Deep Learning PhD course
HW#2

Muhammad Osama

June 14, 2019

1 MNIST

1.1

- Network Architecture: Dense 100 - Dense 10 (Taken similar to what I used in homework $1B$)

- Optimizer: SGD with rate $\eta = 0.25$

- Loss: Categorical cross entropy

- Metrics: Accuracy

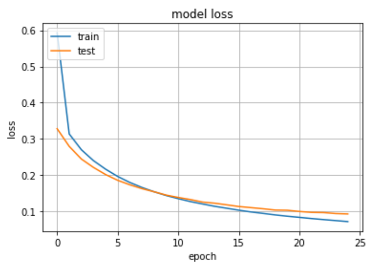
- Total number of parameters = $(784{*}100 + 100) + (100{*}10 + 10) = 79510$

- Epochs = 25, batch size = 100. Computation time ≈ 50 *sec.* In HW 1b, 6 number of epochs were used and same batch size but only the first epoch takes much longer than a minute. As far as accuracy is concerned, in HW1b at the end of 6 epochs the accuracy is around 97% whereas with this implementation, the accuracy is around 94% at the end of 6 epochs. But the final accuracy is similar.

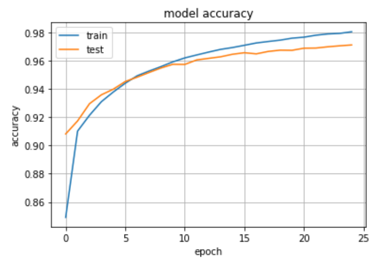
1.2

Total number of parameters:

$$\underbrace{(3*3*1+1)*8}_{1^{st}\ conv.\ layer} + \underbrace{(3*3*8+1)*16}_{2^{nd}\ conv.\ layer} + \underbrace{(3*3*16+1)*32}_{3^{rd}\ conv.\ layer} + \underbrace{7*7*32*10+10}_{full\ layer} = 21578$$



(a) Loss



(b) Accuracy

Figure 1

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_310 (Conv2D)          (None, 28, 28, 8)         80
_________________________________________________________________
activation_360 (Activation)  (None, 28, 28, 8)         0
_________________________________________________________________
max_pooling2d_148 (MaxPoolin (None, 14, 14, 8)         0
_________________________________________________________________
conv2d_311 (Conv2D)          (None, 14, 14, 16)        1168
_________________________________________________________________
activation_361 (Activation)  (None, 14, 14, 16)        0
_________________________________________________________________
max_pooling2d_149 (MaxPoolin (None, 7, 7, 16)          0
_________________________________________________________________
conv2d_312 (Conv2D)          (None, 7, 7, 32)          4640
_________________________________________________________________
activation_362 (Activation)  (None, 7, 7, 32)          0
_________________________________________________________________
flatten_6 (Flatten)          (None, 1568)              0
_________________________________________________________________
dense_8 (Dense)              (None, 10)                15690
_________________________________________________________________
activation_363 (Activation)  (None, 10)                0
=================================================================
Total params: 21,578
Trainable params: 21,578
Non-trainable params: 0
```

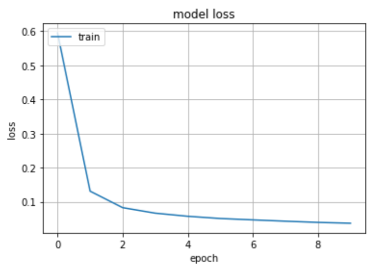
Figure 2

The number of parameters are much less as compared to the network in section 1.1.

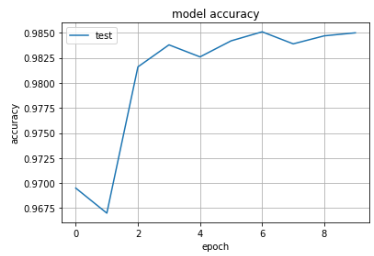
1.3

Activation function: tanh
Optimizer: SGD with rate $\eta = 0.25$

The only difference that can be seen from figure 4 output is that when the



(a) Training loss



(b) Test accuracy

Figure 3

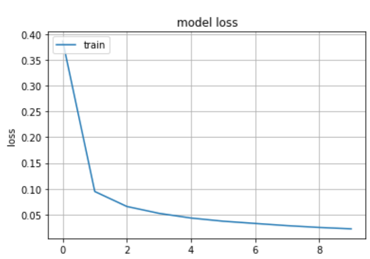
```
-- Activation layer before the pooling layer

