Handwritten Image Detection with Keras using MNIST data

In this exercise we will work with image data: specifically the famous MNIST data set. This data set contains 70,000 images of handwritten digits in grayscale (0=black, 255 = white). The images are 28 pixels by 28 pixels for a total of 784 pixels. This is quite small by image standards. Also, the images are well centered and isolated. This makes this problem solvable with standard fully connected neural nets without too much pre-work.

In the first part of this notebook, we will walk you through loading in the data, building a network, and training it. Then it will be your turn to try different models and see if you can improve performance

In [2]:

```
# Preliminaries

from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
Using TensorFlow backend.
```

Let's explore the dataset a little bit

In [3]:

```
# Load the data, shuffled and split between train and test sets (x_train
and y_rain)
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

In [4]:

```
x_train[0].shape
```

Out[4]:

```
(28, 28)
```

In [5]:
```
#Let's just look at a particular example to see what is inside

x_train[333]  ## Just a 28 x 28 numpy array of ints from 0 to 255
```

```
Out[5]: array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  87, 13
8,
                170, 253, 201, 244, 212, 222, 138,  86,  22,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  95, 253, 25
2,
                252, 252, 252, 253, 252, 252, 252, 252, 245,  80,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0,  68, 246, 205,   6
9,
                 69,  69,  69,  69,  69,  69,  69, 205, 253, 240,  50,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0, 187, 252, 218,   3
4,
                  0,   0,   0,   0,   0,   0,   0, 116, 253, 252,  69,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0, 116, 248, 252, 253,   9
2,
                  0,   0,   0,   0,   0,   0,  95, 230, 253, 157,   6,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0, 116, 249, 253, 189,   4
2,
                  0,   0,   0,   0,  36, 170, 253, 243, 158,   0,   0,   0,
0,
                  0,   0],
               [  0,   0,   0,   0,   0,   0,   0,   0,   0, 133, 252, 245, 14
0,
```

```
          34,   0,   0,  57, 219, 252, 235,  60,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,   0,  25, 205, 253, 25
2,
         234, 184, 184, 253, 240, 100,  44,   0,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  21, 161, 21
9,
         252, 252, 252, 234,  37,   0,   0,   0,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 20
3,
         252, 252, 252, 251, 135,   0,   0,   0,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,   0,   9,  76, 255, 25
3,
         205, 168, 220, 255, 253, 137,   5,   0,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,   0, 114, 252, 249, 13
2,
          25,   0,   0, 180, 252, 252,  45,   0,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,  51, 220, 252, 199,
0,
           0,   0,   0,  38, 186, 252, 154,   7,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0, 184, 252, 252,  21,
0,
           0,   0,   0,   0,  67, 252, 252,  22,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0, 184, 252, 200,   0,
0,
           0,   0,   0,   0,  47, 252, 252,  22,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0, 185, 253, 201,   0,
0,
           0,   0,   0,   3, 118, 253, 245,  21,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0, 163, 252, 252,   0,
0,
           0,   0,   0,  97, 252, 252,  87,   0,   0,   0,   0,   0,
0,
           0,   0],
       [   0,   0,   0,   0,   0,   0,   0,   0,  51, 240, 252, 123,   7
0,
          70, 112, 184, 222, 252, 170,  13,   0,   0,   0,   0,   0,
0,
```

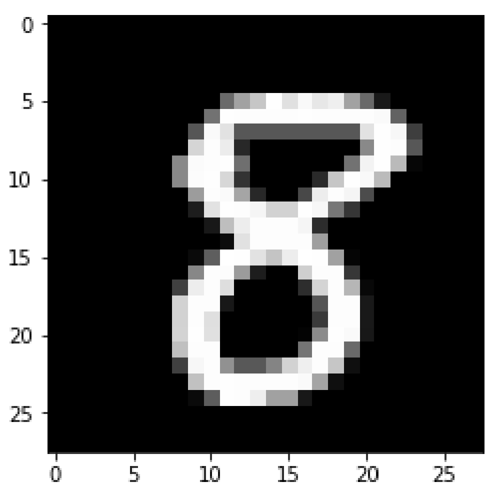
```
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 165, 252, 253, 25
2,
        252, 252, 252, 245, 139,  13,   0,   0,   0,   0,   0,   0,
0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   9,  75, 253, 25
2,
        221, 137, 137,  21,   0,   0,   0,   0,   0,   0,   0,   0,
0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
         0,   0]], dtype=uint8)
```

In [6]:
```
# What is the corresponding label in the training set?
y_train[333]
```

Out[6]: 8

In [7]:
```
# Let's see what this image actually looks like

plt.imshow(x_train[333], cmap='Greys_r')
```

Out[7]: <matplotlib.image.AxesImage at 0x1283ad0f0>



In [8]:
```
# this is the shape of the np.array x_train
# it is 3 dimensional.
print(x_train.shape, 'train samples')
print(x_test.shape, 'test samples')
```

