Copy_of_Keras_Mnist

March 4, 2020

0.1 Keras -- MLPs on MNIST

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

```
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]: %matplotlib inline
```

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

```
In [6]: print("Number of training examples :", X_train.shape[0], "and each image is of shape (%
        print("Number of training examples :", X_test.shape[0], "and each image is of shape (%
```

```
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 [8]:
```python
# 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 (%
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%
```

```
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 [9]:
```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 [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 [11]:
```
# 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 [12]: # 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 [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 constr

        # 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_
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_r
# 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 activatio

# 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 [0]:
```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [0]:
```
# start building a model
model = Sequential()

# The model needs to know what input shape it should expect.
# For this reason, the first layer in a Sequential model
# (and only the first, because following layers can do automatic shape inference)
# needs to receive information about its input shape.
# you can use input_shape and input_dim to pass the shape of input
```

```
# output_dim represent the number of nodes need in that layer
# here we have 10 nodes

model.add(Dense(output_dim, input_dim=input_dim, activation='softmax'))
```

In [17]:
```
# Before training a model, you need to configure the learning process, which is done

# It receives three arguments:
# An optimizer. This could be the string identifier of an existing optimizer , https:
# A loss function. This is the objective that the model will try to minimize., https:
# A list of metrics. For any classification problem you will want to set this to metr


# Note: when using the categorical_crossentropy loss, your targets should be in categ
# (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional
# for a 1 at the index corresponding to the class of the sample).

# that is why we converted out labels into vectors

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

# Keras models are trained on Numpy arrays of input data and labels.
# For training a model, you will typically use the  fit function

# fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, val
# validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_
# validation_steps=None)

# fit() function Trains the model for a fixed number of epochs (iterations on a datas

# it returns A History object. Its History.history attribute is a record of training
# metrics values at successive epochs, as well as validation loss values and validati

# https://github.com/openai/baselines/issues/20

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

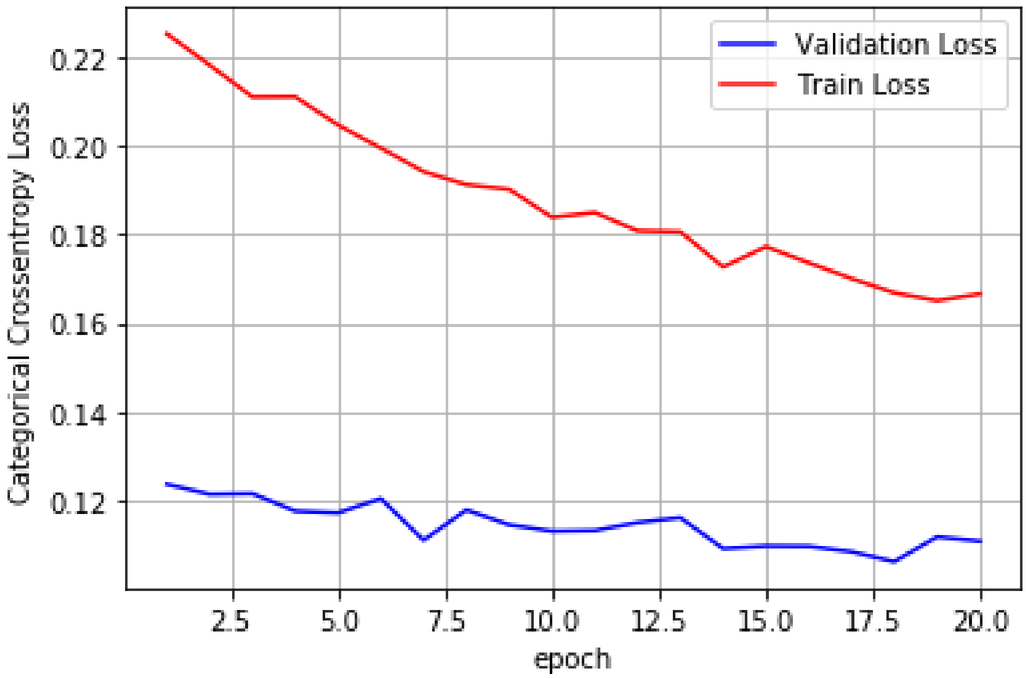
```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/optimizers.py:793: The nam

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_core/python/ops/math_
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

Train on 60000 samples, validate on 10000 samples
Epoch 1/20
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend

60000/60000 [==============================] - 11s 180us/step - loss: 1.2521 - acc: 0.7184 - va
Epoch 2/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.7087 - acc: 0.8454 - val_
Epoch 3/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.5826 - acc: 0.8623 - val_
Epoch 4/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.5221 - acc: 0.8703 - val_
Epoch 5/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4852 - acc: 0.8758 - val_
Epoch 6/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4598 - acc: 0.8803 - val_
Epoch 7/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.4410 - acc: 0.8842 - val_
Epoch 8/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.4263 - acc: 0.8869 - val_
Epoch 9/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4144 - acc: 0.8892 - val_
Epoch 10/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4046 - acc: 0.8916 - val_
Epoch 11/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3962 - acc: 0.8931 - val_
Epoch 12/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3891 - acc: 0.8946 - val_
Epoch 13/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3828 - acc: 0.8958 - val_
Epoch 14/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3772 - acc: 0.8972 - val_
Epoch 15/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3722 - acc: 0.8978 - val_
Epoch 16/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3677 - acc: 0.8991 - val_
Epoch 17/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3637 - acc: 0.8997 - val_
Epoch 18/20
```

```
60000/60000 [==============================] - 1s 24us/step - loss: 0.3599 - acc: 0.9009 - val_
Epoch 19/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3566 - acc: 0.9017 - val_
Epoch 20/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3534 - acc: 0.9025 - val_
```