Train on 60000 samples, validate on 10000 samples
Epoch 1/4
60000/60000 [==============================] - 28s 463us/step - loss: 0.2949 - acc: 0.9070 - val_loss: 0.0936 - val_acc: 0.9704
Epoch 2/4
60000/60000 [==============================] - 25s 416us/step - loss: 0.0859 - acc: 0.9731 - val_loss: 0.0612 - val_acc: 0.9793
Epoch 3/4
60000/60000 [==============================] - 25s 419us/step - loss: 0.0613 - acc: 0.9808 - val_loss: 0.0404 - val_acc: 0.9861
Epoch 4/4
60000/60000 [==============================] - 25s 414us/step - loss: 0.0503 - acc: 0.9847 - val_loss: 0.0397 - val_acc: 0.9866

--Activation layer after the pooling layer

Train on 60000 samples, validate on 10000 samples
Epoch 1/4
60000/60000 [==============================] - 28s 469us/step - loss: 0.2947 - acc: 0.9071 - val_loss: 0.0936 - val_acc: 0.9705
Epoch 2/4
60000/60000 [==============================] - 24s 394us/step - loss: 0.0859 - acc: 0.9731 - val_loss: 0.0611 - val_acc: 0.9792
Epoch 3/4
60000/60000 [==============================] - 24s 397us/step - loss: 0.0612 - acc: 0.9808 - val_loss: 0.0404 - val_acc: 0.9860
Epoch 4/4
60000/60000 [==============================] - 24s 396us/step - loss: 0.0502 - acc: 0.9847 - val_loss: 0.0397 - val_acc: 0.9865
```

Figure 4

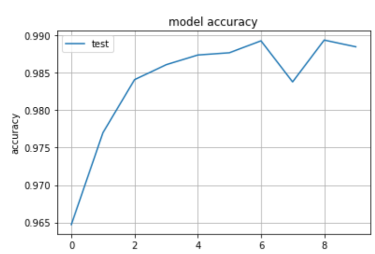
activation layer is after the pooling layer the computational time is slightly less. This is possible because the activation is computed for a smaller image in the later case and hence some time is saved.

1.4

Activation function: relu
Optimizer: Adam with default values of rate $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$.



(a) Training loss



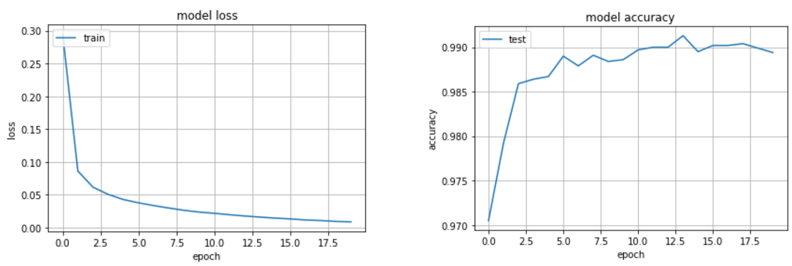
(b) Test accuracy

Figure 5

1.5

1.5.1

Activation changed from 'relu' to 'tanh'. Figure 6.

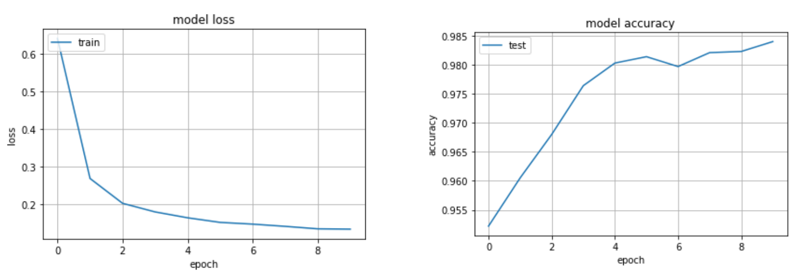


(a) Training loss (b) Test accuracy

Figure 6

1.5.2

Activation: relu
Drop 50% of the weights. Figure 7

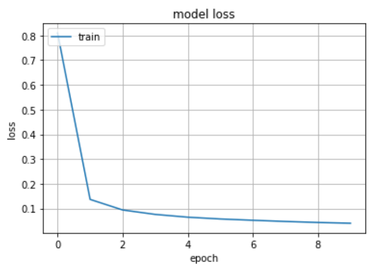


(a) Training loss (b) Test accuracy

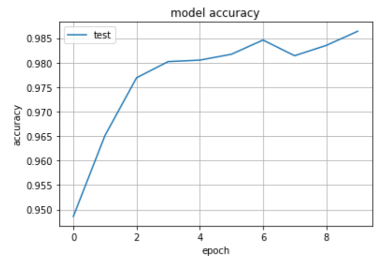
Figure 7

1.5.3

Optimizer:SGD with rate $\eta = 0.001$ and momentum $\alpha = 0.8$. Figure 8
Confusion matrix for the best model which was obtained with the activation function 'tanh' and is shown in figure 9. A few examples of misclassified images are shown in figure 10.



(a) Training Loss



(b) Test accuracy

Figure 8

