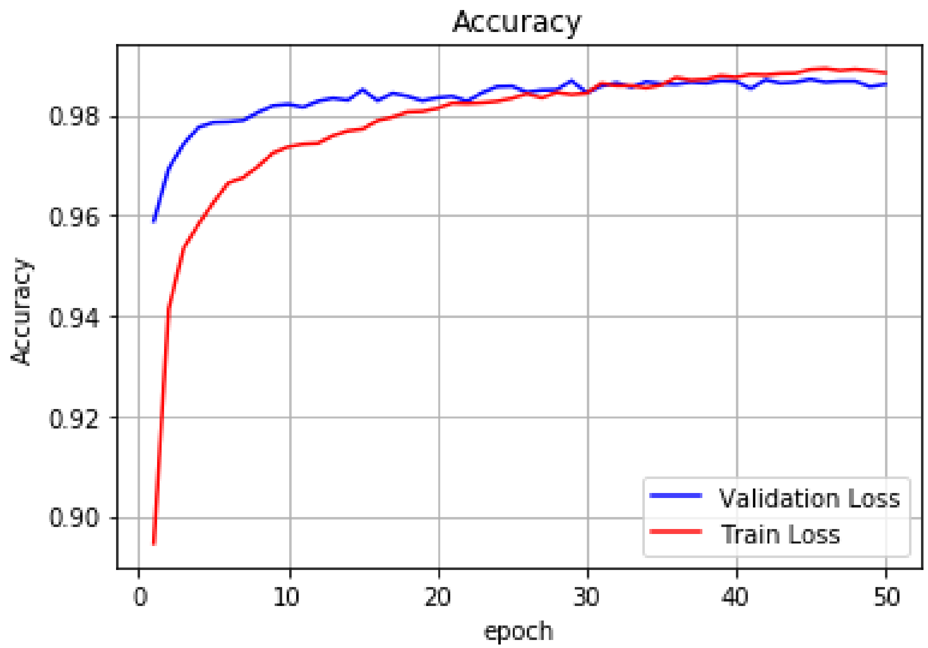
```
60000/60000 [==============================] - 6s 93us/step - loss: 0.0368 - acc: 0.987
5 - val_loss: 0.0488 - val_acc: 0.9867
Epoch 41/50
60000/60000 [==============================] - 6s 94us/step - loss: 0.0362 - acc: 0.988
1 - val_loss: 0.0519 - val_acc: 0.9852
Epoch 42/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0365 - acc: 0.987
9 - val_loss: 0.0465 - val_acc: 0.9870
Epoch 43/50
60000/60000 [==============================] - 5s 92us/step - loss: 0.0347 - acc: 0.988
2 - val_loss: 0.0508 - val_acc: 0.9863
Epoch 44/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0348 - acc: 0.988
3 - val_loss: 0.0472 - val_acc: 0.9865
Epoch 45/50
60000/60000 [==============================] - 5s 91us/step - loss: 0.0335 - acc: 0.989
0 - val_loss: 0.0470 - val_acc: 0.9871
Epoch 46/50
60000/60000 [==============================] - 6s 92us/step - loss: 0.0316 - acc: 0.989
2 - val_loss: 0.0473 - val_acc: 0.9865
Epoch 47/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0334 - acc: 0.988
8 - val_loss: 0.0516 - val_acc: 0.9867
Epoch 48/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0316 - acc: 0.989
1 - val_loss: 0.0476 - val_acc: 0.9867
Epoch 49/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0327 - acc: 0.988
8 - val_loss: 0.0518 - val_acc: 0.9856
Epoch 50/50
60000/60000 [==============================] - 6s 93us/step - loss: 0.0336 - acc: 0.988
4 - val_loss: 0.0510 - val_acc: 0.9861
Test accuracy: 0.9861
```

In [0]:
```
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

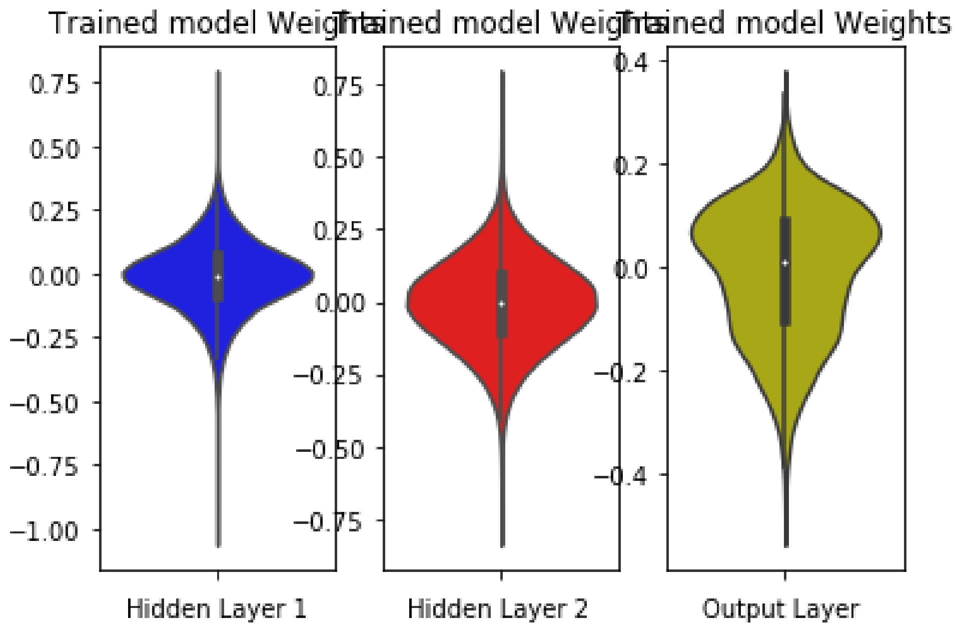
```
Test score: 0.050988600608311 75
Test accuracy: 0.9861
```



In [0]:
```
fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Accuracy')
ax.set_title('Accuracy')

x = list(range(1,nb_epoch+1))

vy = history.history['val_acc']
ty = history.history['acc']
plt_dynamic(x, vy, ty, ax)
```



In [82]:
```
w_after = models['784-512-512-10_50'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



This model is best so far with 98.61% accuracy and 0.051 loss. Let us print some mis-classified images to see where it went wrong.

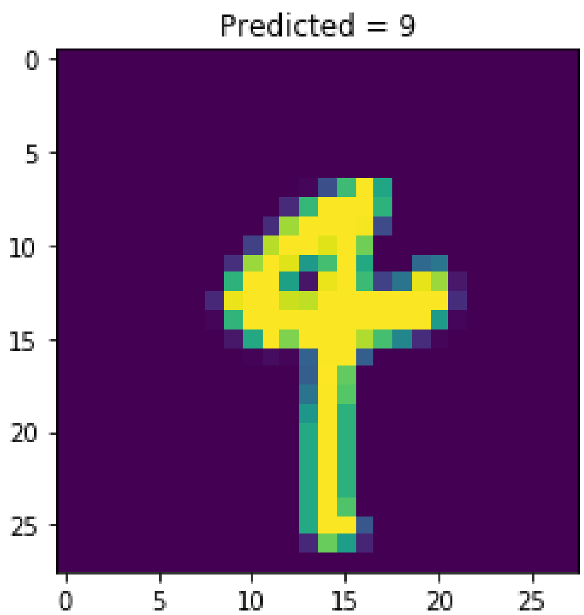
In [0]:
```
y_pred = model.predict(X_test)
print(len(y_pred))
```

