Keras -- MLPs on MNIST

Refrences: Applied Ai course Google Stackoverflow

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

In [0]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4
# https://stackoverflow.com/a/14434334
# this function is used to update the plots for each epoch and error
def plt_dynamic(x, vy, ty, ax, colors=['b']):
    ax.plot(x, vy, 'b', label="Validation Loss")
    ax.plot(x, ty, 'r', label="Train Loss")
    plt.legend()
    plt.grid()
    fig.canvas.draw()
```

In [0]:
```
%matplotlib inline
```

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

In [39]:
```
print("Number of training examples :", X_train.shape[0], "and each image is of shape (%d, %d)"%(X_train.shape[1], X_train.shape[2]))
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d, %d)"%(X_test.shape[1], X_test.shape[2]))
```

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

In [0]:
```
# if you observe the input shape its 2 dimensional vector
# for each image we have a (28*28) vector
# we will convert the (28*28) vector into single dimensional vector of 1 * 784

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1]*X_train.shape[2])
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1]*X_test.shape[2])
```

In [41]:
```
# after converting the input images from 3d to 2d vectors

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

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

In [42]:
```
# An example data point
print(X_train[0])
```

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

In [0]:
```
# if we observe the above matrix each cell is having a value between 0-255
# before we move to apply machine learning algorithms lets try to normalize th
e data
# X => (X - Xmin)/(Xmax-Xmin) = X/255

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

In [44]:
```
# 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 [45]:

```
# here we are having a class number for each image
print("Class label of first image :", y_train[0])

# lets convert this into a 10 dimensional vector
# ex: consider an image is 5 convert it into 5 => [0, 0, 0, 0, 0, 1, 0, 0, 0,
 0]
# this conversion needed for MLPs

Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)

print("After converting the output into a vector : ",Y_train[0])
```

```
Class label of first image : 5
After converting the output into a vector :  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Softmax classifier

In [0]:
```
# https://keras.io/getting-started/sequential-model-guide/

# The Sequential model is a linear stack of layers.
# you can create a Sequential model by passing a list of layer instances to th
e constructor:

# model = Sequential([
#     Dense(32, input_shape=(784,)),
#     Activation('relu'),
#     Dense(10),
#     Activation('softmax'),
# ])

# You can also simply add layers via the .add() method:

# model = Sequential()
# model.add(Dense(32, input_dim=784))
# model.add(Activation('relu'))

###

# https://keras.io/layers/core/

# keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer
='glorot_uniform',
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, ac
tivity_regularizer=None,
# kernel_constraint=None, bias_constraint=None)

# Dense implements the operation: output = activation(dot(input, kernel) + bia
s) where
# activation is the element-wise activation function passed as the activation
 argument,
# kernel is a weights matrix created by the layer, and
# bias is a bias vector created by the layer (only applicable if use_bias is T
rue).

# 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 a
ctivation argument supported by all forward layers:

# from keras.layers import Activation, Dense

# model.add(Dense(64))
# model.add(Activation('tanh'))

# This is equivalent to:
# model.add(Dense(64, activation='tanh'))

# there are many activation functions ar available ex: tanh, relu, softmax
```

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

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

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

Assignment

BN + Relu + Dropout2 Layers

In [0]:
```
```

In [48]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormal
ization-function-in-keras
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condi
tion with σ=√(2/(ni+ni+1).
# 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 import Dropout
# import BatchNormalization
from keras.layers.normalization import BatchNormalization


model_assign = Sequential()

model_assign.add(Dense(332, activation='relu', input_shape=(input_dim,), kerne
l_initializer=RandomNormal(mean=0.0, stddev=0.042, seed=None)))
model_assign.add(BatchNormalization())
model_assign.add(Dropout(0.5))

model_assign.add(Dense(56, activation='relu', kernel_initializer=RandomNormal(
mean=0.0, stddev=0.071, seed=None)) )
model_assign.add(BatchNormalization())
model_assign.add(Dropout(0.5))

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


model_assign.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_28 (Dense)             (None, 332)               260620    
_________________________________________________________________
batch_normalization_11 (Batc (None, 332)               1328      
_________________________________________________________________
dropout_11 (Dropout)         (None, 332)               0         
_________________________________________________________________
dense_29 (Dense)             (None, 56)                18648     
_________________________________________________________________
batch_normalization_12 (Batc (None, 56)                224       
_________________________________________________________________
dropout_12 (Dropout)         (None, 56)                0         
_________________________________________________________________
dense_30 (Dense)             (None, 10)                570       
=================================================================
Total params: 281,390
Trainable params: 280,614
Non-trainable params: 776
_________________________________________________________________
```

