Keras_Mnist

March 18, 2019

0.1 Keras – MLPs on MNIST

```
In [69]: # if you keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow" use
         from keras.utils import np_utils
         from keras.datasets import mnist
         import seaborn as sns
         from keras.initializers import RandomNormal

In [70]: import warnings
         warnings.filterwarnings("ignore")

In [71]: %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 [72]: # the data, shuffled and split between train and test sets
         (X_train, y_train), (X_test, y_test) = mnist.load_data()

In [73]: 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 [74]: # 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 [75]:
```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 [76]:
```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 [77]:
```
# 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 [78]:
```
# 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 [79]:
```python
# 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 [80]:
```python
# 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 const

# 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='gloro
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_
# kernel_constraint=None, bias_constraint=None)

# Dense implements the operation: output = activation(dot(input, kernel) + bias) wher
# activation is the element-wise activation function passed as the activation argumen
# 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 activati

# 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 [81]:
```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [82]:
```
# 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 [83]:
```
# 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
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 36us/step - loss: 1.2915 - acc: 0.6926 - val_
Epoch 2/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.7156 - acc: 0.8402 - val_
Epoch 3/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.5865 - acc: 0.8598 - val_
Epoch 4/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.5246 - acc: 0.8691 - val_
Epoch 5/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.4871 - acc: 0.8750 - val_
```

```
Epoch 6/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.4612 - acc: 0.8795 - val_
Epoch 7/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.4420 - acc: 0.8835 - val_
Epoch 8/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.4272 - acc: 0.8863 - val_
Epoch 9/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.4151 - acc: 0.8889 - val_
Epoch 10/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.4052 - acc: 0.8906 - val_
Epoch 11/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3968 - acc: 0.8923 - val_
Epoch 12/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3895 - acc: 0.8938 - val_
Epoch 13/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3832 - acc: 0.8953 - val_
Epoch 14/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3775 - acc: 0.8965 - val_
Epoch 15/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3725 - acc: 0.8979 - val_
Epoch 16/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3680 - acc: 0.8992 - val_
Epoch 17/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3639 - acc: 0.9001 - val_
Epoch 18/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3602 - acc: 0.9012 - val_
Epoch 19/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3568 - acc: 0.9019 - val_
Epoch 20/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.3537 - acc: 0.9028 - val_
```

In [84]:
```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: 0.3358456688821316
Test accuracy: 0.9079


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

MLP + Sigmoid activation + SGDOptimizer

In [85]:
```
# Multilayer perceptron

model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shape=(input_dim,)))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_61 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_62 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_63 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [86]:
```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accu

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 [==============================] - 4s 66us/step - loss: 2.2698 - acc: 0.2236 - val_
Epoch 2/20
60000/60000 [==============================] - 3s 50us/step - loss: 2.1756 - acc: 0.4934 - val_
Epoch 3/20
60000/60000 [==============================] - 3s 50us/step - loss: 2.0566 - acc: 0.6066 - val_
Epoch 4/20
60000/60000 [==============================] - 3s 50us/step - loss: 1.8853 - acc: 0.6513 - val_
Epoch 5/20
60000/60000 [==============================] - 3s 51us/step - loss: 1.6610 - acc: 0.6828 - val_
Epoch 6/20
60000/60000 [==============================] - 3s 50us/step - loss: 1.4197 - acc: 0.7165 - val_
Epoch 7/20
60000/60000 [==============================] - 3s 51us/step - loss: 1.2065 - acc: 0.7502 - val_
Epoch 8/20
60000/60000 [==============================] - 3s 54us/step - loss: 1.0401 - acc: 0.7756 - val_
Epoch 9/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.9144 - acc: 0.7946 - val_
Epoch 10/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.8201 - acc: 0.8101 - val_
Epoch 11/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.7475 - acc: 0.8224 - val_
Epoch 12/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.6904 - acc: 0.8322 - val_
Epoch 13/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.6447 - acc: 0.8397 - val_
Epoch 14/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.6075 - acc: 0.8461 - val_
Epoch 15/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.5766 - acc: 0.8517 - val_
Epoch 16/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.5506 - acc: 0.8560 - val_
Epoch 17/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.5284 - acc: 0.8614 - val_
Epoch 18/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.5093 - acc: 0.8649 - val_
Epoch 19/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.4925 - acc: 0.8691 - val_
Epoch 20/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.4779 - acc: 0.8720 - val_
```

In [87]:
```
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, 