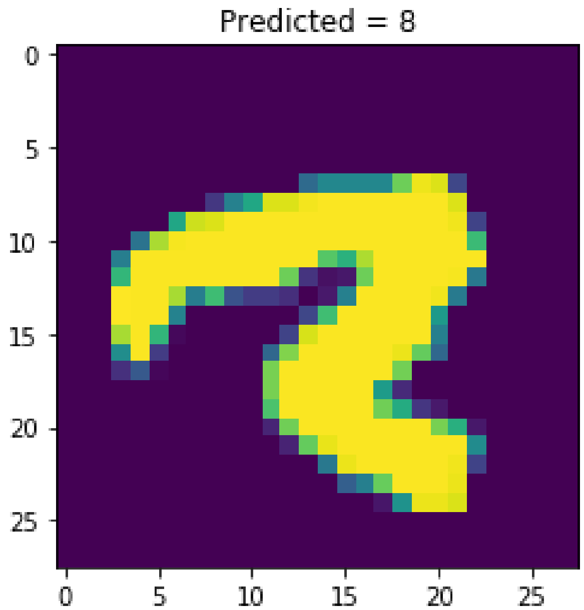
```
10000
```

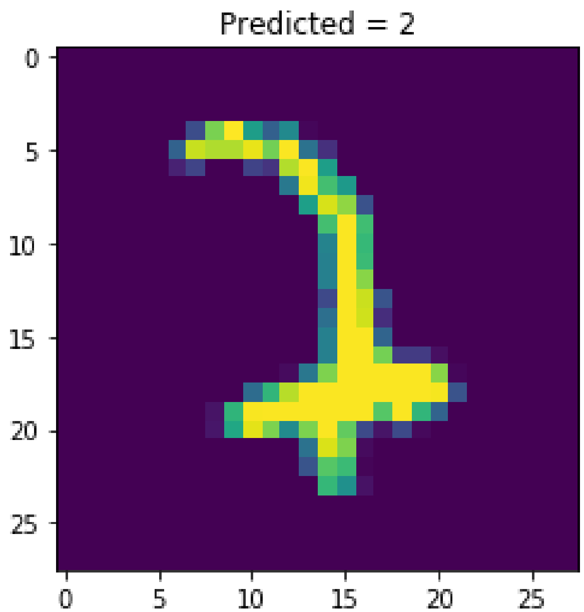
In [0]:
```
ind_arr = np.array(list(range(10000)))
miss_ind = [np.argmax(y_pred[i]) != y_test[i] for i in range(10000)]
ind_arr = ind_arr[miss_ind]
```

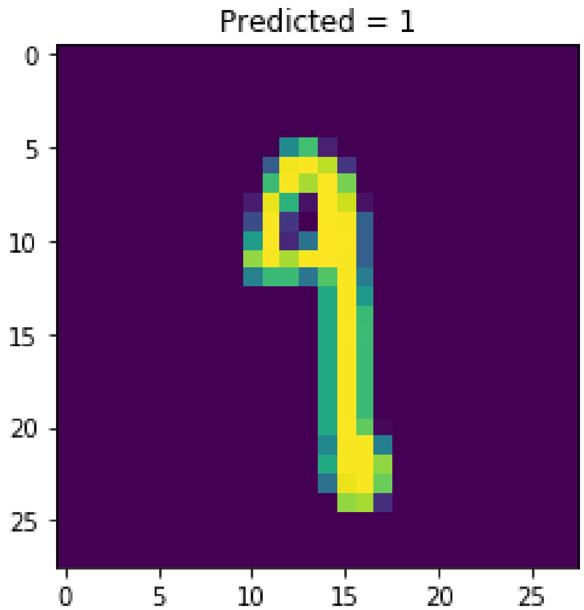
Titles of each image has its predicted values.

In [0]:
```
import random
for _ in range(10):
  ind = random.choice(ind_arr)
  plt.imshow((X_test[ind].reshape(28, 28))*255)
  plt.title(f"Predicted = {np.argmax(y_pred[ind])}")
  plt.show()
```