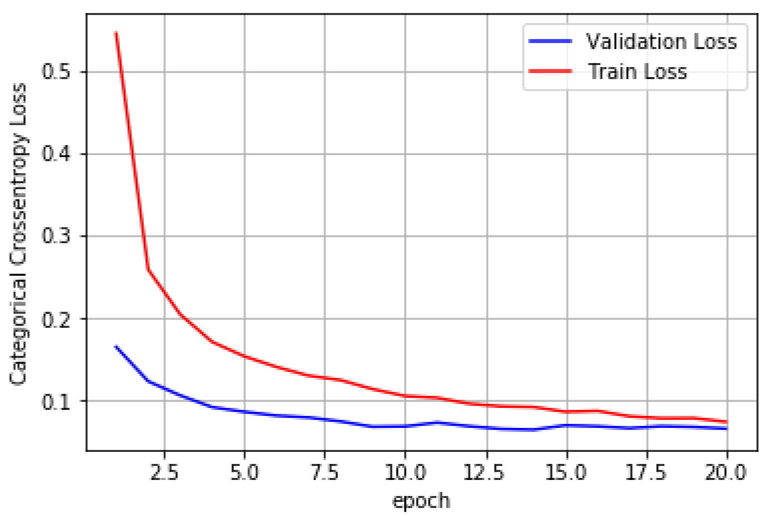
In [49]:
```
model_assign.compile(optimizer='adam', loss='categorical_crossentropy', metric
s=['accuracy'])

history = model_assign.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_
epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.5452 -
acc: 0.8381 - val_loss: 0.1653 - val_acc: 0.9490
Epoch 2/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2592 -
acc: 0.9250 - val_loss: 0.1235 - val_acc: 0.9626
Epoch 3/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.2047 -
acc: 0.9413 - val_loss: 0.1064 - val_acc: 0.9673
Epoch 4/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.1713 -
acc: 0.9503 - val_loss: 0.0921 - val_acc: 0.9712
Epoch 5/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.1539 -
acc: 0.9558 - val_loss: 0.0865 - val_acc: 0.9729
Epoch 6/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1410 -
acc: 0.9595 - val_loss: 0.0819 - val_acc: 0.9758
Epoch 7/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1304 -
acc: 0.9616 - val_loss: 0.0796 - val_acc: 0.9758
Epoch 8/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1250 -
acc: 0.9634 - val_loss: 0.0748 - val_acc: 0.9765
Epoch 9/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1141 -
acc: 0.9671 - val_loss: 0.0684 - val_acc: 0.9795
Epoch 10/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1057 -
acc: 0.9690 - val_loss: 0.0688 - val_acc: 0.9785
Epoch 11/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1035 -
acc: 0.9690 - val_loss: 0.0734 - val_acc: 0.9782
Epoch 12/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0963 -
acc: 0.9716 - val_loss: 0.0689 - val_acc: 0.9797
Epoch 13/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0931 -
acc: 0.9721 - val_loss: 0.0657 - val_acc: 0.9808
Epoch 14/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0923 -
acc: 0.9722 - val_loss: 0.0647 - val_acc: 0.9798
Epoch 15/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0865 -
acc: 0.9735 - val_loss: 0.0701 - val_acc: 0.9787
Epoch 16/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0876 -
acc: 0.9734 - val_loss: 0.0689 - val_acc: 0.9802
Epoch 17/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0810 -
acc: 0.9762 - val_loss: 0.0666 - val_acc: 0.9803
Epoch 18/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0786 -
acc: 0.9764 - val_loss: 0.0690 - val_acc: 0.9808
Epoch 19/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0788 -
```

```
acc: 0.9763 - val_loss: 0.0681 - val_acc: 0.9798
Epoch 20/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0744 -
acc: 0.9776 - val_loss: 0.0661 - val_acc: 0.9812
```

In [50]:
```
score = model_assign.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, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.06606815920227091
Test accuracy: 0.9812
```



3 Hidden layers

In [51]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormal
ization-function-in-keras
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condi
tion with σ=√(2/(ni+ni+1).
# 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 import Dropout

model_assign3 = Sequential()

model_assign3.add(Dense(512, activation='relu', input_shape=(input_dim,), kern
el_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_assign3.add(BatchNormalization())
model_assign3.add(Dropout(0.5))

model_assign3.add(Dense(128, activation='relu', input_shape=(input_dim,), kern
el_initializer=RandomNormal(mean=0.0, stddev=0.056, seed=None)))
model_assign3.add(BatchNormalization())
model_assign3.add(Dropout(0.5))

model_assign3.add(Dense(64, activation='relu', kernel_initializer=RandomNormal
(mean=0.0, stddev=0.10, seed=None)) )
model_assign3.add(BatchNormalization())
model_assign3.add(Dropout(0.5))