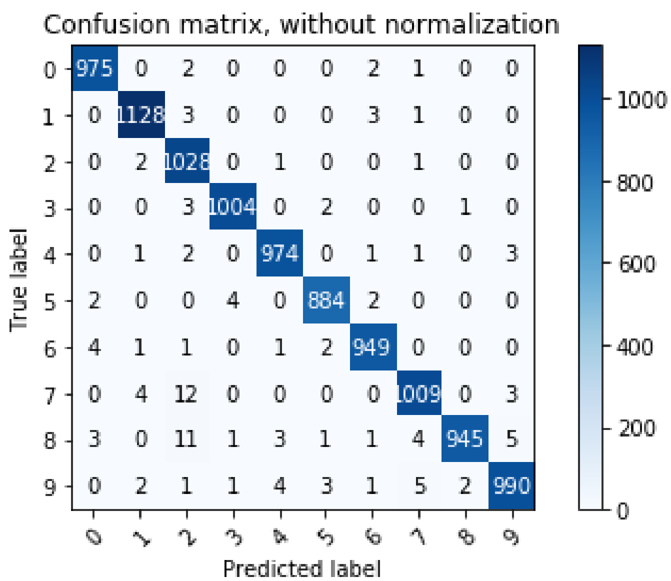


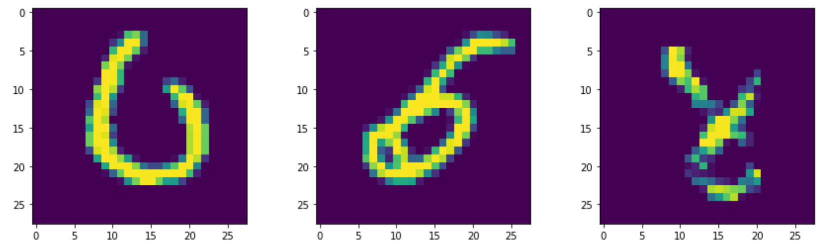
Figure 9: Confusion matrix for MNIST

2 WARWICK

2.1

The model summary for the model used is given in figure 11. The activation used was *relu*. The optimizer is *Adam* with it default values of learning rate $\eta = 0.001,\ \beta_1 = 0.9,\ \beta_2 = 0.999$ and $\epsilon = 10^{-7}$.

The loss function was modified to handle pixel wise class maps by defining the a new function. A function to calculate the Dice coefficient was also defined as shown in figure 12.



(a) True label 0, predicted label 6 (b) True label 5, predicted label 6 (c) True label 8, predicted label 2

Figure 10

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_275 (Conv2D)          (None, 128, 128, 8)       224
_________________________________________________________________
activation_308 (Activation)  (None, 128, 128, 8)       0
_________________________________________________________________
max_pooling2d_130 (MaxPoolin (None, 64, 64, 8)         0
_________________________________________________________________
conv2d_276 (Conv2D)          (None, 64, 64, 16)        1168
_________________________________________________________________
activation_309 (Activation)  (None, 64, 64, 16)        0
_________________________________________________________________
max_pooling2d_131 (MaxPoolin (None, 32, 32, 16)        0
_________________________________________________________________
conv2d_277 (Conv2D)          (None, 32, 32, 32)        4640
_________________________________________________________________
activation_310 (Activation)  (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_transpose_122 (Conv2D (None, 64, 64, 16)        2064
_________________________________________________________________
activation_311 (Activation)  (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_transpose_123 (Conv2D (None, 128, 128, 8)       520
_________________________________________________________________
activation_312 (Activation)  (None, 128, 128, 8)       0
_________________________________________________________________
conv2d_278 (Conv2D)          (None, 128, 128, 1)       9
_________________________________________________________________
activation_313 (Activation)  (None, 128, 128, 1)       0
=================================================================
Total params: 8,625
Trainable params: 8,625
Non-trainable params: 0
```