# 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: 0.4539696149110794
Test accuracy: 0.8789
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

In [88]:
```
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()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

MLP + Sigmoid activation + ADAM

In [89]:
```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shape=(input_dim,)))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()

model_sigmoid.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['ac

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_64 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_65 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_66 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.5393 - acc: 0.8587 - val_
Epoch 2/20
60000/60000 [==============================] - 4s 75us/step - loss: 0.2208 - acc: 0.9353 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.1602 - acc: 0.9524 - val_
Epoch 4/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.1232 - acc: 0.9632 - val_
Epoch 5/20
```

```
60000/60000 [==============================] - 5s 77us/step - loss: 0.0966 - acc: 0.9718 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0768 - acc: 0.9770 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0622 - acc: 0.9812 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0512 - acc: 0.9848 - val_
Epoch 9/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0402 - acc: 0.9881 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0333 - acc: 0.9908 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0264 - acc: 0.9927 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0210 - acc: 0.9944 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0169 - acc: 0.9959 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0137 - acc: 0.9968 - val_
Epoch 15/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0111 - acc: 0.9974 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0089 - acc: 0.9980 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0067 - acc: 0.9987 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0056 - acc: 0.9989 - val_
Epoch 19/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0042 - acc: 0.9992 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0045 - acc: 0.9987 - val_
```

In [90]:
```python
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, 

# 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: 0.08765230814126844
Test accuracy: 0.9772
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [91]: 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()
```

```
<IPython.core.display.Javascript object>
```

<IPython.core.display.HTML object>

MLP + ReLU +SGD

```
In [92]: # Multilayer perceptron

         # https://arxiv.org/pdf/1707.09725.pdf#page=95
         # for relu layers
         # If we sample weights from a normal distribution N(0,) we satisfy this condition wit
         # h1 =>  =(2/(fan_in) = 0.062  => N(0,) = N(0,0.062)
         # h2 =>  =(2/(fan_in) = 0.125  => N(0,) = N(0,0.125)
         # out =>  =(2/(fan_in+1) = 0.120  => N(0,) = N(0,0.120)

         model_relu = Sequential()
         model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initial
         model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0
         model_relu.add(Dense(output_dim, activation='softmax'))

         model_relu.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_67 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_68 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_69 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

```
In [93]: model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accura

         history = model_relu.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 [==============================] - 4s 72us/step - loss: 0.7704 - acc: 0.7838 - val_
Epoch 2/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.3508 - acc: 0.9014 - val_
Epoch 3/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.2871 - acc: 0.9194 - val_
Epoch 4/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.2525 - acc: 0.9293 - val_
```

```
Epoch 5/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.2288 - acc: 0.9355 - val_
Epoch 6/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.2110 - acc: 0.9410 - val_
Epoch 7/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.1965 - acc: 0.9442 - val_
Epoch 8/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1843 - acc: 0.9483 - val_
Epoch 9/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1734 - acc: 0.9514 - val_
Epoch 10/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1641 - acc: 0.9542 - val_
Epoch 11/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1559 - acc: 0.9568 - val_
Epoch 12/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1483 - acc: 0.9584 - val_
Epoch 13/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1415 - acc: 0.9604 - val_
Epoch 14/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1356 - acc: 0.9622 - val_
Epoch 15/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1299 - acc: 0.9639 - val_
Epoch 16/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1247 - acc: 0.9653 - val_
Epoch 17/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1200 - acc: 0.9668 - val_
Epoch 18/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1153 - acc: 0.9677 - val_
Epoch 19/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1112 - acc: 0.9690 - val_
Epoch 20/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1073 - acc: 0.9703 - val_
```