```
(60000, 28, 28) train samples
(10000, 28, 28) test samples
```

In [9]:
```
## For our purposes, these images are just a vector of 784 inputs, so le
t's convert
x_train = x_train.reshape(len(x_train), 28*28)
x_test = x_test.reshape(len(x_test), 28*28)

## Keras works with floats, so we must cast the numbers to floats
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

## Normalize the inputs so they are between 0 and 1
x_train /= 255
x_test /= 255
```

In [10]:
```
# convert class vectors to binary class matrices
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

y_train[333]  # now the digit k is represented by a 1 in the kth entry
 (0-indexed) of the length 10 vector
```

Out[10]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 1., 0.], dtype=float32)
```

In [11]:
```
# We will build a model with two hidden layers of size 64
# Fully connected inputs at each layer
# We will use dropout of .2 to help regularize
model_1 = Sequential()
model_1.add(Dense(64, activation='relu', input_shape=(784,)))
model_1.add(Dropout(0.2))
model_1.add(Dense(64, activation='relu'))
model_1.add(Dropout(0.2))
model_1.add(Dense(10, activation='softmax'))
```

```
WARNING:tensorflow:From /anaconda3/envs/cecs551/lib/python3.6/site-pack
ages/tensorflow/python/framework/op_def_library.py:263: colocate_with
(from tensorflow.python.framework.ops) is deprecated and will be remove
d in a future version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /anaconda3/envs/cecs551/lib/python3.6/site-pack
ages/keras/backend/tensorflow_backend.py:3445: calling dropout (from te
nsorflow.python.ops.nn_ops) with keep_prob is deprecated and will be re
moved in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate =
1 - keep_prob`.
```

In [12]:
```
## Note that this model has a LOT of parameters
model_1.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 64)                50240     
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160      
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0         
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650       
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

In [13]:
```
# Let's compile the model
learning_rate = .001
model_1.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=learning_rate),
              metrics=['accuracy'])
# note that `categorical cross entropy` is the natural generalization
# of the loss function we had in binary classification case, to multi cl
ass case
```

In [14]:
```
# And now let's fit.