In [74]:
```python
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')

# 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, 

# 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 

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

```
Test score: 0.335641682690382
Test accuracy: 0.9094
```



MLP + ReLu activation + Adam Optimizer

```
In [31]: # Multilayer perceptron

         model_sigmoid = Sequential()
         model_sigmoid.add(Dense(226, activation='relu', input_shape=(input_dim,)))
         model_sigmoid.add(Dense(132, activation='relu'))
         model_sigmoid.add(Dense(output_dim, activation='softmax'))

         model_sigmoid.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 226)               177410
_________________________________________________________________
dense_14 (Dense)             (None, 132)               29964
_________________________________________________________________
dense_15 (Dense)             (None, 10)                1330
=================================================================
Total params: 208,704
Trainable params: 208,704
Non-trainable params: 0
_________________________________________________________________
```

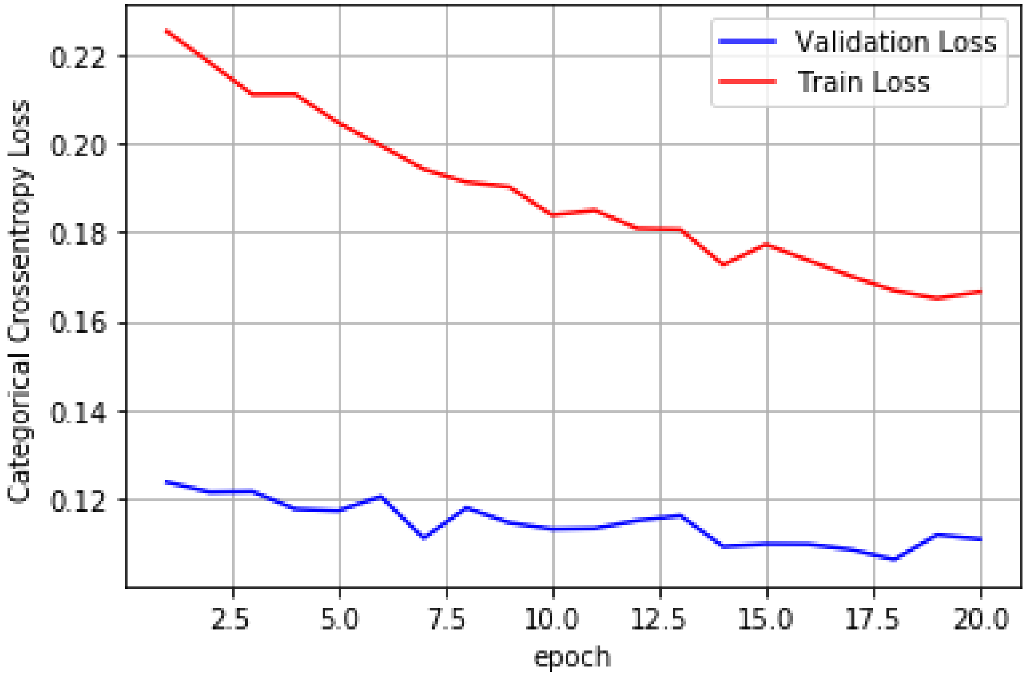
In [32]:
```
model_sigmoid.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['acc

