# Foundations of Data Mining: Assignment 4

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

**Deadline:** Thursday, April 12, 2018

In [1]:

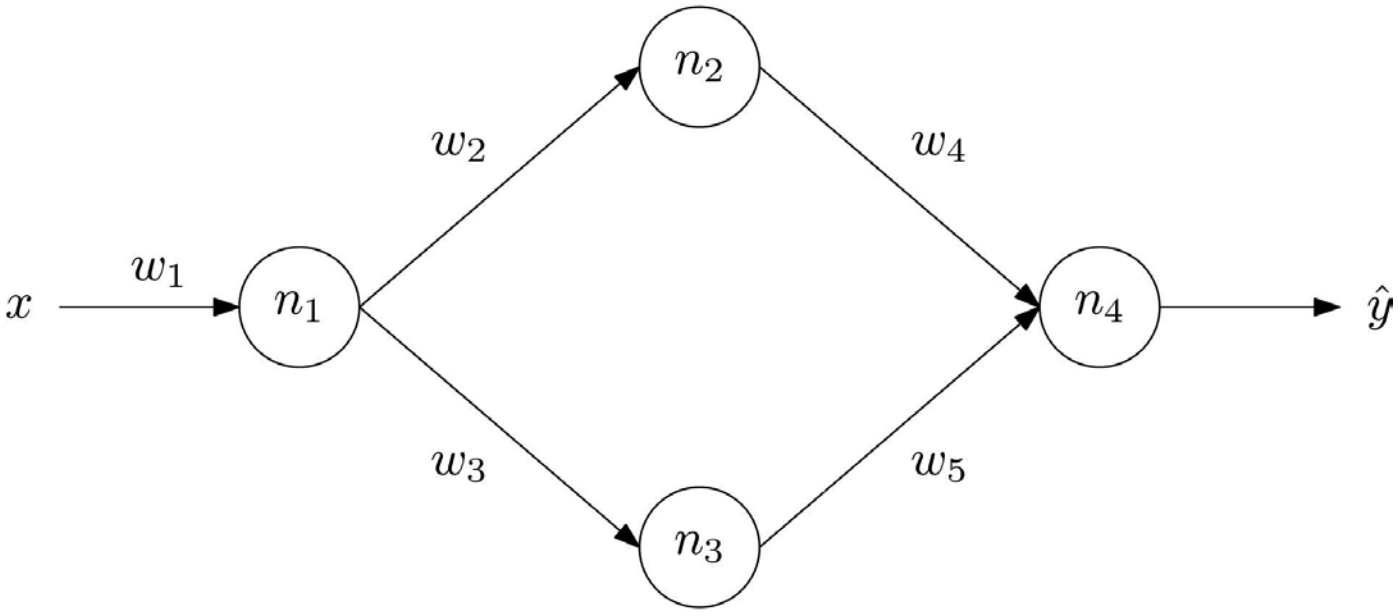
```
# Please fill in your names here
NAME_STUDENT_1 = "Emma van Zoelen"
NAME_STUDENT_2 = "Jules Klomp"
```

