Foundations of Data Mining: Assignment 4

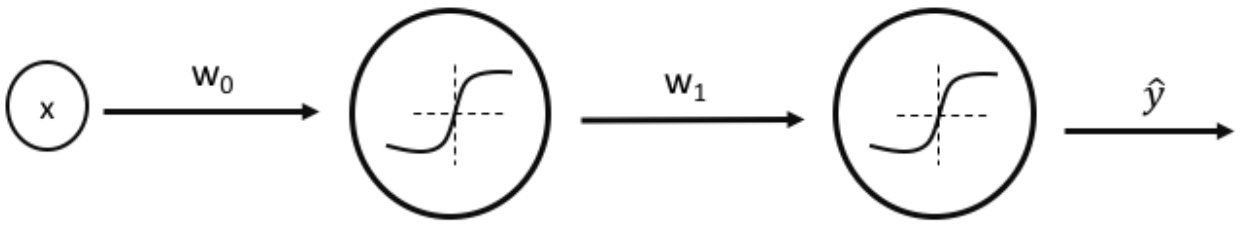
- N.A.F. van de L'Isle (1022588)
- E.Peer (0746176)

Please complete all assignments in this notebook. You should submit this notebook, as well as a PDF version (See File > Download as).

In [50]:
```
%matplotlib inline
from preamble import *
plt.rcParams['savefig.dpi'] = 100 # This controls the size of your 
figures
# Comment out and restart notebook if you only want the last output 
of each cell.
# InteractiveShell.ast_node_interactivity = "all"
```

Backpropagation (3 points)

Figure 1 illustrates a simple neural network model.



It has single input x, two layers with one neuron each. The activation function of both layers is ReLU.

The parameters w_0 and w_1 (no biases) are initialized to the following values $w_0 = 1$ and $w_1 = 2$. Implement a single update step of the gradient descent algorithm by hand. Run the update state for the following two data points:

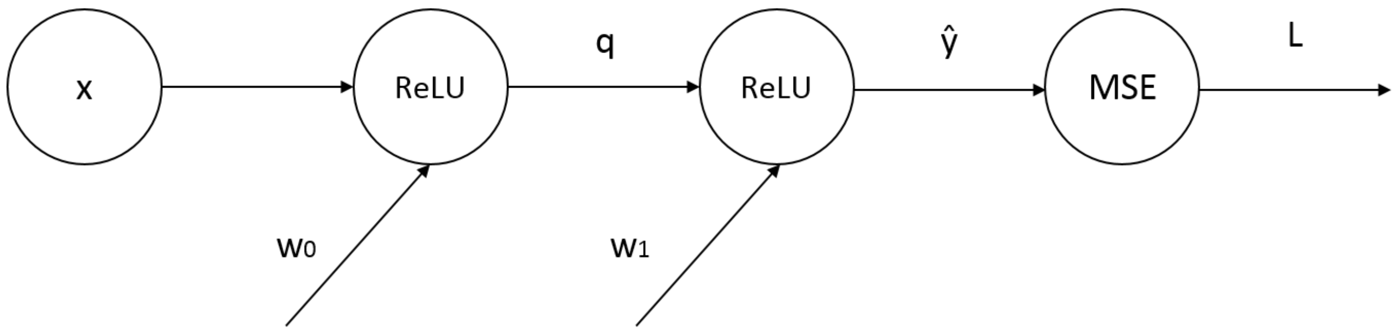
- $(1, 2)$
- $(2, 3)$

The goal is to model the relationship between two continuous variables. The learning rate is set to 0.1

Provide the solution in the following format:

- A choice for a loss function
- Compute graph for training the neural network
- Partial derivative expression for each of the parameters in the model
- The update expression for each of the parameters for each of the data-points
- The final value of both parameters after the single step in the gradient descent algorithm

- loss function: MSE
- Compute graph:



We can use two approaches to compute new values for the weights w_0 and w_1. The first one is to update the weights after each observation, thus taking batchsize = 1. The second is to take the average of the computed weights and thus, only update once, taking batchsize = 2. We will show how to do the first approach. Note that the weights would decrease less if we take batchsize = 2 because of the values for the first observation being equal to zero. Moreover, the change for the weights would then take all observations into account, thus, being more accurate.

Observations: $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (2, 3)$ Weights: $w_0 = 1$ and $w_1 = 2$

We use the MSE as loss function: $L = \frac{1}{2n}\sum_{i=0}^{n}(\hat{y}_i - y_i)^2$.

The first approach(batchsize = 1): \begin{align} \hat{y}_1 &= w_1(w_0x_1)\ &= 211 = 2\ L &= \frac{1}{2}(2-2)^2\ & = 0\ \end{align}

Note that we now go over the data points one by one. Thus, the loss function does not have to average out over n. As $L = 0$ the derivatives will all be zero and, therefore, the weights will not be updated. Thus, we continue to the next observation.

$$\begin{aligned}
\hat{y}_2 &= w_1(w_0x_2)\\
&= 2*1*2 = 4\\
L &= \tfrac{1}{2}(4-3)^2 = \tfrac{1}{2}\\
\frac{\partial L}{\partial \hat{y}_2} &= \left[\tfrac{1}{2}(\hat{y}_2 - y_2)^2\right]'\\
&= 2*\tfrac{1}{2}(\hat{y}_2 - y_2)*1\\
&= \hat{y}_2 - y_2\\
&= 1\\
\frac{\partial L}{\partial w_1} &= \frac{\partial \hat{y}_2}{\partial w_1}\frac{\partial L}{\partial \hat{y}_2}\\
&= w_0x_2 * \frac{\partial L}{\partial \hat{y}_2}\\
&= 2*1\\
&= 2\\
\frac{\partial L}{\partial q} &= \frac{\partial \hat{y}_2}{\partial q}\frac{\partial L}{\partial \hat{y}_2}\\
&= w_1 * 1\\
&= 2\\
\frac{\partial L}{\partial w_0} &= \frac{\partial q}{\partial w_0}\frac{\partial L}{\partial q}\\
&= x_2 * \frac{\partial L}{\partial q}\\
&= 2*2 = 4
\end{aligned}$$

Thus, $w_1^* = w_1 - 0.1*2 = 1.8$ and $w_0^* = w_0 - 0.1*4 = 0.6$.

Training Deep Models (3 points)

The model in the example code below performs poorly as its depth increases. Train this model on the MNIST digit detection task.