Figure 11

2.1.1

The Dice coefficient goes above 0.7 in approximately 200 iterations as shown in figure 13.
Figures 14 and 15 show two examples of test images which had low Dice coefficient. I think the reason that these test images have low Dice coefficient is because in most of the training images the glands are distributed over the space. But in these specific cases in the test data, the gland area is concentrated. This is also suggested by the predicted class map since it sort of gives a spatially

```
from keras import backend as K
from keras.activations import softmax


_EPSILON = K.epsilon()
def pixel_wise_loss(y_true, y_pred):
    y_pred = K.clip(y_pred, _EPSILON, 1.0-_EPSILON)

    loss = y_true*K.log(y_pred) + (1-y_true) * K.log(1-y_pred)
    #loss = K.sum(loss,axis=1)
    #loss = K.sum(loss,axis=1)
    return -loss

def dice_coef(y_true, y_pred):
    """
    Dice = (2*|X & Y|)/ (|X|+ |Y|)
         =  2*sum(|A*B|)/(sum(A^2)+sum(B^2))
    """
    y_true = K.flatten(y_true)
    y_pred = K.flatten(y_pred)
    intersection = K.sum(y_true * y_pred)
    return (2 * intersection) / (K.sum(y_true) + K.sum(y_pred))
```

Figure 12

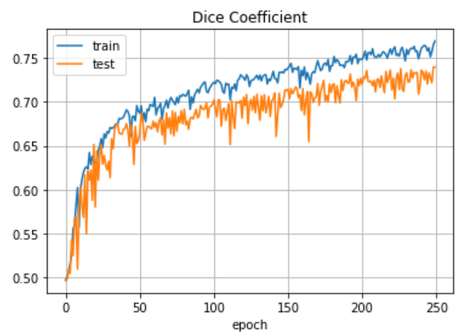


Figure 13

distributed class map.

