Keras -- MLPs on MNIST

```
# if you keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
from keras.initializers import he_normal
from keras.layers.normalization import BatchNormalization
from keras.layers import Dropout
```

```
%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()
```

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

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

```
print("Number of training examples :", X_train.shape[0], "and each image is of
print("Number of test 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 test examples : 10000 and each image is of shape (28, 28)
```

```
# 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])
```

```
# after converting the input images from 3d to 2d vectors

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

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

```
# 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]
```

```
# 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
# X => (X - Xmin)/(Xmax-Xmin) = X/255

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

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

```
 0.58823529 0.10588235 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
 0.99215686 0.81176471 0.00784314 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.15294118 0.58039216
 0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
 0.99215686 0.78823529 0.30588235 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.09019608 0.25882353 0.83529412 0.99215686
 0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.07058824 0.67058824
 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
 0.31372549 0.03529412 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
 0.99215686 0.95686275 0.52156863 0.04313725 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
```

```
# 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

```
# 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

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

# Dense implements the operation: output = activation(dot(input, kernel) + bias
```

```
# activation is the element-wise activation function passed as the activation a
# 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 Tr

# 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 ac

# 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

```

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

```
# 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 inferenc
# 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'))
```

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

```
# Before training a model, you need to configure the learning process, which is

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


# Note: when using the categorical_crossentropy loss, your targets should be in
# (e.g. if you have 10 classes, the target for each sample should be a 10-dimen
# 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=['accur

# 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=Non
# validation_data=None, shuffle=True, class_weight=None, sample_weight=None, in
# validation_steps=None)

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

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

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

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

```

```
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/t

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

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

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

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

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

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

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

60000/60000 [==============================] - 11s 176us/step - loss: 1.2651 -
Epoch 2/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.7152 - a
Epoch 3/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.5881 - a
Epoch 4/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.5267 - a
Epoch 5/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4892 - a
Epoch 6/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4633 - a
Epoch 7/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4441 - a
Epoch 8/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4291 - a
Epoch 9/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4170 - a
Epoch 10/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4069 - a
Epoch 11/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3984 - a
Epoch 12/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3910 - a
Epoch 13/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3846 - a
Epoch 14/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3789 - a
Epoch 15/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3738 - a
Epoch 16/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3693 - a
Epoch 17/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3651 - a
Epoch 18/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.3614 - a
Epoch 19/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3579 - a
Epoch 20/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3547 - a
```

```
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_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.33514436384439467
Test accuracy: 0.909
```

MLP + Sigmoid activation + SGDOptimizer

```
# 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()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               401920
```

```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 30us/step - loss: 2.2663 - a
Epoch 2/20
60000/60000 [==============================] - 2s 26us/step - loss: 2.1799 - a
Epoch 3/20
60000/60000 [==============================] - 2s 27us/step - loss: 2.0670 - a
Epoch 4/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.9053 - a
Epoch 5/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.6921 - a
Epoch 6/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.4557 - a
Epoch 7/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.2397 - a
Epoch 8/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.0686 - a
Epoch 9/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.9402 - a
Epoch 10/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.8436 - a
Epoch 11/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.7690 - a
Epoch 12/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.7099 - a
Epoch 13/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.6624 - a
Epoch 14/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.6232 - a
Epoch 15/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.5907 - a
Epoch 16/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.5633 - a
Epoch 17/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.5398 - a
Epoch 18/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.5195 - a
Epoch 19/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.5018 - a
Epoch 20/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4862 - a
```

```
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
```

```
# 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_e

# we will get val_loss and val_acc only when you pass the paramter validation_da
# 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 numbe

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

```
Test score: 0.4634165374755859
Test accuracy: 0.8754
```

```
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 + Sigmoid activation + ADAM

```
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', metric

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_5 (Dense)              (None, 512)               401920    
```

```
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_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.06385514608082886
Test accuracy: 0.9824
```

```
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()
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:588: FutureWarni
  kde_data = remove_na(group_data)
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:816: FutureWarni
  violin_data = remove_na(group_data)
```

MLP + ReLU +SGD

```
# Multilayer perceptron

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

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

model_relu.summary()
```

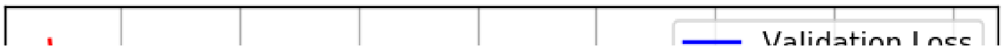
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_8 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_9 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

```
model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.7579 - a
Epoch 2/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.3535 - a
Epoch 3/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.2900 - a
```

```
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_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.12405014228336513
Test accuracy: 0.9631
```



