Retinal_OCT_Scan_Classification_Beating_the_MedMNIST_Benchmark-Project-Final

August 13, 2021

1 Retinal OCT Scan Classification - Beating the MedMNIST Benchmark

Project Team: Laurin Ganser, Radu Diaconescu Emails: laurin.ganser@gmail.com, radu.diaconescu@mars.uni-freiburg.de Course Coordinator: Dr. Ines Dedovic University College Freiburg

2 About the Project

The MedMNIST dataset was published in 2020 to serve as a benchmark for neural network-powered software that works with medical imaging recognition and classification. It includes 10 diverse subsets, including tissue treatments, skin images and CT scans of various parts of the body. One of the largest subsets is OCTMNIST, a dataset of roughly 100,000 optical computer tomography (OCT) scans of human retinas; 25% are healthy retinas, and the other 75% are evenly divided between three pathologies: choroidal neovascularization, diabetic macular edema, drusen. In our project, we built neural networks that are able to recognize and classify such OCT scans with a satisfactory degree of accuracy. While the reference/benchmark classification accuracies in the publication were obtained using several large commercially-available high-performing neural nets, we show that it is possible to build far smaller neural nets that are able to outperform the MedMNIST benchmark.

Structure 1: Getting Set Up
2: Data Prepocessing
3: Data Visualisation
4: Data Augmentation
5: Model Architectures
5.1: Small
5.2: Medium
5.3: Large
5.4: The Winner
6: Results
6.1: MedMNIST Benchmark
6.2: List of Results
7: Discussion

3 Getting Set Up

[2]:
```
#Here we import the necessary libraries
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
```

[3]:
```
#This line of code gets the data we need from the zenodo page
path = tf.keras.utils.get_file('oct_retinal.npz', origin='https://zenodo.org/
↪record/4269852/files/octmnist.npz?download=1')
```

[4]:
```
#This piece of code separates the image and label arrays into the required␣
↪subsets according to the preexisting dictionary of the dataset
with np.load(path) as data:
    train_images = data['train_images']
    train_labels = data['train_labels']
    val_images = data['val_images']
    val_labels = data['val_labels']
    test_images = data['test_images']
    test_labels = data['test_labels']

#Defines class names for classification
class_names = ['choroidal neovascularization', 'diabetic macular edema',␣
↪'drusen', 'normal']
```

4 Data Preprocessing

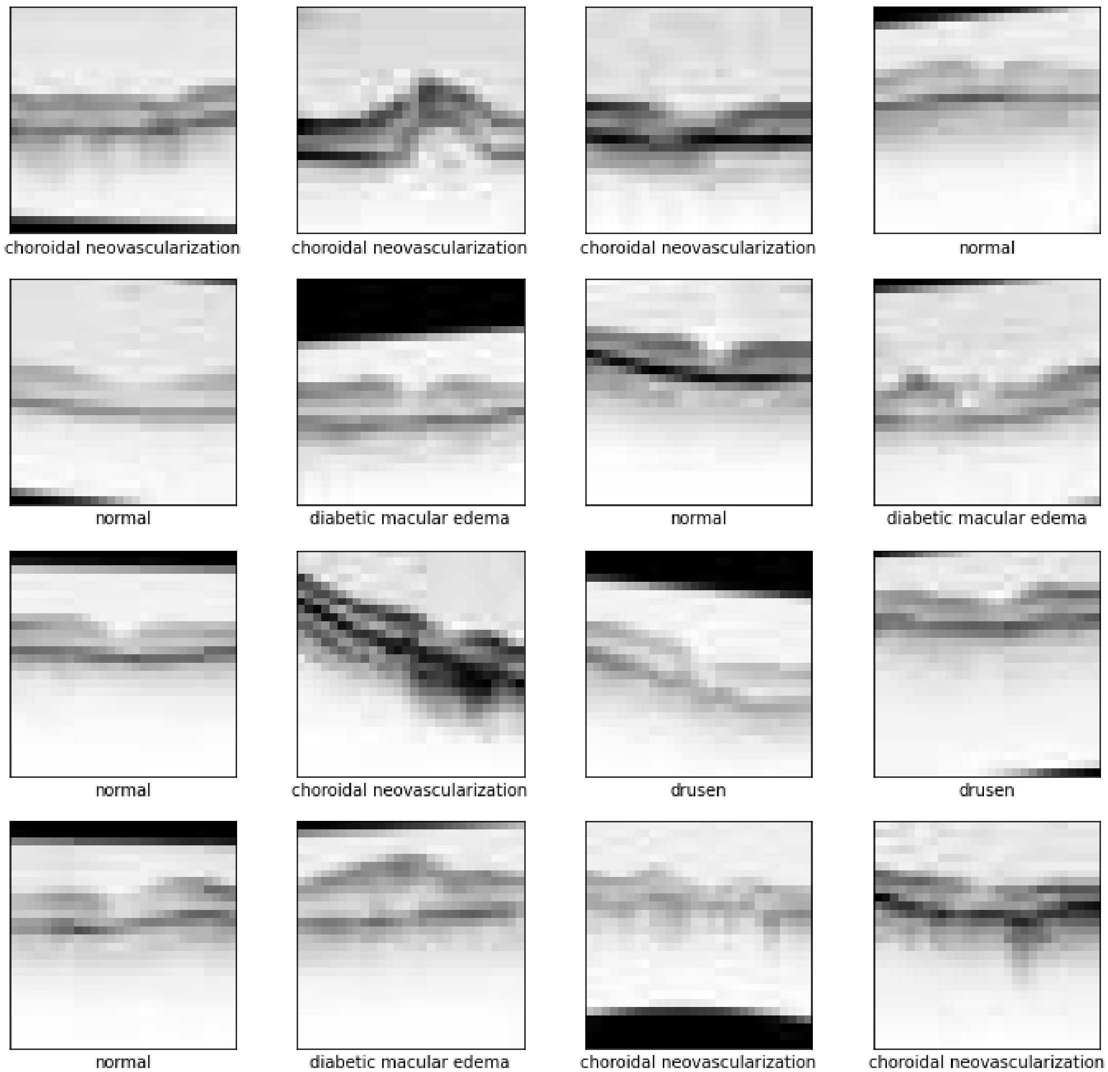
[5]:
```
#This normalises the image values to 0-1, which are the values Tensorflow␣
↪expects

train_images = train_images/255.0
val_images = val_images/255.0
test_images = test_images/255.0

#This adds an extra dimension to the images; this is necessary  because␣
↪convolutional layers
```

```
#expect a minimum number of 3 dimensions for an image (the third one being the␣
↪color channels)

train_images = np.expand_dims(train_images, -1)
val_images = np.expand_dims(val_images, -1)
test_images = np.expand_dims(test_images, -1)
```

