Build various MLP architectures for MNIST dataset.

Exercise :

1. Use MNIST dataset, which is present in keras datasets.
2. Try 2-hidden layer, 3-hidden layer and 5-hidden layer MLP.
3. Also use dropout and batch normalization and plot train-test error vs epochs for each model.
4. Write your observations in English as crisply and unambiguously as possible. Always quantify your results.

Information regarding data set :

1. **Title**: MNIST database of handwritten digits
2. **Sources**: Modified National Institute of Standards and Technology(MNIST)
3. **Relevant Information**: The MNIST database of handwritten digits, available from the page(http://yann.lecun.com/exdb/mnist/), has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image..

In [119]:
```
import warnings
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

# For plotting purposes
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from keras.utils import to_categorical
from keras.models import Sequential
from keras.initializers import he_normal
from keras.layers import BatchNormalization, Dense, Dropout

# Import MNIST Dataset
from keras.datasets import mnist
```

In [120]:
```
# Load and split MNIST dataset
(x_train,y_train),(x_test,y_test) = mnist.load_data()
```

In [121]:
```
print("x_train shape: ", x_train.shape)
print("x_test shape: ", x_test.shape)
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 testing examples :", x_test.shape[0], "and each image is of shape (%d, %d)"%(x_test.shape[1], x_test.shape[2]))
```

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

If we observe the input shape its 3 dimensional vector, so for each image we have a (28*28) vector.

We will convert the (2828) vector into single dimensional vector of 1 784

In [122]:
```
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])

# after converting the input images from 3d to 2d vectors
print("x_train shape: ", x_train.shape)
print("x_test shape: ", x_test.shape)
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]))
```

```
x_train shape:  (60000, 784)
x_test shape:  (10000, 784)
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 [123]:

```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 [124]:

```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_std => (X - Xmin)/(Xmax-Xmin)
# X_scaled = X_std * (Xmax - Xmin) + Xmin
minMaxScaler = MinMaxScaler()

x_train = minMaxScaler.fit_transform(x_train)
x_test = minMaxScaler.transform(x_test)

# x_train 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.         0.         0.
 0.         0.         0.         0.        ]
```

In [125]:

```
temp = y_train[0]
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)

print("After converting the output {0} into a vector : {1}".format(temp,y_train[0]))
```

```
After converting the output 5 into a vector : [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

In [166]:
```
# Plot train and cross validation loss
def plot_train_cv_loss(trained_model, epochs, colors=['b']):
    fig, ax = plt.subplots(1,1)
    ax.set_xlabel('epoch')
    ax.set_ylabel('Categorical Crossentropy Loss')
    x_axis_values = list(range(1,epochs+1))

    validation_loss = trained_model.history['val_loss']
    train_loss = trained_model.history['loss']

    ax.plot(x_axis_values, validation_loss, 'b', label="Validation Loss")
    ax.plot(x_axis_values, train_loss, 'r', label="Train Loss")
    plt.legend()
    plt.grid()
    fig.canvas.draw()


# Plot weight distribution using violin plot
def plot_weights(model):
    w_after = model.get_weights()

    o1_w = w_after[0].flatten().reshape(-1,1)
    o2_w = w_after[2].flatten().reshape(-1,1)
    out_w = w_after[4].flatten().reshape(-1,1)


    fig = plt.figure(figsize=(10,7))
    plt.title("Weight matrices after model trained\n")
    plt.subplot(1, 3, 1)
    plt.title("Trained model\n Weights")
    ax = sns.violinplot(y=o1_w,color='b')
    plt.xlabel('Hidden Layer 1')

    plt.subplot(1, 3, 2)
    plt.title("Trained model\n Weights")
    ax = sns.violinplot(y=o2_w, color='r')
    plt.xlabel('Hidden Layer 2 ')

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

With 2-Hidden Layers

In [128]:
```
# Batch size
batch_size = 128

# Number of time whole data is trained
epochs = 20

# Input layer dimension
input_dimension = x_train.shape[1]

# Output layer dimension
output_dimension = y_train.shape[1]
```