```
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()
```

Trained model Weights Trained model Weights Trained model Weights

MLP + ReLU + ADAM

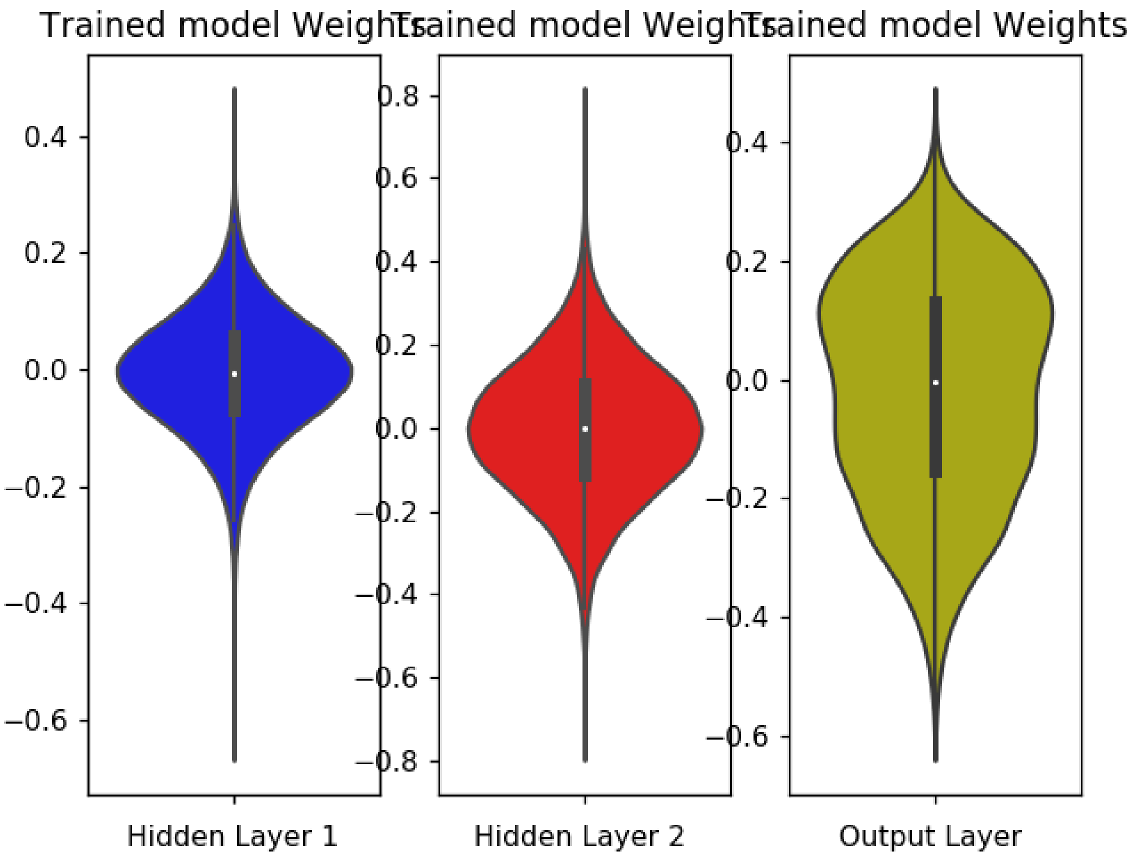
```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_i
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(me
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_11 (Dense)             (None, 512)               401920    
_________________________________________________________________
dense_12 (Dense)             (None, 128)               65664     
_________________________________________________________________
dense_13 (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 [==============================] - 7s 121us/step - loss: 0.2341 - 
Epoch 2/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0878 - a
Epoch 3/20
60000/60000 [==============================] - 5s 75us/step - loss: 0.0544 - a
Epoch 4/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0354 - a
Epoch 5/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0266 - a
Epoch 6/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0200 - a
Epoch 7/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0155 - a
Epoch 8/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0140 - a
Epoch 9/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0143 - a
Epoch 10/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0128 - 
Epoch 11/20
60000/60000 [==============================] - 7s 125us/step - loss: 0.0081 - 
Epoch 12/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0121 - 
Epoch 13/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.0107 - 
Epoch 14/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0113 - 
Epoch 15/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0058 - a
Epoch 16/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0040 - a
Epoch 17/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0119 - a
Epoch 18/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0105 - a
Epoch 19/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0064 - a
Epoch 20/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0056 - a
```

```
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_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb


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

```
Test score: 0.10294274219236926
Test accuracy: 0.9805
```

```
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()
```