batch_size = 128  # mini-batch with 128 examples
epochs = 30
history = model_1.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
WARNING:tensorflow:From /anaconda3/envs/cecs551/lib/python3.6/site-pack
ages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflow.
python.ops.math_ops) is deprecated and will be removed in a future vers
ion.
Instructions for updating:
Use tf.cast instead.
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 1s 24us/step - loss: 0.5
123 - acc: 0.8484 - val_loss: 0.2117 - val_acc: 0.9369
Epoch 2/30
60000/60000 [==============================] - 1s 19us/step - loss: 0.2
467 - acc: 0.9285 - val_loss: 0.1557 - val_acc: 0.9532
Epoch 3/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
994 - acc: 0.9409 - val_loss: 0.1273 - val_acc: 0.9613
Epoch 4/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
698 - acc: 0.9503 - val_loss: 0.1151 - val_acc: 0.9661
Epoch 5/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
529 - acc: 0.9557 - val_loss: 0.1059 - val_acc: 0.9685
Epoch 6/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
384 - acc: 0.9589 - val_loss: 0.1070 - val_acc: 0.9703
Epoch 7/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
308 - acc: 0.9609 - val_loss: 0.1056 - val_acc: 0.9715
Epoch 8/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
207 - acc: 0.9640 - val_loss: 0.0995 - val_acc: 0.9730
Epoch 9/30
60000/60000 [==============================] - 1s 19us/step - loss: 0.1
176 - acc: 0.9652 - val_loss: 0.0995 - val_acc: 0.9739
Epoch 10/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
141 - acc: 0.9662 - val_loss: 0.1035 - val_acc: 0.9732
Epoch 11/30
60000/60000 [==============================] - 1s 19us/step - loss: 0.1
063 - acc: 0.9680 - val_loss: 0.0969 - val_acc: 0.9746
Epoch 12/30
60000/60000 [==============================] - 1s 21us/step - loss: 0.1
070 - acc: 0.9676 - val_loss: 0.0981 - val_acc: 0.9742
Epoch 13/30
60000/60000 [==============================] - 1s 21us/step - loss: 0.1
033 - acc: 0.9695 - val_loss: 0.0946 - val_acc: 0.9742
Epoch 14/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.1
025 - acc: 0.9699 - val_loss: 0.0950 - val_acc: 0.9756
Epoch 15/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
978 - acc: 0.9717 - val_loss: 0.0972 - val_acc: 0.9738
Epoch 16/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
967 - acc: 0.9711 - val_loss: 0.0959 - val_acc: 0.9752
Epoch 17/30
60000/60000 [==============================] - 1s 21us/step - loss: 0.0
```

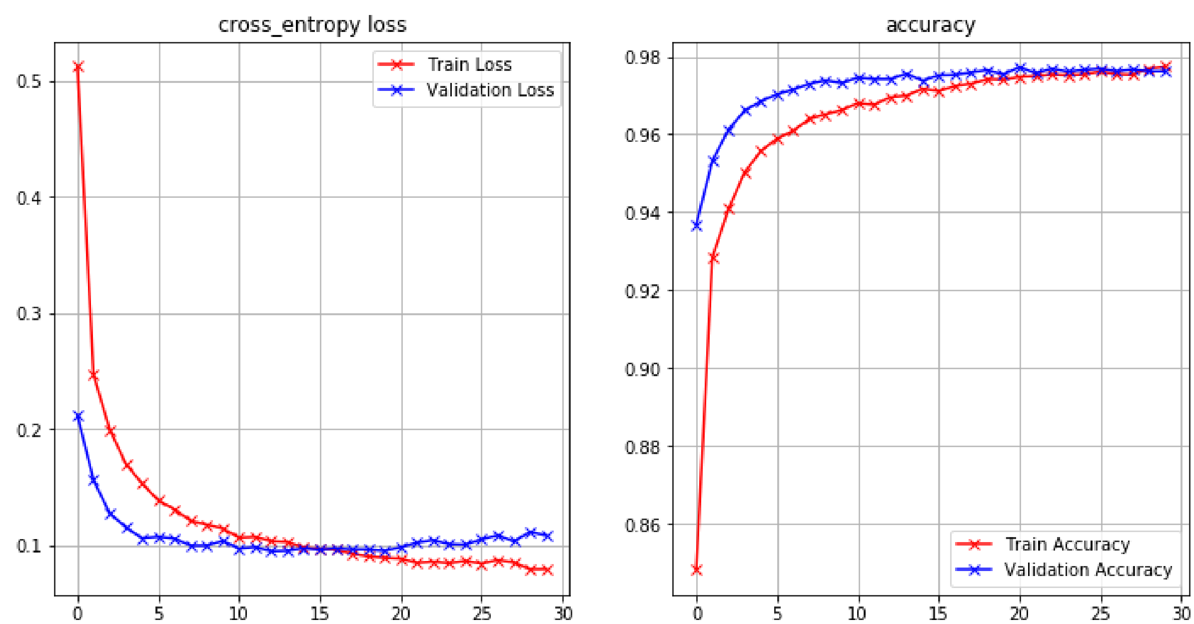
```
959 - acc: 0.9724 - val_loss: 0.0972 - val_acc: 0.9753
Epoch 18/30
60000/60000 [==============================] - 1s 20us/step - loss: 0.0
925 - acc: 0.9730 - val_loss: 0.0963 - val_acc: 0.9759
Epoch 19/30
60000/60000 [==============================] - 1s 20us/step - loss: 0.0
900 - acc: 0.9742 - val_loss: 0.0961 - val_acc: 0.9765
Epoch 20/30
60000/60000 [==============================] - 1s 20us/step - loss: 0.0
890 - acc: 0.9741 - val_loss: 0.0950 - val_acc: 0.9754
Epoch 21/30
60000/60000 [==============================] - 1s 19us/step - loss: 0.0
881 - acc: 0.9748 - val_loss: 0.0981 - val_acc: 0.9773
Epoch 22/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
846 - acc: 0.9750 - val_loss: 0.1019 - val_acc: 0.9757
Epoch 23/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
855 - acc: 0.9754 - val_loss: 0.1042 - val_acc: 0.9769
Epoch 24/30
60000/60000 [==============================] - 1s 17us/step - loss: 0.0
843 - acc: 0.9752 - val_loss: 0.1006 - val_acc: 0.9761
Epoch 25/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
862 - acc: 0.9754 - val_loss: 0.1002 - val_acc: 0.9766
Epoch 26/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
840 - acc: 0.9761 - val_loss: 0.1048 - val_acc: 0.9769
Epoch 27/30
60000/60000 [==============================] - 1s 19us/step - loss: 0.0
869 - acc: 0.9754 - val_loss: 0.1084 - val_acc: 0.9762
Epoch 28/30
60000/60000 [==============================] - 1s 19us/step - loss: 0.0
849 - acc: 0.9753 - val_loss: 0.1031 - val_acc: 0.9767
Epoch 29/30
60000/60000 [==============================] - 1s 18us/step - loss: 0.0
790 - acc: 0.9770 - val_loss: 0.1112 - val_acc: 0.9762
Epoch 30/30
60000/60000 [==============================] - 1s 17us/step - loss: 0.0
791 - acc: 0.9774 - val_loss: 0.1082 - val_acc: 0.9762
```

In [15]:

```
## We will use Keras evaluate function to evaluate performance on the te
st set

score = model_1.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.10819254868056233
Test accuracy: 0.9762
```

In [16]:
```
def plot_loss_accuracy(history):
    fig = plt.figure(figsize=(12, 6))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(history.history["loss"],'r-x', label="Train Loss")
    ax.plot(history.history["val_loss"],'b-x', label="Validation Loss")
    ax.legend()
    ax.set_title('cross_entropy loss')
    ax.grid(True)


    ax = fig.add_subplot(1, 2, 2)
    ax.plot(history.history["acc"],'r-x', label="Train Accuracy")
    ax.plot(history.history["val_acc"],'b-x', label="Validation Accurac
y")
    ax.legend()
    ax.set_title('accuracy')
    ax.grid(True)


plot_loss_accuracy(history)
```



This is reasonably good performance, but we can do even better! Next you will build an even bigger network and compare the performance.

Exercise

Your Turn: Build your own model

Use the Keras "Sequential" functionality to build `model_2` with the following specifications:

1. Two hidden layers.
2. First hidden layer of size 400 and second of size 300
3. Dropout of .4 at each layer
4. How many parameters does your model have? How does it compare with the previous model?
5. Train this model for 20 epochs with RMSProp at a learning rate of .001 and a batch size of 128

In [17]:
```
### Build your model here

# Model 2 has two hidden layers of sizes 400 and 300 respectively
# Fully connected inputs at each layer
# It will use dropout of .4 to help regularize
model_2 = Sequential()
model_2.add(Dense(400, activation='relu', input_shape=(784,)))
model_2.add(Dropout(0.4))
model_2.add(Dense(300, activation='relu'))
model_2.add(Dropout(0.4))
model_2.add(Dense(10, activation='softmax'))
```

In [18]:
```
# Summary of model 2
model_2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 400)               314000
_________________________________________________________________
dropout_3 (Dropout)          (None, 400)               0
_________________________________________________________________
dense_5 (Dense)              (None, 300)               120300
_________________________________________________________________
dropout_4 (Dropout)          (None, 300)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                3010
=================================================================
Total params: 437,310
Trainable params: 437,310
Non-trainable params: 0
_________________________________________________________________
```

In [19]:
```
# We will now compile model 2
learning_rate = .001
model_2.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=learning_rate),
              metrics=['accuracy'])
```

In [20]:
```
# The final step is to fit the model