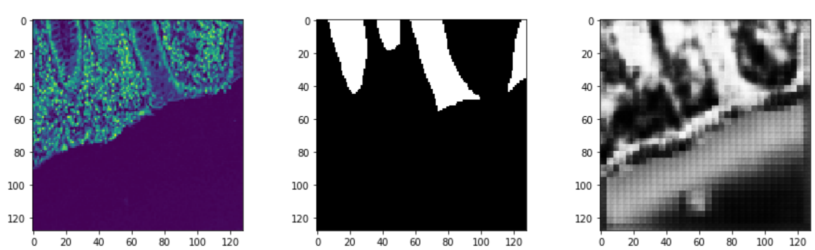


Figure 14: (a) Actual image (b) True class map (c) Predicted class map

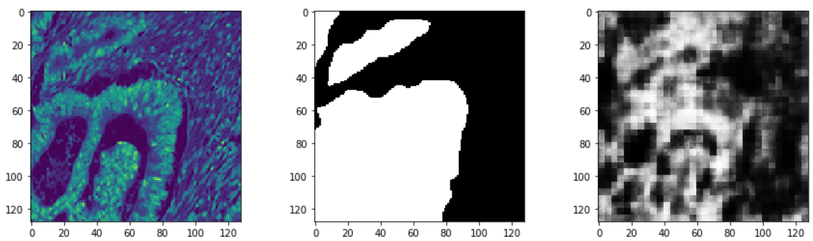
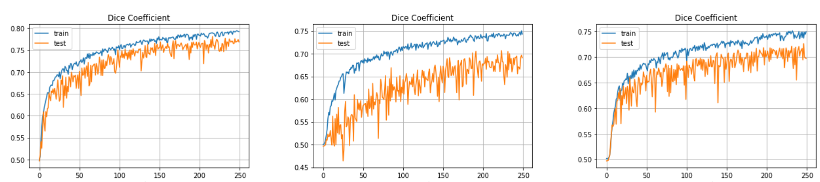


Figure 15: (a) Actual image (b) True class map (c) Predicted class map

2.2

The Dice coefficient for the three different approaches is shown in figure 16.
(a) Data augmentation: Horizontally flipped the images and class maps of the training data and augmented to the original. The training data size doubles and its advantage can be clearly seen in figure 16a. The dice coefficient on the test data becomes greater than 0.7 in only 25 epochs where as in figure 13 it took almost 200 epochs.
(b) Dropping out 30% of the units in each convolution layer. The number of parameters in the original network were only approximately 8000 therefore when we drop out the units that the Dice coefficient increases slowly.
(c) Weight regularization using a $\lambda = 10^{-4}$



(a) (b) (c)

Figure 16: (a) Data augmentation (b) Dropout(0.3) (c) Regularization

Appendix

Task1.1

```
import load_mnist as data
import numpy as np
from keras.models import Sequential
from keras.layers import Dense,Dropout
from keras import optimizers
import matplotlib.pyplot as plt

#load data
#X_train, Y_train, X_test, Y_test = data.
    load_mnist();

#reshape
#X_train = X_train.reshape(60000,28,28,1)
#X_test = X_test.reshape(10000,28,28,1)

# fix random seed for reproducibility
seed = 7
np.random.seed(seed)

#create model
model = Sequential()
#add model layers
model.add(Dense(100, activation='sigmoid
    ',input_dim=784))
model.add(Dense(10, activation='softmax')
    )

#optimizer
sgd = optimizers.SGD(lr=0.25, momentum
    =0.0, decay=0.0, nesterov=False);

#compile model using accuracy to measure
    model performance
model.compile(optimizer=sgd , loss='
    categorical_crossentropy', metrics=['
    accuracy'])

#train the model
history = model.fit(X_train, Y_train,
    validation_data = (X_test,Y_test),
    epochs=25,batch_size = 100)
```

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

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.grid()
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper
    left')
plt.show()
```

Task1.2

```
import load_mnist as data
import numpy as np
from keras.models import Sequential
from keras.layers import Conv2D,
    Activation, MaxPooling2D,
    ZeroPadding2D, Flatten,Dense
from keras import optimizers
import matplotlib.pyplot as plt
import funcs_segmentation as fs

#load data
#X_train, Y_train, X_test, Y_test = data.
    load_mnist();

#reshape
#X_train = X_train.reshape(60000,28,28,1)
#X_test = X_test.reshape(10000,28,28,1)