MLP + Batch-Norm on hidden Layers + AdamOptimizer

```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

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

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

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_14 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dense_15 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

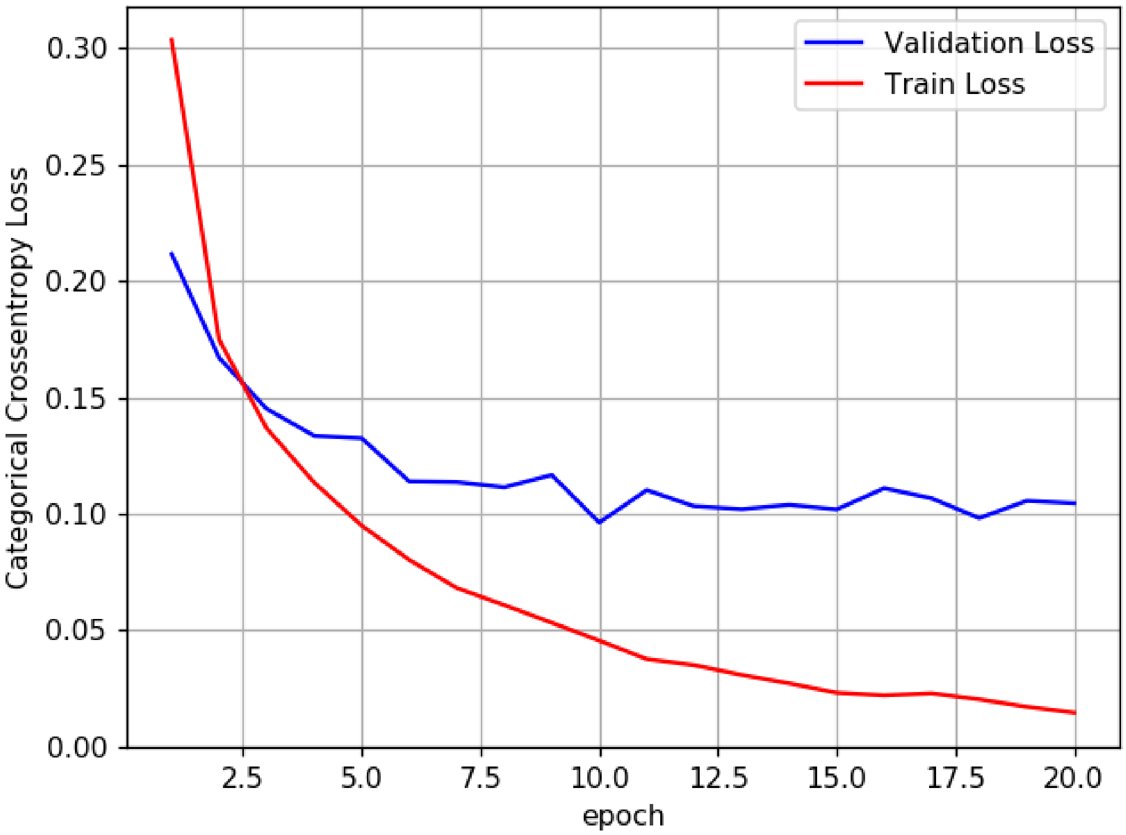
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.3036 -
Epoch 2/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.1747 -
Epoch 3/20
60000/60000 [==============================] - 13s 220us/step - loss: 0.1367 -
Epoch 4/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.1134 -
Epoch 5/20
60000/60000 [==============================] - 13s 211us/step - loss: 0.0949 -
Epoch 6/20
60000/60000 [==============================] - 7s 119us/step - loss: 0.0802 -
Epoch 7/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.0682 -
Epoch 8/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.0608 -
Epoch 9/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0532 -
Epoch 10/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.0455 -
Epoch 11/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0376 -
Epoch 12/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.0350 -
Epoch 13/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.0308 -
Epoch 14/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.0271 -
Epoch 15/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.0231 -
Epoch 16/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.0220 -
Epoch 17/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0229 -
Epoch 18/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.0203 -
Epoch 19/20
60000/60000 [==============================] - 7s 125us/step - loss: 0.0171 -
Epoch 20/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0146 -
```

```
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))