batch_size = 128  # mini-batch with 128 examples
epochs = 20       # run the model for 20 epochs
history = model_2.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.3
289 - acc: 0.9005 - val_loss: 0.1326 - val_acc: 0.9571
Epoch 2/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.1
551 - acc: 0.9539 - val_loss: 0.0904 - val_acc: 0.9728
Epoch 3/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.1
183 - acc: 0.9658 - val_loss: 0.0817 - val_acc: 0.9746
Epoch 4/20
60000/60000 [==============================] - 4s 58us/step - loss: 0.1
042 - acc: 0.9698 - val_loss: 0.0838 - val_acc: 0.9743
Epoch 5/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0
928 - acc: 0.9732 - val_loss: 0.0728 - val_acc: 0.9793
Epoch 6/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0
831 - acc: 0.9759 - val_loss: 0.0770 - val_acc: 0.9800
Epoch 7/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0
779 - acc: 0.9768 - val_loss: 0.0793 - val_acc: 0.9814
Epoch 8/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0
743 - acc: 0.9789 - val_loss: 0.0823 - val_acc: 0.9797
Epoch 9/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0
706 - acc: 0.9810 - val_loss: 0.0952 - val_acc: 0.9776
Epoch 10/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0
672 - acc: 0.9812 - val_loss: 0.0868 - val_acc: 0.9817
Epoch 11/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0
644 - acc: 0.9816 - val_loss: 0.0870 - val_acc: 0.9795
Epoch 12/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0
643 - acc: 0.9826 - val_loss: 0.0865 - val_acc: 0.9811
Epoch 13/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0
612 - acc: 0.9829 - val_loss: 0.0829 - val_acc: 0.9821
Epoch 14/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0
596 - acc: 0.9846 - val_loss: 0.0869 - val_acc: 0.9828
Epoch 15/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0
580 - acc: 0.9846 - val_loss: 0.0799 - val_acc: 0.9835
Epoch 16/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0
587 - acc: 0.9852 - val_loss: 0.0906 - val_acc: 0.9818
Epoch 17/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0
563 - acc: 0.9852 - val_loss: 0.0885 - val_acc: 0.9832
Epoch 18/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0
539 - acc: 0.9860 - val_loss: 0.0922 - val_acc: 0.9832
Epoch 19/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0
```

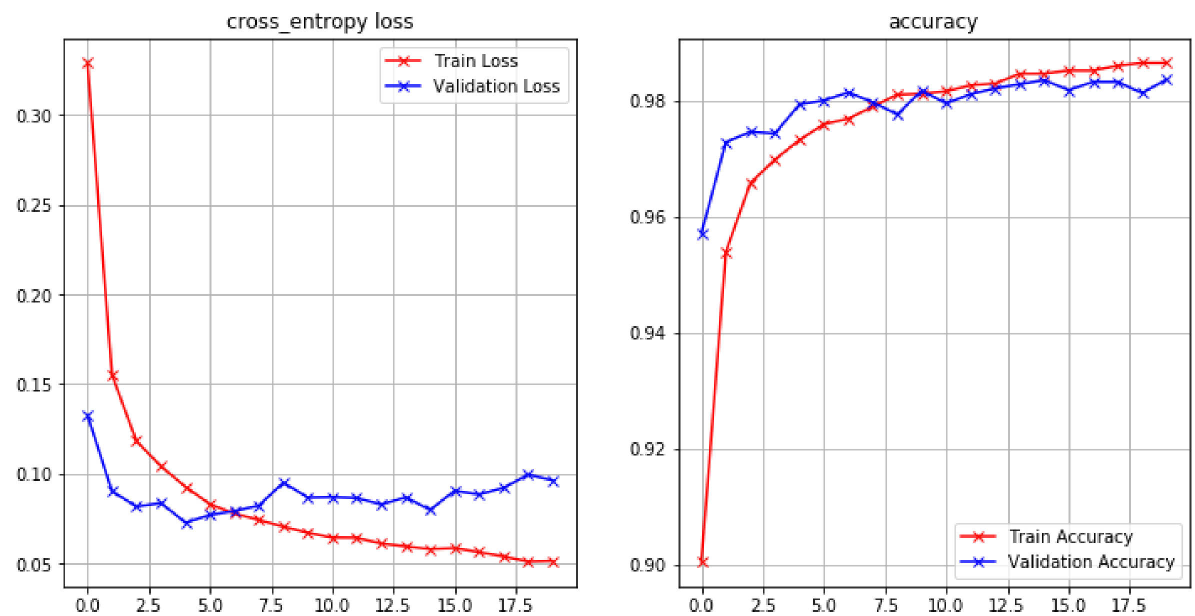
```
512 - acc: 0.9865 - val_loss: 0.0995 - val_acc: 0.9813
Epoch 20/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0
516 - acc: 0.9865 - val_loss: 0.0965 - val_acc: 0.9836
```

SOLUTION

In [21]:
```
## We will again use Keras to evaluate the performance of our model 2 on
the test set
score = model_2.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.09645749624104888
Test accuracy: 0.9836
```

In [22]:
```
def plot_loss_accuracy(history):
    fig = plt.figure(figsize=(12, 6))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(history.history["loss"],'r-x', label="Train Loss")
    ax.plot(history.history["val_loss"],'b-x', label="Validation Loss")
    ax.legend()
    ax.set_title('cross_entropy loss')
    ax.grid(True)