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

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

# 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: 0.12111208859682084
Test accuracy: 0.964


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

In [95]:
```
w_after = model_relu.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()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

MLP + ReLU + ADAM

```
In [96]: model_relu = Sequential()
         model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initiali
         model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0
         model_relu.add(Dense(output_dim, activation='softmax'))

         print(model_relu.summary())

         model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_70 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_71 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_72 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2296 - acc: 0.9312 - val_
Epoch 2/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0876 - acc: 0.9735 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0556 - acc: 0.9827 - val_
Epoch 4/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0390 - acc: 0.9875 - val_
Epoch 5/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0271 - acc: 0.9915 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0201 - acc: 0.9938 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0186 - acc: 0.9940 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0142 - acc: 0.9954 - val_
Epoch 9/20
```

```
60000/60000 [==============================] - 5s 80us/step - loss: 0.0138 - acc: 0.9954 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0104 - acc: 0.9964 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0117 - acc: 0.9962 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0085 - acc: 0.9970 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0121 - acc: 0.9961 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0086 - acc: 0.9970 - val_
Epoch 15/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0082 - acc: 0.9972 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0088 - acc: 0.9969 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0061 - acc: 0.9980 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0055 - acc: 0.9983 - val_
Epoch 19/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0068 - acc: 0.9980 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0097 - acc: 0.9965 - val_
```

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

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

# 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: 0.08695607365363076
Test accuracy: 0.981


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

```
In [98]: w_after = model_relu.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()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

MLP + Batch-Norm on hidden Layers + AdamOptimizer </2>

```
In [99]: # Multilayer perceptron
```

```python
# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,) we satisfy this condition wit
# h1 =>  =(2/(ni+ni+1) = 0.039  => N(0,) = N(0,0.039)
# h2 =>  =(2/(ni+ni+1) = 0.055  => N(0,) = N(0,0.055)
# h1 =>  =(2/(ni+ni+1) = 0.120  => N(0,) = N(0,0.120)

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_ini
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal(mean
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_73 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_9 (Batch (None, 512)               2048
_________________________________________________________________
dense_74 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_10 (Batc (None, 128)               512
_________________________________________________________________
dense_75 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [100]:
```python
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accu

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.3040 - acc: 0.9098 - val
Epoch 2/20
```

```
60000/60000 [==============================] - 5s 87us/step - loss: 0.1766 - acc: 0.9480 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1379 - acc: 0.9597 - val_
Epoch 4/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1140 - acc: 0.9665 - val_
Epoch 5/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0951 - acc: 0.9708 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0818 - acc: 0.9749 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0700 - acc: 0.9787 - val_
Epoch 8/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0580 - acc: 0.9820 - val_
Epoch 9/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0501 - acc: 0.9838 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0440 - acc: 0.9860 - val_
Epoch 11/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0394 - acc: 0.9872 - val_
Epoch 12/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0325 - acc: 0.9900 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0280 - acc: 0.9911 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0255 - acc: 0.9921 - val_
Epoch 15/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0244 - acc: 0.9920 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0218 - acc: 0.9932 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0211 - acc: 0.9932 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0201 - acc: 0.9935 - val_
Epoch 19/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0180 - acc: 0.9939 - val_
Epoch 20/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0169 - acc: 0.9941 - val_
```

In [101]:
```
score = model_batch.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: 0.09116820345263986
Test accuracy: 0.9743


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

In [102]:
```
w_after = model_batch.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()
```

```
<IPython.core.display.Javascript object>



<IPython.core.display.HTML object>
```

