Keras -- MLPs on MNIST

In [2]:

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

In [3]:

```
%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 [4]:

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

In [5]:

```
print("Number of training examples :", X_train.shape[0], "and each image is of shape (%d, %d)"%(X_
train.shape[1], X_train.shape[2]))
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d,
%d)"%(X_test.shape[1], X_test.shape[2]))
```

```
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 [6]:

```
# if you observe the input shape its 3 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 [7]:

```
# 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)"%(X_train.shape[1]))
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d)"%(X_test.
shape[1]))
```

```
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 [8]:

```python
# An example data point
print(X_train[0])
```

```
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255
  247 127   0   0   0   0   0   0   0   0   0   0   0   0  30  36  94 154
  170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0   0   0
    0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82
   82  56  39   0   0   0   0   0   0   0   0   0   0   0   0  18 219 253
  253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  35 241
  225 160 108   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
  253 207   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253
  253 201  78   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0  18 171 219 253 253 253 253 195
   80   9   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0 136 253 253 253 212 135 132  16
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
```

In [9]:

```python
# 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 [10]:

```python
# example data point after normlizing
print(X_train[0])
```

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

```
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.01176471 0.07058824 0.07058824 0.07058824
 0.49411765 0.53333333 0.68627451 0.10196078 0.65098039 1.
 0.96862745 0.49803922 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.11764706 0.14117647 0.36862745 0.60392157
 0.66666667 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
 0.88235294 0.6745098  0.99215686 0.94901961 0.76470588 0.25098039
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.19215686
 0.93333333 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
 0.99215686 0.99215686 0.99215686 0.98431373 0.36470588 0.32156863
 0.32156863 0.21960784 0.15294118 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.07058824 0.85882353 0.99215686
 0.99215686 0.99215686 0.99215686 0.99215686 0.77647059 0.71372549
 0.96862745 0.94509804 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.31372549 0.61176471 0.41960784 0.99215686
 0.99215686 0.80392157 0.04313725 0.         0.16862745 0.60392157
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.05490196 0.00392157 0.60392157 0.99215686 0.35294118
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.54509804 0.99215686 0.74509804 0.00784314 0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.04313725
 0.74509804 0.99215686 0.2745098  0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.1372549  0.94509804
 0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.31764706 0.94117647 0.99215686
 0.99215686 0.46666667 0.09803922 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.17647059 0.72941176 0.99215686 0.99215686
 0.58823529 0.10588235 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
 0.99215686 0.81176471 0.00784314 0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

```
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.15294118 0.58039216
 0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
 0.99215686 0.78823529 0.30588235 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.09019608 0.25882353 0.83529412 0.99215686
 0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.07058824 0.67058824
 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
 0.31372549 0.03529412 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
 0.99215686 0.95686275 0.52156863 0.04313725 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
```

In [11]:

```
# 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.]
```

Softmax classifier

In [12]:

```
# 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_uniform',
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None,
# 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 argument s
upported by all forward layers:

# 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
```

In [13]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

MLP + ReLU + ADAM with 2 Layers

In [14]:

```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization-function-in-
keras
from keras.layers.normalization import BatchNormalization
from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55,
seed=None)))
```

```
seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(output_dim, activation='softmax'))


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               65664
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [15]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

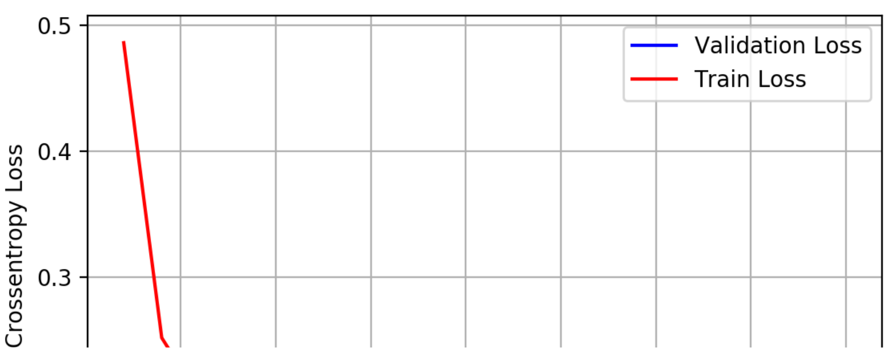
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.4862 - acc: 0.8509 -
val_loss: 0.1624 - val_acc: 0.9497
Epoch 2/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.2521 - acc: 0.9245 -
val_loss: 0.1230 - val_acc: 0.9632
Epoch 3/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.2041 - acc: 0.9388 -
val_loss: 0.1000 - val_acc: 0.9696
Epoch 4/20
60000/60000 [==============================] - 8s 135us/step - loss: 0.1719 - acc: 0.9479 -
val_loss: 0.0915 - val_acc: 0.9712
Epoch 5/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.1554 - acc: 0.9534 -
val_loss: 0.0873 - val_acc: 0.9709
Epoch 6/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.1381 - acc: 0.9583 -
val_loss: 0.0824 - val_acc: 0.9734
Epoch 7/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1308 - acc: 0.9608 -
val_loss: 0.0772 - val_acc: 0.9760
Epoch 8/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.1188 - acc: 0.9637 -
val_loss: 0.0720 - val_acc: 0.9768
Epoch 9/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.1142 - acc: 0.9658 -
val_loss: 0.0717 - val_acc: 0.9794
Epoch 10/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.1047 - acc: 0.9671 - val_l
oss: 0.0678 - val_acc: 0.9790
Epoch 11/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.1010 - acc: 0.9693 -
val_loss: 0.0680 - val_acc: 0.9787
Epoch 12/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0976 - acc: 0.9698 -
val_loss: 0.0639 - val_acc: 0.9808
```