    ax = fig.add_subplot(1, 2, 2)
    ax.plot(history.history["acc"],'r-x', label="Train Accuracy")
    ax.plot(history.history["val_acc"],'b-x', label="Validation Accurac
y")
    ax.legend()
    ax.set_title('accuracy')
    ax.grid(True)


plot_loss_accuracy(history)
```



My answer: Model 1 vs Model 2

1) model_2 has a total of **437,310** parameters, while model_1 only had **55,050** parameters. This means that model_2 has nearly **8 times** the parameters than model_1.

2) Also, model_2 outperformed model_1 on the test set. The accuracy for model_1 was **97.62%** while that of model_2 was **98.28%**.

Think about the following questions

1) How do model_1 and model_2 compare? Which do you prefer? If you were going to put one into production, which would you choose and why?

2) Compare the trajectories of the loss function on the training set and test set for each model? How do they compare? What does that suggest about each model? Do the same for accuracy? Which do you think is more meaningful, the loss or the accuracy?

3) Suggest an improvement to one of the models (changing structure, learning rate, number of epochs, etc.) that you think will result in a better model. Try it out below? Did it improve the performance?

My answers:

1) As mentioned above, model_2 outperformed model_1 on the test set by yielding a smaller loss and a better accuracy. I would prefer model_1. If I was going into production, I would have chosen *model_1* because it has significantly **lesser parameters**, and the accuracy is comparable to *model_2*. Had the difference in the accuracy been major, I would have gone with *model_2*, but in this case, it is only **0.74%**.

2) Discussion:

- **LOSS:** The trajectories of the training loss for both the models are very comparable but *model_2* reaches a smaller loss in lesser number of epochs. The validation loss for *model_1* smoothly hits a minima and then starts to increase. This also happens in *model_2* but is more exaggerated. The exaggeration could be attributed to overfitting.
- **ACCURACY:** Like the loss, the training accuracies of both the models increase smoothly indicating learning. But the validation accuracy for *model_1* plateaus after a certain point, which shows a **saturation** in the model. On the other hand, *model_2* shows **overfitting** due to the gap between the training and validation accuracy after around 10 epochs.
- I think it is more meaningful to use ***accuracy*** to compare performance since the most important goal of out model is to reduce overfitting, or the gap between the training and validation accuracy.

3) We can use regularization. In my experiments below, following obervations were made:

- **Learning Rate**: I tried to compile model_1 with a smaller learning rate (lr = 0.0001) and it perfomed very poorly, giving a test accuracy of **8.92%** and loss of **14.68**.
- **Regularization**: In the next experiment, I created my own model with *L2 regularization* in each layer, and it performed similar to model_1, giving an accuracy of **96.78%** and a loss of **0.14**. Although, the best accuracy so far is still achieved by model_2, but model_3 performs the most consistently.

In [30]:
```
# We will now compile model 1 with a smaller learning rate
learning_rate = .0001
model_1.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=learning_rate),
              metrics=['accuracy'])