In [129]:
```
# Instantiate sequential model
model = Sequential()

# Add 1st hidden layer : dense Layer
dense_layer1 = Dense(512,
                       activation="relu",
                       input_shape=(input_dimension,),
                       kernel_initializer= he_normal(seed=None))
model.add(dense_layer1)

# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add 2nd hidden layer : dense Layer
dense_layer2 = Dense(128,
                       activation="relu",
                       kernel_initializer= he_normal(seed=None))
model.add(dense_layer2)

# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add output layer : dense Layer
dense_layer3 = Dense(output_dimension, activation='softmax')
model.add(dense_layer3)

# Summary of the model
print("Model Summary: \n")
model.summary()
print()
print()

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Run the model
trained_model = model.fit(x_train, y_train, batch_size = batch_size, epochs = epochs, verbose=1, valid
ation_data=(x_test, y_test))
```

```
Model Summary:

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 512)               401920    
_________________________________________________________________
batch_normalization_4 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 128)               65664     
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512       
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________


Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 163us/step - loss: 0.4283 - acc: 0.8700 - val_loss: 0.1467 - val_acc: 0.9534
Epoch 2/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.2024 - acc: 0.9391 - val_loss: 0.1050 - val_acc: 0.9665
Epoch 3/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.1585 - acc: 0.9518 - val_loss: 0.0935 - val_acc: 0.9712
Epoch 4/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.1350 - acc: 0.9588 - val_loss: 0.0785 - val_acc: 0.9756
Epoch 5/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.1214 - acc: 0.9628 - val_loss: 0.0749 - val_acc: 0.9768
Epoch 6/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.1077 - acc: 0.9669 - val_loss: 0.0720 - val_acc: 0.9779
Epoch 7/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.1003 - acc: 0.9692 - val_loss: 0.0676 - val_acc: 0.9800
Epoch 8/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0917 - acc: 0.9724 - val_loss: 0.0677 - val_acc: 0.9792
Epoch 9/20
60000/60000 [==============================] - 8s 130us/step - loss: 0.0868 - acc: 0.9729 - val_loss: 0.0625 - val_acc: 0.9805
Epoch 10/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.0767 - acc: 0.9757 - val_loss: 0.0623 - val_acc: 0.9814
Epoch 11/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0771 - acc: 0.9756 - val_loss: 0.0641 - val_acc: 0.9810
Epoch 12/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.0750 - acc: 0.9770 - val_loss: 0.0587 - val_acc: 0.9815
Epoch 13/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0689 - acc: 0.9781 - val_loss: 0.0635 - val_acc: 0.9810
Epoch 14/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.0696 - acc: 0.9778 - val_loss: 0.0609 - val_acc: 0.9827
Epoch 15/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.0648 - acc: 0.9797 - val_loss: 0.0565 - val_acc: 0.9826
Epoch 16/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.0617 - acc: 0.9809 - val_loss: 0.0569 - val_acc: 0.9834
Epoch 17/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.0589 - acc: 0.9810 - val_loss: 0.0547 - val_acc: 0.9834
Epoch 18/20
60000/60000 [==============================] - 10s 171us/step - loss: 0.0561 - acc: 0.9823 - val_loss: 0.0541 - val_acc: 0.9846
Epoch 19/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.0525 - acc: 0.9827 - val_loss: 0.0572 - val_acc: 0.9834
Epoch 20/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0538 - acc: 0.9833 - val_loss: 0.0527 - val_acc: 0.9835
```