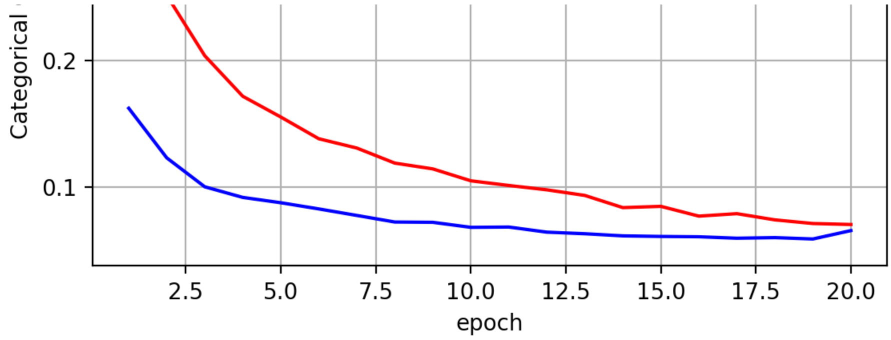
```
Epoch 13/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0931 - acc: 0.9714 -
val_loss: 0.0627 - val_acc: 0.9801
Epoch 14/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0834 - acc: 0.9747 -
val_loss: 0.0610 - val_acc: 0.9811
Epoch 15/20
60000/60000 [==============================] - 8s 125us/step - loss: 0.0844 - acc: 0.9743 -
val_loss: 0.0605 - val_acc: 0.9823
Epoch 16/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.0767 - acc: 0.9754 -
val_loss: 0.0603 - val_acc: 0.9814
Epoch 17/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.0786 - acc: 0.9751 -
val_loss: 0.0591 - val_acc: 0.9833
Epoch 18/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0737 - acc: 0.9767 -
val_loss: 0.0596 - val_acc: 0.9816
Epoch 19/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0708 - acc: 0.9782 -
val_loss: 0.0585 - val_acc: 0.9830
Epoch 20/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0700 - acc: 0.9779 -
val_loss: 0.0652 - val_acc: 0.9821
```

In [16]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
print("Large  Error: %.2f%%" % (100-score[1]*100))

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))

# we will get val_loss and val_acc only when you pass the paramter validation_data
# val_loss : validation loss
# val_acc : validation accuracy

# loss : training loss
# acc : train accuracy
# for each key in histrory.histrory we will have a list of length equal to number of epochs