model_assign3.add(Dense(output_dim, activation='softmax'))


model_assign3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_31 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_13 (Batc (None, 512)               2048      
_________________________________________________________________
dropout_13 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_32 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_14 (Batc (None, 128)               512       
_________________________________________________________________
dropout_14 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_33 (Dense)             (None, 64)                8256      
_________________________________________________________________
batch_normalization_15 (Batc (None, 64)                256       
_________________________________________________________________
dropout_15 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_34 (Dense)             (None, 10)                650       
=================================================================
Total params: 479,306
Trainable params: 477,898
Non-trainable params: 1,408
_________________________________________________________________
```

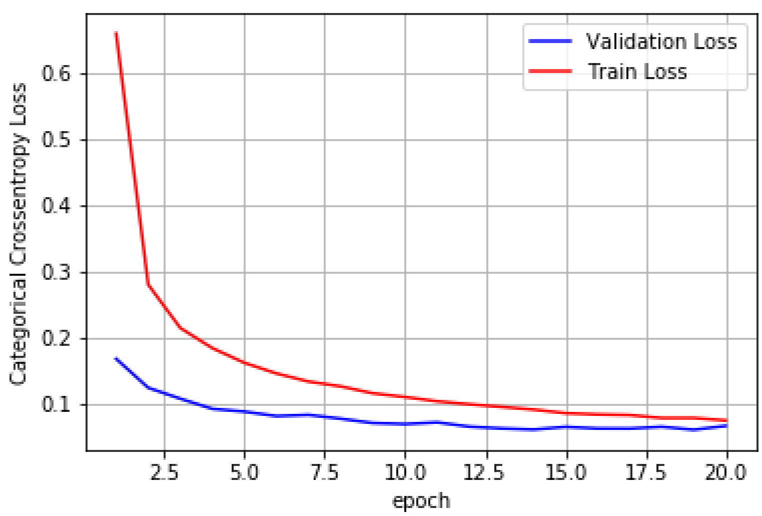
In [52]:
```
model_assign3.compile(optimizer='adam', loss='categorical_crossentropy', metri
cs=['accuracy'])

history = model_assign3.fit(X_train, Y_train, batch_size=batch_size, epochs=nb
_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.6582 -
acc: 0.8005 - val_loss: 0.1684 - val_acc: 0.9460
Epoch 2/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.2803 -
acc: 0.9203 - val_loss: 0.1247 - val_acc: 0.9613
Epoch 3/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.2149 -
acc: 0.9399 - val_loss: 0.1083 - val_acc: 0.9669
Epoch 4/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1845 -
acc: 0.9487 - val_loss: 0.0929 - val_acc: 0.9711
Epoch 5/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1624 -
acc: 0.9553 - val_loss: 0.0888 - val_acc: 0.9718
Epoch 6/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1463 -
acc: 0.9585 - val_loss: 0.0820 - val_acc: 0.9760
Epoch 7/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1341 -
acc: 0.9629 - val_loss: 0.0838 - val_acc: 0.9747
Epoch 8/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1268 -
acc: 0.9642 - val_loss: 0.0781 - val_acc: 0.9755
Epoch 9/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1165 -
acc: 0.9673 - val_loss: 0.0716 - val_acc: 0.9783
Epoch 10/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1107 -
acc: 0.9688 - val_loss: 0.0700 - val_acc: 0.9801
Epoch 11/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1043 -
acc: 0.9700 - val_loss: 0.0725 - val_acc: 0.9788
Epoch 12/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0998 -
acc: 0.9714 - val_loss: 0.0661 - val_acc: 0.9808
Epoch 13/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0959 -
acc: 0.9729 - val_loss: 0.0634 - val_acc: 0.9815
Epoch 14/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0918 -
acc: 0.9741 - val_loss: 0.0617 - val_acc: 0.9819
Epoch 15/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0864 -
acc: 0.9750 - val_loss: 0.0657 - val_acc: 0.9819
Epoch 16/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0845 -
acc: 0.9763 - val_loss: 0.0633 - val_acc: 0.9822
Epoch 17/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0836 -
acc: 0.9757 - val_loss: 0.0632 - val_acc: 0.9825
Epoch 18/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0791 -
acc: 0.9767 - val_loss: 0.0659 - val_acc: 0.9826
Epoch 19/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0792 -
```