```

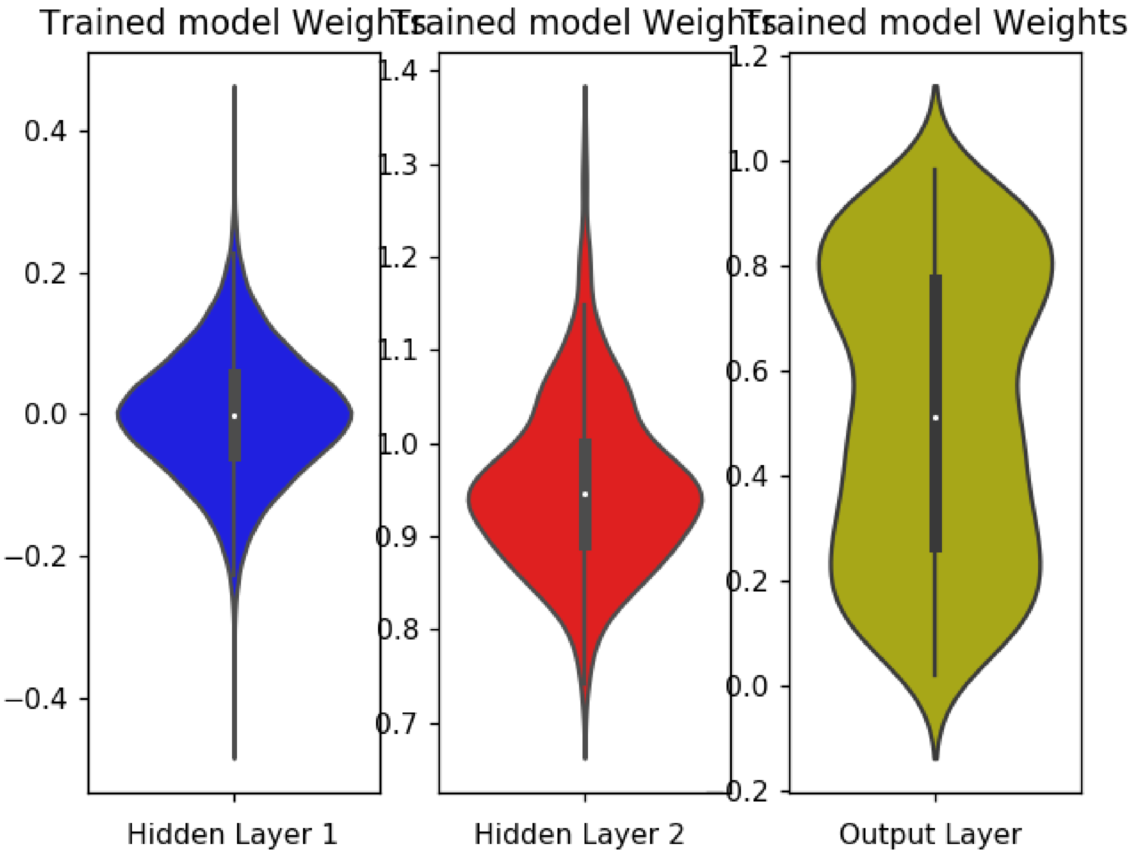
```
# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.10456635547156475
Test accuracy: 0.9732
```




```
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()
```



5. MLP + Dropout + AdamOptimizer

```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormali

from keras.layers import Dropout

model_drop = Sequential()

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

```

```
model_drop.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal
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_17 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_18 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_19 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

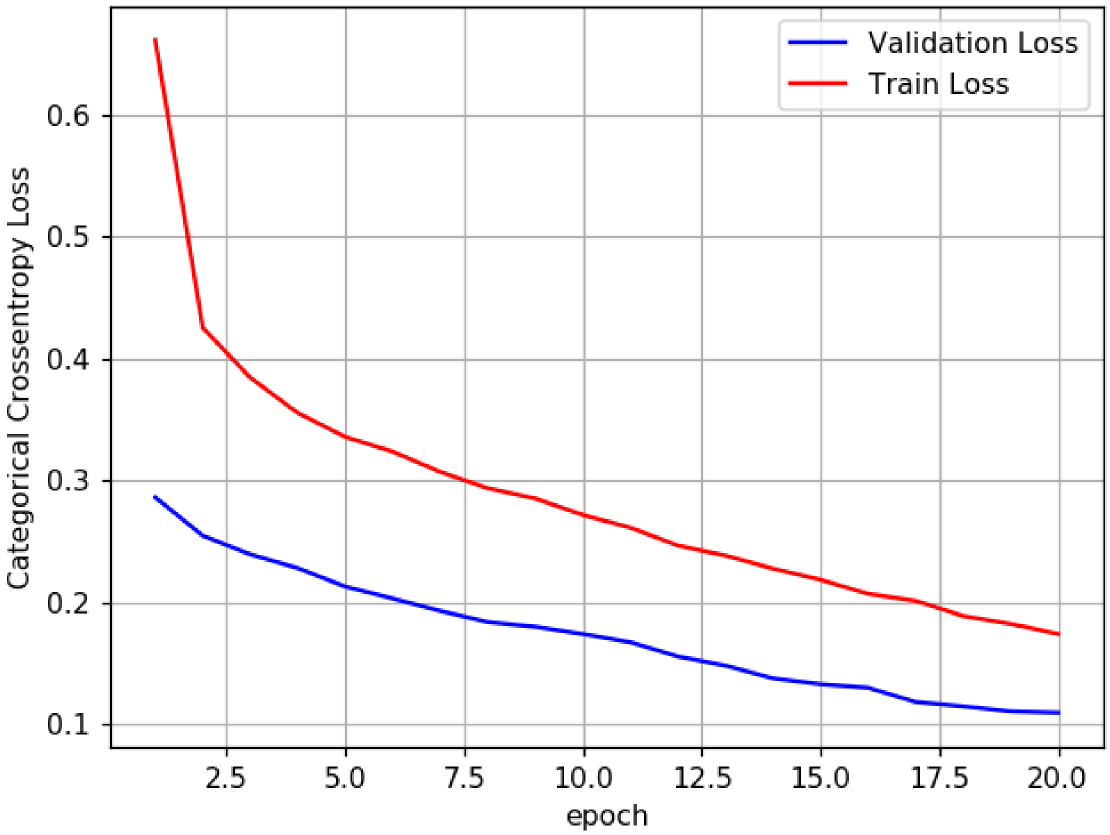
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 227us/step - loss: 0.6612 -
Epoch 2/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.4250 -
Epoch 3/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.3841 -
Epoch 4/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.3551 -
Epoch 5/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.3355 -
Epoch 6/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.3234 -
Epoch 7/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.3068 -
Epoch 8/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.2933 -
Epoch 9/20
60000/60000 [==============================] - 13s 222us/step - loss: 0.2850 -
Epoch 10/20
60000/60000 [==============================] - 14s 236us/step - loss: 0.2715 -
Epoch 11/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.2611 -
Epoch 12/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.2464 -
Epoch 13/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.2382 -
Epoch 14/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.2275 -
Epoch 15/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.2183 -
Epoch 16/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.2068 -
Epoch 17/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.2011 -
Epoch 18/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.1886 -
Epoch 19/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.1821 -
Epoch 20/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.1739 -
```

```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))