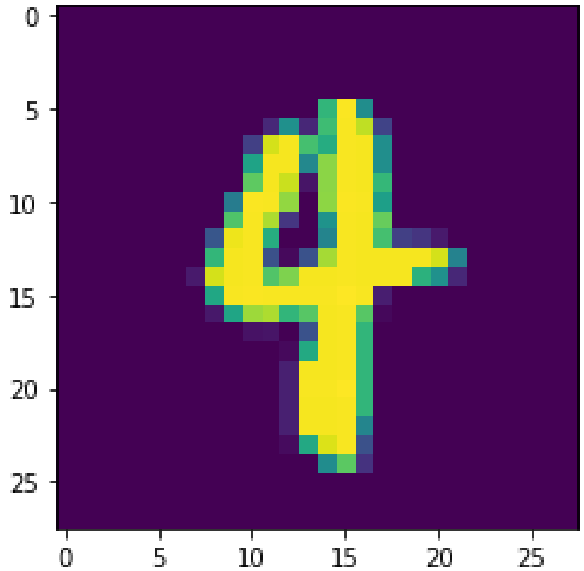

Predicted = 8
0
5
10
15
20
25

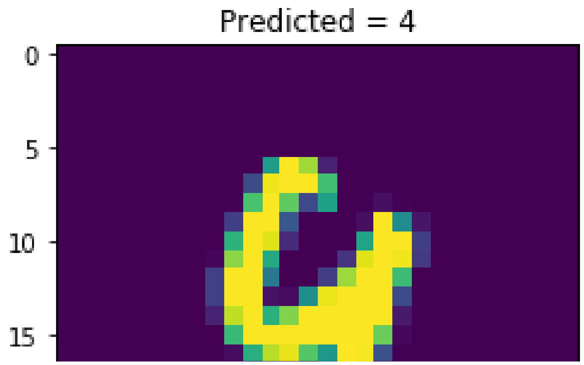

Predicted = 2
0
5
10
15
20
25

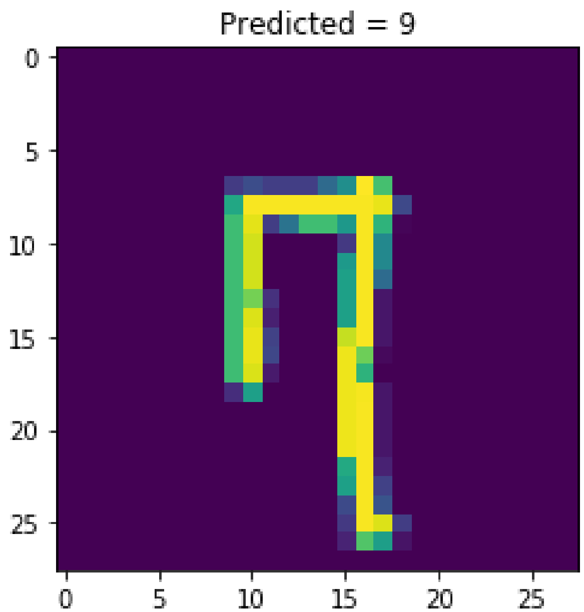

Predicted = 1
0
5
10
15
20
25
0
5
10
15
20
25

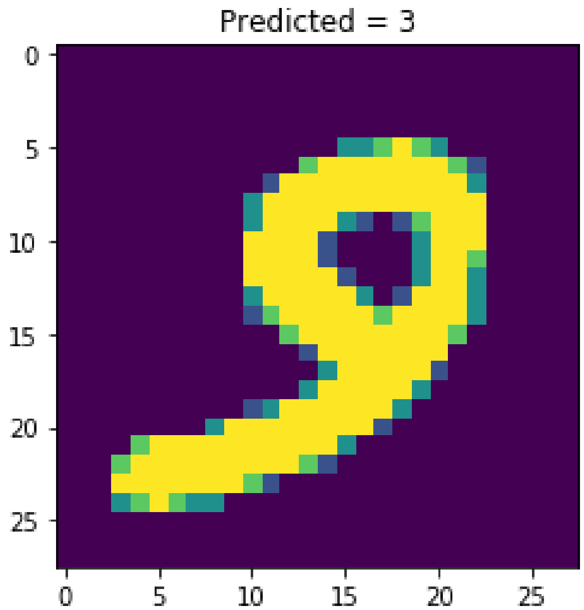
Predicted = 4

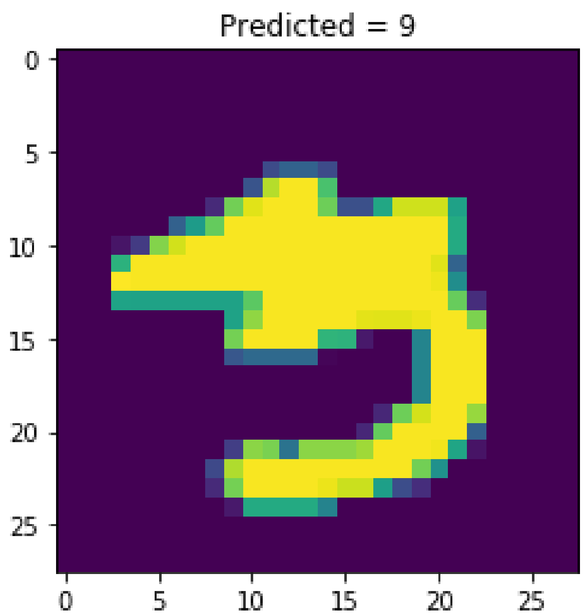
Predicted = 9




Predicted = 4
0
5
10
15

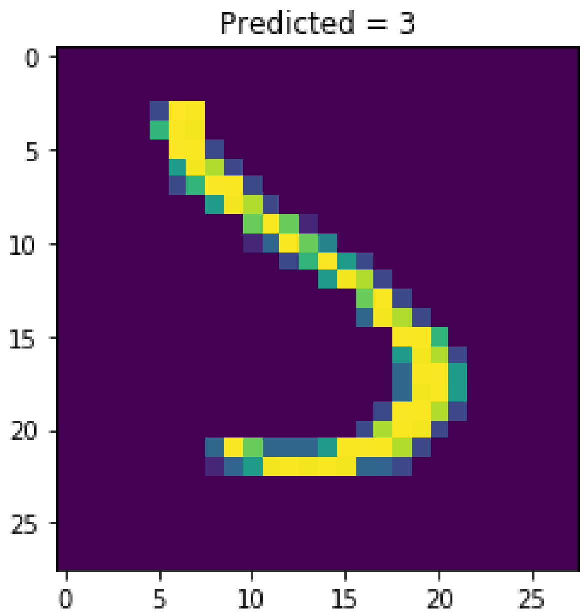

Predicted = 9
0
5
10
15
20
25
0 5 10 15 20 25

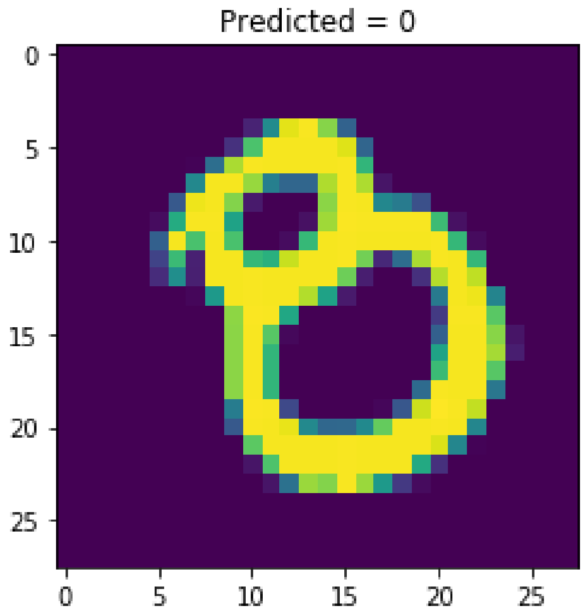


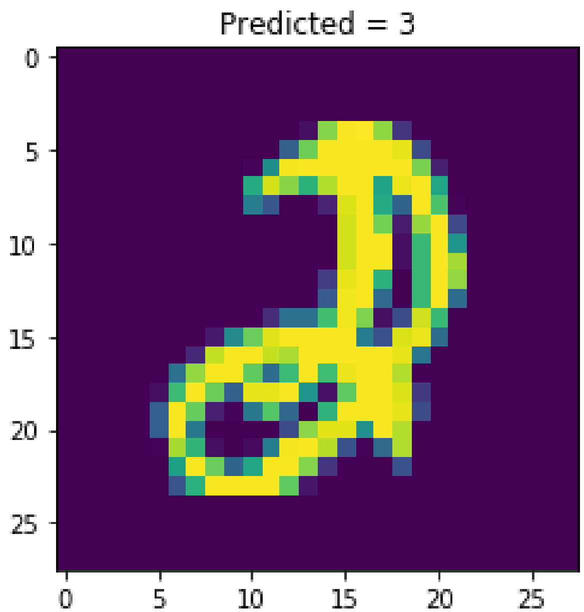


Other models wrong predictions.

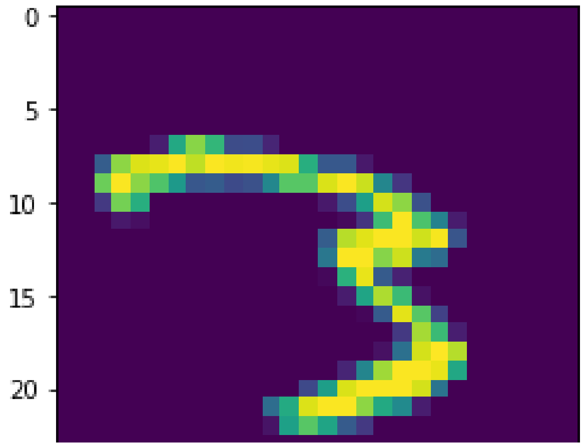
In [0]:
```
import random
for _ in range(10):
  ind = random.choice(ind_arr)
  plt.imshow((X_test[ind].reshape(28, 28))*255)
  plt.title(f"Predicted = {np.argmax(y_pred[ind])}")
  plt.show()
```