5. MLP + Dropout + AdamOptimizer

```
In [103]: # https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalizatio

        from keras.layers import Dropout

        model_drop = Sequential()

        model_drop.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_ini
        model_drop.add(BatchNormalization())
        model_drop.add(Dropout(0.5))

        model_drop.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal(mean
        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_76 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_11 (Batc (None, 512)               2048
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_77 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_12 (Batc (None, 128)               512
_________________________________________________________________
dropout_6 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_78 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
```

```
Non-trainable params: 1,280
_________________________________________________________________
```

```
In [104]: model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accu

          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 123us/step - loss: 0.6731 - acc: 0.7937 - va
Epoch 2/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.4348 - acc: 0.8666 - val_
Epoch 3/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.3857 - acc: 0.8836 - val_
Epoch 4/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.3588 - acc: 0.8920 - val_
Epoch 5/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3374 - acc: 0.8982 - val_
Epoch 6/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.3259 - acc: 0.9021 - val_
Epoch 7/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.3059 - acc: 0.9061 - val_
Epoch 8/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.2948 - acc: 0.9112 - val_
Epoch 9/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.2832 - acc: 0.9154 - val_
Epoch 10/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2676 - acc: 0.9192 - val_
Epoch 11/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2573 - acc: 0.9230 - val_
Epoch 12/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2500 - acc: 0.9240 - val_
Epoch 13/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.2348 - acc: 0.9287 - va
Epoch 14/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.2261 - acc: 0.9322 - va
Epoch 15/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.2195 - acc: 0.9334 - va
Epoch 16/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.2053 - acc: 0.9375 - va
Epoch 17/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.1955 - acc: 0.9411 - va
Epoch 18/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.1906 - acc: 0.9427 - va
Epoch 19/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1828 - acc: 0.9460 - va
Epoch 20/20
```

```
60000/60000 [==============================] - 6s 97us/step - loss: 0.1720 - acc: 0.9483 - val_
```

```
In [105]: 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: 0.10899472893364727
Test accuracy: 0.9675
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [106]: 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()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Hyper-parameter tuning of Keras models using Sklearn

```
In [39]: from keras.optimizers import Adam,RMSprop,SGD
         def best_hyperparameters(activ):

             model = Sequential()
             model.add(Dense(512, activation=activ, input_shape=(input_dim,), kernel_initialize
             model.add(Dense(128, activation=activ, kernel_initializer=RandomNormal(mean=0.0, s
             model.add(Dense(output_dim, activation='softmax'))


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

             return model
```

```
In [40]: # https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models

         activ = ['sigmoid','relu']

         from keras.wrappers.scikit_learn import KerasClassifier
         from sklearn.model_selection import GridSearchCV

         model = KerasClassifier(build_fn=best_hyperparameters, epochs=nb_epoch, batch_size=ba
         param_grid = dict(activ=activ)

         # if you are using CPU
         # grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
         # if you are using GPU dont use the n_jobs parameter
```

```
        grid = GridSearchCV(estimator=model, param_grid=param_grid)
        grid_result = grid.fit(X_train, Y_train)
```

In [41]:
```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.976000 using {'activ': 'relu'}
0.975467 (0.001158) with: {'activ': 'sigmoid'}
0.976000 (0.001809) with: {'activ': 'relu'}
```

6. MLP + Dropout + AdamOptimizer for 2 hidden layers

In [219]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalizatio
from keras.layers import Dropout

def MLP(layers_perceptron_count, BN, dropout): #activation = 'relu', varying no. of
    #from keras.models import Sequential
    global model_BN_Dropout
    model_BN_Dropout = Sequential()
    for i, perceptrons in enumerate(layers_perceptron_count):

        #Taking 256 perceptrons for H1.
        model_BN_Dropout.add(Dense(perceptrons, activation='relu', input_shape=(inpu

        if BN:
            model_BN_Dropout.add(BatchNormalization())

        if dropout:
            model_BN_Dropout.add(Dropout(dropout))

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

    return model_BN_Dropout
```