```
acc: 0.9778 - val_loss: 0.0614 - val_acc: 0.9837
Epoch 20/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0755 -
acc: 0.9780 - val_loss: 0.0672 - val_acc: 0.9813
```

In [53]:
```
score = model_assign3.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, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.06720245788121829
Test accuracy: 0.9813
```



5 hidden layers

In [54]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormal
ization-function-in-keras
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condi
tion with σ=√(2/(ni+ni+1).
# 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 import Dropout

model_assign5 = Sequential()

model_assign5.add(Dense(512, activation='relu', input_shape=(input_dim,), kern
el_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_assign5.add(BatchNormalization())
model_assign5.add(Dropout(0.5))

model_assign5.add(Dense(256, activation='relu', kernel_initializer=RandomNorma
l(mean=0.0, stddev=0.051, seed=None)) )
model_assign5.add(BatchNormalization())
model_assign5.add(Dropout(0.5))

model_assign5.add(Dense(128, activation='relu', kernel_initializer=RandomNorma
l(mean=0.0, stddev=0.072, seed=None)) )
model_assign5.add(BatchNormalization())
model_assign5.add(Dropout(0.5))

model_assign5.add(Dense(64, activation='relu', kernel_initializer=RandomNormal
(mean=0.0, stddev=0.10, seed=None)) )
model_assign5.add(BatchNormalization())
model_assign5.add(Dropout(0.5))

model_assign5.add(Dense(32, activation='relu', kernel_initializer=RandomNormal
(mean=0.0, stddev=0.144, seed=None)) )
model_assign5.add(BatchNormalization())
model_assign5.add(Dropout(0.5))


model_assign5.add(Dense(output_dim, activation='softmax'))


model_assign5.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_16 (Batc (None, 512)               2048
_________________________________________________________________
dropout_16 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 256)               131328
_________________________________________________________________
batch_normalization_17 (Batc (None, 256)               1024
_________________________________________________________________
dropout_17 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_37 (Dense)             (None, 128)               32896
_________________________________________________________________
batch_normalization_18 (Batc (None, 128)               512
_________________________________________________________________
dropout_18 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_38 (Dense)             (None, 64)                8256
_________________________________________________________________
batch_normalization_19 (Batc (None, 64)                256
_________________________________________________________________
dropout_19 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_39 (Dense)             (None, 32)                2080
_________________________________________________________________
batch_normalization_20 (Batc (None, 32)                128
_________________________________________________________________
dropout_20 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_40 (Dense)             (None, 10)                330
=================================================================
Total params: 580,778
Trainable params: 578,794
Non-trainable params: 1,984
_________________________________________________________________
```

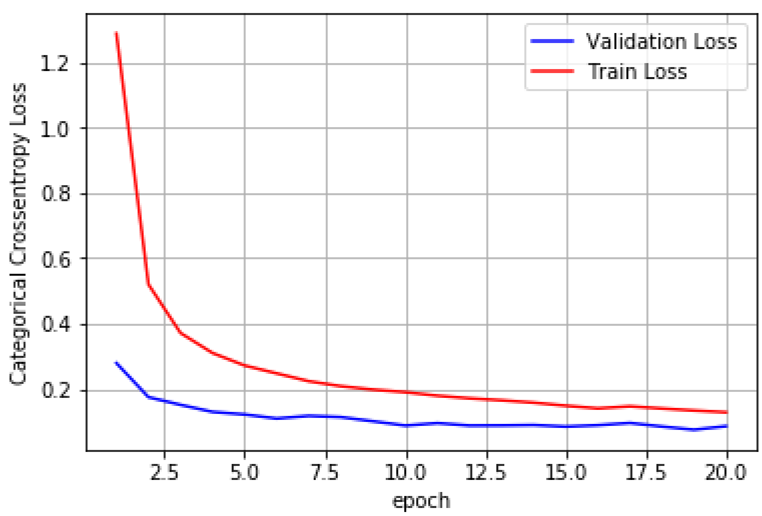
In [55]:
```
model_assign5.compile(optimizer='adam', loss='categorical_crossentropy', metri
cs=['accuracy'])

history = model_assign5.fit(X_train, Y_train, batch_size=batch_size, epochs=nb
_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 124us/step - loss: 1.2889 -
acc: 0.5871 - val_loss: 0.2800 - val_acc: 0.9286
Epoch 2/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.5212 -
acc: 0.8593 - val_loss: 0.1755 - val_acc: 0.9521
Epoch 3/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.3722 -
acc: 0.9079 - val_loss: 0.1519 - val_acc: 0.9609
Epoch 4/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.3107 -
acc: 0.9253 - val_loss: 0.1302 - val_acc: 0.9677
Epoch 5/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.2724 -
acc: 0.9364 - val_loss: 0.1224 - val_acc: 0.9663
Epoch 6/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.2483 -
acc: 0.9423 - val_loss: 0.1099 - val_acc: 0.9738
Epoch 7/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.2243 -
acc: 0.9488 - val_loss: 0.1181 - val_acc: 0.9716
Epoch 8/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.2091 -
acc: 0.9518 - val_loss: 0.1143 - val_acc: 0.9731
Epoch 9/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.1986 -
acc: 0.9546 - val_loss: 0.1015 - val_acc: 0.9741
Epoch 10/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1906 -
acc: 0.9569 - val_loss: 0.0883 - val_acc: 0.9782
Epoch 11/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1798 -
acc: 0.9594 - val_loss: 0.0958 - val_acc: 0.9770
Epoch 12/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1719 -
acc: 0.9615 - val_loss: 0.0883 - val_acc: 0.9786
Epoch 13/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1658 -
acc: 0.9629 - val_loss: 0.0887 - val_acc: 0.9792
Epoch 14/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1586 -
acc: 0.9647 - val_loss: 0.0900 - val_acc: 0.9789
Epoch 15/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1491 -
acc: 0.9662 - val_loss: 0.0851 - val_acc: 0.9799
Epoch 16/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1410 -
acc: 0.9681 - val_loss: 0.0894 - val_acc: 0.9796
Epoch 17/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.1473 -
acc: 0.9672 - val_loss: 0.0963 - val_acc: 0.9793
Epoch 18/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.1407 -
acc: 0.9677 - val_loss: 0.0853 - val_acc: 0.9798
Epoch 19/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1344 -
```

```
acc: 0.9695 - val_loss: 0.0755 - val_acc: 0.9823
Epoch 20/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1291 -
acc: 0.9703 - val_loss: 0.0864 - val_acc: 0.9796
```