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch,
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.2741 - acc: 0.9215 - val_
Epoch 2/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.1078 - acc: 0.9671 - val_
Epoch 3/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0704 - acc: 0.9785 - val_
Epoch 4/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0525 - acc: 0.9838 - val_
Epoch 5/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0383 - acc: 0.9881 - val_
Epoch 6/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0303 - acc: 0.9904 - val_
Epoch 7/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0231 - acc: 0.9925 - val_
Epoch 8/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0189 - acc: 0.9941 - val_
Epoch 9/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0163 - acc: 0.9944 - val_
Epoch 10/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0138 - acc: 0.9955 - val_
Epoch 11/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0157 - acc: 0.9947 - val_
Epoch 12/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0113 - acc: 0.9965 - val_
Epoch 13/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0087 - acc: 0.9971 - val_
Epoch 14/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0108 - acc: 0.9963 - val_
Epoch 15/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0104 - acc: 0.9962 - val_
Epoch 16/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0060 - acc: 0.9979 - val_
Epoch 17/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0083 - acc: 0.9971 - val_
Epoch 18/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0072 - acc: 0.9976 - val_
Epoch 19/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0071 - acc: 0.9974 - val_
Epoch 20/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0094 - acc: 0.9968 - val_
```

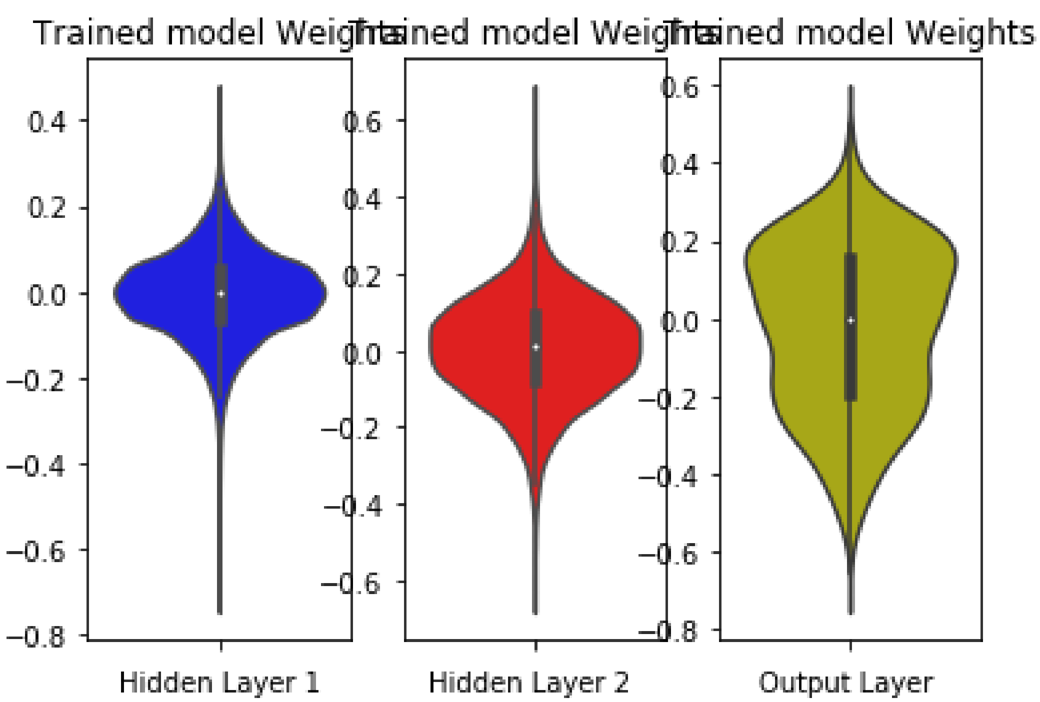
In [73]:
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
```

```
print('Test accuracy:', score[1])

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

# 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, v

# 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 e

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

```
Test score: 0.09268326033384333
Test accuracy: 0.9798
```



In [72]:
```
w_after = model_sigmoid.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 activation + BatchNormalisation + Dropout + ADAM with 2 layers

In [35]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization

from keras.layers import Dropout
from keras.layers.normalization import BatchNormalization
model_drop = Sequential()

model_drop.add(Dense(226, activation='relu', input_shape=(input_dim,), kernel_initiali
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(132, activation='relu', kernel_initializer=RandomNormal(mean=0.0
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 226)               177410
_________________________________________________________________
batch_normalization_3 (Batch (None, 226)               904
_________________________________________________________________
dropout_3 (Dropout)          (None, 226)               0
_________________________________________________________________
dense_17 (Dense)             (None, 132)               29964
_________________________________________________________________
batch_normalization_4 (Batch (None, 132)               528
_________________________________________________________________
dropout_4 (Dropout)          (None, 132)               0
_________________________________________________________________
dense_18 (Dense)             (None, 10)                1330
=================================================================
Total params: 210,136
Trainable params: 209,420
Non-trainable params: 716
_________________________________________________________________
```

In [71]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
```