Examine its training performance by gradually increasing its depth:

- Set the depth to 1 hidden layer
- Set the depth to 2 hidden layers
- Set the depth to 3 hidden layers

Modify the model such that you improve its performance when its depth increases. Train the new model again for the different depths:

- Set the depth to 1 hidden layer
- Set the depth to 2 hidden layers
- Set the depth to 3 hidden layers

In [2]:
```
# (You don't need to change this part of the code)
from __future__ import print_function
import numpy as np
np.random.seed(1234)

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from keras.utils import np_utils


import matplotlib.pyplot as plt

batch_size = 128
nb_classes = 10
nb_epoch = 10
```

```
Using TensorFlow backend.
```

In [3]:
```
# (You don't need to change this part of the code)
# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

```
60000 train samples
10000 test samples
```

In [4]:
```
# Use this parameter to change the depth of the model
number_hidden_layers = 1  # Number of hidden layers
```

In [45]:
```python
acc_results = []
valacc_results = []

for layer in range(1,4):
    number_hidden_layers = layer
    # Model
    model = Sequential()
    model.add(Dense(512, input_shape=(784,), activation='sigmoid'))
    model.add(Dropout(0.2))

    while number_hidden_layers > 1:
        model.add(Dense(512))
        model.add(Activation('sigmoid'))
        model.add(Dropout(0.2))
        number_hidden_layers -= 1


    model.add(Dense(10))
    model.add(Activation('softmax'))

    model.summary()

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

    # Training (You don't need to change this part of the code)
    history = model.fit(X_train, Y_train,
                        batch_size=batch_size, nb_epoch=nb_epoch,
                        verbose=1, validation_data=(X_test, Y_test)
)
    score = model.evaluate(X_test, Y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    acc_results.append(history.history['acc'])
    valacc_results.append(history.history['val_acc'])
```

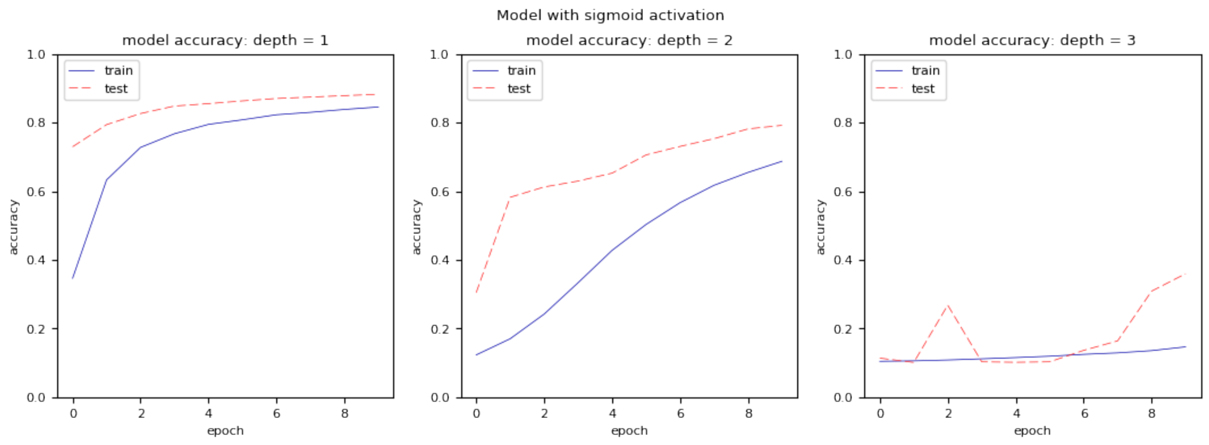
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_30 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_20 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_31 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_20 (Activation)   (None, 10)                0
=================================================================
Total params: 407,050.0
Trainable params: 407,050.0
Non-trainable params: 0.0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 5s - loss: 2.0181 -
acc: 0.3463 - val_loss: 1.6341 - val_acc: 0.7296
Epoch 2/10
60000/60000 [==============================] - 5s - loss: 1.4710 -
```

```
acc: 0.6330 - val_loss: 1.1844 - val_acc: 0.7941
Epoch 3/10
60000/60000 [==============================] - 5s - loss: 1.1223 -
acc: 0.7278 - val_loss: 0.9180 - val_acc: 0.8264
Epoch 4/10
60000/60000 [==============================] - 5s - loss: 0.9223 -
acc: 0.7676 - val_loss: 0.7616 - val_acc: 0.8483
Epoch 5/10
60000/60000 [==============================] - 5s - loss: 0.7959 -
acc: 0.7949 - val_loss: 0.6642 - val_acc: 0.8553
Epoch 6/10
60000/60000 [==============================] - 5s - loss: 0.7150 -
acc: 0.8081 - val_loss: 0.5981 - val_acc: 0.8633
Epoch 7/10
60000/60000 [==============================] - 5s - loss: 0.6545 -
acc: 0.8229 - val_loss: 0.5507 - val_acc: 0.8703
Epoch 8/10
60000/60000 [==============================] - 5s - loss: 0.6126 -
acc: 0.8299 - val_loss: 0.5159 - val_acc: 0.8744
Epoch 9/10
60000/60000 [==============================] - 5s - loss: 0.5792 -
acc: 0.8383 - val_loss: 0.4872 - val_acc: 0.8786
Epoch 10/10
60000/60000 [==============================] - 5s - loss: 0.5536 -
acc: 0.8455 - val_loss: 0.4655 - val_acc: 0.8825
Test score: 0.465494298959
Test accuracy: 0.8825
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_32 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_21 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_33 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_21 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_22 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_34 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_22 (Activation)   (None, 10)                0
=================================================================
Total params: 669,706.0
Trainable params: 669,706.0
Non-trainable params: 0.0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 10s - loss: 2.3288 -
acc: 0.1229 - val_loss: 2.2327 - val_acc: 0.3044
Epoch 2/10
60000/60000 [==============================] - 10s - loss: 2.2493 -
acc: 0.1699 - val_loss: 2.1499 - val_acc: 0.5823
Epoch 3/10
60000/60000 [==============================] - 11s - loss: 2.1580 -
acc: 0.2418 - val_loss: 2.0408 - val_acc: 0.6124
Epoch 4/10
60000/60000 [==============================] - 9s - loss: 2.0333 -
```

```
acc: 0.3334 - val_loss: 1.8857 - val_acc: 0.6296
Epoch 5/10
60000/60000 [==============================] - 10s - loss: 1.8638 -
acc: 0.4276 - val_loss: 1.6820 - val_acc: 0.6527
Epoch 6/10
60000/60000 [==============================] - 9s - loss: 1.6614 -
acc: 0.5031 - val_loss: 1.4556 - val_acc: 0.7058
Epoch 7/10
60000/60000 [==============================] - 11s - loss: 1.4529 -
acc: 0.5665 - val_loss: 1.2480 - val_acc: 0.7306
Epoch 8/10
60000/60000 [==============================] - 12s - loss: 1.2743 -
acc: 0.6173 - val_loss: 1.0848 - val_acc: 0.7531
Epoch 9/10
60000/60000 [==============================] - 10s - loss: 1.1325 -
acc: 0.6547 - val_loss: 0.9583 - val_acc: 0.7816
Epoch 10/10
60000/60000 [==============================] - 9s - loss: 1.0264 -
acc: 0.6872 - val_loss: 0.8651 - val_acc: 0.7920
Test score: 0.865106915092
Test accuracy: 0.792
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_23 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_23 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_24 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_24 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_25 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_38 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_25 (Activation)   (None, 10)                0
=================================================================
Total params: 932,362.0
Trainable params: 932,362.0
Non-trainable params: 0.0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 14s - loss: 2.3602 -
acc: 0.1041 - val_loss: 2.2998 - val_acc: 0.1135
Epoch 2/10
60000/60000 [==============================] - 13s - loss: 2.3469 -
acc: 0.1055 - val_loss: 2.2981 - val_acc: 0.1010
Epoch 3/10
60000/60000 [==============================] - 13s - loss: 2.3374 -
acc: 0.1080 - val_loss: 2.2889 - val_acc: 0.2679
Epoch 4/10
60000/60000 [==============================] - 13s - loss: 2.3281 -
```

```
acc: 0.1114 - val_loss: 2.2884 - val_acc: 0.1028
Epoch 5/10
60000/60000 [==============================] - 15s - loss: 2.3195 -
acc: 0.1150 - val_loss: 2.2806 - val_acc: 0.1014
Epoch 6/10
60000/60000 [==============================] - 15s - loss: 2.3125 -
acc: 0.1191 - val_loss: 2.2739 - val_acc: 0.1028
Epoch 7/10
60000/60000 [==============================] - 14s - loss: 2.3029 -
acc: 0.1245 - val_loss: 2.2685 - val_acc: 0.1361
Epoch 8/10
60000/60000 [==============================] - 14s - loss: 2.2967 -
acc: 0.1288 - val_loss: 2.2631 - val_acc: 0.1637
Epoch 9/10
60000/60000 [==============================] - 15s - loss: 2.2889 -
acc: 0.1353 - val_loss: 2.2533 - val_acc: 0.3087
Epoch 10/10
60000/60000 [==============================] - 14s - loss: 2.2783 -
acc: 0.1463 - val_loss: 2.2424 - val_acc: 0.3595
Test score: 2.24242398682
Test accuracy: 0.3595
```