# fix random seed for reproducibility
seed = 7
np.random.seed(seed)

#create model
model = Sequential()
#add model layers
#1
#model.add(ZeroPadding2D(padding=1,
    data_format = 'channels_last'))
model.add(Conv2D(8,kernel_size=(3,3),
    strides=1,padding='same',data_format='
    channels_last',use_bias='True',
    kernel_initializer='random_normal',
    bias_initializer='zeros',input_shape
    =(28,28,1)))
#2
model.add(Activation('relu'))
#3
model.add(MaxPooling2D(pool_size=(2, 2),
    strides=2, padding='valid',
    data_format='channels_last'))
#4
#model.add(ZeroPadding2D(padding=1,
    data_format = 'channels_last'))
model.add(Conv2D(16,kernel_size=(3,3),
```

```
    strides=1,padding='same',data_format='
    channels_last',use_bias='True',
    kernel_initializer='random_normal',
    bias_initializer='zeros'))
#5
model.add(Activation('relu'))
#6
model.add(MaxPooling2D(pool_size=(2, 2),
    strides=2, padding='valid',
    data_format='channels_last'))
#7
#model.add(ZeroPadding2D(padding=1,
    data_format = 'channels_last'))
model.add(Conv2D(32,kernel_size=(3,3),
    strides=1,padding='same',data_format='
    channels_last',use_bias='True',
    kernel_initializer='random_normal',
    bias_initializer='zeros'))
#8
model.add(Activation('relu'))
#
model.add(Flatten(data_format='
    channels_last'))
#9
model.add(Dense(10,use_bias='True',
    kernel_initializer='random_normal',
    bias_initializer='zeros'))
#10
model.add(Activation('softmax'))

#optimizer
sgd = optimizers.SGD(lr=0.25, momentum
    =0.0, decay=0.0, nesterov=False);

#compile model using accuracy to measure
    model performance
model.compile(optimizer=sgd , loss='
    categorical_crossentropy', metrics=['
    accuracy'])

#train the model
history = model.fit(X_train, Y_train,
    validation_data = (X_test,Y_test),
    epochs=4,batch_size = 100)

#get accuracy on test data
```

```
#model.evaluate(X_test, Y_test)

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

# summarize history for loss
plt.plot(history.history['loss'])
#plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.grid()
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()
```

For tasks 1.3 to 1.5 small changes were done to the file for task1.2 as mentioned in the report. Therefore it would just be redundant to paste their code here.

Task 2-Warwick

```
import load_warwick_augmented as data
#import numpy as np
from keras.models import Sequential
from keras.layers import Conv2D,
    Activation, MaxPooling2D,
    Conv2DTranspose#, Dropout
from keras import optimizers
#from keras.regularizers import l2
import funcs_segmentation as fs
import matplotlib.pyplot as plt

#load data
X_train, Y_train, X_test, Y_test = data.
    load_warwick();

# fix random seed for reproducibility
#seed = 7
#np.random.seed(seed)

#create model
model = Sequential()
#add model layers
#####
model.add(Conv2D(8,kernel_size=(3,3),
    strides=1,padding='same',data_format='
    channels_last',input_shape=(128,128,3)
    ))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2),
    strides=2, padding='valid',
    data_format='channels_last'))

#####
model.add(Conv2D(16,kernel_size=(3,3),
    strides=1,padding='same',data_format='
    channels_last'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2),
    strides=2, padding='valid',
    data_format='channels_last'))
```

```
#####
model.add(Conv2D(32,kernel_size=(3,3),
    strides=1,padding='same', data_format
    ='channels_last'))
model.add(Activation('relu'))

#####
model.add(Conv2DTranspose(16, kernel_size
    =(2,2), strides=(2, 2), padding='same
    ',  data_format='channels_last'));
model.add(Activation('relu'))

model.add(Conv2DTranspose(8, kernel_size
    =(2,2), strides=(2, 2), padding='same
    ', data_format='channels_last'));
model.add(Activation('relu'))


model.add(Conv2D(1,kernel_size = (1,1)))

model.add(Activation('sigmoid'))

#optimizer
sgd = optimizers.SGD(lr=0.001, momentum
    =0.0, decay=0.0, nesterov=False);

#compile model using accuracy to measure
    model performanc
model.compile(optimizer='adam' , loss=fs.
    pixel_wise_loss , metrics=[fs.dice_coef
    ])