In [56]:
```
score = model_assign5.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, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.08644420035893563
Test accuracy: 0.9796
```



Without Dropout and BN

2 layers

In [57]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormal
ization-function-in-keras
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condi
tion with σ=√(2/(ni+ni+1).
# 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 import Dropout

model_assign = Sequential()

model_assign.add(Dense(332, activation='relu', input_shape=(input_dim,), kerne
l_initializer=RandomNormal(mean=0.0, stddev=0.042, seed=None)))

model_assign.add(Dense(56, activation='relu', kernel_initializer=RandomNormal(
mean=0.0, stddev=0.071, seed=None)) )

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


model_assign.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_41 (Dense)             (None, 332)               260620    
_________________________________________________________________
dense_42 (Dense)             (None, 56)                18648     
_________________________________________________________________
dense_43 (Dense)             (None, 10)                570       
=================================================================
Total params: 279,838
Trainable params: 279,838
Non-trainable params: 0
_________________________________________________________________
```

In [58]:
```
model_assign.compile(optimizer='adam', loss='categorical_crossentropy', metric
s=['accuracy'])

history = model_assign.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_
epoch, verbose=1, validation_data=(X_test, Y_test))
```

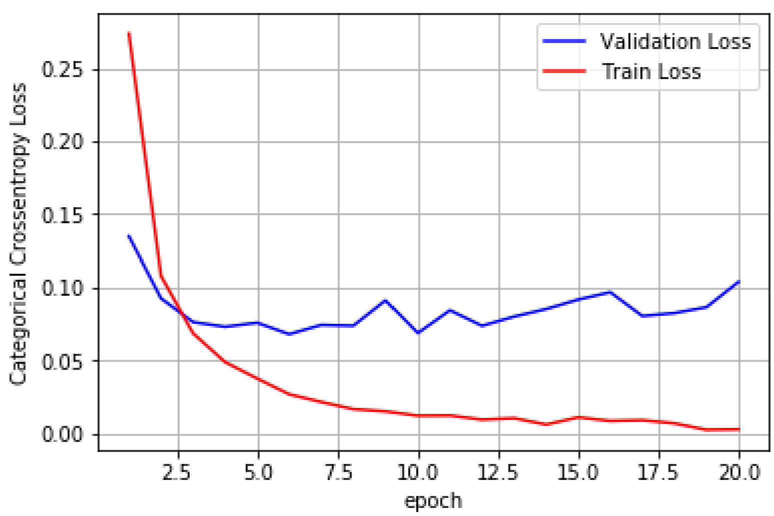
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2737 -
acc: 0.9219 - val_loss: 0.1348 - val_acc: 0.9589
Epoch 2/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.1079 -
acc: 0.9680 - val_loss: 0.0925 - val_acc: 0.9705
Epoch 3/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0686 -
acc: 0.9786 - val_loss: 0.0763 - val_acc: 0.9774
Epoch 4/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0489 -
acc: 0.9850 - val_loss: 0.0731 - val_acc: 0.9782
Epoch 5/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0376 -
acc: 0.9880 - val_loss: 0.0758 - val_acc: 0.9771
Epoch 6/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0268 -
acc: 0.9921 - val_loss: 0.0680 - val_acc: 0.9793
Epoch 7/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0215 -
acc: 0.9934 - val_loss: 0.0743 - val_acc: 0.9778
Epoch 8/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0166 -
acc: 0.9950 - val_loss: 0.0738 - val_acc: 0.9789
Epoch 9/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0151 -
acc: 0.9951 - val_loss: 0.0911 - val_acc: 0.9767
Epoch 10/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0121 -
acc: 0.9960 - val_loss: 0.0688 - val_acc: 0.9812
Epoch 11/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0123 -
acc: 0.9961 - val_loss: 0.0843 - val_acc: 0.9808
Epoch 12/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0094 -
acc: 0.9969 - val_loss: 0.0737 - val_acc: 0.9824
Epoch 13/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0104 -
acc: 0.9966 - val_loss: 0.0800 - val_acc: 0.9809
Epoch 14/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0061 -
acc: 0.9981 - val_loss: 0.0852 - val_acc: 0.9790
Epoch 15/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0110 -
acc: 0.9961 - val_loss: 0.0917 - val_acc: 0.9781
Epoch 16/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0086 -
acc: 0.9971 - val_loss: 0.0967 - val_acc: 0.9776
Epoch 17/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0091 -
acc: 0.9970 - val_loss: 0.0804 - val_acc: 0.9811
Epoch 18/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0069 -
acc: 0.9974 - val_loss: 0.0824 - val_acc: 0.9802
Epoch 19/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0025 -
```

```
acc: 0.9993 - val_loss: 0.0865 - val_acc: 0.9822
Epoch 20/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0028 -
acc: 0.9991 - val_loss: 0.1038 - val_acc: 0.9789
```