In [46]:
```
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
fig.suptitle('Model with sigmoid activation')

for ax, k, result in zip(axes, ['depth = 1','depth = 2','depth = 3'
], range(0,3)):
    ax.plot(acc_results[result])
    ax.plot(valacc_results[result])
    ax.set_title('model accuracy: '+k)
    ax.set_ylabel('accuracy')
    ax.set_xlabel('epoch')
    ax.legend(['train', 'test'], loc='upper left')
    ax.set_ylim([0,1])
```



In [41]:
```
acc_relu = []
valacc_relu = []

# now try with different activation functions
for layer in range(1,4):
    number_hidden_layers = layer
    # Model
    model = Sequential()
    model.add(Dense(512, input_shape=(784,), activation='relu'))
    model.add(Dropout(0.2))

    while number_hidden_layers > 1:
        model.add(Dense(512))
        model.add(Activation('relu'))
        model.add(Dropout(0.2))
        number_hidden_layers -= 1


    model.add(Dense(10))
    model.add(Activation('softmax'))

    model.summary()

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

    # Training (You don't need to change this part of the code)
    history = model.fit(X_train, Y_train,
                        batch_size=batch_size, nb_epoch=nb_epoch,
                        verbose=1, validation_data=(X_test, Y_test)
)
    score = model.evaluate(X_test, Y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    # list all data in history
    print(history.history.keys())
    # summarize history for accuracy
    #plt.plot(history.history['acc'])
    acc_relu.append(history.history['acc'])

    #plt.plot(history.history['val_acc'])
    valacc_relu.append(history.history['val_acc'])

    #plt.title('model accuracy')
    #plt.ylabel('accuracy')
    #plt.xlabel('epoch')
    #plt.legend(['train', 'test'], loc='upper left')
    #plt.show()
```

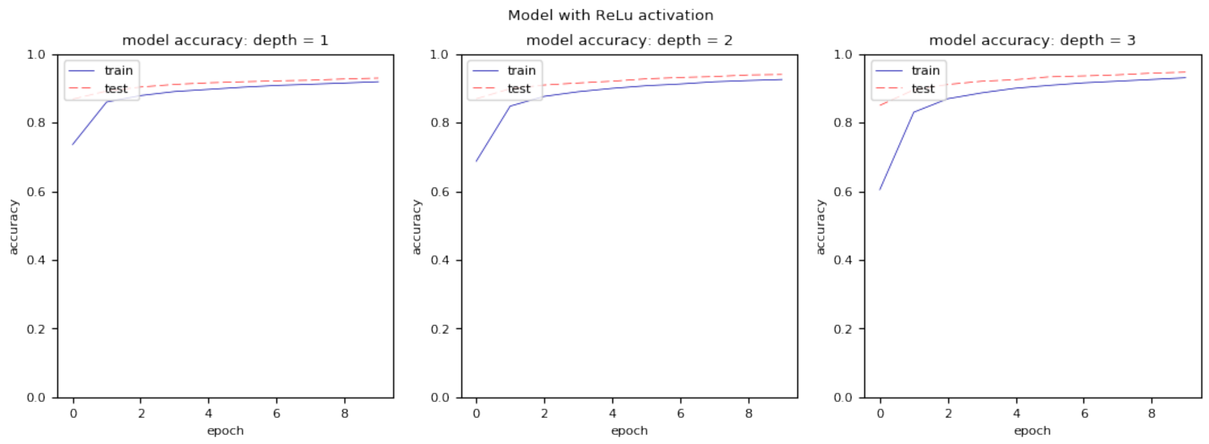
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_21 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_14 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_22 (Dense)             (None, 10)                5130
```

```
_________________________________________________________________
activation_14 (Activation)   (None, 10)                0
=================================================================
Total params: 407,050.0
Trainable params: 407,050.0
Non-trainable params: 0.0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 6s - loss: 1.1182 -
acc: 0.7361 - val_loss: 0.5950 - val_acc: 0.8684
Epoch 2/10
60000/60000 [==============================] - 6s - loss: 0.5460 -
acc: 0.8600 - val_loss: 0.4301 - val_acc: 0.8919
Epoch 3/10
60000/60000 [==============================] - 6s - loss: 0.4445 -
acc: 0.8788 - val_loss: 0.3693 - val_acc: 0.9037
Epoch 4/10
60000/60000 [==============================] - 6s - loss: 0.3969 -
acc: 0.8905 - val_loss: 0.3360 - val_acc: 0.9113
Epoch 5/10
60000/60000 [==============================] - 5s - loss: 0.3665 -
acc: 0.8968 - val_loss: 0.3145 - val_acc: 0.9159
Epoch 6/10
60000/60000 [==============================] - 6s - loss: 0.3451 -
acc: 0.9032 - val_loss: 0.2970 - val_acc: 0.9189
Epoch 7/10
60000/60000 [==============================] - 6s - loss: 0.3263 -
acc: 0.9083 - val_loss: 0.2839 - val_acc: 0.9217
Epoch 8/10
60000/60000 [==============================] - 7s - loss: 0.3125 -
acc: 0.9117 - val_loss: 0.2729 - val_acc: 0.9236
Epoch 9/10
60000/60000 [==============================] - 7s - loss: 0.2989 -
acc: 0.9153 - val_loss: 0.2625 - val_acc: 0.9272
Epoch 10/10
60000/60000 [==============================] - 6s - loss: 0.2886 -
acc: 0.9191 - val_loss: 0.2537 - val_acc: 0.9297
Test score: 0.253701646206
Test accuracy: 0.9297
dict_keys(['loss', 'acc', 'val_acc', 'val_loss'])
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_15 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_24 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_15 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_16 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_25 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_16 (Activation)   (None, 10)                0
=================================================================
Total params: 669,706.0
Trainable params: 669,706.0
```