#train the model
history = model.fit(X_train, Y_train,
    validation_data = (X_test,Y_test),
    epochs=250, batch_size = 5)

#get accuracy on test data
#model.evaluate(X_test,Y_test)

# list all data in history
#print(history.history.keys())

# summarize history for accuracy
plt.plot(history.history['dice_coef'])
plt.plot(history.history['val_dice_coef
```

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

# summarize history for loss
#plt.plot(history.history['loss'])
#plt.plot(history.history['val_loss'])
#plt.title('model loss')
#plt.grid()
#plt.ylabel('loss')
#plt.xlabel('epoch')
#plt.legend(['train'], loc='upper left')
#plt.show()
```

Supporting functions for task2

```
from keras import backend as K
from keras.activations import softmax


_EPSILON = K.epsilon()
def pixel_wise_loss(y_true, y_pred):
y_pred = K.clip(y_pred, _EPSILON, 1.0-
    _EPSILON)

loss = y_true*K.log(y_pred) + (1-y_true)
    * K.log(1-y_pred)
#loss = K.sum(loss,axis=1)
#loss = K.sum(loss,axis=1)
return -loss

def dice_coef(y_true, y_pred):
"""
Dice = (2*|X & Y|)/ (|X|+ |Y|)
= 2*sum(|A*B|)/(sum(A^2)+sum(B^2))
"""
y_true = K.flatten(y_true)
y_pred = K.flatten(y_pred)
intersection = K.sum(y_true * y_pred)
return (2 * intersection) / (K.sum(y_true
    ) + K.sum(y_pred))


def my_cross_entrp(y_true,y_pred):
y_pred = K.clip(y_pred, _EPSILON, 1.0-
    _EPSILON)

loss = K.sum(y_true*K.log(y_pred),axis=1)

return -loss




def dice_loss(y_true,y_pred):

dloss = 1-dice_coef(y_true,y_pred)

return dloss
```

```
def softMaxAxis1(x):
return softmax(x, axis=2)
```

Data Augmentation

```
import numpy as np
import matplotlib.pyplot as misc

def load_warwick():
# Loads the MNIST dataset from png images

NUM_TEST_IMAGES = 60
# create list of image objects
test_images = []
test_class_maps = []

for label in np.arange(1,NUM_TEST_IMAGES
    +1):
if label<10:
image_path= "WARWICK/WARWICK/Test/image_"
     + str(0)+str(label) + ".png"
class_map_path= "WARWICK/WARWICK/Test/
    label_" + str(0)+str(label) + ".png"
else:
image_path= "WARWICK/WARWICK/Test/image_"
     + str(label) + ".png"
class_map_path= "WARWICK/WARWICK/Test/
    label_" + str(label) + ".png"

image = misc.imread(image_path)
class_map = misc.imread(class_map_path)
test_images.append(image)
test_class_maps.append(class_map)

# create list of image objects
NUM_TRN_IMAGES = 85

train_images = []
train_class_maps = []

for label in np.arange(1,NUM_TRN_IMAGES
    +1):
if label<10:
image_path= "WARWICK/WARWICK/Train/image_
    " + str(0)+str(label) + ".png"
class_map_path= "WARWICK/WARWICK/Train/
    label_" + str(0)+str(label) + ".png"
else:
image_path= "WARWICK/WARWICK/Train/image_
```

```
    " + str(label) + ".png"
class_map_path= "WARWICK/WARWICK/Train/
    label_" + str(label) + ".png"

image = misc.imread(image_path)
class_map = misc.imread(class_map_path)
train_images.append(image)
train_class_maps.append(class_map)
train_images.append(np.fliplr(image))
train_class_maps.append(np.fliplr(
    class_map))

X_train= np.array(train_images).reshape
    (85*2,128,128,3)
Y_train= np.array(train_class_maps).
    reshape(85*2,128,128,1)
X_test= np.array(test_images).reshape
    (60,128,128,3)
Y_test= np.array(test_class_maps).reshape
    (60,128,128,1)

return X_train, Y_train, X_test, Y_test
```