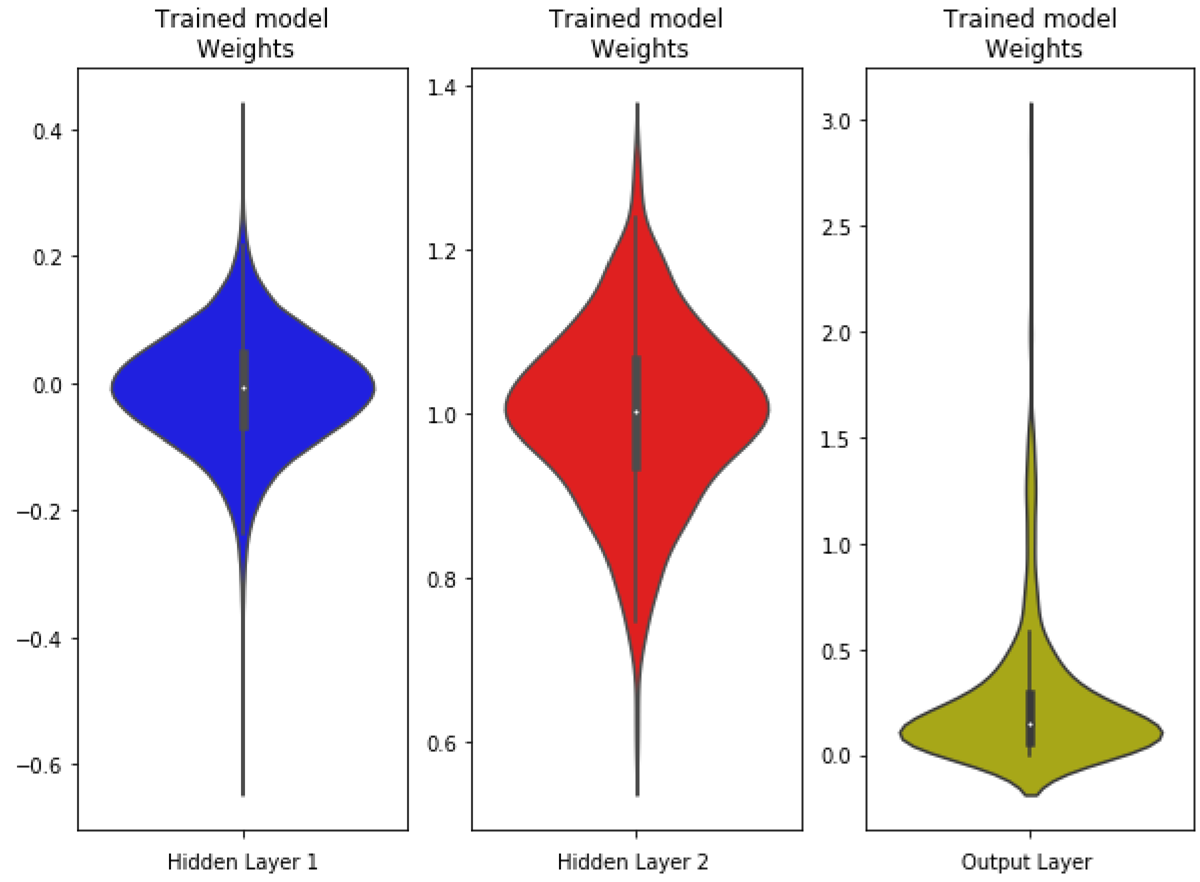
In [150]:
```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy: {0:.2f}%'.format(score[1]*100))
```

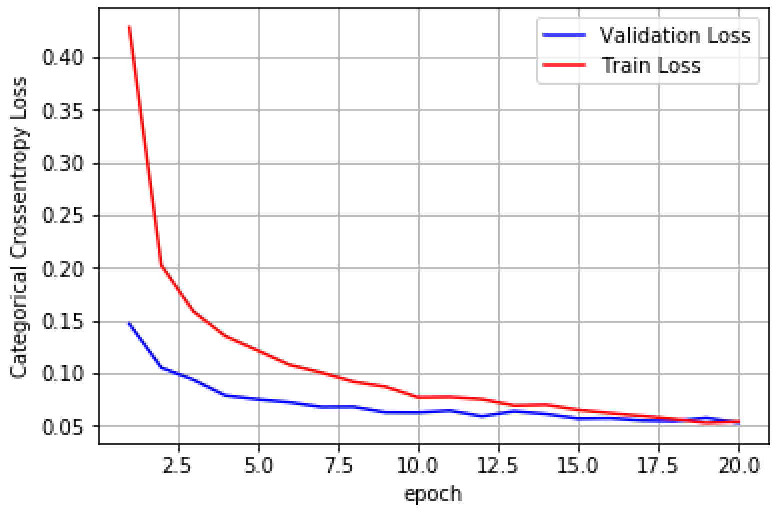
```
Test score: 0.052687030650049566
Test accuracy: 98.35%
```

In [167]:
```
print()
print()
# Plot weight distribution using violin plot
plot_weights(model)

print()
print()

# Plot train and cross validation error
plot_train_cv_loss(trained_model, epochs)
```





On 16th epoch we find that validation error and train error comes together, so best value for epoch is 16-18