```

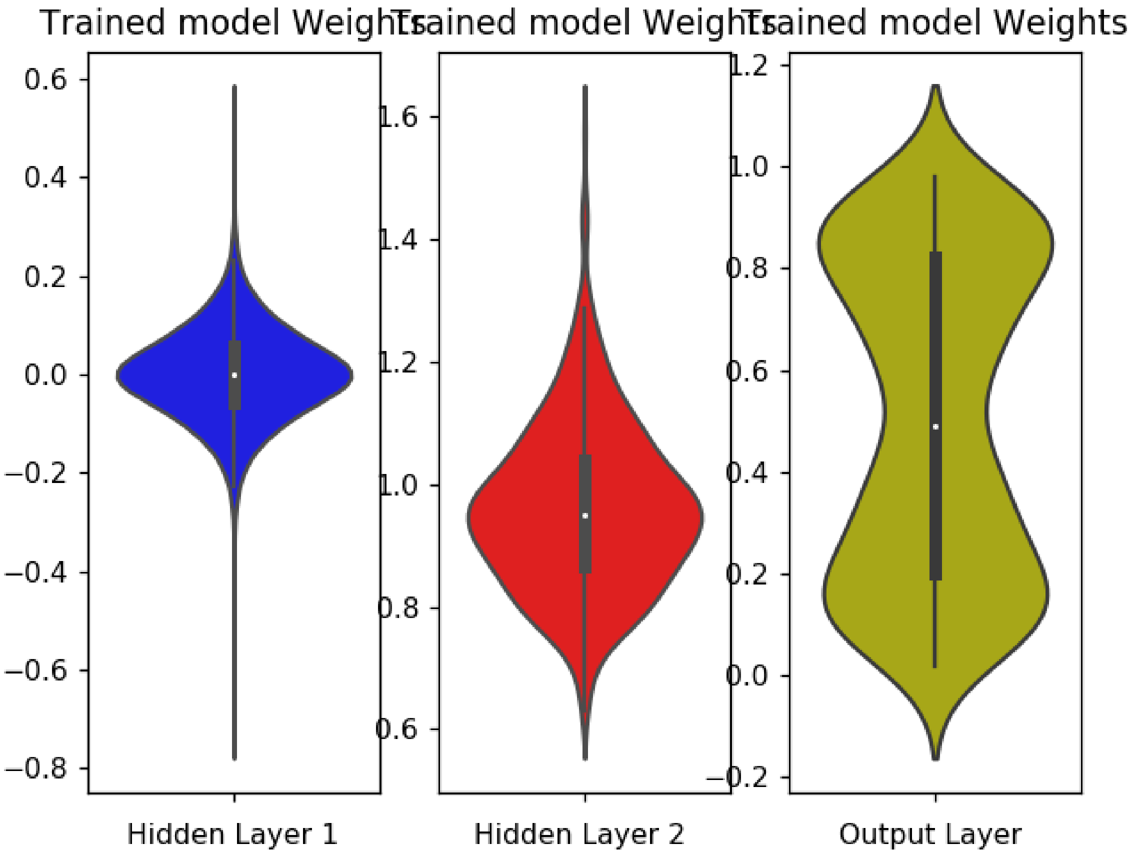
```
# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.1093290721397847
Test accuracy: 0.9679
```




```
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()
```



Hyper-parameter tuning of Keras models using Sklearn

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

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


    model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimi
```

```
    return model
```

```
# https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-r

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_s
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)
```