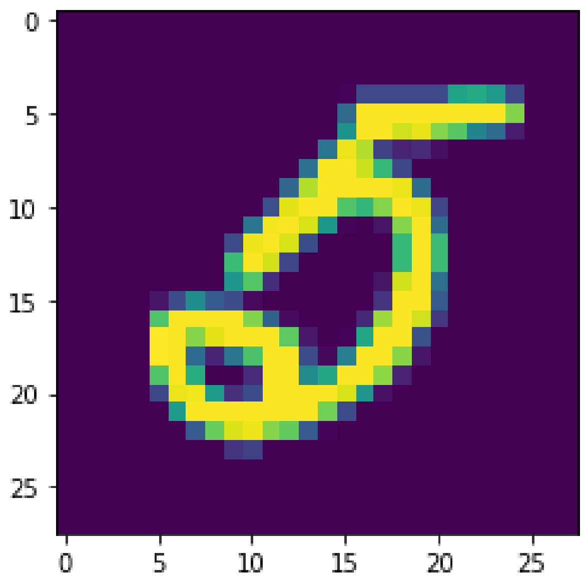

Predicted = 0
0
5
10
15
20
25
0
5
10
15
20
25

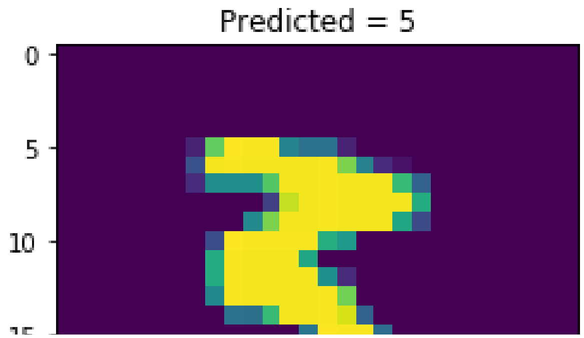

Predicted = 3
0
5
10
15
20
25
0
5
10
15
20
25

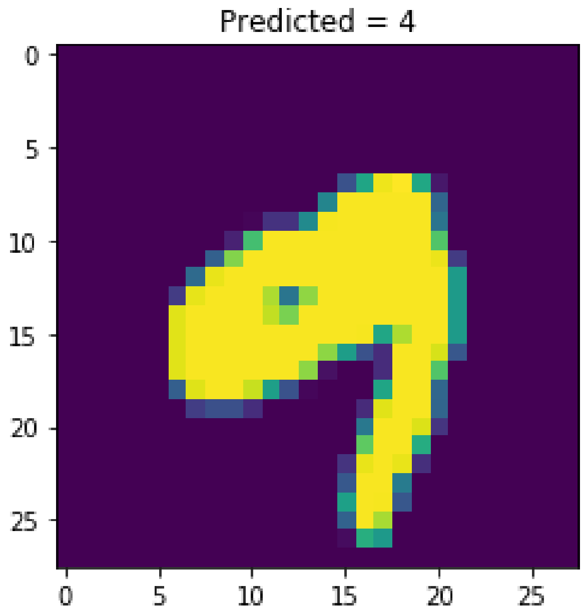
Predicted = 7

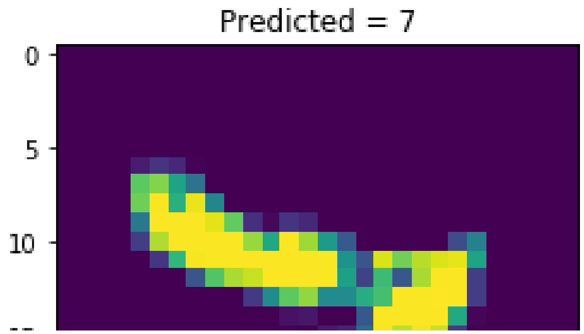


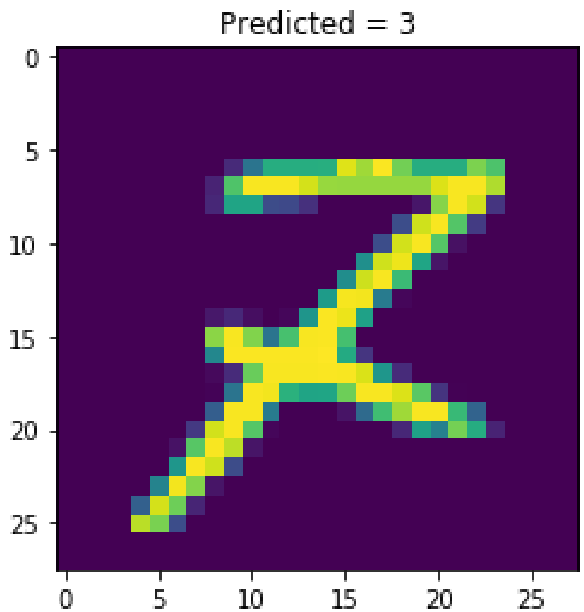
Predicted = 0

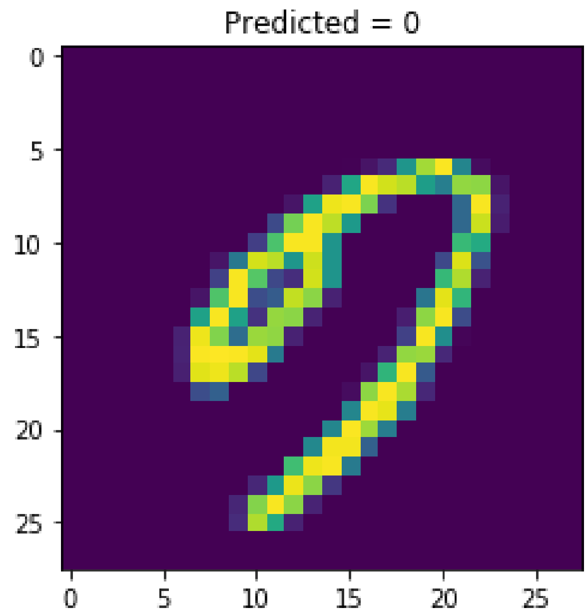



Predicted = 5
0
5
10


Predicted = 4
0
5
10
15
20
25
0
5
10
15
20
25


Predicted = 7
0
5
10


Predicted = 3
0
5
10
15
20
25
0
5
10
15
20
25



Testing other models with no batch normalization and no dropout and changing other parameters like activation functions and optimizers

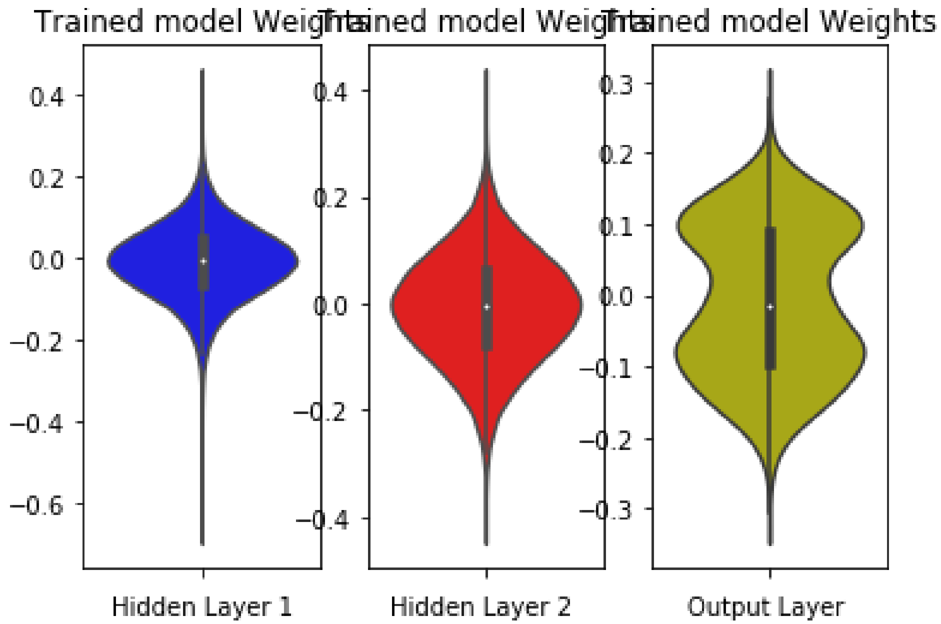
Taking 784-512-256-10 as the model and continuing to experiment on it with No Batch normalizaions and Dropouts. Below are the results of main model which have BN and Dropouts and we see the difference between this model and coming models (All are trained on 20 epochs).

Test score: 0.053256970743143756

Test accuracy: 0.9844

And Weight distributions of 784-512-256-10 (previously trained) are plotted below again for reference.

In [85]:
```
w_after = models['784-512-256-10'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



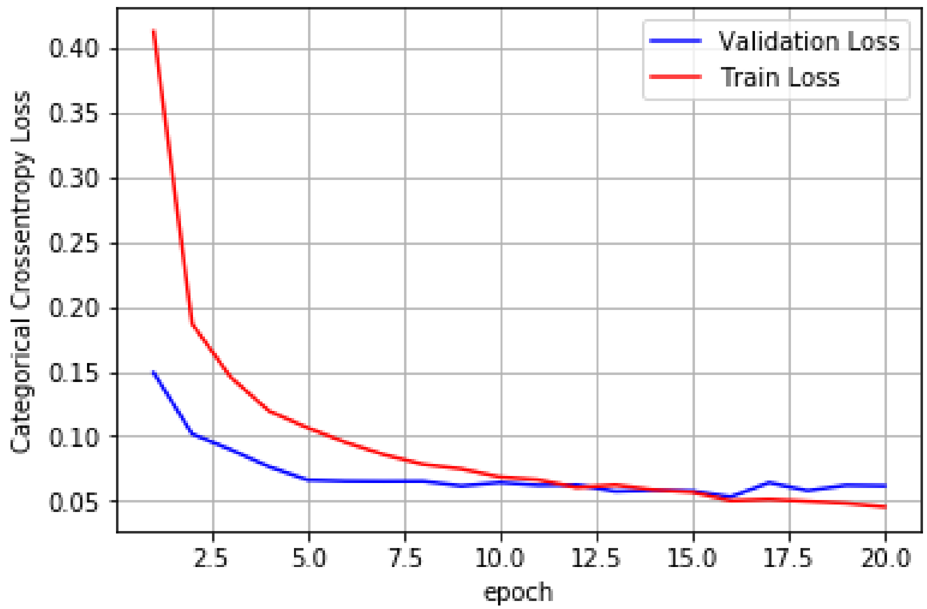
Without Batch Normalization

In [0]:
```
nb_batch = 20
```

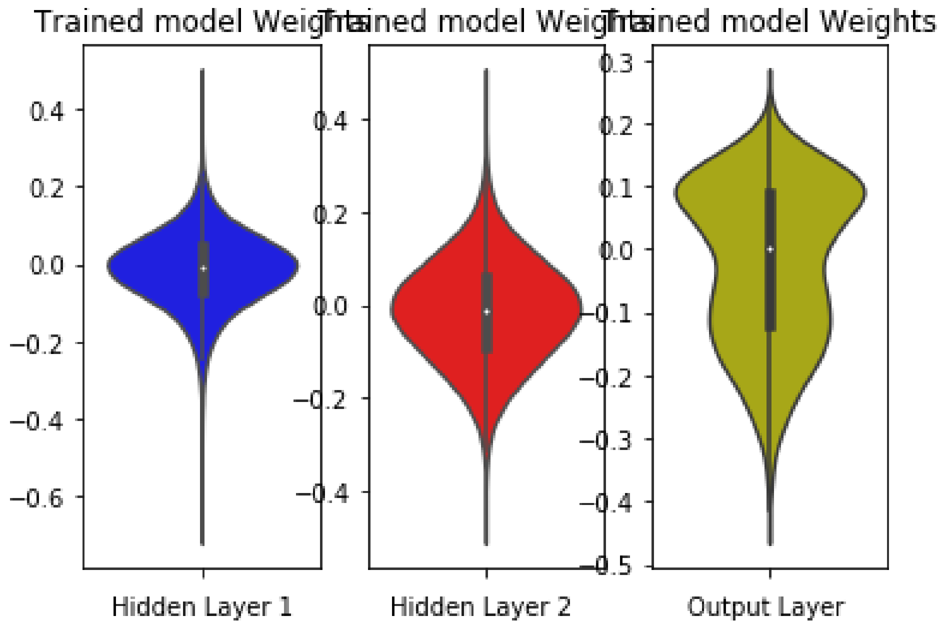
In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_NOBN'] = model_relu
histories['784-512-256_10_NOBN'] = history
```