With 3-Hidden Layers

In [168]:

```
# Instantiate sequential model
model = Sequential()

# Add 1st hidden layer : dense Layer
dense_layer1 = Dense(512,
                       activation="relu",
                       input_shape=(input_dimension,),
                       kernel_initializer= he_normal(seed=None))
model.add(dense_layer1)

# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add 2nd hidden layer : dense Layer
dense_layer2 = Dense(256,
                       activation="relu",
                       kernel_initializer= he_normal(seed=None))
model.add(dense_layer2)

# Add 3rd hidden layer : dense Layer
dense_layer3 = Dense(128,
                       activation="relu",
                       kernel_initializer= he_normal(seed=None))
model.add(dense_layer3)


# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add output layer : dense Layer
dense_layer4 = Dense(output_dimension, activation='softmax')
model.add(dense_layer4)

# Summary of the model
print("Model Summary: \n")
model.summary()
print()
print()

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Run the model
trained_model = model.fit(x_train, y_train, batch_size = batch_size, epochs = epochs, verbose=1, valid
ation_data=(x_test, y_test))
```

```
Model Summary:

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 512)               401920    
_________________________________________________________________
batch_normalization_6 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 256)               131328    
_________________________________________________________________
dense_9 (Dense)              (None, 128)               32896     
_________________________________________________________________
batch_normalization_7 (Batch (None, 128)               512       
_________________________________________________________________
dropout_6 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290      
=================================================================
Total params: 569,994
Trainable params: 568,714
Non-trainable params: 1,280
_________________________________________________________________


Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 165us/step - loss: 0.4073 - acc: 0.8767 - val_loss: 0.1355 - val_acc: 0.9572
Epoch 2/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.1851 - acc: 0.9446 - val_loss: 0.1001 - val_acc: 0.9683
Epoch 3/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.1462 - acc: 0.9558 - val_loss: 0.0864 - val_acc: 0.9730
Epoch 4/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.1237 - acc: 0.9622 - val_loss: 0.0697 - val_acc: 0.9779
Epoch 5/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.1073 - acc: 0.9665 - val_loss: 0.0728 - val_acc: 0.9766
Epoch 6/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.0998 - acc: 0.9697 - val_loss: 0.0634 - val_acc: 0.9804
Epoch 7/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.0913 - acc: 0.9722 - val_loss: 0.0616 - val_acc: 0.9804
Epoch 8/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.0824 - acc: 0.9744 - val_loss: 0.0672 - val_acc: 0.9793
Epoch 9/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0794 - acc: 0.9755 - val_loss: 0.0582 - val_acc: 0.9817
Epoch 10/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0726 - acc: 0.9768 - val_loss: 0.0648 - val_acc: 0.9804
Epoch 11/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.0670 - acc: 0.9790 - val_loss: 0.0575 - val_acc: 0.9824
Epoch 12/20
60000/60000 [==============================] - 10s 164us/step - loss: 0.0663 - acc: 0.9793 - val_loss: 0.0575 - val_acc: 0.9818
Epoch 13/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.0623 - acc: 0.9805 - val_loss: 0.0586 - val_acc: 0.9822
Epoch 14/20
60000/60000 [==============================] - 10s 167us/step - loss: 0.0583 - acc: 0.9822 - val_loss: 0.0591 - val_acc: 0.9834
Epoch 15/20
60000/60000 [==============================] - 10s 169us/step - loss: 0.0559 - acc: 0.9816 - val_loss: 0.0531 - val_acc: 0.9830
Epoch 16/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.0541 - acc: 0.9827 - val_loss: 0.0531 - val_acc: 0.9846
Epoch 17/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.0524 - acc: 0.9833 - val_loss: 0.0553 - val_acc: 0.9843
Epoch 18/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.0483 - acc: 0.9851 - val_loss: 0.0526 - val_acc: 0.9850
Epoch 19/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.0464 - acc: 0.9847 - val_loss: 0.0554 - val_acc: 0.9839
Epoch 20/20
60000/60000 [==============================] - 12s 194us/step - loss: 0.0441 - acc: 0.9854 - val_loss:
```

```
0.0559 - val_acc: 0.9845
```