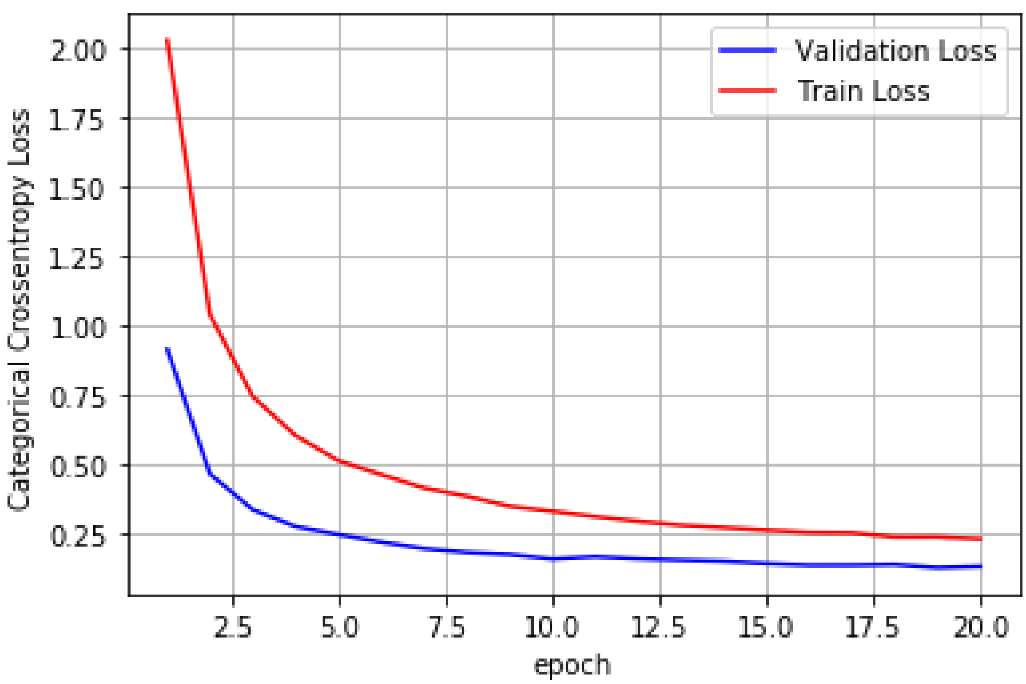
```
history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ve
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.2252 - acc: 0.9416 - va
Epoch 2/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2182 - acc: 0.9431 - val_
Epoch 3/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2109 - acc: 0.9455 - val_
Epoch 4/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.2110 - acc: 0.9453 - val_
Epoch 5/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2046 - acc: 0.9467 - val_
Epoch 6/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1994 - acc: 0.9486 - val_
Epoch 7/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1941 - acc: 0.9502 - val_
Epoch 8/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1912 - acc: 0.9513 - val_
Epoch 9/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1901 - acc: 0.9510 - val_
Epoch 10/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1838 - acc: 0.9525 - val_
Epoch 11/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1849 - acc: 0.9516 - val_
Epoch 12/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1808 - acc: 0.9528 - val_
Epoch 13/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1806 - acc: 0.9520 - val_
Epoch 14/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1726 - acc: 0.9549 - val_
Epoch 15/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1772 - acc: 0.9540 - val_
Epoch 16/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1736 - acc: 0.9540 - val_
Epoch 17/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1700 - acc: 0.9561 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.1668 - acc: 0.9564 - val_
Epoch 19/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.1650 - acc: 0.9571 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 92us/step - loss: 0.1665 - acc: 0.9567 - val_
```