In [105]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.061633928584580645
Test accuracy: 0.9841
```



In [117]:
```
w_after = models['784-512-256-10_NOBN'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[2].flatten().reshape(-1,1)
out_w = w_after[4].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



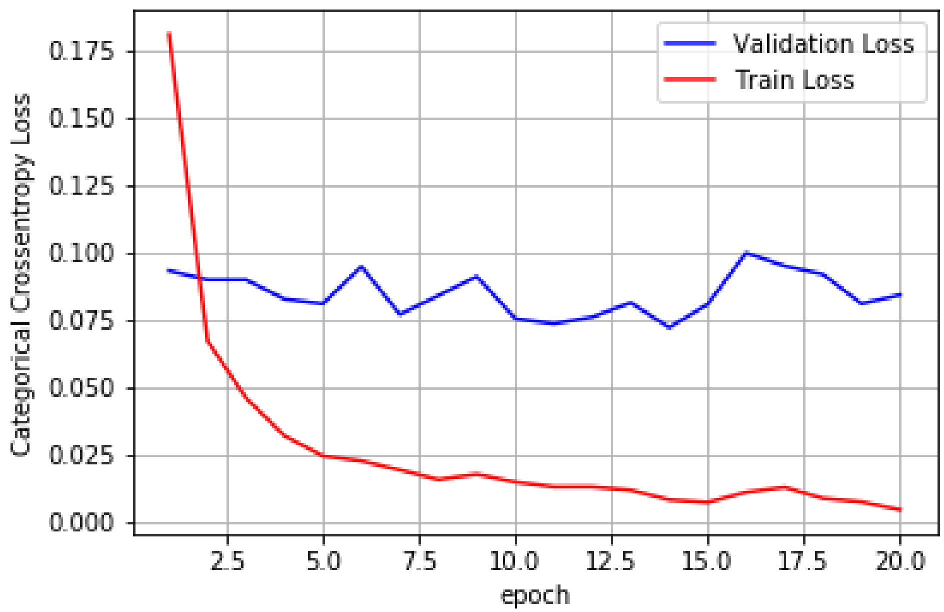
After removing Batch normalization accuracy is slightly less than the main model and loss is also slightly more. Weight distribution looks almost similar for hidden layers but for output layer there seems to be more positive weights than the main model.

Without Dropouts

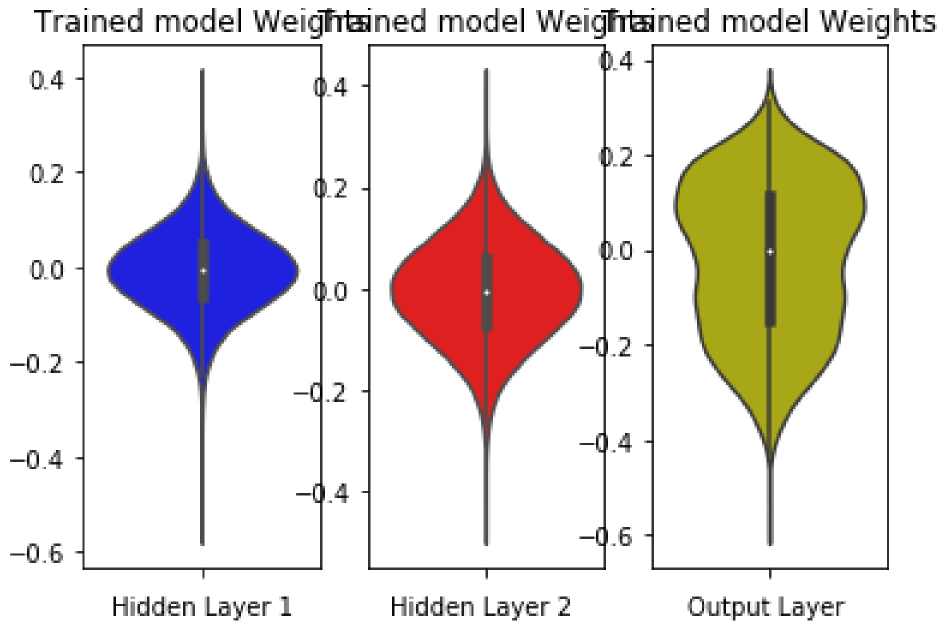
In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_NODR'] = model_relu
histories['784-512-256-10_NODR'] = history
```

In [91]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.08391701467761513
Test accuracy: 0.9813
```



In [126]:
```
w_after = models['784-512-256-10_NODR'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



After removing dropouts we can see a lot of difference between the results. Loss is very high and accuracy seems to be fine. But train loss is so less (neary 0.01) this indicates that our model is overfitting a lot and the loss of test is also fluctualting a lot showing that there are no improvements in testing accuracy as it is overfitted on training set. weight distributions are almost same except for output layer which again seems to have lot of positive values.

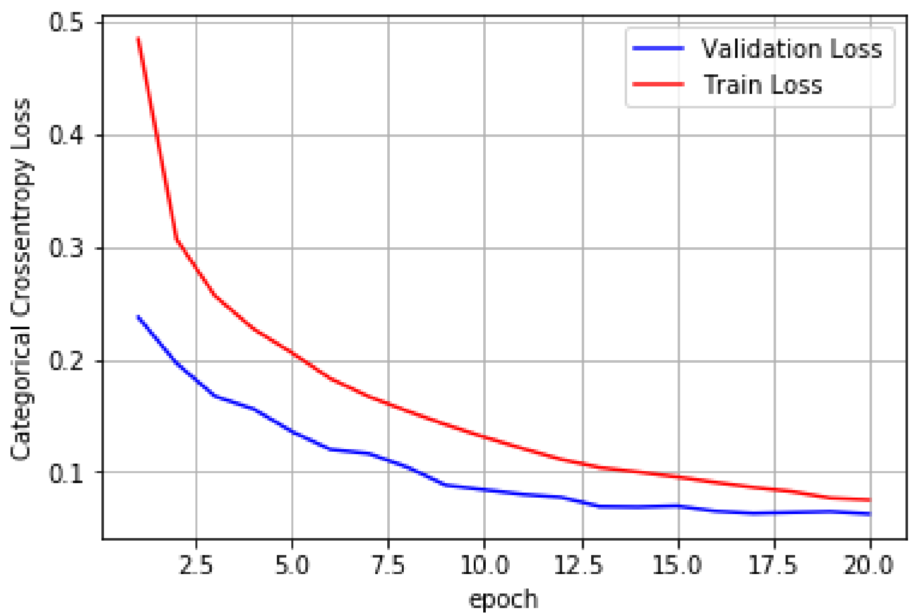
Activation -> Sigmoid

In [0]:

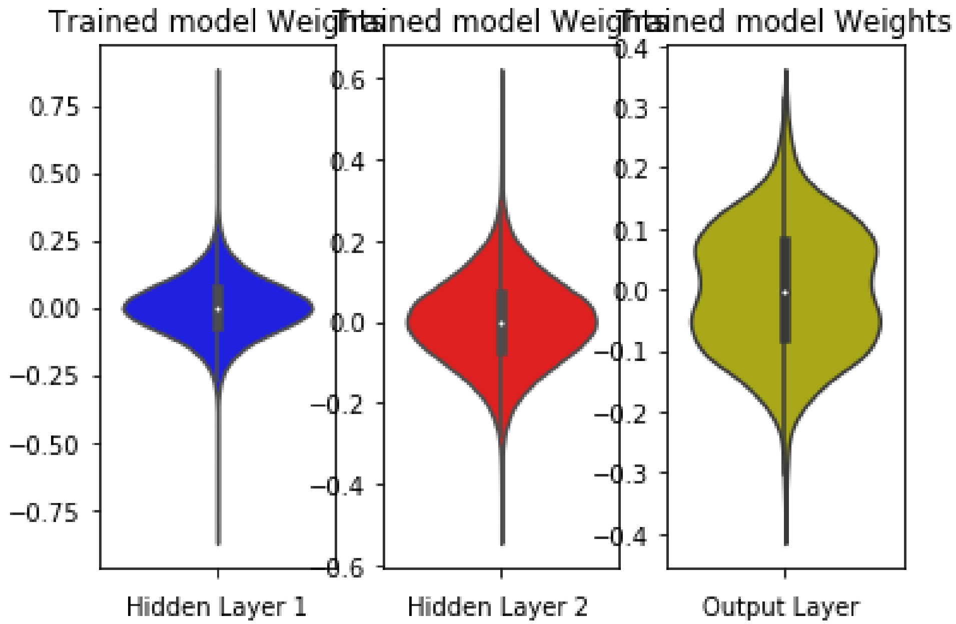
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_initialize
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='sigmoid', kernel_initializer=initializers.he_normal(s
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_sigmoid'] = model_relu
histories['784-512-256-10_sigmoid'] = history
```