```
Non-trainable params: 0.0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 10s - loss: 1.2180 -
acc: 0.6873 - val_loss: 0.5434 - val_acc: 0.8694
Epoch 2/10
60000/60000 [==============================] - 9s - loss: 0.5353 -
acc: 0.8480 - val_loss: 0.3778 - val_acc: 0.8987
Epoch 3/10
60000/60000 [==============================] - 9s - loss: 0.4270 -
acc: 0.8764 - val_loss: 0.3259 - val_acc: 0.9095
Epoch 4/10
60000/60000 [==============================] - 10s - loss: 0.3778 -
acc: 0.8899 - val_loss: 0.2955 - val_acc: 0.9154
Epoch 5/10
60000/60000 [==============================] - 9s - loss: 0.3436 -
acc: 0.8998 - val_loss: 0.2754 - val_acc: 0.9206
Epoch 6/10
60000/60000 [==============================] - 9s - loss: 0.3196 -
acc: 0.9075 - val_loss: 0.2563 - val_acc: 0.9276
Epoch 7/10
60000/60000 [==============================] - 9s - loss: 0.3010 -
acc: 0.9125 - val_loss: 0.2408 - val_acc: 0.9315
Epoch 8/10
60000/60000 [==============================] - 9s - loss: 0.2819 -
acc: 0.9189 - val_loss: 0.2286 - val_acc: 0.9342
Epoch 9/10
60000/60000 [==============================] - 9s - loss: 0.2670 -
acc: 0.9226 - val_loss: 0.2166 - val_acc: 0.9388
Epoch 10/10
60000/60000 [==============================] - 9s - loss: 0.2546 -
acc: 0.9257 - val_loss: 0.2085 - val_acc: 0.9402
Test score: 0.208472842467
Test accuracy: 0.9402
dict_keys(['loss', 'acc', 'val_acc', 'val_loss'])
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_26 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_17 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_27 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_17 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_18 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_28 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_18 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_19 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_29 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_19 (Activation)   (None, 10)                0
=================================================================
Total params: 932,362.0
```

```
Trainable params: 932,362.0
Non-trainable params: 0.0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 13s - loss: 1.4003 -
acc: 0.6044 - val_loss: 0.5623 - val_acc: 0.8501
Epoch 2/10
60000/60000 [==============================] - 13s - loss: 0.5711 -
acc: 0.8303 - val_loss: 0.3675 - val_acc: 0.8955
Epoch 3/10
60000/60000 [==============================] - 13s - loss: 0.4371 -
acc: 0.8695 - val_loss: 0.3067 - val_acc: 0.9110
Epoch 4/10
60000/60000 [==============================] - 14s - loss: 0.3787 -
acc: 0.8867 - val_loss: 0.2763 - val_acc: 0.9205
Epoch 5/10
60000/60000 [==============================] - 13s - loss: 0.3397 -
acc: 0.9003 - val_loss: 0.2531 - val_acc: 0.9252
Epoch 6/10
60000/60000 [==============================] - 13s - loss: 0.3094 -
acc: 0.9086 - val_loss: 0.2327 - val_acc: 0.9335
Epoch 7/10
60000/60000 [==============================] - 14s - loss: 0.2882 -
acc: 0.9158 - val_loss: 0.2175 - val_acc: 0.9360
Epoch 8/10
60000/60000 [==============================] - 15s - loss: 0.2683 -
acc: 0.9207 - val_loss: 0.2040 - val_acc: 0.9392
Epoch 9/10
60000/60000 [==============================] - 14s - loss: 0.2508 -
acc: 0.9257 - val_loss: 0.1911 - val_acc: 0.9441
Epoch 10/10
60000/60000 [==============================] - 13s - loss: 0.2367 -
acc: 0.9307 - val_loss: 0.1797 - val_acc: 0.9471
Test score: 0.179688579264
Test accuracy: 0.9471
dict_keys(['loss', 'acc', 'val_acc', 'val_loss'])
```

```
In [44]: fig, axes = plt.subplots(1, 3, figsize=(13, 4))
         fig.suptitle('Model with ReLu activation')

         for ax, k, result in zip(axes, ['depth = 1','depth = 2','depth = 3'
         ], range(0,3)):
             ax.plot(acc_relu[result])
             ax.plot(valacc_relu[result])
             ax.set_title('model accuracy: '+k)
             ax.set_ylabel('accuracy')
             ax.set_xlabel('epoch')
             ax.legend(['train', 'test'], loc='upper left')
             ax.set_ylim([0,1])
```



Submit an explanation for the limitation of the original model. Explain your modification.

The problem the original model faces when depth increases is known as the **vanishing gradient problem**. Basically, the speed of learning drops heavily in the earlier layers resulting in a model that cannot learn well from the inputs: resulting in a worse performance. Note that the deeper the model, the worse the performance (the vanishing gradient problem gets worse). The gradients get so small that even a large difference in inputs does not result in a (significant) change in outputs.

This problem arises from the choice of activation function. In the original model, the 'Sigmoid' function is used, and to understand what goes wrong we need to look at the gradients and how they are computed/updated. When computing the gradients in the first layers, by backpropagating (and applying the chain rule) we multiply by terms that are weigths, biases and derivatives of the activation functions (here sigmoid!). Since this derivative of the sigmoid function is at most 0.25, multiplying very often with this term results in an exponential decrease. Would this derivative be (much) bigger than one, we could encounter another problem called the 'exploding gradient', where exactly the opposite happens.

The applied modification is simply changing the activation function Sigmoid to ReLU. This activation function addresses the vanishing gradient function. Actually, the derivative of this function is always 1 when the output of the neuron is > 0 (otherwise it is zero), so both the exploding and vanishing gradient problems are addressed by this activation function. Interesting note here is that, since the derivative can get zero, certain neurons can 'die' out (which does not necessarily means bad news).

Convolutional Neural Networks for Filtering (2 points)

Convolutional neural networks are well suited for analyzing images. They can be used to apply various image filtering operations.

The goal of this exercise is to design a CNN model that applies 2 filters to its input images. The input images are 128x128 RGB color images, encoded as 128x128x3 tensor with floating point value normalized between 0 and 1. The RGB format is such that the pixels address by: [:, :, 0] encode the red pixels of the image, the pixels addressed by [:, :, 1] define the green pixels and pixels addressed by [:, :, 2] define the blue pixels.

Design a convolutional neural network that will:

1. Apply the sepia filter to the image
2. Apply Gaussian smoothing to the image

Use the specification of the sepia and the Gaussian filter below.

You answer should contain:

```
- The definition of the architecture of the CNN
    - Number of layers
    - Number of filters per layer
    - Shape of the filter per layer
- Values of each of the parameters of the CNN when using a 5x5 Gaussian
smoothing filter
- The dimensions of the output image when a 5x5 Gaussian smothing is ap
plied
```

The sepia effect gives warmth and a feel of vintage to photographs. The sepia filter is defined as:

$$R_o = (R_i * .393) + (G_i * .769) + (B_i * .189) G_o = (R_i * .349) + (G_i * .686) + (B_i * .168) B_o = (R_i * .272) + (G_i * .534) + (B_i * .131)$$

Gaussian blurring is an effect that reduces the noise and details in an image. Gaussian smoothing filter:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

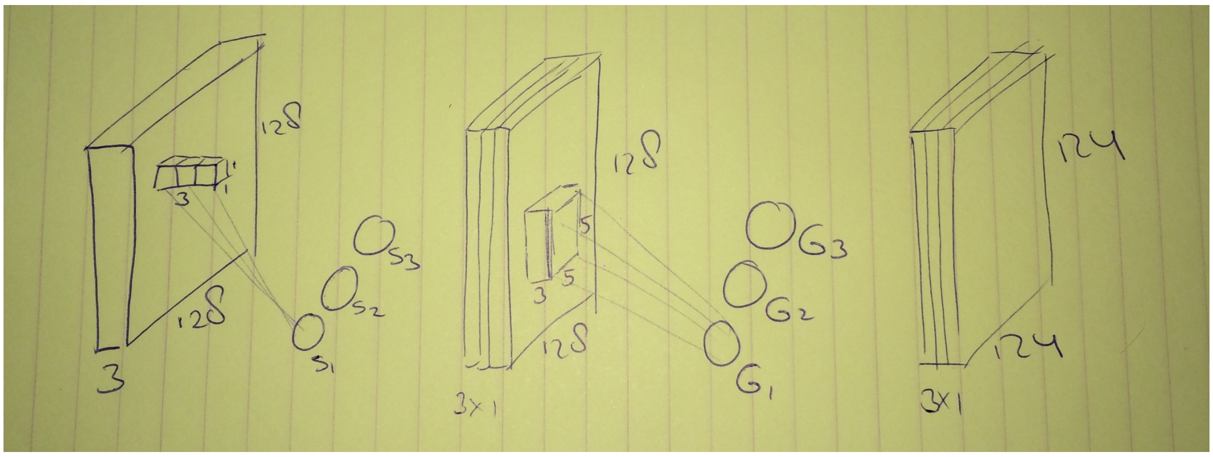
- A discretized version of the filter is given by the following table:

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

- To normalize the filter response, each value should divided by 273. This is a truncated discretized Gaussian filter with a σ of 1.

In [106]:
```python
from IPython.display import Image
Image(filename='IMG_7518.jpg')
```

Out[106]:



The CNN we've designed for this task is sketched in the image above. Basically our network consists of two layers, whereas the first layer applies the Sepia effect and the second applies the Gaussian Smoothing Filter. Below we will describe the network in more detail.

- Definition of the architecture of the CNN:
    - Number of layers: 2 layers.
    - Number filters per layer: 1 per layer
    - Shape of the filter per layer:
        - Layer 1 (SEPIA): 1x1x3
        - Layer 2 (GAUSSIAN): 5x5x3

- Values of each of the parameters of the CNN.
    - For the first layer, each of the 3 neurons will have specific (fixed) weights attached to the filter of 1x1x3, in order to apply the Sepia filter. Note that the order of colors in the tensor we assume to be [R G B], and that the weights correspond to the values in the formulas as described above.
        - neuron 1 weights of the filter: [.393 .769 0.189]
        - neuron 2 weights of the filter: [.349 .686 0.168]
        - neuron 3 weights of the filter: [.272 .534 0.131]

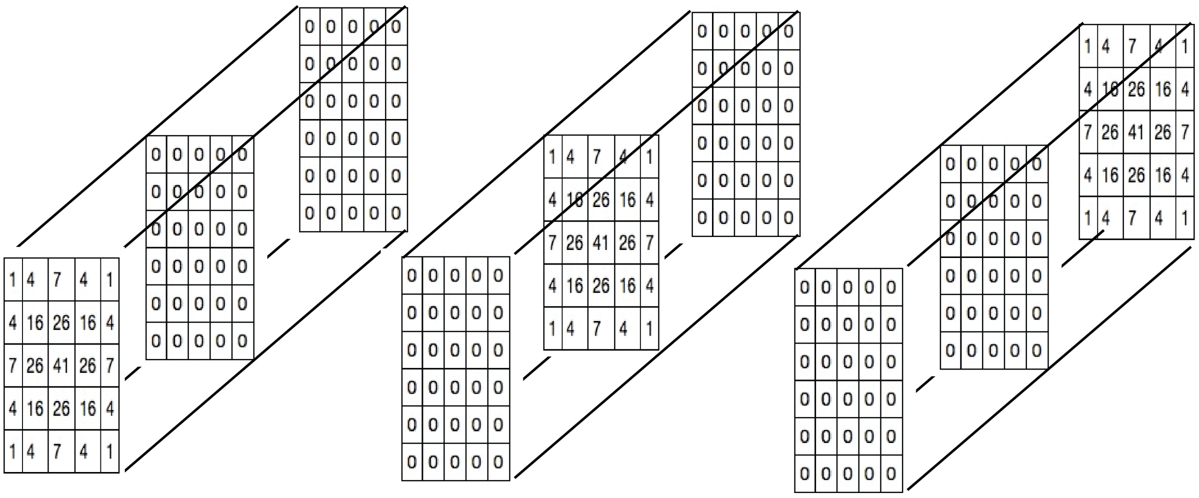
      The output of this layer with the 1x1x3 filter will again be a tensor of 128x128x3, just like the original input.
    - For the second layer, we again have 3 neurons, but now use a filter of size 5x5x3. The weights are fixed according to the following procedure. Again we assume the colors in the tensor are ordered in the [R G B] order.
        - Neuron 1 will gaussian-blur the RED layer of the image tensor. In total there are 3 planes of 5x5 elements of the filter behind eachother which all get a weight. The first plane of 5x5 weights are set with the discretized filter values / 273 (to directly normalize the filter response). The two subsequent planes of 5x5 will get weights of 0.
        - Neuron 2 will gaussian-blur the GREEN layer of the image tensor. Now the 1st and 3rd layer will get weights of 0, whereas the middle layer will get weights according to the given discretized filter values (each again divided by 273 for normalization).
        - Neuron 3 will gaussian-blur the BLUE layer of the image tensor. This time the weights of the first and set layer of the filter will be set to zero, and the 3rd will get weights according to the given discretized filter values (also divided by 273 for normalization).

          This procedure is visualized in the image below. NOTE: in the visualization the raw discretized filter values are shown, wheras in our model we divide each of these values by 273 for direct normalization of the filter response.
- The output this model provides will be a 124x124x3 output image which is in sepia and has been Gaussian Blurred.

In [110]:
```
print("Image: weights for the Gaussian filter. Left neuron 1, middl
e neuron 2 right neuron 3. Note that the filter values all will be
normalized by dividing them by 273")
from IPython.display import Image
Image(filename='weights.png')
```