In [2]:

```
%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 (6 points)

Figure 1 illustrates a simple neural network model.




It has single input $x$, and three layers with respectively one, two, and one neurons. The activation function of the neurons is ReLU.

The parameters $w_1$, $w_2$, $w_3$, $w_4$, and $w_5$ (no biases) are initialized to the following values $w_1 = 2, w_2 = 1$, $w_3 = 2$, $w_4 = 4$, and $w_5 = 1$. Implement a single update step of the gradient descent algorithm by hand. Run the update state for the data point $(x = 2, y = 3)$:

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 all four parameters after the single step in the gradient descent algorithm

The Python code for simple computational graph nodes, as seen in the tutorial session, is provided in the cell below (run the cell to load the code, and again to run the code). Extend the nodes so they can be used to implement the network described above. Implement the network with the same initial weights and the correct learning rate, and verify your hand-made calculations. Add comments to your code or provide a separate description to explain the changes you have made.


Figure 2

MSE is chosen for the loss function: $L(\mathbf{x}, y; \mathbf{W}) = \frac{1}{2n} \sum_{i=0}^{n} (z - y)^2$

Partial derivative expression for each of the parameters in the model

- $\frac{\partial L}{\partial L} = 1$
- $\frac{\partial L}{\partial z} = \frac{2(z-y)}{2} * 1 = (z - y)$
- $\frac{\partial L}{\partial \gamma_1} = \frac{\partial z}{\partial \gamma_1} \frac{\partial L}{\partial z} = 1 * (z - y)$
- $\frac{\partial L}{\partial \gamma_2} = \frac{\partial z}{\partial \gamma_2} \frac{\partial L}{\partial z} = 1 * (z - y)$
- $\frac{\partial L}{\partial w_4} = \frac{\partial \gamma_1}{\partial w_4} \frac{\partial L}{\partial \gamma_1} = \beta_1 * (z - y) = (z - y) * x * w_1 * w_2$
- $\frac{\partial L}{\partial w_5} = \frac{\partial \gamma_2}{\partial w_5} \frac{\partial L}{\partial \gamma_2} = \beta_2 * (z - y) = (z - y) * x * w_1 * w_3$
- $\frac{\partial L}{\partial \beta_1} = \frac{\partial \gamma_1}{\partial \beta_1} \frac{\partial L}{\partial \gamma_1} = w_4 * (z - y)$
- $\frac{\partial L}{\partial \beta_2} = \frac{\partial \gamma_2}{\partial \beta_2} \frac{\partial L}{\partial \gamma_1} = w_5 * (z - y)$
- $\frac{\partial L}{\partial w_2} = \frac{\partial \beta_1}{\partial w_2} \frac{\partial L}{\partial \beta_1} = x * w_1 * w_4 * (z - y)$
- $\frac{\partial L}{\partial w_3} = \frac{\partial \beta_2}{\partial w_3} \frac{\partial L}{\partial \beta_2} = x * w_1 * w_5 * (z - y)$
- $\frac{\partial L}{\partial \alpha} = \frac{\partial \beta_1}{\partial \alpha} \frac{\partial L}{\partial \beta_1} + \frac{\partial \beta_2}{\partial \alpha} \frac{\partial L}{\partial \beta_2} = (w_2 * w_4 + w_3 * w_5) * (z - y)$
- $\frac{\partial L}{\partial w_1} = \frac{\partial \alpha}{\partial w_1} \frac{\partial L}{\partial \alpha} = x * w_1 * (z - y)$
- $\frac{\partial L}{\partial w_0} = \frac{\partial q}{\partial w_0} \frac{\partial L}{\partial q} = x * (w_2 * w_4 + w_3 * w_5) * (z - y)$

It is given that $\lambda = 0.1$ (the learning rate).

$$w_1 \leftarrow w_1 - \lambda \frac{\partial}{\partial w_1} L(\mathbf{x}, y; \mathbf{w_1}, b)$$
$$w_2 \leftarrow w_1 - \lambda \frac{\partial}{\partial w_2} L(\mathbf{x}, y; \mathbf{w_2}, b)$$
$$w_3 \leftarrow w_0 - \lambda \frac{\partial}{\partial w_3} L(\mathbf{x}, y; \mathbf{w_3}, b)$$
$$w_4 \leftarrow w_0 - \lambda \frac{\partial}{\partial w_4} L(\mathbf{x}, y; \mathbf{w_4}, b)$$
$$w_5 \leftarrow w_0 - \lambda \frac{\partial}{\partial w_5} L(\mathbf{x}, y; \mathbf{w_5}, b)$$

- The new $w_1$ will be:

$$w_1 = 2 - 0.1 * (x * (w_2 * w_4 + w_3 * w_5) * (z - y))$$

This will result into

$$w_1 = 2 - 0.1 * (2((1 * 4 + 2 * 1) * (21 - 3))) = -19.6$$

- The new $w_2$ will be:

  $$w_2 = 1 - 0.1 * (x * w_1 * w_4 * (z - y))$$

  This will result into

  $$w_2 = 1 - 0.1 * (2 * 2 * 4 * (21 - 3))) = -27.8$$

- The new $w_3$ will be:

  $$w_3 = 2 - 0.1 * (x * w_1 * w_5 * (z - y))$$

  This will result into

  $$w_3 = 2 - 0.1 * (2 * 2 * 1 * (21 - 3))) = -5.2$$

- The new $w_4$ will be:

  $$w_4 = 4 - 0.1 * (x * w_1 * w_2 * (z - y))$$

  This will result into

  $$w_4 = 4 - 0.1 * (2 * 2 * 1 * (21 - 3))) = -3.2$$

- And the new $w_5$ will be:

  $$w_5 = 1 - 0.1 * (x * w_1 * w_3 * (z - y))$$

  This will result into

  $$w_5 = 1 - 0.1 * (2 * 2 * 2 * (21 - 3))) = -13.4$$

In [131]:

```
# %load basic_graph.py
'''
Implementations of nodes for a computation graph. Each node
has a forward pass and a backward pass function, allowing
for the evaluation and backpropagation of data.
'''

from abc import ABC, abstractmethod
import math
import time


class Node(object):

    def __init__(self, inputs):
        self.inputs = inputs

    @abstractmethod
    def forward(self):
        ''' Feed-forward the result '''
        raise NotImplementedError("Missing forward-propagation method.")

    @abstractmethod
    def backward(self, d):
        ''' Back-propagate the error
            d is the delta of the subsequent node in the network '''
        raise NotImplementedError("Missing back-propagation method.")


class ConstantNode(Node):

    def __init__(self, value):
        self.output = value

    def forward(self):
        return self.output

    def backward(self, d):
        pass


class VariableNode(Node):

    def __init__(self, value):
        self.output = value

    def forward(self):
        return self.output

    def backward(self, d):
        self.output -= 0.1 * d # Gradient Descent


class AdditionNode(Node):

    def forward(self):
        self.output = sum([i.forward() for i in self.inputs])
        return self.output
```