In [94]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.06232662117220461
Test accuracy: 0.9814
```



In [119]:
```
w_after = models['784-512-256-10_sigmoid'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



Changing activation functions of 2 hidden layers to `sigmoid` decreased our accuracy and increased our loss as well. So Relu did good than sigmoid even when we dont have very deep network. The weights distributions are almost same and the weights of output layer are nearer to zero in this model.

Optimizer -> AdaDelta

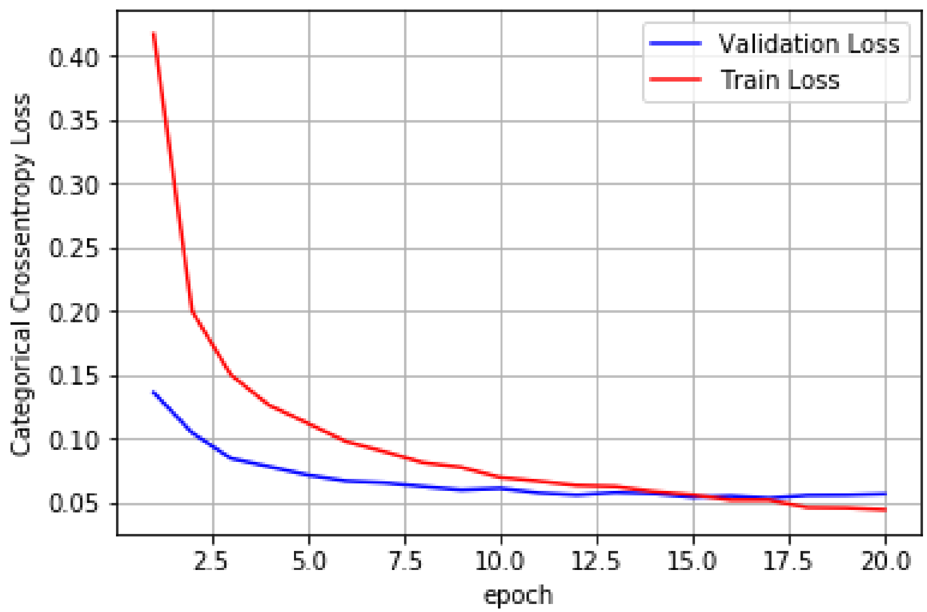
Choosing AdaDelta as this optimizer is good when we have sparse data (we have lot of pixels of value 0 in our pictures).

In [0]:

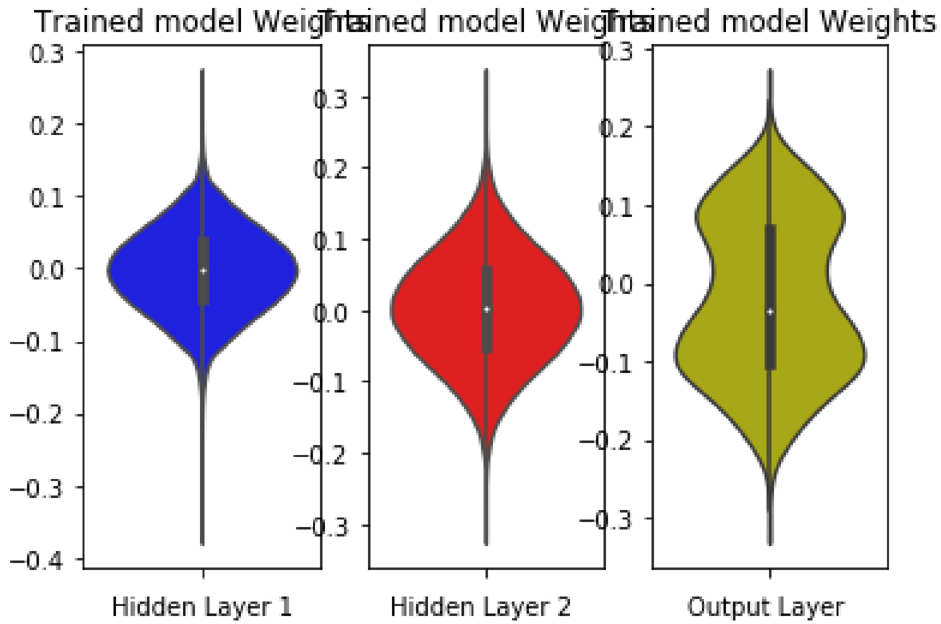
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adadelta', loss='categorical_crossentropy', metrics=['accurac

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_adadelta'] = model_relu
histories['784-512-256-10_adadelta'] = history
```

In [97]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.05676332595801214
Test accuracy: 0.9846
```