```
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: 0.975633 using {'activ': 'relu'}
0.974650 (0.001138) with: {'activ': 'sigmoid'}
0.975633 (0.002812) with: {'activ': 'relu'}
```

==============Assignment======================

1. Number of hidden layers = 2 (550,450) + adam optimizer + BN + D

```
# for relu layers, directly using he_normal() initializer
model_one = Sequential()

model_one.add(Dense(550, activation='relu', input_shape=(input_dim,), kernel_in
model_one.add(BatchNormalization())
model_one.add(Dropout(0.6))

model_one.add(Dense(450, activation='relu', kernel_initializer=he_normal(seed=N
model_one.add(BatchNormalization())
model_one.add(Dropout(0.5))

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


model_one.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_40 (Dense)             (None, 550)               431750    
_________________________________________________________________
batch_normalization_25 (Batc (None, 550)               2200      
_________________________________________________________________
dropout_25 (Dropout)         (None, 550)               0         
_________________________________________________________________
dense_41 (Dense)             (None, 450)               247950    
_________________________________________________________________
batch_normalization_26 (Batc (None, 450)               1800      
_________________________________________________________________
dropout_26 (Dropout)         (None, 450)               0         
_________________________________________________________________
dense_42 (Dense)             (None, 10)                4510      
=================================================================
Total params: 688,210
Trainable params: 686,210
Non-trainable params: 2,000
_________________________________________________________________
```

```
model_one.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

history = model_one.fit(X_train, Y_train, batch_size=batch_size, epochs=50, ver
```

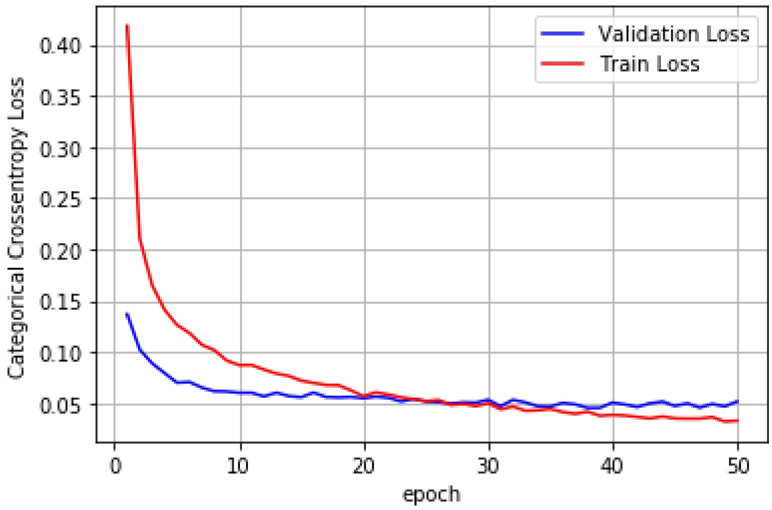
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 5s 78us/step - loss: 0.4183 - a
Epoch 2/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.2105 - a
Epoch 3/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.1653 - a
Epoch 4/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.1413 - a
Epoch 5/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.1264 - a
Epoch 6/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.1186 - a
Epoch 7/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.1070 - a
Epoch 8/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.1018 - a
Epoch 9/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0917 - a
Epoch 10/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0872 - a
Epoch 11/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0873 - a
Epoch 12/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0828 - a
Epoch 13/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0789 - a
Epoch 14/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0766 - a
Epoch 15/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0720 - a
Epoch 16/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0697 - a
Epoch 17/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0677 - a
Epoch 18/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0675 - a
Epoch 19/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0625 - a
Epoch 20/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0568 - a
Epoch 21/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0605 - a
Epoch 22/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.0584 - a
Epoch 23/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0555 - a
Epoch 24/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0539 - a
Epoch 25/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0520 - a
Epoch 26/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0530 - a
Epoch 27/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0482 - a
Epoch 28/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0493 - a
Epoch 29/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0470 - a
Epoch 30/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0498 - a
```

```
Epoch 31/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0440 - a
Epoch 32/50
60000/60000 [==============================] - 3s 53us/step - loss: 0.0467 - a
Epoch 33/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0425 - a
Epoch 34/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0430 - a
Epoch 35/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0443 - a
Epoch 36/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0413 - a
Epoch 37/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0396 - a
Epoch 38/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0416 - a
Epoch 39/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0376 - a
Epoch 40/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0385 - a
Epoch 41/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.0378 - a
Epoch 42/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.0365 - a
Epoch 43/50
60000/60000 [==============================] - 3s 58us/step - loss: 0.0351 - a
Epoch 44/50
60000/60000 [==============================] - 3s 56us/step - loss: 0.0369 - a
Epoch 45/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0351 - a
Epoch 46/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0348 - a
Epoch 47/50
60000/60000 [==============================] - 3s 53us/step - loss: 0.0349 - a
Epoch 48/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.0363 - a
Epoch 49/50
60000/60000 [==============================] - 3s 55us/step - loss: 0.0321 - a
Epoch 50/50
60000/60000 [==============================] - 3s 53us/step - loss: 0.0328 - a
```