```
Image: weights for the Gaussian filter. Left neuron 1, middle neuro
n 2 right neuron 3. Note that the filter values all will be normali
zed by dividing them by 273
```

Out[110]:



Model Design (2 Points)

Various decisions need to be made in a modeling process to address specific properties of the data and the modeling goal. In this task, you are given a description of a data structure and a goal for which you need to design a model.

Produce a figure depicting your model. Briefly explain the figure and justify all decisions made in the modeling process. In detail, describe at least:

- Input data format
- Number of layers
- Type of layers (Dense, Recurrent, Convolutional - 1D, 2D, 3D)
- Regularization
- Model output
- Loss function

The training and execution procedures for the model may differ, so you can use different descriptions for both.

Data and goal description:

The goal of this task is to generate captions for short video clips.

The video data is structured as sequences of color images. The model needs to be able to process a number of consecutive images that form a short video clip. The training data consists of video clips (few seconds) and a short caption (5-10 words).

For simplicity, the accuracy of the model is evaluated on the exact prediction of the caption. In other words, the model needs to produce correctly the specific words in a specific order for each video.

Model Design Answer:

Model Design Answer:

To model this, we use that fact that we want to update the sequence word by word. Thus, the 'time' component is in creating the sequence. Hence, for this part we will use a RNN. To be able to create a sequence, we first need to transform the video. We want to create one data structure that summarizes the entire video (sequence of images). For this we have a CNN layer. These two layers will be trained seperately. The CNN will be trained on an image database. This ensures that we extract the correct features from each image to get a correct summary of the video. The RNN will be trained using a dictionary of words. Note that this dictionary needs to contain an indicator that the sequence has ended. Thus, there is a tag in the dictionary of words with END.

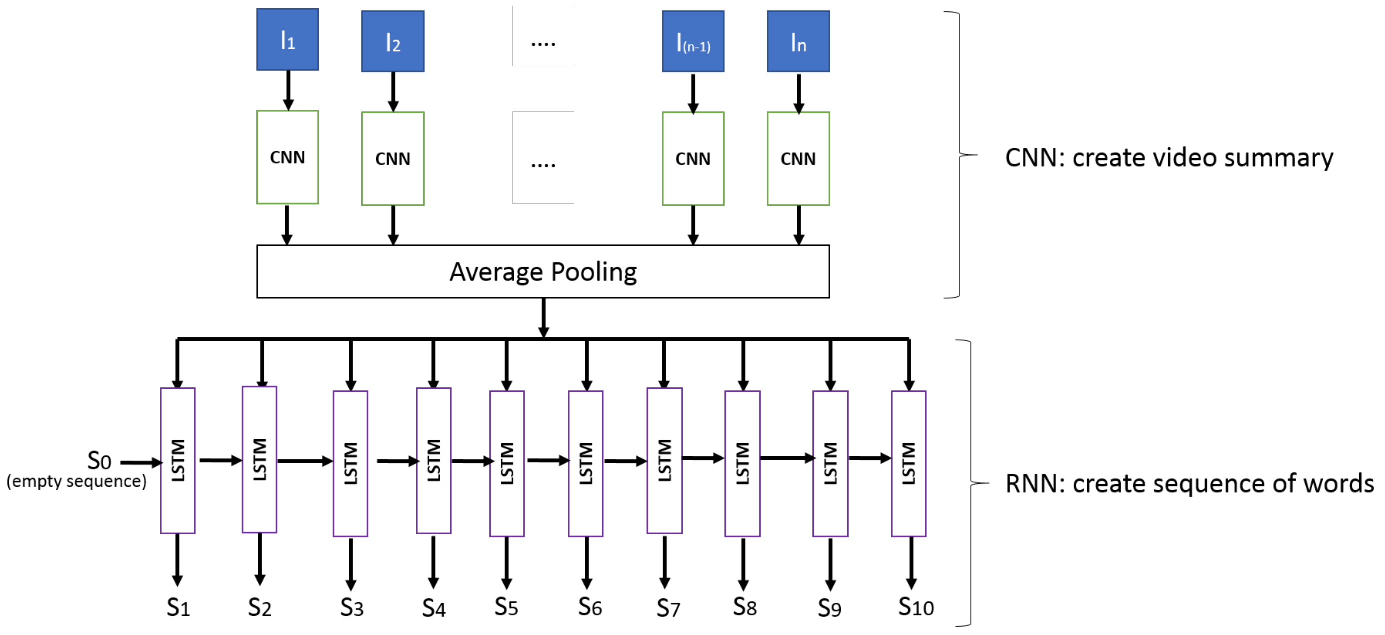
- Input data format: The data format will be a tensor of images. Each image is 3D, thus each sequence of images for each video will be 4D tensor.
- Number of layers: The model consists of two layers. A CNN and a RNN layer.
- Type of layers:
  - CNN layer. The setup of the CNN is as we have seen in the lecture, type D. However, we end with with a single vector after the two FC 4096. Hence, we do not have the FC 1000 and softmax at the end. Thus, for each image (video frame!) we run a CNN, each image (frame) becomes a FC 4096. Then we run a mean pooling over it to get one vector representing the total video. We choose to use average pooling afterwards, because we want a summary of the video. For example, the video shows a boy playing football. However, the video shows only the ball and the boy on some images but there are also images with crowd, goal and keeper clearly visible. Therefore, we want to take the average such that every part of the video is captured by the FC 4096. Thus, the vector is a summary of the entire video. This single vector is then the input for the next layer.
  - RNN layer. This layer consists of the a recurrent neural network using the LSTM setup for each cell. The main idea here is to generate a word at each time stamp. This word comes from a dictionary(Dict) for which the model is already pre-trained. The input at each cell is the output from the previous cell and the summary vector of the images. The output is a word that is choses with the highest probability as seen in equation (1). Note that when END is part of the sequence, we want to ensure that the probability that END is chosen next equals one. This was, we can not add new words to the sequence when we have reached the end of a sequence. Moreover, by training correctly, we can ensure that this probability for END being chosen equals zero when the number of previous words is less than 5.
  - These layers can be seen in the figure 2

$$\max_i \Pr[\text{word}_{i \in D} | \text{previous words chosen}] \quad (1)$$

- Regularization: The implementation of the model would make use of a dropout layer in the CNN model. This ensures that the network becomes less sensitive to the specific weights of neurons.
- Model output: The model outputs a sequence of words. This sequence is smaller than or equal to 10. Output = $S = S_1, .., S_{10}$
- Loss function: The loss function is the 0/1-loss function as we want the exact caption. Thus, if the caption S that is ouputted is only slighly different from the actual sequence $\hat{S}$, the loss will still equal one as it is not completely correct:

$$L = \begin{cases} 1 & \text{if } \hat{S} \neq S \\ 0 & \text{otherwise} \end{cases}$$

Figure 2:



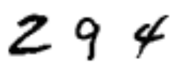
MNIST Calculator (5 points)

During the lectures you have seen a CNN model that can be successfully trained to classify the MNIST images. You have also seen how a RNN model that can be trained to implement addition of two numbers.

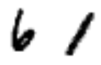
Using the KERAS (or TensorFlow) library, design and train a model that can learn how to add numbers directly from the MNIST image data. More specifically, the model should input a sequence of a set of images and produces a cumulative sum of the numbers represented by the digits in the images.

For example:

Input 1:



Input 2:



Output: 355

Setup We have chosen to first build and train a CNN, then build and train an RNN and finally combine these two in a combined model, that is able classify and sum directly from the images, using the two pretrained models. By doing the training separately, we are able speed up the training process to a very large extend. Moreover, it is more clear for each part of the network what it should learn.

Data preparation In order for the model to learn, we need a lot of data preprocessing.

Inputs For inputs we have chosen to limit ourselves to 3 digit numbers. Thus, our model restricts itself to sum up two numbers of maximum 3 digits. Note that the 0 can be picked as first (or first and second) number, thus smaller digits are also possible. The training data is created by randomly selecting 6 images from the mnist database, and putting them together. The first three represent the first number to add, the second three the second number.

CNN Our CNN takes as trainingsdata-input this tensor of 60000 samples of 6x28x28, and outputs a one-hot encoding of these 6 digits. Because we can train this part separately from the adder, the learning can be more focussed on the digit recognition, rather than doing all tasks together.

RNN We train the RNN using a one hot encoding of the true labels in the training set as input, and the true summation of the first and second 3-digit number. Again, by doing this separately, this submodel can be trained more specific for this task in less time.

Total Model Our total model simply consists of the two models behind eachother. Because we pre-trained both submodels, further tuning is not necessarily needed to achieve a good performance.

In [7]:
```
## Imports
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Flatten, Reshape, Input, T
imeDistributed
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import numpy as np