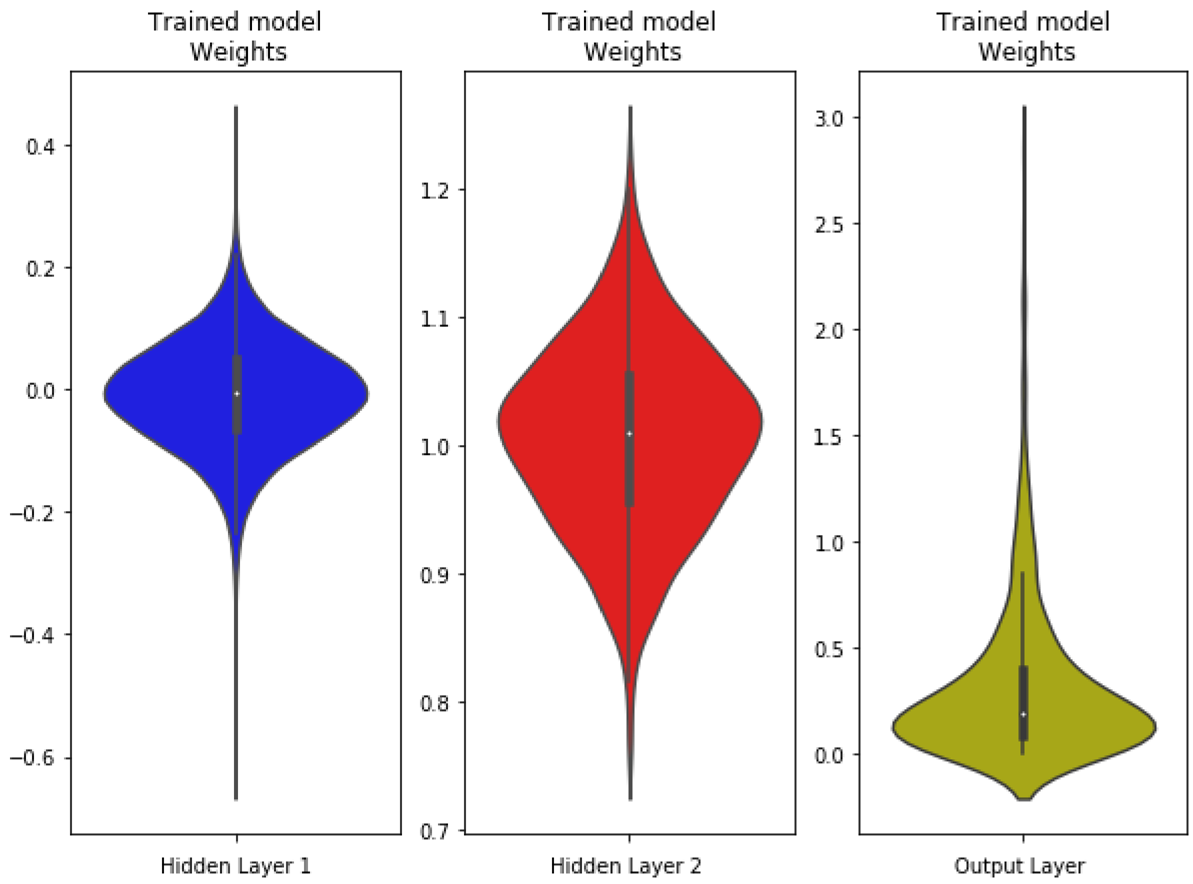
In [169]:
```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy: {0:.2f}%'.format(score[1]*100))
```

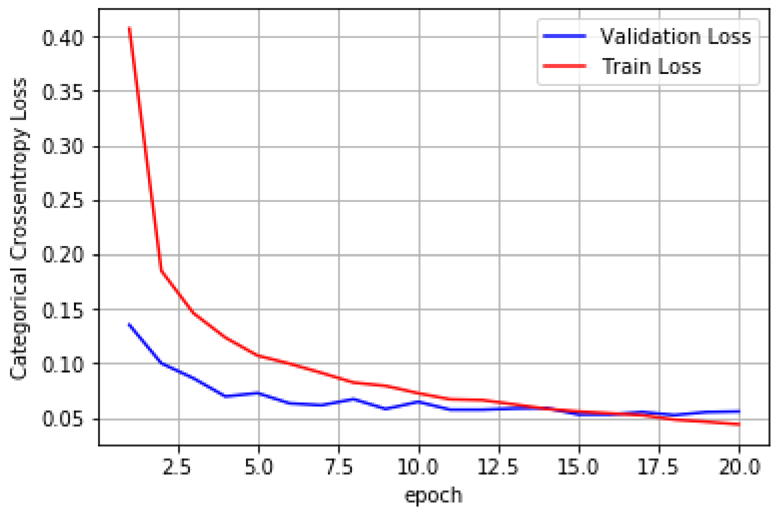
```
Test score: 0.05587759582863073
Test accuracy: 98.45%
```