```
    def backward(self, d):
        for i in self.inputs:
            i.backward(d)


class MultiplicationNode(Node):

    def forward(self):
        self.output = self.inputs[0].forward() * self.inputs[1].forward()
        return self.output

    def backward(self, d):
        self.inputs[0].backward(d * self.inputs[1].output)
        self.inputs[1].backward(d * self.inputs[0].output)


class MSENode(Node):

    def forward(self):
        self.output = 0.5 * (
            self.inputs[0].forward() - self.inputs[1].forward())**2
        return self.output

    def backward(self, d):
        self.inputs[0].backward(d * (self.inputs[0].output - self.inputs[1].output))
        self.inputs[1].backward(d * (self.inputs[1].output - self.inputs[0].output))


#class SigmoidNode(Node):
#
 #   def forward(self):
  #      self.output = 1.0 / (1.0 + math.exp(-self.inputs[0].forward()))
   #     return self.output
#
 #   def backward(self, d):
  #      self.inputs[0].backward(d * self.output * (1.0 - self.output))

class ReLUNode(Node):

    def forward(self):
        self.output = np.maximum(0, self.inputs[0].forward())
        return self.output
        #raise NotImplementedError("Forward pass for ReLU activation node has not been impl

    def backward(self, d):
        self.inputs[0].backward(d * self.output)

        #raise NotImplementedError("Backward pass for ReLU activation node has not been imp

#class TanhNode(object):
#
 #   def forward(self):
  #      raise NotImplementedError("Forward pass for tanh activation node has not been impl
#
 #   def backward(self, d):
  #      raise NotImplementedError("Backward pass for tanh activation node has not been imp
#
# Example graph as shown in MLP lecture slides
#class SampleGraph(object):
#
 #   def __init__(self, x, y, w, b):
```

```
  #      ''' x: input
   #         y: expected output
    #        w: initial weight
     #       b: initial bias '''
      #  self.w = VariableNode(w)
       # self.b = VariableNode(b)
        #self.graph = MSENode([
         #   AdditionNode([
          #      MultiplicationNode([
           #         ConstantNode(x),
            #        self.w
             #   ]),
              #  MultiplicationNode([
               #     self.b,
                #    ConstantNode(1)
                #])
            # ]),
             #ConstantNode(y)
       # ])

#    def forward(self):
 #       return self.graph.forward()
#
 #   def backward(self, d):
  #      self.graph.backward(d)


class Neuron(Node):

    def __init__(self, inputs, weights, activation):
        ''' weights: list of initial weights, same length as inputs '''
        self.inputs = inputs
        # Initialize a weight for each input
        self.weights = [VariableNode(weight) for weight in weights]
        # Neurons normally have a bias, ignore for this assignment
        #self.bias = VariableNode(bias, "b")

        # Multiplication node for each pair of inputs and weights
        mults = [MultiplicationNode([i, w]) for i, w, in zip(self.inputs, self.weights)]
        # Neurons normally have a bias, ignore for this assignment
        #mults.append(MultiplicationNode([self.bias, ConstantNode(1)]))

        # Sum all multiplication results
        added = AdditionNode(mults)

        # Apply activation function
        if activation == 'sigmoid':
            self.graph = SigmoidNode([added])
        elif activation == 'relu':
            self.graph = ReLUNode([added])
        elif activation == 'tanh':
            self.graph = TanhNode([added])
        else:
            raise ValueError("Unknown activation function.")

    def forward(self):
        return self.graph.forward()

    def backward(self, d):
        self.graph.backward(d)
```

```
    def set_weights(self, new_weights):
        for i in len(new_weights):
            self.weights[i].output = new_weights[i]

    def get_weights(self):
        return [weight.output for weight in self.weights]

class InputLayer(Node):
    def __init__(self):
        self.nodes = [ConstantNode(2)]

    def forward(self):
        pass

    def backward(self, i):
        for node in self.nodes:
            for output in node.outputs:
                outputs.backward(None)

    def set_inputs(self, values):
        for node, value in zip(self.nodes, values):
            node.output = value

class HiddenLayer(Node):

    def __init__(self, inputs, num_neurons, weights, activation='relu'):
        self.neurons = [Neuron(inputs.nodes, weights = weights, activation=activation) for

    def forward(self):
        pass

    def backward(self, i):
        pass



class OutputLayer(Node):

    def __init__(self, inputs):
        self.expected = [ConstantNode(0) for i in inputs]
        self.nodes = [MSENode([i, e])
                      for i, e in zip(input.nodes, self.expected)]
        self.graph = AdditionNode(self.nodes)

    def forward(self):
        self.output = self.graph.forward()
        return self.output

    def backward(self, i):
        pass

    def set_expected(self, values):
        for node, value in zip(self.expected, values):
            node.output = value

#network_in = InputLayer(1)
#network_out = HiddenLayer(network_in, 1)
#network_out = HiddenLayer(network_in, 2)
#network_out = HiddenLayer(network_in, 1)
#network_out = OutputLayer(network_out)
#network_in.set_inputs([2])
```

```
#network_out.set_expected([3])
#network_out.forward()
#network_in.backward(1)

if __name__ == '__main__':
    print("Loaded simple graph nodes")

    # Example network
    #sg = SampleGraph(2, 2, 2, 1)
    #prediction = sg.forward()
    #print("Initial prediction is", prediction)
    #sg.backward(1)
    #print("w has new value", sg.w.output)
    #print("b has new value", sg.b.output)