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

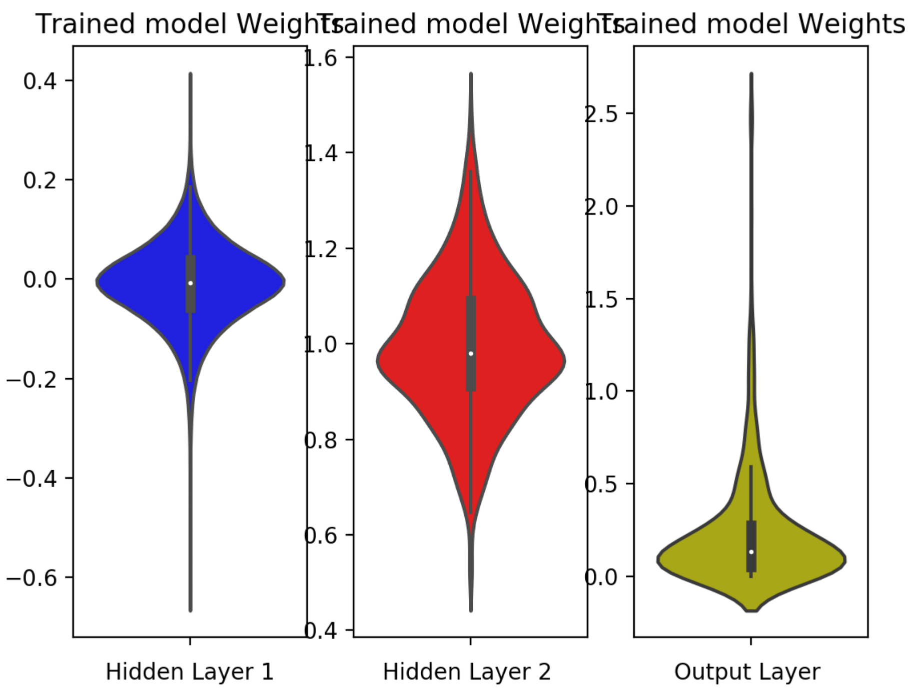
```
Test score: 0.06517606317649478
Test accuracy: 0.9821
Large  Error: 1.79%
```





In [17]:

```
w_after = model_drop.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()
```



MLP + ReLU + ADAM with 3 Layers

In [18]:

```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization-function-in-keras

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(360, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(output_dim, activation='softmax'))


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 512)               401920    
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 360)               184680    
_________________________________________________________________
batch_normalization_4 (Batch (None, 360)               1440      
_________________________________________________________________
dropout_4 (Dropout)          (None, 360)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 128)               46208     
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512       
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290      
=================================================================
Total params: 638,098
Trainable params: 636,098
Non-trainable params: 2,000
_________________________________________________________________
```