In [170]:
```
print()
print()
# Plot weight distribution using violin plot
plot_weights(model)

print()
print()

# Plot train and cross validation error
plot_train_cv_loss(trained_model, epochs)
```





On 12th epoch we find that validation error and train error comes together, so best value for epoch is 12-13

With 5-Hidden Layers

In [173]:

```
# Instantiate sequential model
model = Sequential()

# Add 1st hidden layer : dense Layer
dense_layer1 = Dense(624,
                     activation="relu",
                     input_shape=(input_dimension,),
                     kernel_initializer= he_normal(seed=None))
model.add(dense_layer1)

# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add 2nd hidden layer : dense Layer
dense_layer2 = Dense(474,
                     activation="relu",
                     kernel_initializer= he_normal(seed=None))
model.add(dense_layer2)

# Add 3rd hidden layer : dense Layer
dense_layer3 = Dense(324,
                     activation="relu",
                     kernel_initializer= he_normal(seed=None))
model.add(dense_layer3)

# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add 4th hidden layer : dense Layer
dense_layer4 = Dense(174,
                     activation="relu",
                     kernel_initializer= he_normal(seed=None))
model.add(dense_layer4)

# Add 5th hidden layer : dense Layer
dense_layer5 = Dense(24,
                     activation="relu",
                     kernel_initializer= he_normal(seed=None))
model.add(dense_layer5)

# Add batch normalization
model.add(BatchNormalization())

# Add dropout
model.add(Dropout(0.5))

# Add output layer : dense Layer
dense_layer4 = Dense(output_dimension, activation='softmax')
model.add(dense_layer4)

# Summary of the model
print("Model Summary: \n")
model.summary()
print()
print()

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Run the model
trained_model = model.fit(x_train, y_train, batch_size = batch_size, epochs = epochs, verbose=1, valid
ation_data=(x_test, y_test))
```

```
Model Summary:

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 624)               489840
_________________________________________________________________
batch_normalization_8 (Batch (None, 624)               2496
_________________________________________________________________
dropout_7 (Dropout)          (None, 624)               0
_________________________________________________________________
dense_12 (Dense)             (None, 474)               296250
_________________________________________________________________
dense_13 (Dense)             (None, 324)               153900
_________________________________________________________________
batch_normalization_9 (Batch (None, 324)               1296
_________________________________________________________________
dropout_8 (Dropout)          (None, 324)               0
_________________________________________________________________
dense_14 (Dense)             (None, 174)               56550
_________________________________________________________________
dense_15 (Dense)             (None, 24)                4200
_________________________________________________________________
batch_normalization_10 (Batc (None, 24)                96