In [220]:
```
#using ADAM optimizer , categorical_crossentropy as loss function and accuracy as sc
def compile_and_history():
    model_BN_Dropout.compile(optimizer='adam', loss='categorical_crossentropy', metr

    history = model_BN_Dropout.fit(X_train, Y_train, batch_size=batch_size, epochs=nb
```

In [221]:
```
def elbow_curve():
    score = model_BN_Dropout.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))

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

In [222]:
```
def layer_weights_plots():
    w_after = model_BN_Dropout.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()
```

In [223]:
```
#Training 2 hidden layer network with H1=364, H2=96, BatchNormalization and dropout
MLP(layers_perceptron_count=[364,96], BN=True, dropout=0.5).summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_169 (Dense)            (None, 364)               285740
_________________________________________________________________
batch_normalization_53 (Batc (None, 364)               1456
_________________________________________________________________
```

```
dropout_47 (Dropout)         (None, 364)              0
_________________________________________________________________
dense_170 (Dense)            (None, 96)               35040
_________________________________________________________________
batch_normalization_54 (Batc (None, 96)               384
_________________________________________________________________
dropout_48 (Dropout)         (None, 96)               0
_________________________________________________________________
dense_171 (Dense)            (None, 10)               970
=================================================================
Total params: 323,590
Trainable params: 322,670
Non-trainable params: 920
_________________________________________________________________
```

```
In [225]: compile_and_history()
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.4378 - acc: 0.8686 - val
Epoch 2/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2184 - acc: 0.9345 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.1717 - acc: 0.9473 - val_
Epoch 4/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1496 - acc: 0.9555 - val_
Epoch 5/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1329 - acc: 0.9596 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.1205 - acc: 0.9630 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.1108 - acc: 0.9662 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1037 - acc: 0.9681 - val_
Epoch 9/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0975 - acc: 0.9709 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0944 - acc: 0.9718 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0860 - acc: 0.9734 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0849 - acc: 0.9733 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0853 - acc: 0.9737 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0777 - acc: 0.9758 - val_
Epoch 15/20
```

```
60000/60000 [==============================] - 5s 86us/step - loss: 0.0749 - acc: 0.9763 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0697 - acc: 0.9784 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0688 - acc: 0.9786 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0708 - acc: 0.9776 - val_
Epoch 19/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0640 - acc: 0.9804 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0631 - acc: 0.9799 - val_
```

In [226]:
```
elbow_curve()
```

```
Test score: 0.05720752552064368
Test accuracy: 0.9828
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

In [227]:
```
layer_weights_plots()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

7. MLP + Dropout + AdamOptimizer for 3 hidden layers

In [228]:
```
#Training 3 hidden layer network with H1=364, H2=256, H3=128 BatchNormalization and 
MLP(layers_perceptron_count=[364,256,128], BN=True, dropout=0.5).summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_175 (Dense)            (None, 364)               285740
_________________________________________________________________
batch_normalization_57 (Batc (None, 364)               1456
_________________________________________________________________
dropout_51 (Dropout)         (None, 364)               0
_________________________________________________________________
dense_176 (Dense)            (None, 256)               93440
_________________________________________________________________
```

```
batch_normalization_58 (Batc (None, 256)               1024
_________________________________________________________________
dropout_52 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_177 (Dense)            (None, 128)               32896
_________________________________________________________________
batch_normalization_59 (Batc (None, 128)               512
_________________________________________________________________
dropout_53 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_178 (Dense)            (None, 10)                1290
=================================================================
Total params: 416,358
Trainable params: 414,862
Non-trainable params: 1,496
_________________________________________________________________
```