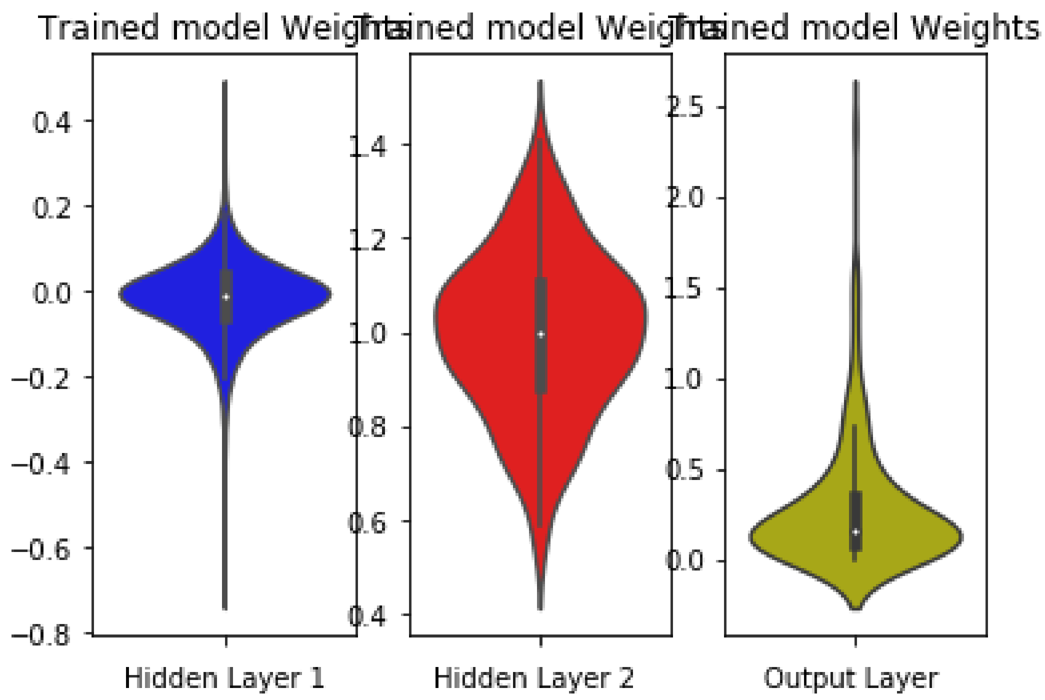
In [70]:
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
```

```
print('Test accuracy:', score[1])

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

# 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, 

# 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 

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

```
Test score: 0.12922746661901474
Test accuracy: 0.9667
```



In [69]:
```
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 activation + BatchNormalisation + Dropout + ADAM with 3 layers

```
In [44]: # https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization

         from keras.layers import Dropout

         model_drop = Sequential()

         model_drop.add(Dense(442, activation='relu', input_shape=(input_dim,), kernel_initial
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(123, activation='relu', kernel_initializer=RandomNormal(mean=0.0
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(82, activation='relu', kernel_initializer=RandomNormal(mean=0.0,
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

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


         model_drop.summary()
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 442)               346970
_________________________________________________________________
batch_normalization_8 (Batch (None, 442)               1768
_________________________________________________________________
dropout_8 (Dropout)          (None, 442)               0
_________________________________________________________________
dense_24 (Dense)             (None, 123)               54489
_________________________________________________________________
batch_normalization_9 (Batch (None, 123)               492
_________________________________________________________________
dropout_9 (Dropout)          (None, 123)               0
_________________________________________________________________
dense_25 (Dense)             (None, 82)                10168
_________________________________________________________________
batch_normalization_10 (Batc (None, 82)                328
_________________________________________________________________
dropout_10 (Dropout)         (None, 82)                0
```

```
_________________________________________________________________
dense_26 (Dense)             (None, 10)                830
=================================================================
Total params: 415,045
Trainable params: 413,751
Non-trainable params: 1,294
_________________________________________________________________
```

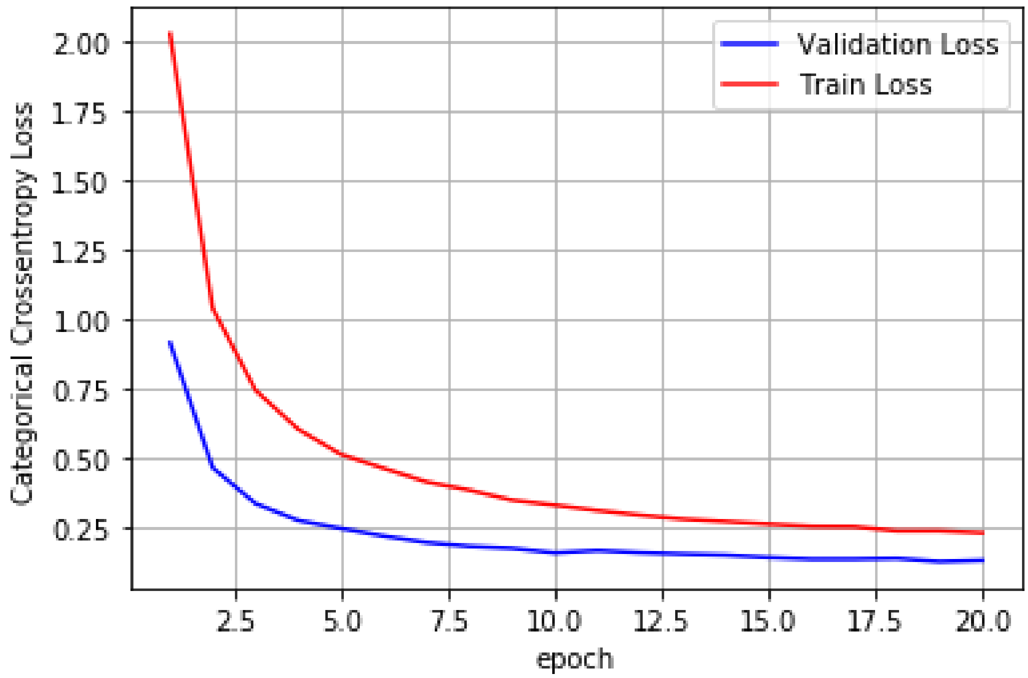
```
In [45]: model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura

         history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ve
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.8982 - acc: 0.7228 - val_
Epoch 2/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.4125 - acc: 0.8783 - val_
Epoch 3/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.3221 - acc: 0.9079 - val_
Epoch 4/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2656 - acc: 0.9241 - val_
Epoch 5/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.2327 - acc: 0.9330 - val_
Epoch 6/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2128 - acc: 0.9402 - val_
Epoch 7/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1942 - acc: 0.9448 - val_
Epoch 8/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1787 - acc: 0.9493 - val_
Epoch 9/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1686 - acc: 0.9521 - val_
Epoch 10/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1577 - acc: 0.9548 - val_
Epoch 11/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.1524 - acc: 0.9563 - val_
Epoch 12/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1422 - acc: 0.9597 - val_
Epoch 13/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1357 - acc: 0.9620 - val_
Epoch 14/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1305 - acc: 0.9627 - val_
Epoch 15/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1242 - acc: 0.9642 - val_
Epoch 16/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1178 - acc: 0.9659 - val_
Epoch 17/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.1110 - acc: 0.9682 - val_
```

```
Epoch 18/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1122 - acc: 0.9675 - val_
Epoch 19/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1022 - acc: 0.9705 - val_
Epoch 20/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1047 - acc: 0.9708 - val_
```

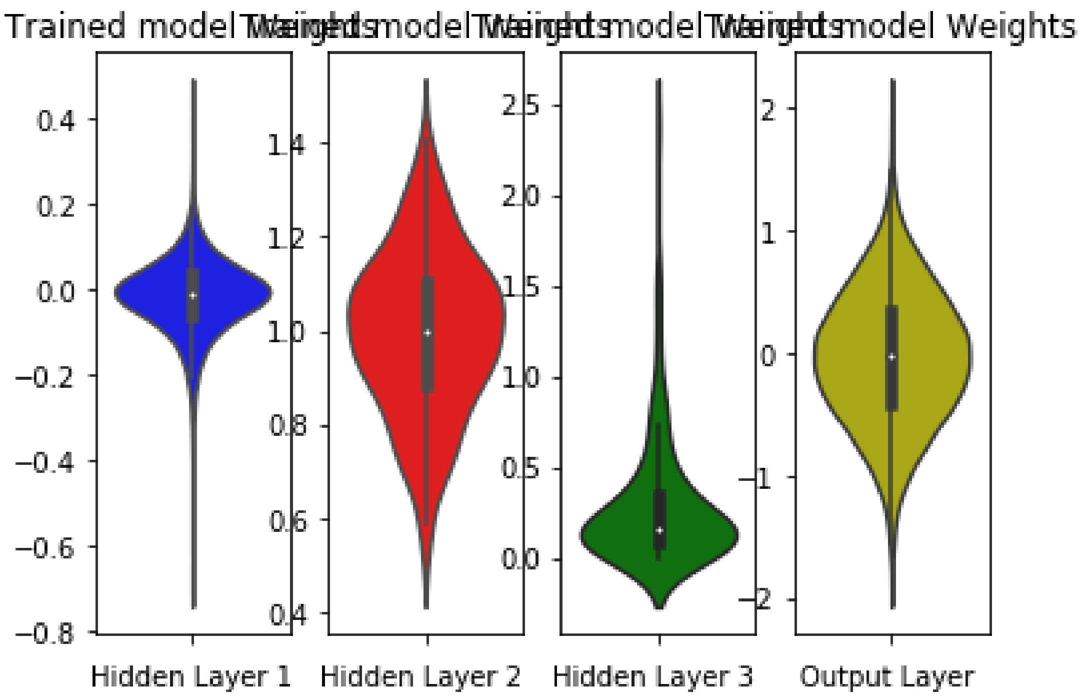
In [66]:
```python
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# 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, 