In [59]:

```
score = model_assign.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, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.1037857443765759
Test accuracy: 0.9789
```



3 Layers

In [60]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormal
ization-function-in-keras
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condi
tion with σ=√(2/(ni+ni+1).
# 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 import Dropout

model_assign3 = Sequential()

model_assign3.add(Dense(512, activation='relu', input_shape=(input_dim,), kern
el_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))

model_assign3.add(Dense(128, activation='relu', input_shape=(input_dim,), kern
el_initializer=RandomNormal(mean=0.0, stddev=0.056, seed=None)))

model_assign3.add(Dense(64, activation='relu', kernel_initializer=RandomNormal
(mean=0.0, stddev=0.10, seed=None)) )

model_assign3.add(Dense(output_dim, activation='softmax'))


model_assign3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_44 (Dense)             (None, 512)               401920    
_________________________________________________________________
dense_45 (Dense)             (None, 128)               65664     
_________________________________________________________________
dense_46 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_47 (Dense)             (None, 10)                650       
=================================================================
Total params: 476,490
Trainable params: 476,490
Non-trainable params: 0
_________________________________________________________________
```

In [61]:
```
model_assign3.compile(optimizer='adam', loss='categorical_crossentropy', metri
cs=['accuracy'])

history = model_assign3.fit(X_train, Y_train, batch_size=batch_size, epochs=nb
_epoch, verbose=1, validation_data=(X_test, Y_test))
```

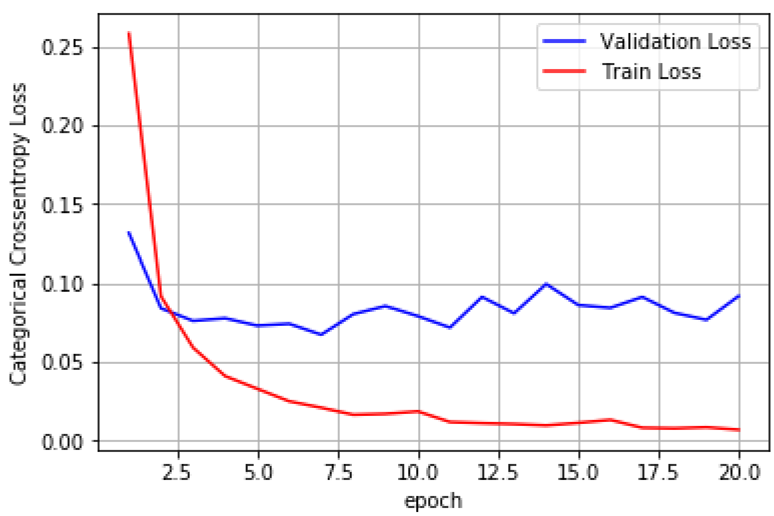
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.2583 -
acc: 0.9237 - val_loss: 0.1316 - val_acc: 0.9590
Epoch 2/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0916 -
acc: 0.9723 - val_loss: 0.0840 - val_acc: 0.9730
Epoch 3/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0591 -
acc: 0.9815 - val_loss: 0.0760 - val_acc: 0.9766
Epoch 4/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0409 -
acc: 0.9874 - val_loss: 0.0777 - val_acc: 0.9755
Epoch 5/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0329 -
acc: 0.9889 - val_loss: 0.0730 - val_acc: 0.9784
Epoch 6/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0249 -
acc: 0.9918 - val_loss: 0.0742 - val_acc: 0.9788
Epoch 7/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0208 -
acc: 0.9930 - val_loss: 0.0672 - val_acc: 0.9799
Epoch 8/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0164 -
acc: 0.9945 - val_loss: 0.0803 - val_acc: 0.9793
Epoch 9/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.0169 -
acc: 0.9944 - val_loss: 0.0854 - val_acc: 0.9787
Epoch 10/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.0185 -
acc: 0.9939 - val_loss: 0.0790 - val_acc: 0.9795