In [120]:
```
w_after = models['784-512-256-10_adadelta'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



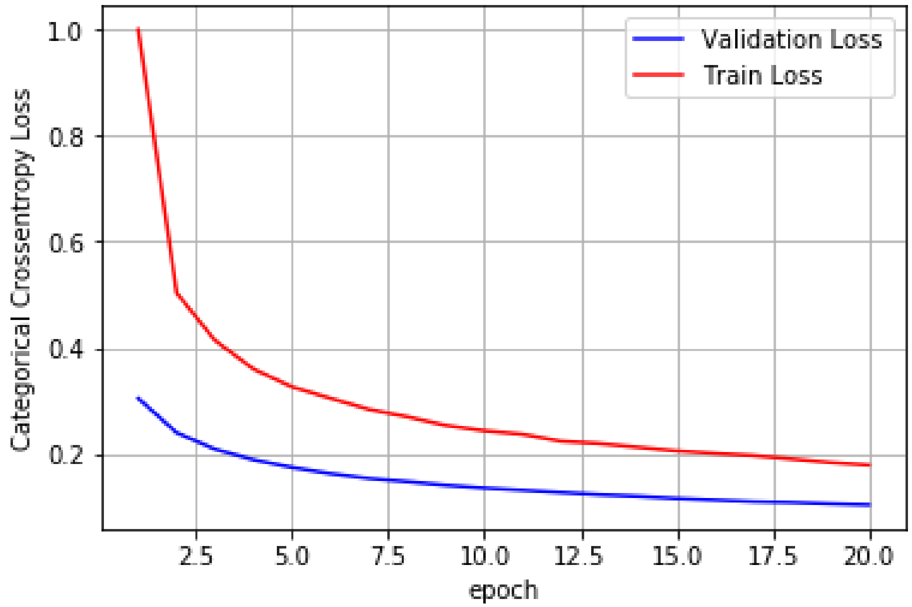
Choosing AdaDelta as our optimizer actually increased our accuracy and also decreased our loss slightly. I think this is due to the reason that agadelta is good with sparse data and we have 0's in our input. the weight distributions are same as the main model which indicates changing our optimizer to another good optimiser didnt change the model much. So let us see for a 'not so good' optimiser so that we can see some difference.

Optimizer -> SGD

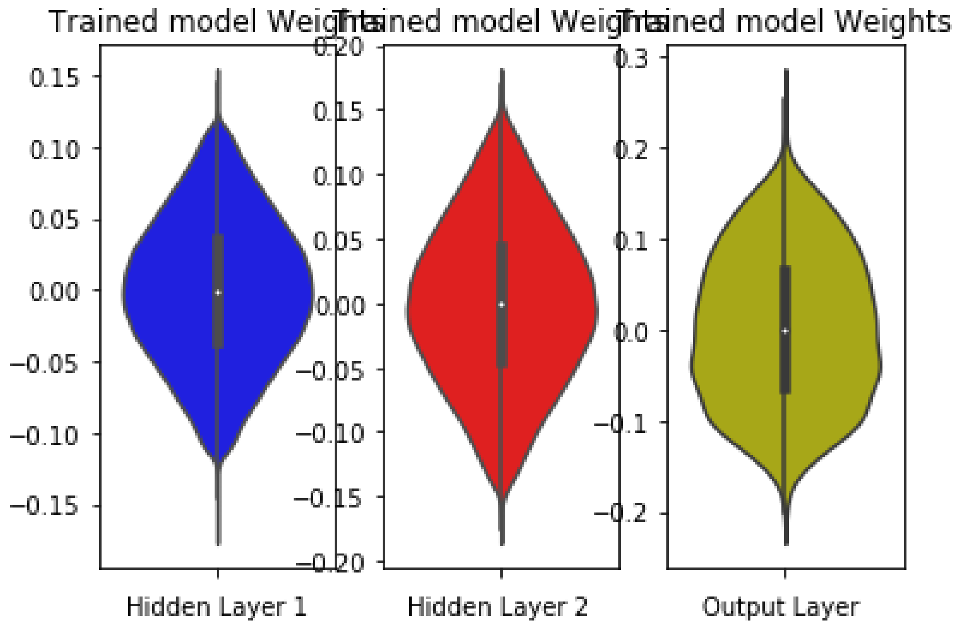
In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.he_normal(seed
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_sgd'] = model_relu
histories['784-512-256-10_sgd'] = history
```

In [110]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.10426200004285202
Test accuracy: 0.9688
```



In [121]:
```
w_after = models['784-512-256-10_sgd'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



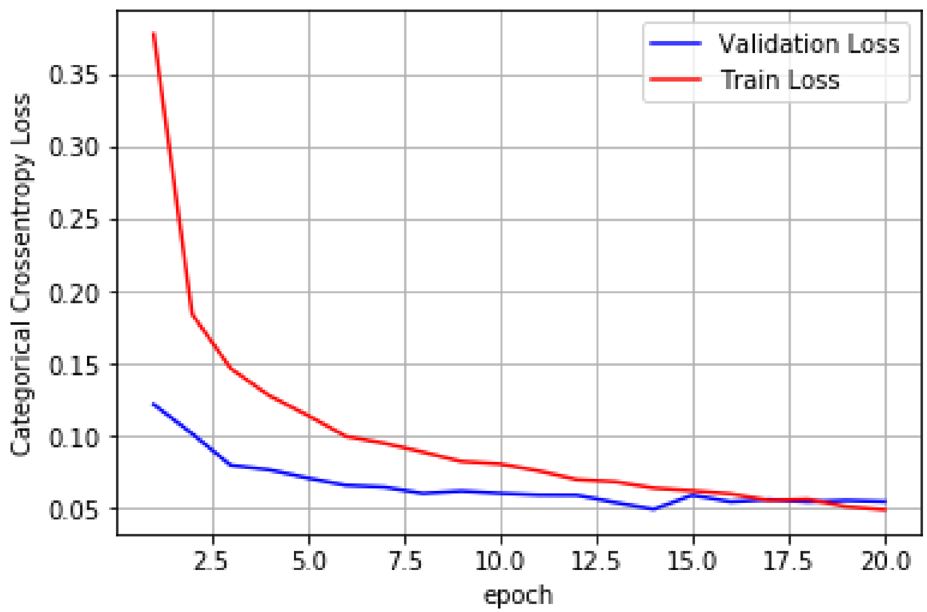
Changing optimisers to SGD is very bad decision as we can see a lot of difference in the models performance and weight distributions. We have least accuracy so far and highest loss so far. And weight distributions are different from the main model and didnt seem to change much from the initializations that are done before training as the output layers weights are not at all same as that of previous models.

Initializers -> Glorot_Normal

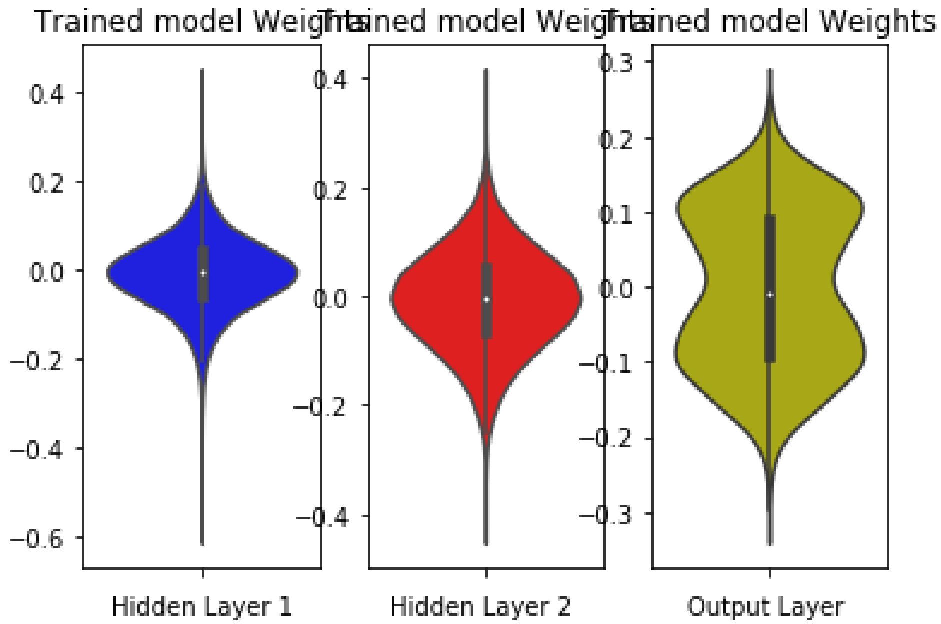
In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.glorot_normal(
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_glorot'] = model_relu
histories['784-512-256-10_glorot'] = history
```

In [101]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.054689733804622664
Test accuracy: 0.9842
```