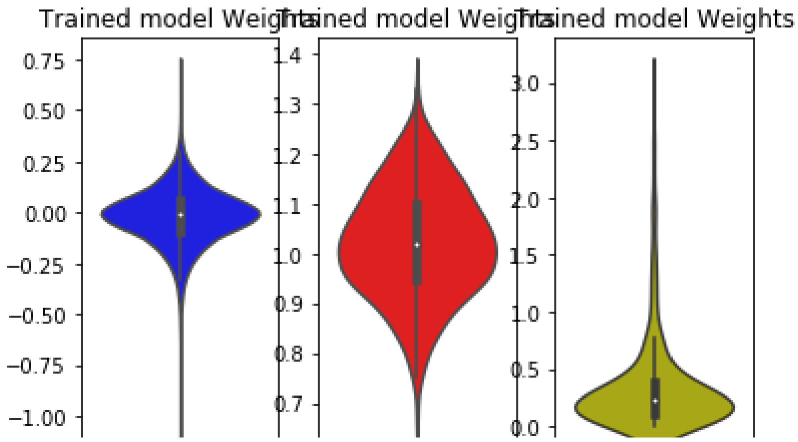
```
%matplotlib inline
score = model_one.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# list of epoch numbers
x = list(range(1,50+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_e

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.051541719751636995
Test accuracy: 0.9857
```




```
w_after = model_one.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()
```



2. Number of hidden layers = 3 (550,450,350) + adam optimizer + BN

```
# for relu layers, directly using he_normal() initializer
model_two = Sequential()

model_two.add(Dense(550, activation='relu', input_shape=(input_dim,), kernel_in
model_two.add(BatchNormalization())
model_two.add(Dropout(0.6))

model_two.add(Dense(450, activation='relu', kernel_initializer=he_normal(seed=N
model_two.add(BatchNormalization())
model_two.add(Dropout(0.5))

model_two.add(Dense(350, activation='relu', kernel_initializer=he_normal(seed=N
model_two.add(BatchNormalization())
model_two.add(Dropout(0.5))

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


model_two.summary()
```

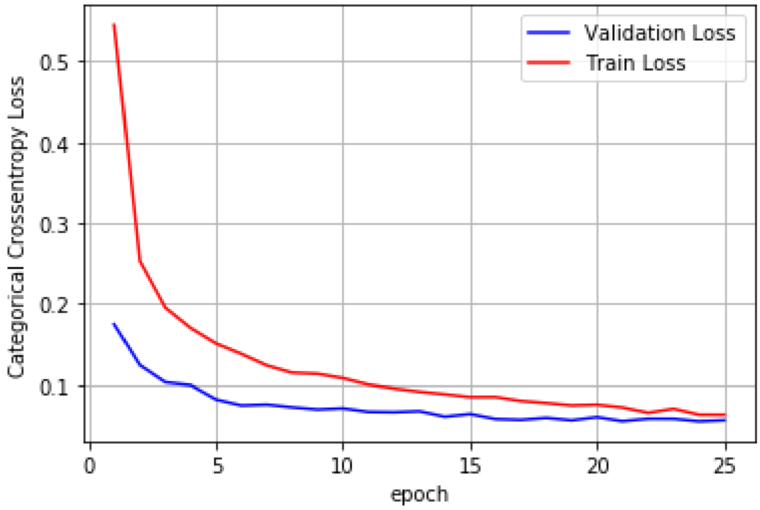
```
WARNING:tensorflow:Large dropout rate: 0.6 (>0.5). In TensorFlow 2.x, dropout(
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_30 (Dense)             (None, 550)               431750
_________________________________________________________________
batch_normalization_17 (Batc (None, 550)               2200
_________________________________________________________________
dropout_17 (Dropout)         (None, 550)               0
_________________________________________________________________
dense_31 (Dense)             (None, 450)               247950
```

```
model_two.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

history = model_two.fit(X_train, Y_train, batch_size=batch_size, epochs=25, ver
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/25
60000/60000 [==============================] - 5s 87us/step - loss: 0.5450 - a
Epoch 2/25
60000/60000 [==============================] - 4s 68us/step - loss: 0.2530 - a
Epoch 3/25
60000/60000 [==============================] - 4s 67us/step - loss: 0.1952 - a
Epoch 4/25
60000/60000 [==============================] - 4s 66us/step - loss: 0.1699 - a
Epoch 5/25
60000/60000 [==============================] - 4s 66us/step - loss: 0.1508 - a
Epoch 6/25
60000/60000 [==============================] - 4s 70us/step - loss: 0.1379 - a
Epoch 7/25
60000/60000 [==============================] - 4s 67us/step - loss: 0.1237 - a
Epoch 8/25
60000/60000 [==============================] - 4s 69us/step - loss: 0.1146 - a
Epoch 9/25
60000/60000 [==============================] - 4s 67us/step - loss: 0.1133 - a
Epoch 10/25
60000/60000 [==============================] - 4s 68us/step - loss: 0.1080 - a
Epoch 11/25
60000/60000 [==============================] - 4s 68us/step - loss: 0.1001 - a
Epoch 12/25
60000/60000 [==============================] - 4s 68us/step - loss: 0.0951 - a
Epoch 13/25
60000/60000 [==============================] - 4s 66us/step - loss: 0.0912 - a
Epoch 14/25
60000/60000 [==============================] - 4s 67us/step - loss: 0.0881 - a
Epoch 15/25
60000/60000 [==============================] - 4s 68us/step - loss: 0.0847 - a
Epoch 16/25
60000/60000 [==============================] - 4s 67us/step - loss: 0.0848 - a
Epoch 17/25
```

```
%matplotlib inline
score = model_two.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(1,25+1))

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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