# Fitting the model again
batch_size = 128  # mini-batch with 128 examples
epochs = 30       # run the model for 30 epochs
history = model_1.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 1s 24us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 2/30
60000/60000 [==============================] - 1s 21us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 3/30
60000/60000 [==============================] - 1s 21us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 4/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 5/30
60000/60000 [==============================] - 1s 20us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 6/30
60000/60000 [==============================] - 1s 20us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 7/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 8/30
60000/60000 [==============================] - 1s 21us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 9/30
60000/60000 [==============================] - 1s 20us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 10/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 11/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 12/30
60000/60000 [==============================] - 1s 20us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 13/30
60000/60000 [==============================] - 1s 20us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 14/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 15/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 16/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 17/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 18/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 19/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
```

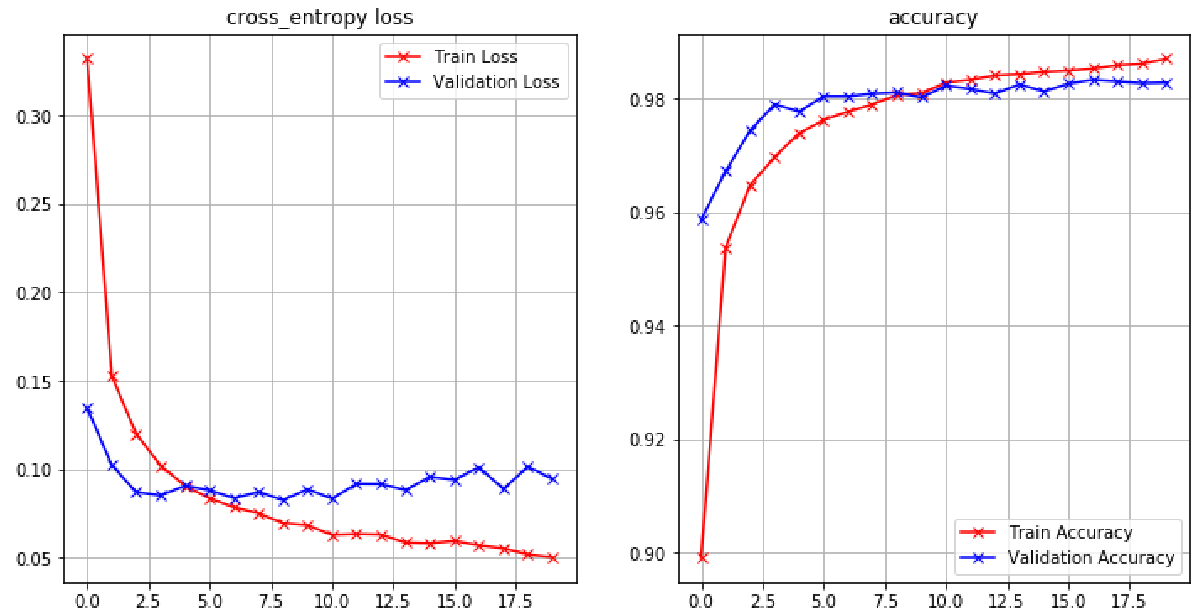
```
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 20/30
60000/60000 [==============================] - 1s 20us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 21/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 22/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 23/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 24/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 25/30
60000/60000 [==============================] - 1s 17us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 26/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 27/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 28/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 29/30
60000/60000 [==============================] - 1s 18us/step - loss: 14.
6618 - acc: 0.0904 - val_loss: 14.6804 - val_acc: 0.0892
Epoch 30/30
60000/60000 [==============================] - 1s 19us/step - loss: 14.
6618 - acc: 0.0903 - val_loss: 14.6804 - val_acc: 0.0892
```

In [31]:

```
## We will again use Keras to evaluate the performance of our model 2 on
the test set
score = model_1.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 14.680361064147949
Test accuracy: 0.0892
```

In [24]:
```
plot_loss_accuracy(history)
```



In [26]:
```
# We will now build a completely new model structure
model_3 = Sequential()

# Model 3 has two hidden layers of sizes 400 and 300 respectively
# Fully connected inputs at each layer
# It will use dropout of .5 to help regularize
# We also use individual layer L2 regularization and it helps in gaining
performance
model_3.add(Dense(64, activation='relu', input_shape=(784,), kernel_regu
larizer=keras.regularizers.l2(0.0001)))
model_3.add(Dropout(0.5))
model_3.add(Dense(16, activation='relu', input_shape=(784,), kernel_regu
larizer=keras.regularizers.l2(0.0001)))
model_2.add(Dropout(0.5))
model_3.add(Dense(10, activation='softmax'))