class CharacterTable(object):
    '''
    Given a set of characters:
    + Encode them to a one hot integer representation
    + Decode the one hot integer representation to their character
output
    + Decode a vector of probabilities to their character output
    '''
    def __init__(self, chars, maxlen):
        self.chars = sorted(set(chars))
        self.char_indices = dict((c, i) for i, c in enumerate(self.
chars))
        self.indices_char = dict((i, c) for i, c in enumerate(self.
chars))
        self.maxlen = maxlen

    def encode(self, C, maxlen=None):
        maxlen = maxlen if maxlen else self.maxlen
        X = np.zeros((maxlen, len(self.chars)))
        for i, c in enumerate(C):
            X[i, self.char_indices[c]] = 1
        return X

    def decode(self, X, calc_argmax=True):
        if calc_argmax:
            X = X.argmax(axis=-1)
        return ''.join(self.indices_char[x] for x in X)

# Training parameters
batch_size = 128
num_classes = 10
epochs = 1



#
#
# Data preparation
#
#

# input image dimensions
img_rows, img_cols = 28, 28

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

Index_train = []
Index_test = []
for i in range(0,6):
    temp = np.random.choice(len(x_train), len(x_train), replace = F
alse)
    temp2 = np.random.choice(len(x_test), len(x_test), replace = Fa
lse)
    Index_train.append(temp)
    Index_test.append(temp2)

# create arrays of 2x3 = 6 images form the MNIST dataset.
xtrain =  np.array([x_train[Index_train[0]], x_train[Index_train[1]
], x_train[Index_train[2]], x_train[Index_train[3]], x_train[Index_
train[4]], x_train[Index_train[5]]])
ytrain =  np.array([y_train[Index_train[0]], y_train[Index_train[1]
], y_train[Index_train[2]], y_train[Index_train[3]], y_train[Index_
train[4]], y_train[Index_train[5]]])

xtest =  np.array([x_test[Index_test[0]], x_test[Index_test[1]], x_
test[Index_test[2]], x_test[Index_test[3]], x_test[Index_test[4]],
x_test[Index_test[5]]])
ytest =  np.array([y_test[Index_test[0]], y_test[Index_test[1]], y_
test[Index_test[2]], y_test[Index_test[3]], y_test[Index_test[4]],
y_test[Index_test[5]]])

# convert class vectors to binary class matrices
y_train_cnn = np.zeros((6,60000,10))
y_test_cnn = np.zeros((6,10000,10))
for i in range(0,6):
    y_train_cnn[i] = keras.utils.to_categorical(ytrain[i], num_clas
ses)
    y_test_cnn[i] = keras.utils.to_categorical(ytest[i], num_classe
s)

y_train = np.swapaxes(y_train_cnn,0,1)
```

```
y_test = np.swapaxes(y_test_cnn,0,1)

x_train = np.swapaxes(xtrain,0,1)
x_test = np.swapaxes(xtest,0,1)

# prepare for cnn
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0],x_train.shape[1], 1,
img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], x_train.shape[1], 1, i
mg_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], i
mg_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], x_train.shape[1], img_
rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Creation of final_y for RNN
y_train_final = np.zeros((60000))
y_test_final = np.zeros((10000))

for i in range(0,60000):
    y_train_final[i] = int(ytrain[0,i]*100+ytrain[3,i]*100+ytrain[1
,i]*10+ytrain[4,i]*10+ytrain[2,i]+ytrain[5,i])

for i in range(0,10000):
    y_test_final[i] = int(ytest[0,i]*100+ytest[3,i]*100+ytest[1,i]*
10+ytest[4,i]*10+ytest[2,i]+ytest[5,i])

chars = '0123456789'
ctable = CharacterTable(chars, 4)

y_train_final2 = np.zeros((60000, 4, 10))
#create the y's
for i in range(0, 60000):
    if y_train_final[i]/1000<1:
        temp = '0{}'.format(int(y_train_final[i]))
        #print(temp)
    else:
        temp = '{}'.format(int(y_train_final[i]))
        #print(temp)
    y_train_final2[i]=ctable.encode(temp)

y_test_final2 = np.zeros((10000, 4, 10))
for i in range(0, 10000):
    if y_test_final[i]/1000<1:
        temp = '0{}'.format(int(y_test_final[i]))
        #print(temp)
    else:
        temp = '{}'.format(int(y_test_final[i]))
        #print(temp)
    y_test_final2[i]=ctable.encode(temp)

print("done preprocessing")
```

```
x_train shape: (60000, 6, 28, 28, 1)
60000 train samples
10000 test samples
done preprocessing
```

In [9]:
```
K.set_learning_phase(1)
# CNN model definition