# 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 

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

```
Test score: 0.12922746661901474
Test accuracy: 0.9667
```



In [65]:
```
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=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()
```



MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 5 layers

```
In [48]: # https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization

         from keras.layers import Dropout

         model_drop = Sequential()

         model_drop.add(Dense(128, activation='relu', input_shape=(input_dim,), kernel_initial
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))
```

```
model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))


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


model_drop.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_27 (Dense)             (None, 128)               100480
_________________________________________________________________
batch_normalization_11 (Batc (None, 128)               512
_________________________________________________________________
dropout_11 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_28 (Dense)             (None, 128)               16512
_________________________________________________________________
batch_normalization_12 (Batc (None, 128)               512
_________________________________________________________________
dropout_12 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_29 (Dense)             (None, 128)               16512
_________________________________________________________________
batch_normalization_13 (Batc (None, 128)               512
_________________________________________________________________
dropout_13 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_30 (Dense)             (None, 128)               16512
_________________________________________________________________
batch_normalization_14 (Batc (None, 128)               512
_________________________________________________________________
dropout_14 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_31 (Dense)             (None, 128)               16512
_________________________________________________________________
batch_normalization_15 (Batc (None, 128)               512
_________________________________________________________________
dropout_15 (Dropout)         (None, 128)               0
```

```
_________________________________________________________________
dense_32 (Dense)             (None, 10)                1290
=================================================================
Total params: 170,378
Trainable params: 169,098
Non-trainable params: 1,280
_________________________________________________________________
```

```
In [49]: model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura

         history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ve
```

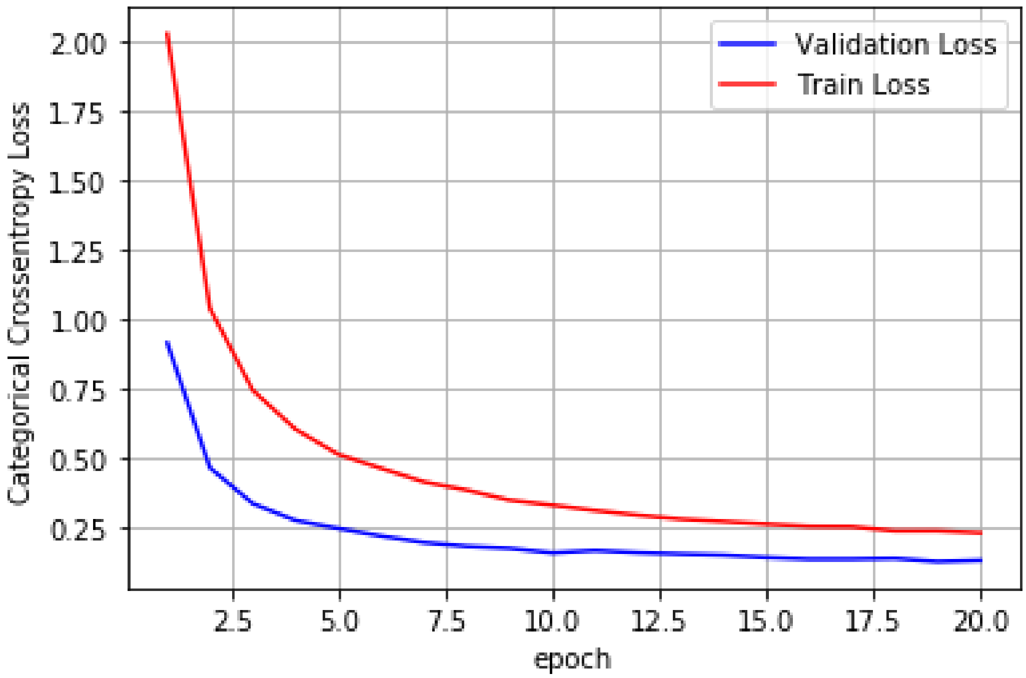
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 124us/step - loss: 2.0302 - acc: 0.3242 - va
Epoch 2/20
60000/60000 [==============================] - 6s 94us/step - loss: 1.0387 - acc: 0.6370 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.7431 - acc: 0.7597 - val_
Epoch 4/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.6024 - acc: 0.8151 - val_
Epoch 5/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.5116 - acc: 0.8481 - val_
Epoch 6/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.4611 - acc: 0.8689 - val_
Epoch 7/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.4127 - acc: 0.8841 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 92us/step - loss: 0.3831 - acc: 0.8939 - val_
Epoch 9/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.3468 - acc: 0.9057 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.3286 - acc: 0.9119 - val_
Epoch 11/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3094 - acc: 0.9169 - val_
Epoch 12/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2925 - acc: 0.9216 - val_
Epoch 13/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.2782 - acc: 0.9266 - val_
Epoch 14/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2696 - acc: 0.9290 - val_
Epoch 15/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2603 - acc: 0.9310 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.2522 - acc: 0.9344 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.2510 - acc: 0.9347 - val_
```