# Model architecture and summary
model_3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 64)                50240
_________________________________________________________________
dropout_5 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_8 (Dense)              (None, 16)                1040
_________________________________________________________________
dense_9 (Dense)              (None, 10)                170
=================================================================
Total params: 51,450
Trainable params: 51,450
Non-trainable params: 0
_________________________________________________________________
```

In [27]:
```
# We will now compile model 1 with a smaller learning rate
learning_rate = 1e-3
model_3.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=learning_rate),
              metrics=['accuracy'])
```

In [28]:
```
# Fitting the model again
batch_size = 128  # mini-batch with 128 examples
epochs = 30
history = model_3.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 3s 43us/step - loss: 0.7
535 - acc: 0.7753 - val_loss: 0.2947 - val_acc: 0.9196
Epoch 2/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.4
124 - acc: 0.8834 - val_loss: 0.2388 - val_acc: 0.9344
Epoch 3/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.3
603 - acc: 0.8981 - val_loss: 0.2141 - val_acc: 0.9424
Epoch 4/30
60000/60000 [==============================] - 2s 28us/step - loss: 0.3
326 - acc: 0.9071 - val_loss: 0.1937 - val_acc: 0.9480
Epoch 5/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.3
186 - acc: 0.9120 - val_loss: 0.1855 - val_acc: 0.9494
Epoch 6/30
60000/60000 [==============================] - 2s 26us/step - loss: 0.3
037 - acc: 0.9157 - val_loss: 0.1752 - val_acc: 0.9543
Epoch 7/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.2
974 - acc: 0.9177 - val_loss: 0.1690 - val_acc: 0.9566
Epoch 8/30
60000/60000 [==============================] - 2s 26us/step - loss: 0.2
857 - acc: 0.9228 - val_loss: 0.1651 - val_acc: 0.9568
Epoch 9/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.2
809 - acc: 0.9223 - val_loss: 0.1613 - val_acc: 0.9594
Epoch 10/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.2
743 - acc: 0.9250 - val_loss: 0.1579 - val_acc: 0.9601
Epoch 11/30
60000/60000 [==============================] - 2s 31us/step - loss: 0.2
678 - acc: 0.9277 - val_loss: 0.1550 - val_acc: 0.9630
Epoch 12/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.2
658 - acc: 0.9273 - val_loss: 0.1532 - val_acc: 0.9626
Epoch 13/30
60000/60000 [==============================] - 2s 28us/step - loss: 0.2
603 - acc: 0.9301 - val_loss: 0.1546 - val_acc: 0.9649
Epoch 14/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.2
605 - acc: 0.9300 - val_loss: 0.1507 - val_acc: 0.9640
Epoch 15/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.2
587 - acc: 0.9296 - val_loss: 0.1491 - val_acc: 0.9634
Epoch 16/30
60000/60000 [==============================] - 2s 26us/step - loss: 0.2
570 - acc: 0.9306 - val_loss: 0.1460 - val_acc: 0.9651
Epoch 17/30
60000/60000 [==============================] - 1s 25us/step - loss: 0.2
523 - acc: 0.9320 - val_loss: 0.1463 - val_acc: 0.9668
Epoch 18/30
60000/60000 [==============================] - 1s 25us/step - loss: 0.2
479 - acc: 0.9334 - val_loss: 0.1485 - val_acc: 0.9659
Epoch 19/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.2
```

```
497 - acc: 0.9330 - val_loss: 0.1468 - val_acc: 0.9668
Epoch 20/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.2
457 - acc: 0.9339 - val_loss: 0.1463 - val_acc: 0.9660
Epoch 21/30
60000/60000 [==============================] - 2s 26us/step - loss: 0.2
440 - acc: 0.9341 - val_loss: 0.1431 - val_acc: 0.9669
Epoch 22/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.2
403 - acc: 0.9358 - val_loss: 0.1470 - val_acc: 0.9659
Epoch 23/30
60000/60000 [==============================] - 2s 29us/step - loss: 0.2
434 - acc: 0.9340 - val_loss: 0.1440 - val_acc: 0.9663
Epoch 24/30
60000/60000 [==============================] - 2s 27us/step - loss: 0.2
405 - acc: 0.9355 - val_loss: 0.1421 - val_acc: 0.9672
Epoch 25/30
60000/60000 [==============================] - 1s 25us/step - loss: 0.2
413 - acc: 0.9351 - val_loss: 0.1416 - val_acc: 0.9682
Epoch 26/30
60000/60000 [==============================] - 2s 25us/step - loss: 0.2
383 - acc: 0.9350 - val_loss: 0.1416 - val_acc: 0.9695
Epoch 27/30
60000/60000 [==============================] - 1s 24us/step - loss: 0.2
384 - acc: 0.9358 - val_loss: 0.1380 - val_acc: 0.9680
Epoch 28/30
60000/60000 [==============================] - 1s 24us/step - loss: 0.2
410 - acc: 0.9366 - val_loss: 0.1390 - val_acc: 0.9690
Epoch 29/30
60000/60000 [==============================] - 2s 26us/step - loss: 0.2
409 - acc: 0.9353 - val_loss: 0.1428 - val_acc: 0.9672
Epoch 30/30
60000/60000 [==============================] - 2s 25us/step - loss: 0.2
365 - acc: 0.9365 - val_loss: 0.1409 - val_acc: 0.9678
```

In [29]:
```
## We will again use Keras to evaluate the performance of our model 2 on the test set
score = model_3.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

plot_loss_accuracy(history)
```

```
Test loss: 0.1409323357641697
Test accuracy: 0.9678
```