In [19]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 190us/step - loss: 0.7516 - acc: 0.7659 - val_loss: 0.2080 - val_acc: 0.9365
Epoch 2/20
60000/60000 [==============================] - 10s 165us/step - loss: 0.3401 - acc: 0.8989 - val_loss: 0.1570 - val_acc: 0.9515
Epoch 3/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.2640 - acc: 0.9219 - val_loss: 0.1278 - val_acc: 0.9612
Epoch 4/20
```

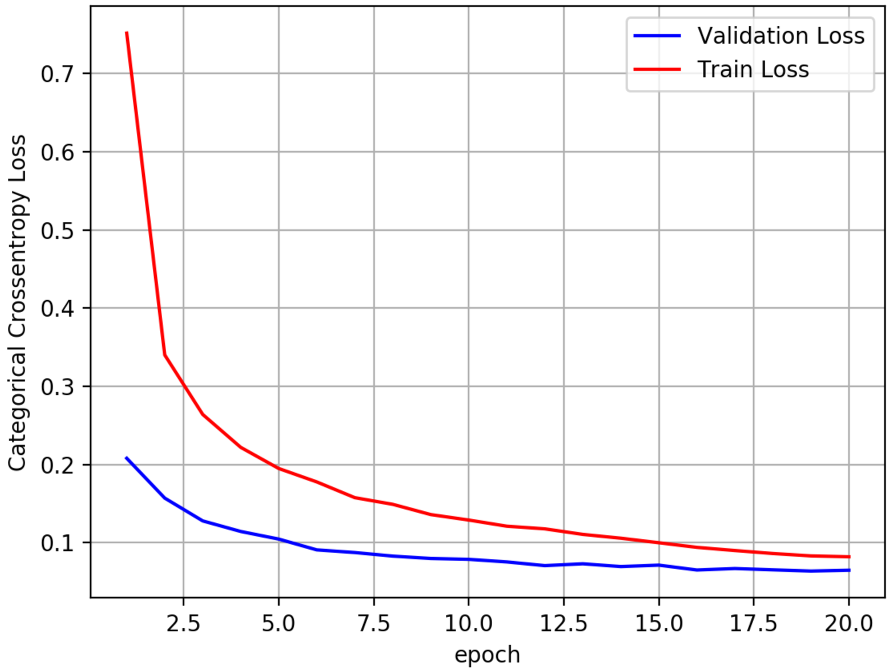
```
60000/60000 [==============================] - 10s 169us/step - loss: 0.2219 - acc: 0.9353 - val_l
oss: 0.1143 - val_acc: 0.9667
Epoch 5/20
60000/60000 [==============================] - 10s 174us/step - loss: 0.1948 - acc: 0.9426 - val_l
oss: 0.1046 - val_acc: 0.9672
Epoch 6/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.1777 - acc: 0.9482 - val_l
oss: 0.0908 - val_acc: 0.9729
Epoch 7/20
60000/60000 [==============================] - 11s 192us/step - loss: 0.1576 - acc: 0.9527 - val_l
oss: 0.0874 - val_acc: 0.9745
Epoch 8/20
60000/60000 [==============================] - 10s 164us/step - loss: 0.1490 - acc: 0.9553 - val_l
oss: 0.0828 - val_acc: 0.9764
Epoch 9/20
60000/60000 [==============================] - 12s 197us/step - loss: 0.1359 - acc: 0.9593 - val_l
oss: 0.0798 - val_acc: 0.9773
Epoch 10/20
60000/60000 [==============================] - 15s 252us/step - loss: 0.1289 - acc: 0.9612 - val_l
oss: 0.0787 - val_acc: 0.9770
Epoch 11/20
60000/60000 [==============================] - 12s 208us/step - loss: 0.1211 - acc: 0.9642 - val_l
oss: 0.0754 - val_acc: 0.9776
Epoch 12/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.1177 - acc: 0.9646 - val_l
oss: 0.0707 - val_acc: 0.9788
Epoch 13/20
60000/60000 [==============================] - 13s 217us/step - loss: 0.1106 - acc: 0.9672 - val_l
oss: 0.0730 - val_acc: 0.9779
Epoch 14/20
60000/60000 [==============================] - 13s 224us/step - loss: 0.1057 - acc: 0.9680 - val_l
oss: 0.0695 - val_acc: 0.9796
Epoch 15/20
60000/60000 [==============================] - 13s 223us/step - loss: 0.0999 - acc: 0.9698 - val_l
oss: 0.0713 - val_acc: 0.9796
Epoch 16/20
60000/60000 [==============================] - 14s 227us/step - loss: 0.0940 - acc: 0.9714 - val_l
oss: 0.0651 - val_acc: 0.9816
Epoch 17/20
60000/60000 [==============================] - 15s 251us/step - loss: 0.0900 - acc: 0.9725 - val_l
oss: 0.0670 - val_acc: 0.9801
Epoch 18/20
60000/60000 [==============================] - 13s 223us/step - loss: 0.0862 - acc: 0.9737 - val_l
oss: 0.0653 - val_acc: 0.9816
Epoch 19/20
60000/60000 [==============================] - 13s 218us/step - loss: 0.0831 - acc: 0.9749 - val_l
oss: 0.0637 - val_acc: 0.9817
Epoch 20/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.0820 - acc: 0.9746 - val_l
oss: 0.0648 - val_acc: 0.9815
```

In [20]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
print(" Error: %.2f%%" % (100-score[1]*100))

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))

# we will get val_loss and val_acc only when you pass the paramter validation_data
# val_loss : validation loss
# val_acc : validation accuracy

# loss : training loss
# acc : train accuracy
# for each key in histrory.histrory we will have a list of length equal to number of epochs
```