```
In [229]: compile_and_history()
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.5142 - acc: 0.8429 - va
Epoch 2/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.2362 - acc: 0.9306 - val
Epoch 3/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.1904 - acc: 0.9428 - val
Epoch 4/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1632 - acc: 0.9514 - val
Epoch 5/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.1478 - acc: 0.9568 - val
Epoch 6/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.1315 - acc: 0.9609 - val
Epoch 7/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1232 - acc: 0.9634 - val
Epoch 8/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1181 - acc: 0.9655 - val
Epoch 9/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1128 - acc: 0.9664 - val
Epoch 10/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.1051 - acc: 0.9685 - val
Epoch 11/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0996 - acc: 0.9701 - val
Epoch 12/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0939 - acc: 0.9715 - val
Epoch 13/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0911 - acc: 0.9727 - val
Epoch 14/20
```

```
60000/60000 [==============================] - 7s 116us/step - loss: 0.0862 - acc: 0.9740 - va
Epoch 15/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0814 - acc: 0.9753 - va
Epoch 16/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0820 - acc: 0.9749 - va
Epoch 17/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0780 - acc: 0.9765 - va
Epoch 18/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0797 - acc: 0.9755 - va
Epoch 19/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0716 - acc: 0.9783 - va
Epoch 20/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.0703 - acc: 0.9784 - va
```

```
In [230]: elbow_curve()
```

```
Test score: 0.058362318481178954
Test accuracy: 0.9833


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

```
In [231]: layer_weights_plots()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

8. MLP + Dropout + AdamOptimizer for 5 hidden layers

```
In [236]: #Training 5 hidden layer network with H1=364, H2=256, H3=128, H4=64, H5=32 BatchNorm
          MLP(layers_perceptron_count=[512,256,128,64,32], BN=True, dropout=0.5).summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_185 (Dense)            (None, 512)               401920
_________________________________________________________________
batch_normalization_65 (Batc (None, 512)               2048
_________________________________________________________________
dropout_59 (Dropout)         (None, 512)               0
_________________________________________________________________
```

```
dense_186 (Dense)            (None, 256)              131328
_________________________________________________________________
batch_normalization_66 (Batc (None, 256)              1024
_________________________________________________________________
dropout_60 (Dropout)         (None, 256)              0
_________________________________________________________________
dense_187 (Dense)            (None, 128)              32896
_________________________________________________________________
batch_normalization_67 (Batc (None, 128)              512
_________________________________________________________________
dropout_61 (Dropout)         (None, 128)              0
_________________________________________________________________
dense_188 (Dense)            (None, 64)               8256
_________________________________________________________________
batch_normalization_68 (Batc (None, 64)               256
_________________________________________________________________
dropout_62 (Dropout)         (None, 64)               0
_________________________________________________________________
dense_189 (Dense)            (None, 32)               2080
_________________________________________________________________
batch_normalization_69 (Batc (None, 32)               128
_________________________________________________________________
dropout_63 (Dropout)         (None, 32)               0
_________________________________________________________________
dense_190 (Dense)            (None, 10)               330
=================================================================
Total params: 580,778
Trainable params: 578,794
Non-trainable params: 1,984
_________________________________________________________________
```

```
In [237]: compile_and_history()
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 13s 217us/step - loss: 1.0597 - acc: 0.6676 - va
Epoch 2/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.4175 - acc: 0.8939 - val
Epoch 3/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.3170 - acc: 0.9236 - val
Epoch 4/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.2778 - acc: 0.9354 - val
Epoch 5/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.2451 - acc: 0.9424 - val
Epoch 6/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.2292 - acc: 0.9473 - va
Epoch 7/20
```

```
60000/60000 [==============================] - 10s 170us/step - loss: 0.2059 - acc: 0.9520 - va
Epoch 8/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.1929 - acc: 0.9554 - val
Epoch 9/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.1837 - acc: 0.9584 - val
Epoch 10/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.1773 - acc: 0.9592 - val
Epoch 11/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.1677 - acc: 0.9619 - va
Epoch 12/20
60000/60000 [==============================] - 10s 164us/step - loss: 0.1584 - acc: 0.9637 - va
Epoch 13/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.1545 - acc: 0.9658 - va
Epoch 14/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.1463 - acc: 0.9660 - val
Epoch 15/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.1459 - acc: 0.9674 - val
Epoch 16/20
60000/60000 [==============================] - 9s 157us/step - loss: 0.1363 - acc: 0.9687 - val
Epoch 17/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.1332 - acc: 0.9698 - val
Epoch 18/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.1334 - acc: 0.9703 - val
Epoch 19/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.1231 - acc: 0.9718 - val
Epoch 20/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.1206 - acc: 0.9727 - val
```