Epoch 11/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0117 -
acc: 0.9960 - val_loss: 0.0717 - val_acc: 0.9828
Epoch 12/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0110 -
acc: 0.9964 - val_loss: 0.0912 - val_acc: 0.9795
Epoch 13/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0104 -
acc: 0.9965 - val_loss: 0.0808 - val_acc: 0.9820
Epoch 14/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0096 -
acc: 0.9968 - val_loss: 0.0994 - val_acc: 0.9770
Epoch 15/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0112 -
acc: 0.9962 - val_loss: 0.0861 - val_acc: 0.9808
Epoch 16/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0131 -
acc: 0.9956 - val_loss: 0.0843 - val_acc: 0.9802
Epoch 17/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0081 -
acc: 0.9976 - val_loss: 0.0912 - val_acc: 0.9802
Epoch 18/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0077 -
acc: 0.9974 - val_loss: 0.0811 - val_acc: 0.9831
Epoch 19/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0084 -
```

```
acc: 0.9972 - val_loss: 0.0767 - val_acc: 0.9833
Epoch 20/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0068 -
acc: 0.9979 - val_loss: 0.0917 - val_acc: 0.9829
```

In [62]:

```
score = model_assign3.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, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.09172725084941308
Test accuracy: 0.9829
```



5 Layers

In [63]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormal
ization-function-in-keras
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condi
tion with σ=√(2/(ni+ni+1).
# 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 import Dropout

model_assign5 = Sequential()

model_assign5.add(Dense(512, activation='relu', input_shape=(input_dim,), kern
el_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))

model_assign5.add(Dense(256, activation='relu', kernel_initializer=RandomNorma
l(mean=0.0, stddev=0.051, seed=None)) )

model_assign5.add(Dense(128, activation='relu', kernel_initializer=RandomNorma
l(mean=0.0, stddev=0.072, seed=None)) )

model_assign5.add(Dense(64, activation='relu', kernel_initializer=RandomNormal
(mean=0.0, stddev=0.10, seed=None)) )

model_assign5.add(Dense(32, activation='relu', kernel_initializer=RandomNormal
(mean=0.0, stddev=0.144, seed=None)) )

model_assign5.add(Dense(output_dim, activation='softmax'))


model_assign5.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_48 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_49 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_50 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_51 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_52 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_53 (Dense)             (None, 10)                330
=================================================================
Total params: 576,810
Trainable params: 576,810
Non-trainable params: 0
_________________________________________________________________
```

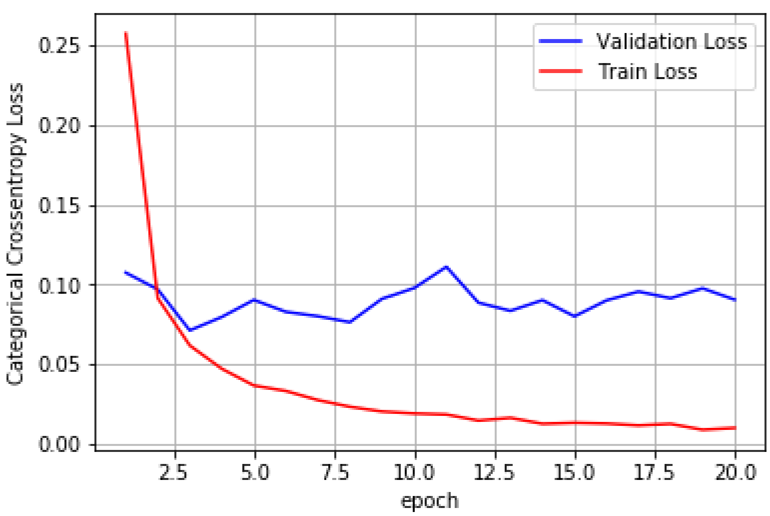
In [64]:
```
model_assign5.compile(optimizer='adam', loss='categorical_crossentropy', metri
cs=['accuracy'])