In [122]:
```
w_after = models['784-512-256-10_glorot'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



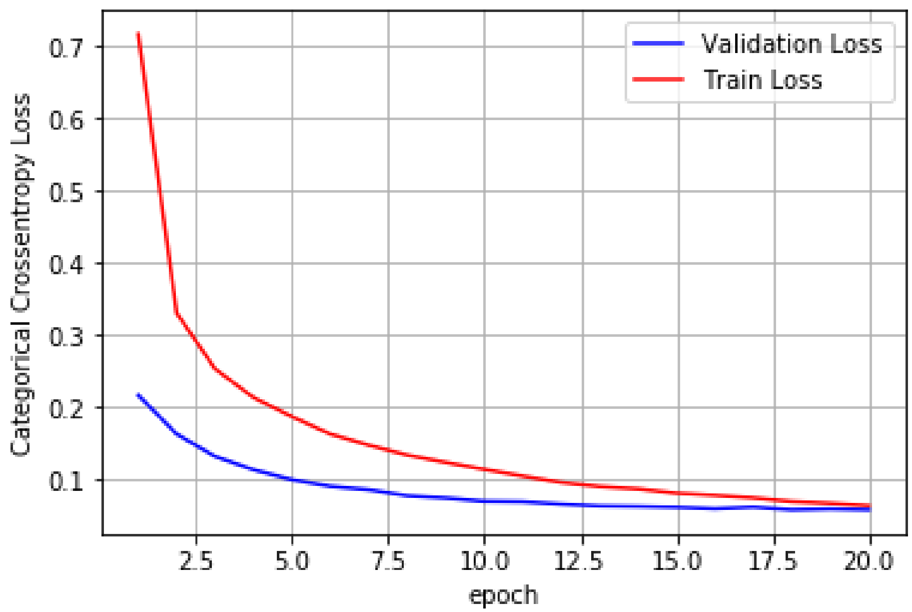
Changing initializations to Glorot normal didnt change results much. accuracy and loss are almost same and weight distributions are also almost same.

Initializers -> Random Uniform (-0.5, 0.5)

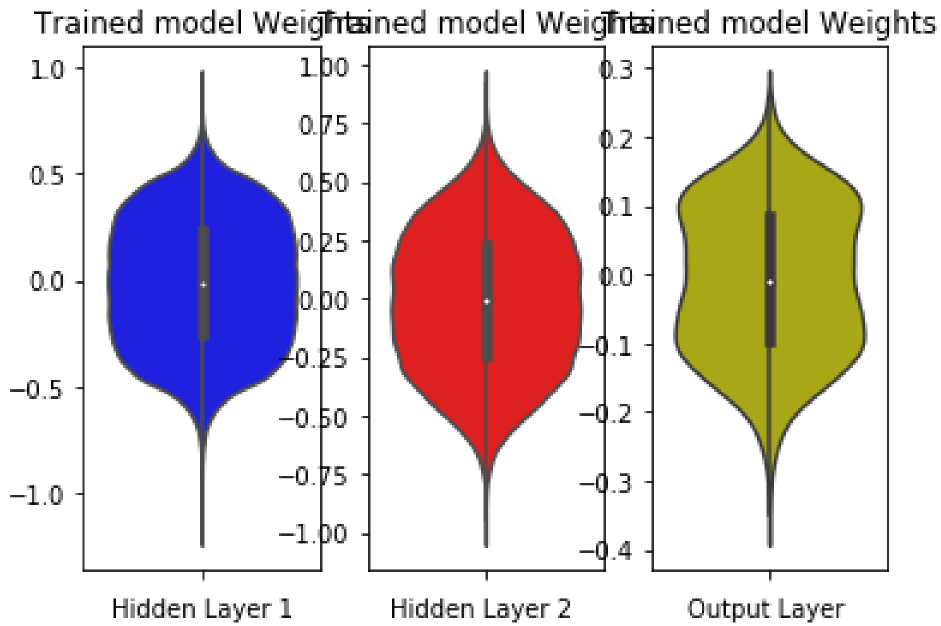
In [0]:
```
import warnings
warnings.filterwarnings('ignore')

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=i
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(256, activation='relu', kernel_initializer=initializers.RandomUniform(
model_relu.add(BatchNormalization())
model_relu.add(Dropout(0.5))
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=

models['784-512-256-10_uniform'] = model_relu
histories['784-512-256-10_uniform'] = history
```

In [115]:
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

x = list(range(1,nb_epoch+1))

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.05820857608325896
Test accuracy: 0.9819
```



In [123]:
```
w_after = models['784-512-256-10_uniform'].get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



Changing initalisations to uniform reduced the accuracy and increased loss. And weight distributions are more flattened than previous models as they are initialised to uniform.

Conclusion:

In [2]:
```
from prettytable import PrettyTable
```

In [3]:
```
table = PrettyTable()
table.field_names = ['Architecture', 'No of hidden layers', 'epochs', 'Cross-entropy Loss',
table.add_row(['784-512-128-10', 2, 20, 0.05295, '98.51%'])
table.add_row(['784-256-128-10', 2, 20, 0.063, '98.11%'])
table.add_row(['784-512-256-10', 2, 20, 0.05326, '98.44%'])
table.add_row(['784-512-256-128-10', 3, 20, 0.06344, '98.26%'])
table.add_row(['784-512-128-64-32-16-10', 5, 20, 0.104, '97.83%'])
table.add_row(['784-512-256-10', 2, 50, 0.049, '98.58%'])
table.add_row(['784-512-256-128-10', 3, 50, 0.05839, '98.61%'])
table.add_row(['784-512-512-10 Dropout: 0.652-0.1', 2, 50, 0.051, '98.61%'])
table.add_row(['784-512-256-10 No BN', 2, 20, 0.06163, '98.41%'])
table.add_row(['784-512-256-10 No Dropout', 2, 20, 0.08392, '98.13%'])
table.add_row(['784-512-256-10 sigmoid', 2, 20, 0.06233, '98.14%'])
table.add_row(['784-512-256-10 AdaDelta', 2, 20, 0.05676, '98.46%'])
table.add_row(['784-512-256-10 SGD', 2, 20, 0.10426, '96.88%'])
table.add_row(['784-512-256-10 Glorot', 2, 20, 0.05469, '98.42%'])
table.add_row(['784-512-256-10 Uniform', 2, 20, 0.05821, '98.19%'])

print(table)
```

```
+-----------------------------------+---------------------+--------+--------------------+
---------------+
|            Architecture           | No of hidden layers | epochs | Cross-entropy Loss |
Test Accuracy |
+-----------------------------------+---------------------+--------+--------------------+
---------------+
|           784-512-128-10          |          2          |   20   |      0.05295       |
98.51%    |
|           784-256-128-10          |          2          |   20   |       0.063        |
98.11%    |
|           784-512-256-10          |          2          |   20   |      0.05326       |
98.44%    |
|         784-512-256-128-10        |          3          |   20   |      0.06344       |
98.26%    |
|      784-512-128-64-32-16-10      |          5          |   20   |       0.104        |
```

				97.83%
784-512-256-10	2	50	0.049	98.58%
784-512-256-128-10	3	50	0.05839	98.61%
784-512-512-10 Dropout: 0.652-0.1	2	50	0.051	98.61%
784-512-256-10 No BN	2	20	0.06163	98.41%
784-512-256-10 No Dropout	2	20	0.08392	98.13%
784-512-256-10 sigmoid	2	20	0.06233	98.14%
784-512-256-10 AdaDelta	2	20	0.05676	98.46%
784-512-256-10 SGD	2	20	0.10426	96.88%
784-512-256-10 Glorot	2	20	0.05469	98.42%
784-512-256-10 Uniform	2	20	0.05821	98.19%

Model with dropout values is obtained by tuning hyper-parameters. models with no extra info have Batch normalization and Dropout layers to them and they have Relu as activation function and Adam as optimizer and weights initialized to He_normal. In above table these values are default for all models except for those it is mentioned

Conclusion:

- **Increase in number of hidden layers didnt increase our accuracies and infact MLP with 5 hidden layers has least accuracy and highest loss.**
- **MLP with 3 hidden layers also didnt give good results as the losses of the models are little high.**

- **The best model with good accuracy and good loss is (784-512-512-10 Dropout: 0.652-0.1) which is obtained by hyper-paramter tuning. Even though MLP with 3 hidden layers (784-512-256-128-10) and 50 epochs gave good accuracy the loss is not that good.**
- **More hyper-parameter tuning might give even better models but accuracy may not be as high as 99% with simple MLPs because lot of model's accuracy is not increasing after certain number of epochs (99% might be achieved by CNN's). This may be due to the reason that images are not good (i.e. have bad handwriting). From above mis-predicted images we can see that mis-predictions have some reasons behind it (i.e. we can see 7 predicted as 3 as there is line in between and 9 predicted as 4 as the image look similar to 4 etc). And even we cant classify few images for example, one of the images (predicted as 3) shown above is not predictable just by looking at image.**
- **The model with optimizer as AdaDelta also seems to give good results. So in furthur hyper-parameter tuning we can consider it as a option. This may be due to the reason that inputs we have are slightly sparse.**
- **Other than optimizer remianing changes didnt give good results than our choices. Which is good.**
- **Removing Batch normalization didnt change the weight distribution much (which I expected to change). The weight distributions of hidden layers are almost normally distributed around 0 in all cases and weight distributions of output layer are also same for almost every model except for some models. Output layer distributions have 2 peaks - one in positive side of zero and other in negative side which i guess due to the activation function softmax.**
- **Distribution of Batch normalization weights are distributed around 1. which I believe they should be as these are scale values that will be multiplied to the results of previous layer.**

In [0]:

```
```