```
Test score: 0.05603201202231867
Test accuracy: 0.9837
```




```python
w_after = model_two.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()
```

3. Number of hidden layers = 5 (650,550,450,350,250) + adam optim Dropout(0.6,0.5,0.5,0.5,0.5)

```
# for relu layers, directly using he_normal() initializer
model_three = Sequential()

model_three.add(Dense(650, activation='relu', input_shape=(input_dim,), kernel_
model_three.add(BatchNormalization())
model_three.add(Dropout(0.6))

model_three.add(Dense(550, activation='relu', kernel_initializer=he_normal(seed=
model_three.add(BatchNormalization())
model_three.add(Dropout(0.5))

model_three.add(Dense(450, activation='relu', kernel_initializer=he_normal(seed=
model_three.add(BatchNormalization())
model_three.add(Dropout(0.5))

model_three.add(Dense(350, activation='relu', kernel_initializer=he_normal(seed=
model_three.add(BatchNormalization())
model_three.add(Dropout(0.5))

model_three.add(Dense(250, activation='relu', kernel_initializer=he_normal(seed=
model_three.add(BatchNormalization())
model_three.add(Dropout(0.5))

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


model_three.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_34 (Dense)             (None, 650)               510250
_________________________________________________________________
batch_normalization_20 (Batc (None, 650)               2600
_________________________________________________________________
dropout_20 (Dropout)         (None, 650)               0
_________________________________________________________________
dense_35 (Dense)             (None, 550)               358050
_________________________________________________________________
batch_normalization_21 (Batc (None, 550)               2200
_________________________________________________________________
dropout_21 (Dropout)         (None, 550)               0
_________________________________________________________________
dense_36 (Dense)             (None, 450)               247950
_________________________________________________________________
batch_normalization_22 (Batc (None, 450)               1800
_________________________________________________________________
dropout_22 (Dropout)         (None, 450)               0
_________________________________________________________________
dense_37 (Dense)             (None, 350)               157850
```

```
model_three.compile(optimizer='adam', loss='categorical_crossentropy', metrics=

history = model_three.fit(X_train, Y_train, batch_size=batch_size, epochs=25, v
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/25
60000/60000 [==============================] - 7s 124us/step - loss: 0.8781 -
Epoch 2/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.3221 - a
Epoch 3/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.2472 - a
Epoch 4/25
60000/60000 [==============================] - 5s 91us/step - loss: 0.2086 - a
Epoch 5/25
60000/60000 [==============================] - 6s 93us/step - loss: 0.1839 - a
Epoch 6/25
60000/60000 [==============================] - 6s 93us/step - loss: 0.1685 - a
Epoch 7/25
60000/60000 [==============================] - 5s 92us/step - loss: 0.1554 - a
Epoch 8/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.1438 - a
Epoch 9/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.1350 - a
Epoch 10/25
60000/60000 [==============================] - 5s 91us/step - loss: 0.1313 - a
Epoch 11/25
60000/60000 [==============================] - 6s 93us/step - loss: 0.1240 - a
Epoch 12/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.1179 - a
Epoch 13/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.1129 - a
Epoch 14/25
60000/60000 [==============================] - 6s 94us/step - loss: 0.1075 - a
Epoch 15/25
60000/60000 [==============================] - 6s 92us/step - loss: 0.1045 - a
Epoch 16/25
60000/60000 [==============================] - 6s 93us/step - loss: 0.1004 - a
Epoch 17/25
```

```
%matplotlib inline
score = model_three.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(1,25+1))

# we will get val_loss and val_acc only when you pass the paramter validation_d
# 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 numb

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