    # Run your network here
    network_in = InputLayer()
    network_out = HiddenLayer(network_in, 1, [2])
    network_out = HiddenLayer(network_out, 2, [1,2])
    network_out = HiddenLayer(network_out, 1, [4,1])
    network_out = OutputLayer(network_out)
    network_in.set_inputs(2)
    network_out.set_expected(3)
    network_out.forward()
    network_in.backward(1)
```

```
Loaded simple graph nodes

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-131-32c77866777c> in <module>()
    259     network_in = InputLayer()
    260     network_out = HiddenLayer(network_in, 1, [2])
--> 261     network_out = HiddenLayer(network_out, 2, [1,2])
    262     network_out = HiddenLayer(network_out, 1, [4,1])
    263     network_out = OutputLayer(network_out)

<ipython-input-131-32c77866777c> in __init__(self, inputs, num_neurons, weig
hts, activation)
    206
    207     def __init__(self, inputs, num_neurons, weights, activation='rel
u'):
--> 208         self.neurons = [Neuron(inputs=inputs.nodes, weights = weight
s, activation=activation) for i in range(num_neurons)]
    209
    210     def forward(self):

<ipython-input-131-32c77866777c> in <listcomp>(.0)
    206
    207     def __init__(self, inputs, num_neurons, weights, activation='rel
u'):
--> 208         self.neurons = [Neuron(inputs=inputs.nodes, weights = weight
s, activation=activation) for i in range(num_neurons)]
    209
    210     def forward(self):

AttributeError: 'HiddenLayer' object has no attribute 'nodes'
```

Explanation for code alterations

There were a few things missing in the code, namely the ReLU class node was not defined, and for the purpose of simulating the graph as given, it needs to be built up from different layers. As to that goal, the code for InputLayer, HiddenLayer, and OutputLayer from the tutorial on computational graphs and MLP was taken.

The ReLU class was defined so that it would put forward its own inputvalue if this is larger than 0, and else 0. Backward, it pass on the value fed to it.

The model as presented for this question was interpreted as having an input layer with 1 input, 3 hidden layers with 1, 2 and 1 node respectively and lastly an output layer where MSE would be used to calculate the loss. The HiddenLayer class was adjusted so that it would accept a list of weights and pass this over to the Neuron nodes.

Unfortunately we weren't able to get the code running...

# 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

Submit an explanation for the limitation of the original model. Explain your modification. Submit your code and 6 plots (can be overlaid) for the training performance of both models with different depths.

In [4]:

```
# (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
```

```
/Users/Jules/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:34: Futu
reWarning: Conversion of the second argument of issubdtype from `float` to `
np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [5]:

```
# (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)
```

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz (http
s://s3.amazonaws.com/img-datasets/mnist.npz)
11493376/11490434 [==============================] - 16s 1us/step
60000 train samples
10000 test samples
```

In [56]:

```
# Use this parameter to change the depth of the model
number_hidden_layers = 3# Number of hidden layers
```

In [57]:

```
# 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('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'])
```

```
_________________________________________________________________
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
Trainable params: 932,362
Non-trainable params: 0
_________________________________________________________________
```

In [58]:

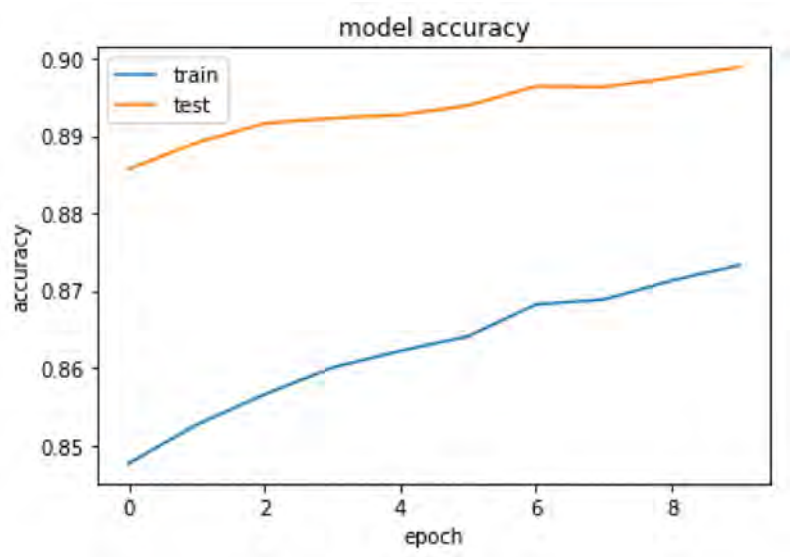
```
# Training (You don't need to change this part of the code)
history = model.fit(X_train, Y_train,
                    batch_size=batch_size, nb_epoch=nb_epoch,
                    verbose=1, validation_data=(X_test, Y_test));
score = model.evaluate(X_test, Y_test, verbose=0);
```

```
/Users/Jules/anaconda3/lib/python3.6/site-packages/keras/models.py:942: User
Warning: The `nb_epoch` argument in `fit` has been renamed `epochs`.
  warnings.warn('The `nb_epoch` argument in `fit` '

Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 21s 344us/step - loss: 2.1660
- acc: 0.2336 - val_loss: 1.7495 - val_acc: 0.6144
Epoch 2/10
60000/60000 [==============================] - 19s 320us/step - loss: 1.5194
- acc: 0.5058 - val_loss: 0.9053 - val_acc: 0.7639
Epoch 3/10
60000/60000 [==============================] - 18s 305us/step - loss: 1.0055
- acc: 0.6631 - val_loss: 0.6214 - val_acc: 0.8210
Epoch 4/10
60000/60000 [==============================] - 18s 303us/step - loss: 0.8068
- acc: 0.7297 - val_loss: 0.5184 - val_acc: 0.8456
Epoch 5/10
60000/60000 [==============================] - 17s 287us/step - loss: 0.7060
- acc: 0.7684 - val_loss: 0.4614 - val_acc: 0.8649
Epoch 6/10
60000/60000 [==============================] - 20s 326us/step - loss: 0.6349
- acc: 0.7934 - val_loss: 0.4305 - val_acc: 0.8723
Epoch 7/10
60000/60000 [==============================] - 18s 294us/step - loss: 0.5930
- acc: 0.8096 - val_loss: 0.4047 - val_acc: 0.8797
Epoch 8/10
60000/60000 [==============================] - 17s 284us/step - loss: 0.5630
- acc: 0.8198 - val_loss: 0.3844 - val_acc: 0.8841
Epoch 9/10
60000/60000 [==============================] - 17s 282us/step - loss: 0.5410
- acc: 0.8280 - val_loss: 0.3676 - val_acc: 0.8882
Epoch 10/10
60000/60000 [==============================] - 17s 287us/step - loss: 0.5104
- acc: 0.8386 - val_loss: 0.3564 - val_acc: 0.8912
```

In [38]:

```
print('Activation: Sigmoid')
print('Depth: 1')
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'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

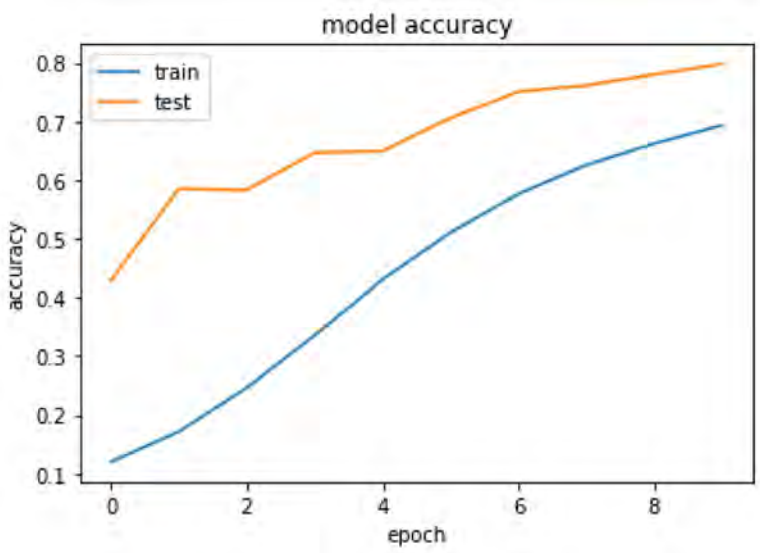
```
Activation: Sigmoid
Depth: 1
Test score: 0.3654909325242043
Test accuracy: 0.8989
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```



In [42]:

```
#Code deleted for brevity
```

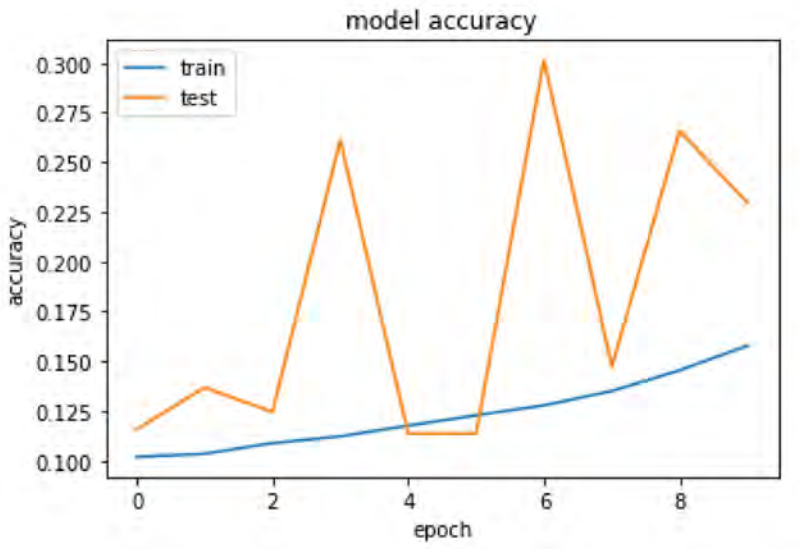
```
Activation: Sigmoid
Depth: 2
Test score: 0.8513415504455566
Test accuracy: 0.7978
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```