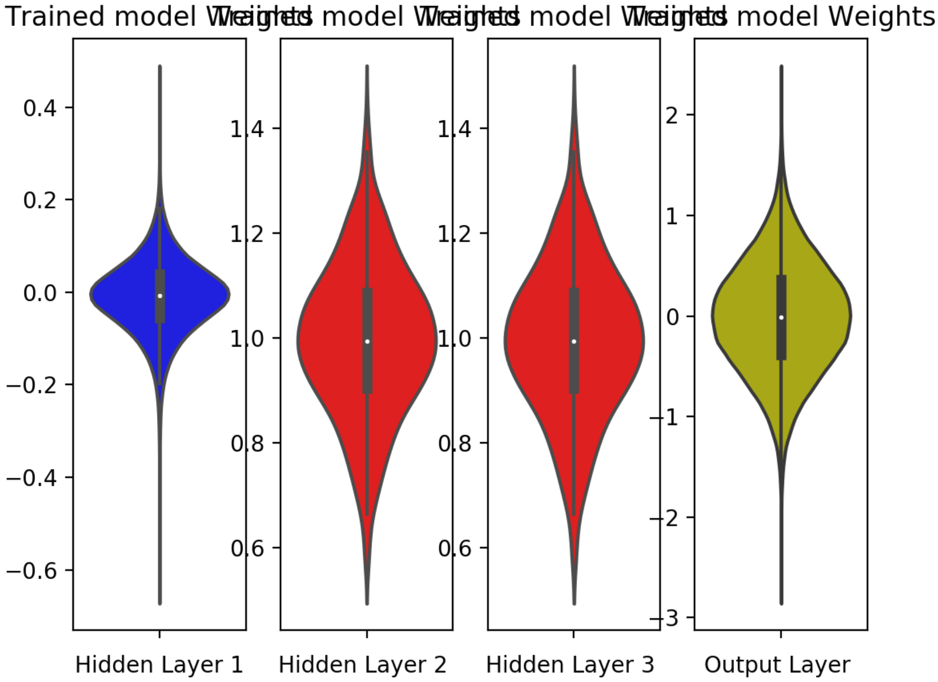
```
vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.06477750879065133
Test accuracy: 0.9815
 Error: 1.85%
```



In [21]:

```
w_after = model_drop.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[2].flatten().reshape(-1,1)
h3_w = w_after[4].flatten().reshape(-1,1)
out_w = w_after[6].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=h2_w, color='r')
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()
```



MLP + ReLU + ADAM with 5 Layers

In [22]:

```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization-function-in-
keras

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(620, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(480, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(350, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(220, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55,
seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55,
seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(output_dim, activation='softmax'))


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 620)               486700
_________________________________________________________________
batch_normalization_6 (Batch (None, 620)               2480
```

```
batch_normalization_6 (Batch (None, 620)              2480
_________________________________________________________________
dropout_6 (Dropout)          (None, 620)              0
_________________________________________________________________
dense_9 (Dense)              (None, 480)              298080
_________________________________________________________________
batch_normalization_7 (Batch (None, 480)              1920
_________________________________________________________________
dropout_7 (Dropout)          (None, 480)              0
_________________________________________________________________
dense_10 (Dense)             (None, 350)              168350
_________________________________________________________________
batch_normalization_8 (Batch (None, 350)              1400
_________________________________________________________________
dropout_8 (Dropout)          (None, 350)              0
_________________________________________________________________
dense_11 (Dense)             (None, 220)              77220
_________________________________________________________________
batch_normalization_9 (Batch (None, 220)              880
_________________________________________________________________
dropout_9 (Dropout)          (None, 220)              0
_________________________________________________________________
dense_12 (Dense)             (None, 128)              28288
_________________________________________________________________
batch_normalization_10 (Batc (None, 128)              512
_________________________________________________________________
dropout_10 (Dropout)         (None, 128)              0
_________________________________________________________________
dense_13 (Dense)             (None, 10)               1290
=================================================================
Total params: 1,067,120
Trainable params: 1,063,524
Non-trainable params: 3,596
_________________________________________________________________
```