_________________________________________________________________
dropout_9 (Dropout)          (None, 24)                0
_________________________________________________________________
dense_16 (Dense)             (None, 10)                250
=================================================================
Total params: 1,004,878
Trainable params: 1,002,934
Non-trainable params: 1,944
_________________________________________________________________


Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 241us/step - loss: 0.6557 - acc: 0.8123 - val_loss:
0.1558 - val_acc: 0.9541
Epoch 2/20
60000/60000 [==============================] - 14s 236us/step - loss: 0.2663 - acc: 0.9330 - val_loss:
0.1104 - val_acc: 0.9671
Epoch 3/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.2048 - acc: 0.9481 - val_loss:
0.1047 - val_acc: 0.9709
Epoch 4/20
60000/60000 [==============================] - 13s 213us/step - loss: 0.1788 - acc: 0.9546 - val_loss:
0.0894 - val_acc: 0.9744
Epoch 5/20
60000/60000 [==============================] - 13s 218us/step - loss: 0.1557 - acc: 0.9598 - val_loss:
0.0827 - val_acc: 0.9761
Epoch 6/20
60000/60000 [==============================] - 13s 221us/step - loss: 0.1415 - acc: 0.9635 - val_loss:
0.0789 - val_acc: 0.9774
Epoch 7/20
60000/60000 [==============================] - 15s 247us/step - loss: 0.1301 - acc: 0.9668 - val_loss:
0.0755 - val_acc: 0.9795
Epoch 8/20
60000/60000 [==============================] - 14s 226us/step - loss: 0.1229 - acc: 0.9690 - val_loss:
0.0759 - val_acc: 0.9800
Epoch 9/20
60000/60000 [==============================] - 14s 232us/step - loss: 0.1139 - acc: 0.9709 - val_loss:
0.0766 - val_acc: 0.9800
Epoch 10/20
60000/60000 [==============================] - 14s 235us/step - loss: 0.1072 - acc: 0.9730 - val_loss:
0.0710 - val_acc: 0.9814
Epoch 11/20
60000/60000 [==============================] - 14s 237us/step - loss: 0.0994 - acc: 0.9748 - val_loss:
0.0657 - val_acc: 0.9819
Epoch 12/20
60000/60000 [==============================] - 17s 283us/step - loss: 0.0934 - acc: 0.9757 - val_loss:
0.0725 - val_acc: 0.9811
Epoch 13/20
60000/60000 [==============================] - 17s 288us/step - loss: 0.0907 - acc: 0.9768 - val_loss:
0.0640 - val_acc: 0.9816
Epoch 14/20
60000/60000 [==============================] - 16s 261us/step - loss: 0.0859 - acc: 0.9782 - val_loss:
0.0638 - val_acc: 0.9833
Epoch 15/20
60000/60000 [==============================] - 16s 263us/step - loss: 0.0845 - acc: 0.9778 - val_loss:
0.0672 - val_acc: 0.9811
Epoch 16/20
60000/60000 [==============================] - 18s 300us/step - loss: 0.0789 - acc: 0.9794 - val_loss:
0.0624 - val_acc: 0.9841
Epoch 17/20
60000/60000 [==============================] - 16s 267us/step - loss: 0.0759 - acc: 0.9799 - val_loss:
0.0649 - val_acc: 0.9848
```

```
Epoch 18/20
60000/60000 [==============================] - 16s 273us/step - loss: 0.0765 - acc: 0.9805 - val_loss:
0.0634 - val_acc: 0.9846
Epoch 19/20
60000/60000 [==============================] - 18s 296us/step - loss: 0.0711 - acc: 0.9817 - val_loss:
0.0642 - val_acc: 0.9840
Epoch 20/20
60000/60000 [==============================] - 17s 290us/step - loss: 0.0679 - acc: 0.9824 - val_loss:
0.0593 - val_acc: 0.9844
```