```
Epoch 18/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.2356 - acc: 0.9386 - val_
Epoch 19/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2358 - acc: 0.9390 - val_
Epoch 20/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.2286 - acc: 0.9405 - val_
```

In [64]:

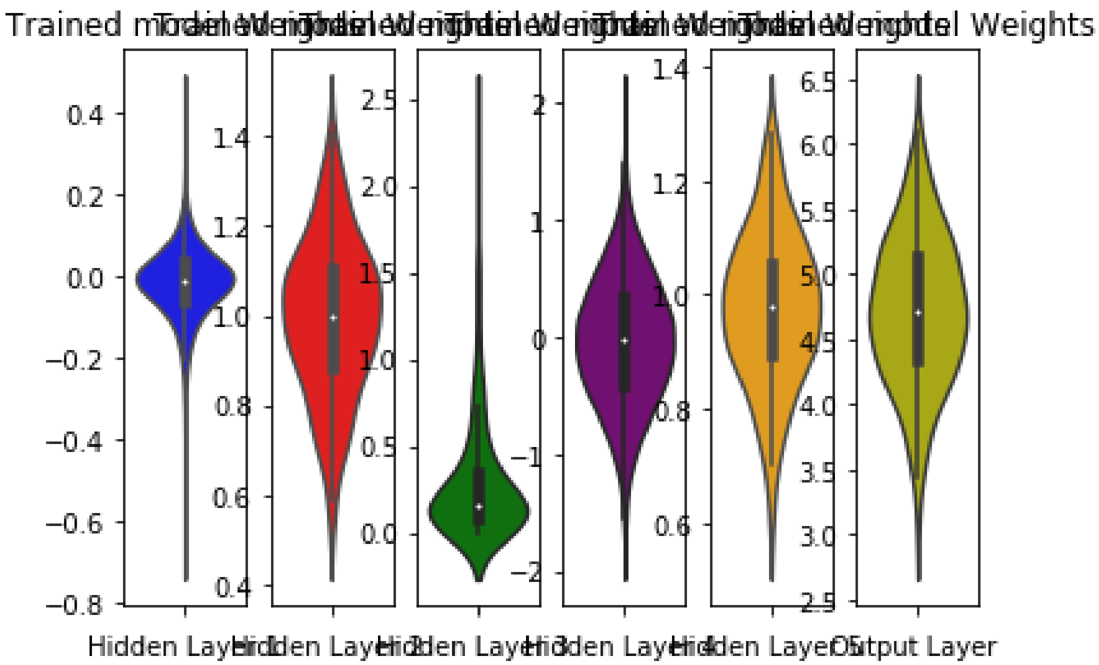
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# 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, 

# 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 

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

```
Test score: 0.12922746661901474
Test accuracy: 0.9667
```



In [63]:
```
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='g')
```

```
        plt.xlabel('Hidden Layer 3 ')

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

        plt.subplot(1, 6, 5)
        plt.title("Trained model Weights")
        ax = sns.violinplot(y=h5_w, color='orange')
        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()
```



Summary

```
In [0]: from prettytable import PrettyTable
        summary = PrettyTable()

In [0]: summary.field_names = ["Model","Dropout","Test Loss","Test Accuracy"]

In [77]: summary.add_row(["MLP + ReLu activation + Adam Optimizer","0.5", "0.092", "0.9798"])
         summary.add_row(["MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 2 
```

```python
summary.add_row(["MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 3 l
summary.add_row(["MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 5 l

print(summary)
```

```
+---------------------------------------------------------------------------+---------+--------
|                                   Model                                   | Dropout | Test Lo
+---------------------------------------------------------------------------+---------+--------
|                   MLP + ReLu activation + Adam Optimizer                  |   0.5   |   0.092
| MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 2 layers |   0.5   |   0.122
| MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 3 layers |   0.5   |   0.122
| MLP + ReLu activation + BatchNormalisation + Dropout + ADAM with 5 layers |   0.5   |   0.129
+---------------------------------------------------------------------------+---------+--------
```