In [46]:

```
#Code deleted for brevity
```

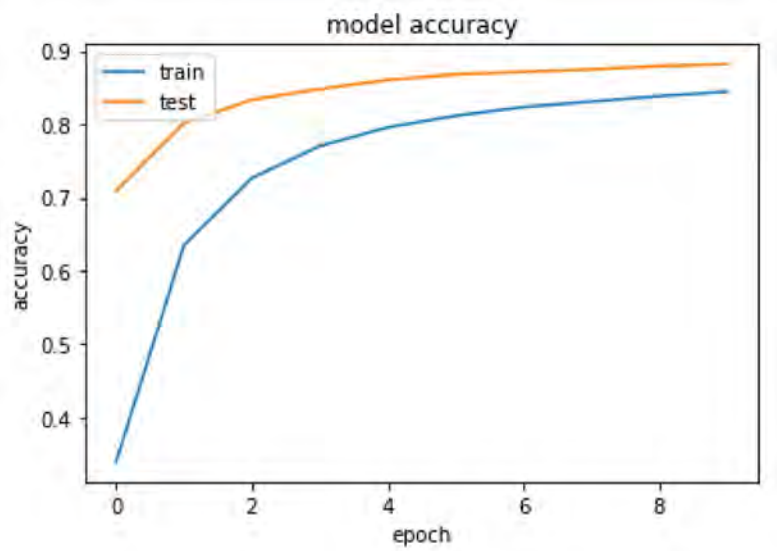
```
Activation: Sigmoid
Depth: 3
Test score: 2.2199254230499266
Test accuracy: 0.2294
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```



In [50]:

```
#Code deleted for brevity
```

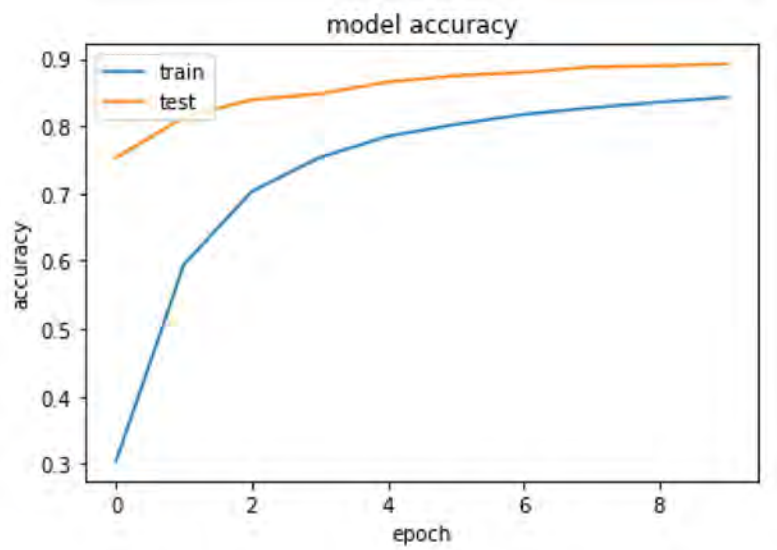
```
Activation: ReLu
Depth: 1
Test score: 0.4640749076843262
Test accuracy: 0.8819
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```




In [55]:

```
#Code deleted for brevity
```

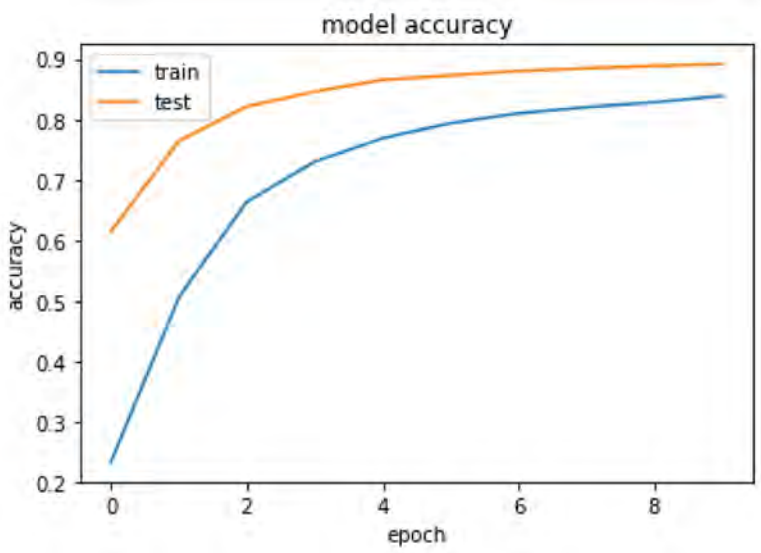
```
Activation: ReLu
Depth: 2
Test score: 0.3757955534338951
Test accuracy: 0.892
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```



In [61]:

```
#Code deleted for brevity
```

```
Activation: Relu
Depth: 3
Test score: 0.35636908671855927
Test accuracy: 0.8912
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```




Answer to 2.

The problem with the initial model, causing its performance to decrease as the depth of the model increases is the chosen activation function of the model. The gradient of the Sigmoid function is always below 1, which in itself isn't necessarily a problem. However, for high input values, the gradient of the sigmoid function will become smaller and by adding layers, these small values will be multiplied, eventually nearing a value of zero. This is called 'vanishing gradients' and is the reason a sigmoid activation function isn't suited well for deep learning models with many layers.