In [174]:

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy: {0:.2f}%'.format(score[1]*100))
```

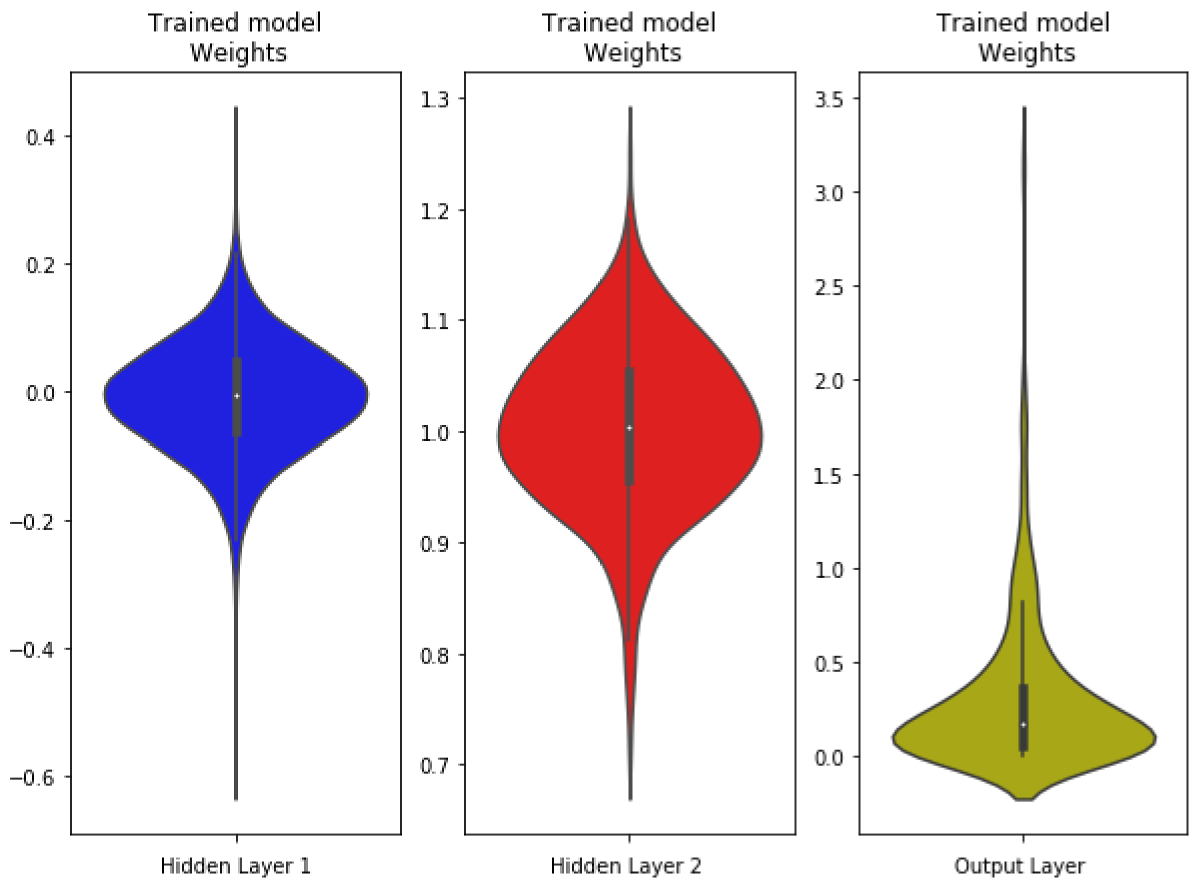
```
Test score: 0.0593049447054018
Test accuracy: 98.44%
```

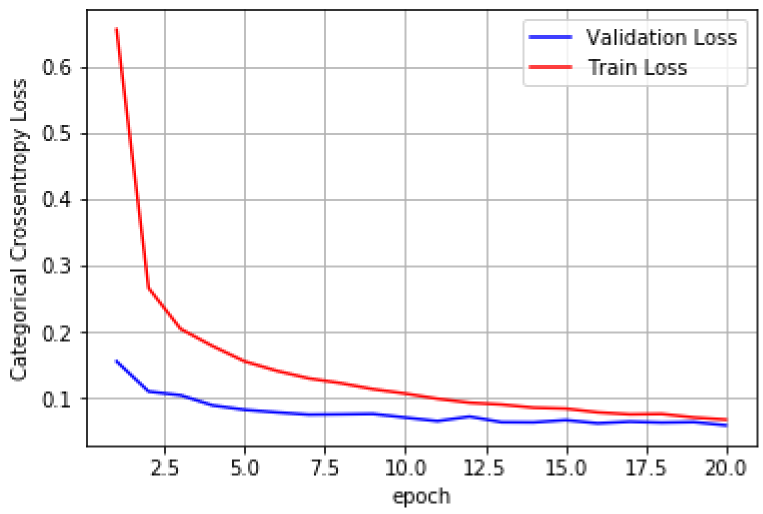
In [175]:

```
print()
print()
# Plot weight distribution using violin plot
plot_weights(model)

print()
print()

# Plot train and cross validation error
plot_train_cv_loss(trained_model, epochs)
```





On 19th epoch we find that validation error and train error comes together, so best value for epoch is 19-20

Observations :

> 1. Tried different MLP architectures on MNIST dataset.
> 2. 'Relu' is used as an activation function to develop MLP.
> 3. 'Adam' is used as an optimizer to develop MLP.
> 4. Introduced batch normalization and dropout in between hidden layers.
> 5. Got 98.35, 98.45 and 98.44 accuracies for 2,3 and 5 hidden layers.