history = model_assign5.fit(X_train, Y_train, batch_size=batch_size, epochs=nb
_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2572 -
acc: 0.9217 - val_loss: 0.1074 - val_acc: 0.9669
Epoch 2/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0914 -
acc: 0.9713 - val_loss: 0.0969 - val_acc: 0.9664
Epoch 3/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0617 -
acc: 0.9806 - val_loss: 0.0712 - val_acc: 0.9782
Epoch 4/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0471 -
acc: 0.9852 - val_loss: 0.0796 - val_acc: 0.9769
Epoch 5/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0366 -
acc: 0.9888 - val_loss: 0.0903 - val_acc: 0.9745
Epoch 6/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0332 -
acc: 0.9895 - val_loss: 0.0828 - val_acc: 0.9766
Epoch 7/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0274 -
acc: 0.9916 - val_loss: 0.0801 - val_acc: 0.9785
Epoch 8/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0233 -
acc: 0.9925 - val_loss: 0.0763 - val_acc: 0.9806
Epoch 9/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0203 -
acc: 0.9939 - val_loss: 0.0910 - val_acc: 0.9793
Epoch 10/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0190 -
acc: 0.9940 - val_loss: 0.0978 - val_acc: 0.9784
Epoch 11/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0184 -
acc: 0.9943 - val_loss: 0.1111 - val_acc: 0.9741
Epoch 12/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0146 -
acc: 0.9954 - val_loss: 0.0886 - val_acc: 0.9806
Epoch 13/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0163 -
acc: 0.9947 - val_loss: 0.0835 - val_acc: 0.9832
Epoch 14/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0126 -
acc: 0.9961 - val_loss: 0.0901 - val_acc: 0.9806
Epoch 15/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0132 -
acc: 0.9961 - val_loss: 0.0800 - val_acc: 0.9816
Epoch 16/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.0127 -
acc: 0.9963 - val_loss: 0.0900 - val_acc: 0.9823
Epoch 17/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0115 -
acc: 0.9967 - val_loss: 0.0955 - val_acc: 0.9808
Epoch 18/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0125 -
acc: 0.9963 - val_loss: 0.0914 - val_acc: 0.9800
Epoch 19/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0088 -
```

```
acc: 0.9974 - val_loss: 0.0976 - val_acc: 0.9801
Epoch 20/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0100 -
acc: 0.9969 - val_loss: 0.0905 - val_acc: 0.9826
```

In [65]:
```
score = model_assign5.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, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.0904728904638242
Test accuracy: 0.9826
```



Pretty Table

In [2]:
```
# http://zetcode.com/python/prettytable/
from prettytable import PrettyTable

#If you get a ModuleNotFoundError error , install prettytable using: pip3 inst
all prettytable

x = PrettyTable()
x.field_names = ["Model", "Training accuracy", "Test Accuracy", "val_loss"]

x.add_row(["MLP(BN+Relu+Dropout) 2 layer", 0.977, 0.98, 0.0661])
x.add_row(["MLP(BN+Relu+Dropout) 3 layer", 0.978, 0.98, 0.0672])
x.add_row(["MLP(BN+Relu+Dropout) 5 layer", 0.9703, 0.9796, 0.0864])
x.add_row(["MLP(Relu) 2 layer", 0.999, 0.97, 0.103])
x.add_row(["MLP(Relu) 3 layer", 0.998, 0.9829, 0.0917])
x.add_row(["MLP(Relu) 5 layer", 0.997, 0.98, 0.0905])
print(x)
```

```
+------------------------------+-------------------+---------------+---------
-+
|            Model             | Training accuracy | Test Accuracy | val_loss
|
+------------------------------+-------------------+---------------+---------
-+
| MLP(BN+Relu+Dropout) 2 layer |       0.977       |      0.98     |  0.0661
|
| MLP(BN+Relu+Dropout) 3 layer |       0.978       |      0.98     |  0.0672
|
| MLP(BN+Relu+Dropout) 5 layer |       0.9703      |     0.9796    |  0.0864
|
|      MLP(Relu) 2 layer       |       0.999       |      0.97     |  0.103
|
|      MLP(Relu) 3 layer       |       0.998       |     0.9829    |  0.0917
|
|      MLP(Relu) 5 layer       |       0.997       |      0.98     |  0.0905
|
+------------------------------+-------------------+---------------+---------
-+
```

ConclusionFrom the preety table we can observe that 1. loss is minimum when model is 2 layer with BN_Relu+Dropout 2. model has maximum accuracy when mlp with 2 layer 3. when we play with different parameters of mlp like BN Droupout our loss will decreaseSteps Taken1. import the necessary library 2. Import mnist dataset 3. loading train+ test data mnist 4. normalization 5. Giving Output a 10 class 7. Applying models with different parameters