The ReLu activation function doesn't have this problem, as it simply returns 0 for input values smaller or equal to 0 or the actual input value for values larger than 0. The gradient isn't always below 1 as is the case and problem for Sigmoid activation, and multiplication over multiple layers doesn't cause the gradients to reach zero. ReLu is therefore a better choice for deep learning models with multiple layers.

# MNIST Calculator (6 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. You now need to build a model that is a combination of convolutional layers and recurrent cells.

Using the KERAS library, design and train a model that produces a sum of a sequence of MNIST images. More specifically, the model should input a sequence of 10 images and compute the cumulative sum of the digits represented by the images.

For example:

Input 1:

Output 1: 15

Input 2:

Output 2: 7

Your solutions should include:

- Python code that formats the MNIST dataset such that it can be used for traning and testing your model
- Implementation in keras of your model (for training and testing)
- Performance on the model on test data
- Justification (in text) of your decisions for the model architecture (type of layers, activation functions, loss function, regularization and training hyperparameters)

Note: Use the 60000/10000 train/test split of the MNIST dataset

In [1]:

```
# Imports
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.models import Model
from keras.layers import Activation, Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Input, BatchNormalization, RepeatVector
from keras.layers import LSTM, GRU
from keras.layers.wrappers import TimeDistributed
from keras import backend as K

import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (10, 10) # Make the figures a bit bigger
import numpy as np
```

```
/Users/Jules/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:34: Futu
reWarning: Conversion of the second argument of issubdtype from `float` to `
np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [2]:

```
# Training parameters
batch_size = 128
num_classes = 10
epochs = 12
```

In [3]:

```
# Data preparation

# Input image dimensions
img_rows, img_cols = 28,28
img_channels = 1 # Greyscale so 1

# The data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
class_names = ("0", "1", "2", "3", "4", "5", "6", "7", "8", "9")

for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.axis('off')
    plt.imshow(x_train[i], cmap='gray', interpolation='none')
    #plt.title("Class: {}".format(class_names[y_train[i][0]]))

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], img_channels, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], img_channels, img_rows, img_cols)
    input_shape = (img_channels, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, img_channels)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, img_channels)
    input_shape = (img_rows, img_cols, img_channels)

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')

print(y_train)
print(y_test)

# Convert class vectors to binary class matrices
#y_train = keras.utils.to_categorical(y_train, num_classes) #one-hot encoding
#y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
60000 train samples
10000 test samples
[5 0 4 ... 5 6 8]
[7 2 1 ... 4 5 6]
```

In [4]:

```
# Convert training and test set into groups of 10 images
x_train_sequence = []
y_train_sequence = []

x_test_sequence = []
y_test_sequence = []

for i in range(0, len(x_train), 10):
    group_x = np.stack(x_train[i:i+10])
    group_y = np.sum(y_train[i:i+10])
    x_train_sequence.append(group_x)
    y_train_sequence.append(group_y)

for i in range(0, len(x_test), 10):
    group_x = np.stack(x_test[i:i+10])
    group_y = np.sum(y_test[i:i+10])
    x_test_sequence.append(group_x)
    y_test_sequence.append(group_y)

x_train_sequence = np.array(x_train_sequence)
y_train_sequence = np.array(y_train_sequence)
x_test_sequence = np.array(x_test_sequence)
y_test_sequence = np.array(y_test_sequence)
```

In [16]:

```
# Model definition
model = Sequential()
model.add(TimeDistributed(Conv2D(32, (3, 3), padding='same'), input_shape=(10,28,28,1)))
model.add(TimeDistributed(Activation('relu')))
model.add(TimeDistributed(Conv2D(32, (3, 3))))
model.add(TimeDistributed(Activation('relu')))
model.add(TimeDistributed(MaxPooling2D(pool_size=(2, 2))))
model.add(TimeDistributed(Dropout(0.25)))

model.add(TimeDistributed(Flatten()))

model.add(LSTM(128, return_sequences=True))

model.add(BatchNormalization())
model.add(Dropout(0.2))

model.add(Flatten())
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('relu'))

model.compile(loss=keras.losses.mean_squared_error,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['mae'])
model.summary()
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| time_distributed_22 (TimeDis | (None, 10, 28, 28, 32) | 320 |
| time_distributed_23 (TimeDis | (None, 10, 28, 28, 32) | 0 |
| time_distributed_24 (TimeDis | (None, 10, 26, 26, 32) | 9248 |
| time_distributed_25 (TimeDis | (None, 10, 26, 26, 32) | 0 |
| time_distributed_26 (TimeDis | (None, 10, 13, 13, 32) | 0 |
| time_distributed_27 (TimeDis | (None, 10, 13, 13, 32) | 0 |
| time_distributed_28 (TimeDis | (None, 10, 5408) | 0 |
| lstm_4 (LSTM) | (None, 10, 128) | 2834944 |
| batch_normalization_4 (Batch | (None, 10, 128) | 512 |
| dropout_11 (Dropout) | (None, 10, 128) | 0 |
| flatten_8 (Flatten) | (None, 1280) | 0 |
| dense_7 (Dense) | (None, 256) | 327936 |
| activation_15 (Activation) | (None, 256) | 0 |
| dropout_12 (Dropout) | (None, 256) | 0 |

```
dense_8 (Dense)              (None, 1)                 257
_________________________________________________________________
activation_16 (Activation)   (None, 1)                 0
=================================================================
Total params: 3,173,217
Trainable params: 3,172,961
Non-trainable params: 256
_________________________________________________________________
```