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

sequences_input = Input(shape=(6,28,28,1))
process_seq = TimeDistributed(model)(sequences_input)
model = Model(input=[sequences_input], output=process_seq)

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

# Training loop
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
/Users/evertjanpeer/anaconda/lib/python3.5/site-packages/ipykernel/
__main__.py:19: UserWarning: Update your `Model` call to the Keras
2 API: `Model(inputs=[<tf.Tenso..., outputs=Tensor("ti...)`
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 543s - loss: 0.3178
- acc: 0.9020 - val_loss: 0.1068 - val_acc: 0.9678
Test loss: 0.105810460305
Test accuracy: 0.966733333302
```

In [13]:
```
# RNN model definition
from keras.layers import RepeatVector, recurrent, Activation, TimeD
istributed, Dropout
RNN = recurrent.SimpleRNN
HIDDEN_SIZE=128
LAYERS = 2
print('Build model...')
rnn = Sequential()
rnn.add(RNN(HIDDEN_SIZE, input_shape=(6,10)))
rnn.add(RepeatVector(3 + 1))
for _ in range(LAYERS):
    rnn.add(RNN(HIDDEN_SIZE, return_sequences=True))

rnn.add(Dropout(.2))

rnn.add(TimeDistributed(Dense(len(chars))))
rnn.add(Activation('softmax'))

rnn.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# Training Loop for RNN
rnn.fit(y_train,y_train_final2,
          batch_size=batch_size,
          epochs=epochs*3,
          verbose=1,
          validation_data=(y_test, y_test_final2))

score = rnn.evaluate(y_test, y_test_final2, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

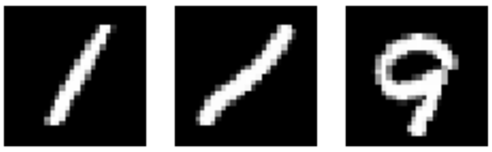
```
Build model...
Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 21s - loss: 1.3750 -
acc: 0.4820 - val_loss: 0.5472 - val_acc: 0.8231
Epoch 2/3
60000/60000 [==============================] - 22s - loss: 0.3048 -
acc: 0.9123 - val_loss: 0.1821 - val_acc: 0.9521
Epoch 3/3
60000/60000 [==============================] - 21s - loss: 0.1351 -
acc: 0.9635 - val_loss: 0.1544 - val_acc: 0.9527
Test loss: 0.157774398899
Test accuracy: 0.951
```

```
In [105]: # Build final model
          print('Build final model...')
          Final_model = Sequential()
          Final_model.add(model)
          Final_model.add(rnn)
          Final_model.compile(loss='categorical_crossentropy',
                        optimizer='adam',
                        metrics=['accuracy'])
          print('Evaluating performance total model...')
          #model is already trained, see how it performs
          score = Final_model.evaluate(x_test, y_test_final2)
          print('Test loss:', score[0])
          print('Test accuracy:', score[1])
```

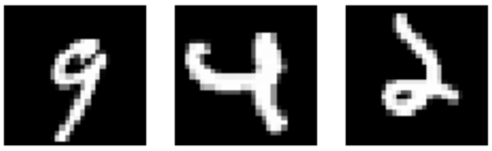
```
Build final model...
Evaluating performance total model...
10000/10000 [==============================] - 32s
Test loss: 1.25394978209
Test accuracy: 0.826225
```

```
In [104]: # EG: let's see the first test example

          # show some examples:

          for examples in range(0,3):
              print("input one:")
              for i in range(3):
                  plt.subplot(1,6,i+1)
                  plt.imshow(x_test[examples][i].reshape((28,28)), cmap='Grey
          s_r')
                  plt.axis('off')
              plt.show()

              print("input two:")
              for i in range(3,6):
                  plt.subplot(1,6,i+1)
                  plt.imshow(x_test[examples][i].reshape((28,28)), cmap='Grey
          s_r')
                  plt.axis('off')
              plt.show()

              print("compute sum...")
              print("computed sum = {}".format(Final_model.predict_classes(np
          .array([x_test[examples]]))))
              print("true sum = {}\n".format(int(y_test_final[examples])))
```

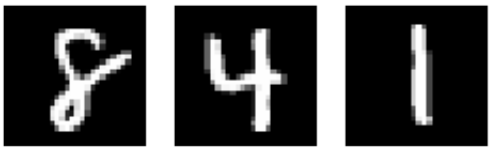
```
input one:
```



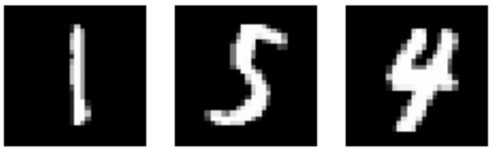
```
input two:
```



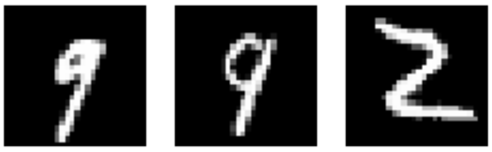
```
compute sum...
1/1 [==============================] - 0s
computed sum = [[1 0 6 1]]
true sum = 1061

input one:
```



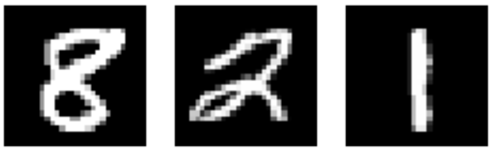
```
input two:
```



```
compute sum...
1/1 [==============================] - 0s
computed sum = [[0 9 9 5]]
true sum = 995

input one:
```



```
input two:
```



```
compute sum...
1/1 [==============================] - 0s
computed sum = [[1 9 1 3]]
true sum = 1813
```

Interpration of the results and room for improvement Our model already, without a lot of training time, performs quite well. There are however some points that could be taken into account to further tune the model.

- The number of Epochs can be increased in order to give the model more training time.
- Multiple iterations can be performed to also tune the model.

In addition, the model is now limited to the summation of two 3-digit numbers, which could be generalized more. However, a lot of padding of the input spaces would be needed for smaller numbers.