```
In [238]: elbow_curve()
```

```
Test score: 0.08220580408470705
Test accuracy: 0.9814
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [239]: layer_weights_plots()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

0.2 [9] Conclusions:

1. MLP for 2 hidden layers achieves Test accuracy: 0.9828.
2. MLP for 3 hidden layers achieves Test accuracy: 0.9833 which is marginally better than 2 layer MLP.
3. MLP for 5 hidden layers achieves Test accuracy: 0.9814 which is worse than both 2 layer and 3 layer MLP.

In [9]:
```
from prettytable import PrettyTable

x = PrettyTable()

x.field_names = ["Model", "Layers", "Activation","Optimizer", "BN", "Dropout", "Test Ac

x.add_row(["Softmax", "0                ", "Sigmoid", "SGDOptimizer", "False", "NA", 0
x.add_row(["MLP", "2 (512,128)          ", "Sigmoid", "SGDOptimizer", "False", "NA", 0
x.add_row(["MLP", "2 (512,128)          ", "Sigmoid", "ADAM", "False", "NA", 0.9772])
x.add_row(["MLP", "2 (512,128)          ", "RELU", "SGDOptimizer", "False", "NA", 0.964
x.add_row(["MLP", "2 (512,128)          ", "RELU", "ADAM", "False", "NA", 0.981])
x.add_row(["MLP", "2 (512,128)          ", "RELU", "ADAM", "True", "NA", 0.9743])
x.add_row(["MLP", "2 (512,128)          ", "RELU", "ADAM", "False", "0.5", 0.9743])
x.add_row(["MLP", "2 (364,96)           ", "RELU", "ADAM", "True", 0.5, 0.9828])
x.add_row(["MLP", "3 (364,256,128)      ", "RELU", "ADAM", "True", 0.5, 0.9833])
x.add_row(["MLP", "5 (512,256,128,64,32)", "RELU", "ADAM", "True", 0.5, 0.9814])

print(x)
```

```
+---------+----------------------+------------+--------------+-------+---------+-------------
|  Model  |        Layers        | Activation |  Optimizer   |   BN  | Dropout | Test Accurac
+---------+----------------------+------------+--------------+-------+---------+-------------
| Softmax |    0                 |  Sigmoid   | SGDOptimizer | False |    NA   |     0.9026
|   MLP   | 2 (512,128)          |  Sigmoid   | SGDOptimizer | False |    NA   |     0.8789
|   MLP   | 2 (512,128)          |  Sigmoid   |     ADAM     | False |    NA   |     0.9772
|   MLP   | 2 (512,128)          |    RELU    | SGDOptimizer | False |    NA   |     0.964
|   MLP   | 2 (512,128)          |    RELU    |     ADAM     | False |    NA   |     0.981
|   MLP   | 2 (512,128)          |    RELU    |     ADAM     |  True |    NA   |     0.9743
|   MLP   | 2 (512,128)          |    RELU    |     ADAM     | False |   0.5   |     0.9743
|   MLP   | 2 (364,96)           |    RELU    |     ADAM     |  True |   0.5   |     0.9828
|   MLP   | 3 (364,256,128)      |    RELU    |     ADAM     |  True |   0.5   |     0.9833
|   MLP   | 5 (512,256,128,64,32) |    RELU    |     ADAM     |  True |   0.5   |     0.9814
+---------+----------------------+------------+--------------+-------+---------+-------------
```

In [ ]:
```
0.9675
```