In [6]:

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.loss = []
        self.val_loss = []
        self.acc = []
        self.val_acc = []

    def on_batch_end(self, batch, logs={}):
        self.loss.append(logs.get('loss'))
        self.acc.append(logs.get('acc'))

lh = LossHistory()
# Training Loop
history = model.fit(x_train_sequence, y_train_sequence,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test_sequence, y_test_sequence),
          callbacks=[lh])
score = model.evaluate(x_test_sequence, y_test_sequence, verbose=1)
print('Test loss:', score[0])
print('Test mean absolute error:', score[1])
```

```
Train on 6000 samples, validate on 1000 samples
Epoch 1/12
6000/6000 [==============================] - 261s 43ms/step - loss: 150.1340
- mean_absolute_error: 8.4179 - val_loss: 94.8874 - val_mean_absolute_error:
8.2298
Epoch 2/12
6000/6000 [==============================] - 249s 41ms/step - loss: 66.0010
- mean_absolute_error: 6.4958 - val_loss: 62.2235 - val_mean_absolute_error:
6.8093
Epoch 3/12
6000/6000 [==============================] - 226s 38ms/step - loss: 54.2638
- mean_absolute_error: 5.8391 - val_loss: 20.8691 - val_mean_absolute_error:
3.5740
Epoch 4/12
6000/6000 [==============================] - 226s 38ms/step - loss: 42.0764
- mean_absolute_error: 5.1334 - val_loss: 56.2081 - val_mean_absolute_error:
6.6213
Epoch 5/12
6000/6000 [==============================] - 224s 37ms/step - loss: 53.7609
- mean_absolute_error: 5.8396 - val_loss: 19.4284 - val_mean_absolute_error:
3.4340
Epoch 6/12
6000/6000 [==============================] - 220s 37ms/step - loss: 52.5882
- mean_absolute_error: 5.7513 - val_loss: 14.7101 - val_mean_absolute_error:
3.0305
Epoch 7/12
6000/6000 [==============================] - 220s 37ms/step - loss: 48.1074
- mean_absolute_error: 5.5240 - val_loss: 16.5363 - val_mean_absolute_error:
3.1628
Epoch 8/12
6000/6000 [==============================] - 218s 36ms/step - loss: 43.7312
- mean_absolute_error: 5.2792 - val_loss: 38.3216 - val_mean_absolute_error:
5.3699
Epoch 9/12
6000/6000 [==============================] - 234s 39ms/step - loss: 39.3844
- mean_absolute_error: 4.9923 - val_loss: 79.3655 - val_mean_absolute_error:
```

```
8.1218
Epoch 10/12
6000/6000 [==============================] - 220s 37ms/step - loss: 40.2006
- mean_absolute_error: 5.0670 - val_loss: 51.1625 - val_mean_absolute_error:
6.3083
Epoch 11/12
6000/6000 [==============================] - 230s 38ms/step - loss: 40.0292
- mean_absolute_error: 5.0106 - val_loss: 45.6675 - val_mean_absolute_error:
5.8124
Epoch 12/12
6000/6000 [==============================] - 220s 37ms/step - loss: 42.3175
- mean_absolute_error: 5.2127 - val_loss: 12.7986 - val_mean_absolute_error:
2.8484
1000/1000 [==============================] - 13s 13ms/step
Test loss: 12.798572811126709
Test mean absolute error: 2.8484191398620604
```

For the problem of learning to predict the right sum for a series of 10 handwritten digits, several types of model layers are necessary. First of all, the information of the images needs to be encoded. For that, convolutional layers were used, as would be done in a task where the digits would have to be classified, since convolutional layers are good at encoding complex data. After that, a maxpooling layer was added to reduce the amount of data points per image, and a flatten layer was added to combine all image data into one dimension. However, since the input consisted of a sequence of 10 images instead of just one image, something had to be added to enable the neural network to loop over the sequence of images. The extra dimension of the sequence was treated as temporal information, and the all layers explained above were wrapped in a time distributed layer to allow for this looping. This way, the images could be encoded individually. To be able to then encode the sequential data into one sum, an LSTM layer was added. After that, Batch Normalization was used to normalize the influence of the individual features on the outcome, and a flatten layer to lose the last extra dimension. Via a dense layer of 256 nodes the final layer of the network consists of only one node with a ReLu activation function, which effectively makes the model do regression. Connected to that, Mean Squared Error was used as a loss function, to be able to minimize the distance between the predicted sum and the actual sum. Mean Absolute Error was used to evaluate the performance of the model, to be able to get an intuitive feel for the performance of the model. Currently, the final performance of the model is a Mean Absolute Error of about 3 on the test set. A better result might be achieved with a higher number of epochs, but considering the time, 12 epochs seemed like a right amount. Also, after about 6 epochs, the improvement with each epoch became smaller and smaller.