In [23]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

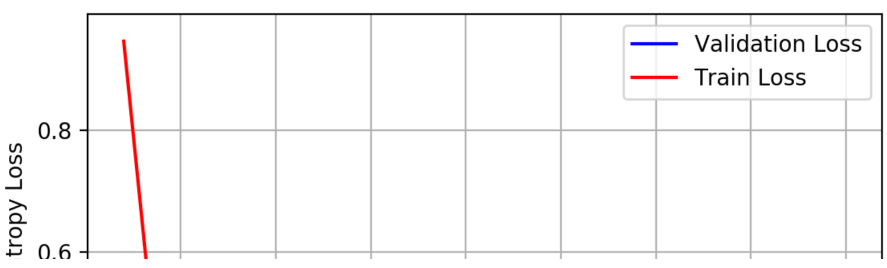
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 22s 365us/step - loss: 0.9467 - acc: 0.7095 - val_l
oss: 0.2148 - val_acc: 0.9358
Epoch 2/20
60000/60000 [==============================] - 18s 298us/step - loss: 0.3351 - acc: 0.9064 - val_l
oss: 0.1470 - val_acc: 0.9572
Epoch 3/20
60000/60000 [==============================] - 22s 375us/step - loss: 0.2519 - acc: 0.9316 - val_l
oss: 0.1209 - val_acc: 0.9648
Epoch 4/20
60000/60000 [==============================] - 24s 392us/step - loss: 0.2065 - acc: 0.9437 - val_l
oss: 0.1061 - val_acc: 0.9696
Epoch 5/20
60000/60000 [==============================] - 20s 339us/step - loss: 0.1819 - acc: 0.9508 - val_l
oss: 0.1079 - val_acc: 0.9715
Epoch 6/20
60000/60000 [==============================] - 21s 347us/step - loss: 0.1664 - acc: 0.9550 - val_l
oss: 0.0953 - val_acc: 0.9747
Epoch 7/20
60000/60000 [==============================] - 20s 334us/step - loss: 0.1548 - acc: 0.9578 - val_l
oss: 0.0864 - val_acc: 0.9768
Epoch 8/20
60000/60000 [==============================] - 20s 340us/step - loss: 0.1418 - acc: 0.9612 - val_l
oss: 0.0824 - val_acc: 0.9782
Epoch 9/20
60000/60000 [==============================] - 22s 372us/step - loss: 0.1332 - acc: 0.9639 - val_l
oss: 0.0868 - val_acc: 0.9765
Epoch 10/20
60000/60000 [==============================] - 18s 292us/step - loss: 0.1226 - acc: 0.9665 - val_l
oss: 0.0788 - val_acc: 0.9794
Epoch 11/20
60000/60000 [==============================] - 19s 318us/step - loss: 0.1133 - acc: 0.9687 - val_l
oss: 0.0815 - val_acc: 0.9788
```

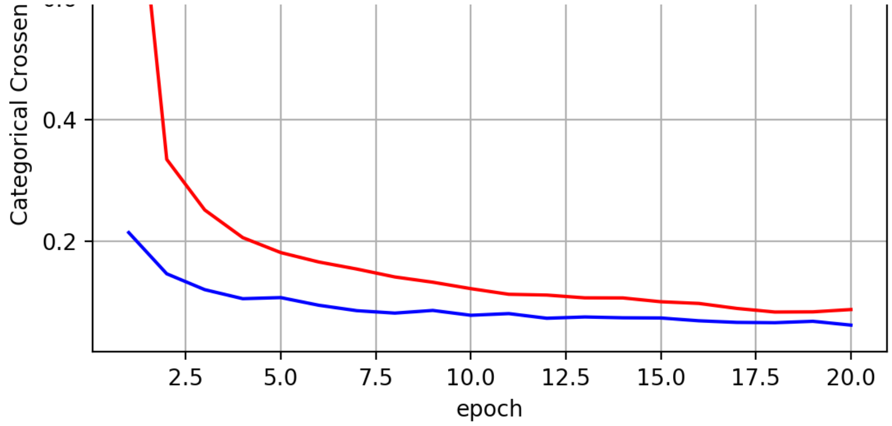
```
Epoch 12/20
60000/60000 [==============================] - 22s 360us/step - loss: 0.1120 - acc: 0.9690 - val_l
oss: 0.0739 - val_acc: 0.9810
Epoch 13/20
60000/60000 [==============================] - 21s 344us/step - loss: 0.1075 - acc: 0.9701 - val_l
oss: 0.0760 - val_acc: 0.9809
Epoch 14/20
60000/60000 [==============================] - 22s 364us/step - loss: 0.1073 - acc: 0.9715 - val_l
oss: 0.0746 - val_acc: 0.9802
Epoch 15/20
60000/60000 [==============================] - 23s 388us/step - loss: 0.1010 - acc: 0.9721 - val_l
oss: 0.0743 - val_acc: 0.9796
Epoch 16/20
60000/60000 [==============================] - 20s 325us/step - loss: 0.0983 - acc: 0.9735 - val_l
oss: 0.0698 - val_acc: 0.9832
Epoch 17/20
60000/60000 [==============================] - 19s 323us/step - loss: 0.0901 - acc: 0.9754 - val_l
oss: 0.0671 - val_acc: 0.9824
Epoch 18/20
60000/60000 [==============================] - 19s 315us/step - loss: 0.0842 - acc: 0.9768 - val_l
oss: 0.0666 - val_acc: 0.9829
Epoch 19/20
60000/60000 [==============================] - 20s 333us/step - loss: 0.0844 - acc: 0.9758 - val_l
oss: 0.0689 - val_acc: 0.9824
Epoch 20/20
60000/60000 [==============================] - 20s 340us/step - loss: 0.0882 - acc: 0.9764 - val_l
oss: 0.0626 - val_acc: 0.9838
```

In [24]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
print(" Error: %.2f%%" % (100-score[1]*100))

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))

# we will get val_loss and val_acc only when you pass the paramter validation_data
# val_loss : validation loss
# val_acc : validation accuracy

# loss : training loss
# acc : train accuracy
# for each key in histrory.histrory we will have a list of length equal to number of epochs

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

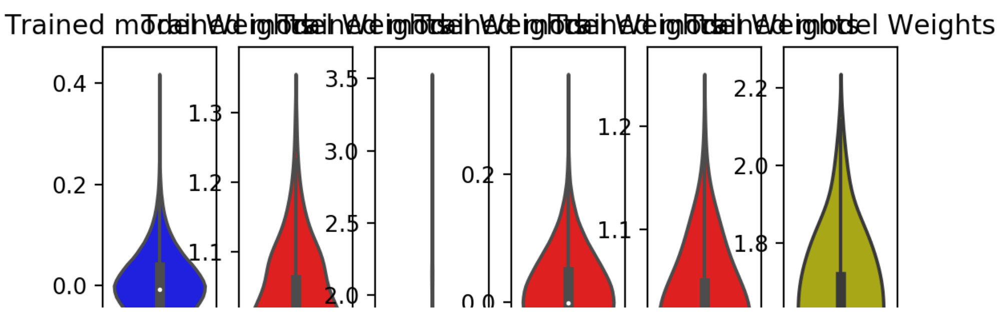
```
Test score: 0.06262384752128274
Test accuracy: 0.9838
 Error: 1.62%
```

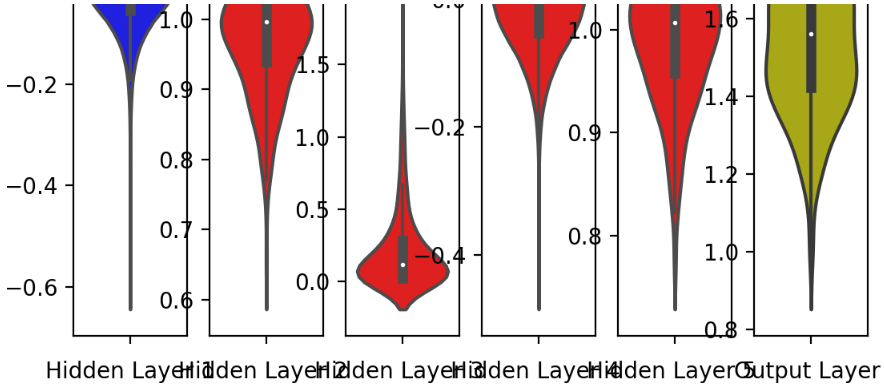




In [25]:

```
w_after = model_drop.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[2].flatten().reshape(-1,1)
h3_w = w_after[4].flatten().reshape(-1,1)
h4_w = w_after[6].flatten().reshape(-1,1)
h5_w = w_after[8].flatten().reshape(-1,1)
out_w = w_after[10].flatten().reshape(-1,1)


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 6, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 6, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 6, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='r')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1, 6, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='r')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='r')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```





Results

In [30]:

```
from prettytable import PrettyTable
x = PrettyTable()

names = ["Relu with 2 layers", "Relu with 3 layers", "Relu with 5 layers"]
Test_Accuracy = [97.91, 97.61, 97.8]
Test_Score = [0.0691,0.080,0.082]
numbering = [1,2,3]
ptable = PrettyTable()
# Adding columns
ptable.add_column("S.NO.",numbering)
ptable.add_column("MODEL",names)

ptable.add_column("Test Score",Test_Score)
ptable.add_column("Test Accuracy",Test_Accuracy)
# Printing the Table
print(ptable)
```

```
+--------+--------------------+------------+---------------+
| S.NO.  |       MODEL        | Test Score | Test Accuracy |
+--------+--------------------+------------+---------------+
|   1    | Relu with 2 layers |   0.0691   |     97.91     |
|   2    | Relu with 3 layers |    0.08    |     97.61     |
|   3    | Relu with 5 layers |   0.082    |      97.8     |
+--------+--------------------+------------+---------------+
```

Reference

In [ ]:

```
https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/exercise-try-different-mlpa
rchitectures-on-mnist-dataset/
```