[6]:
```
#Convert the label arrays from the original format into a simple  numpy array␣
↪of values
train_labels = [train_labels[x][0] for x in range(0, len(train_labels))]
val_labels = [val_labels[x][0] for x in range(0, len(val_labels))]
test_labels = [test_labels[x][0] for x in range(0, len(test_labels))]

train_labels = np.array(train_labels)
val_labels=np.array(val_labels)
test_labels=np.array(test_labels)
```

5 Data Visualisation

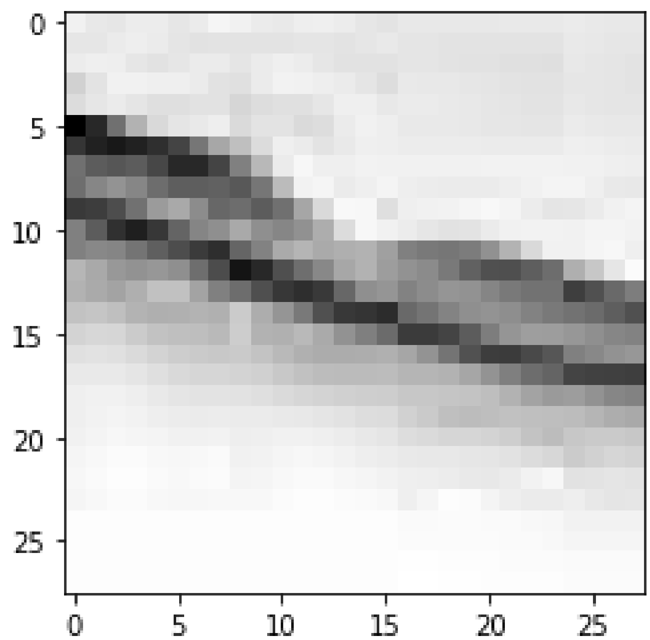
[7]:
```
#Visualize n random images from the dataset

HOW_MANY_VISUALIZE = 16

img_list = np.random.randint(97477, size=(HOW_MANY_VISUALIZE)) #takes n amount␣
↪of random numbers from a pool
                                                               #that has the␣
↪size of the dataset

fig = plt.figure(figsize=(12, 3*HOW_MANY_VISUALIZE)) #roughly scales the images␣
↪according to n

#plots the selected images and their labels
for x in range(0, len(img_list)):
    #ax[x] = fig.addsubplot(gs[x//4], gs[x%4])
    plt.subplot(HOW_MANY_VISUALIZE, 4, x+1) #specifies place in multiplot
    plt.imshow(train_images[img_list[x]], cmap=plt.cm.binary) #gets and plots␣
↪image
    plt.xlabel(class_names[train_labels[img_list[x]]]) #gets and plots label
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

plt.show()
```



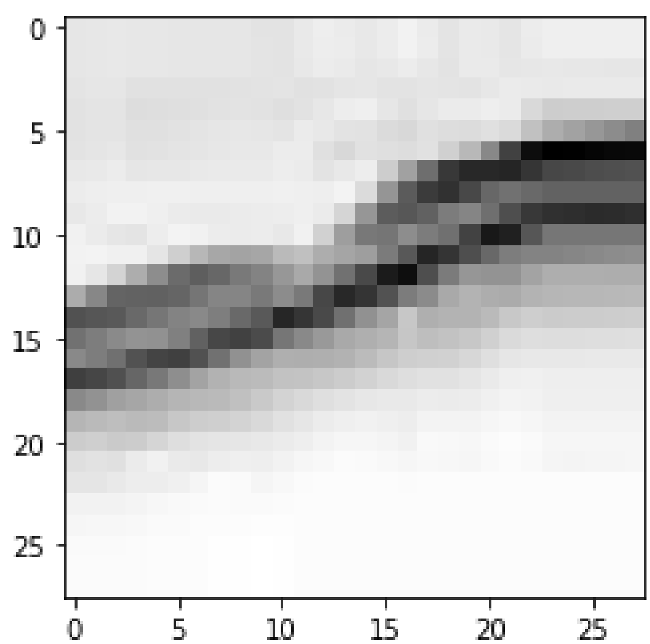
6 Data Augmentation

[8]:
```
#DATA AUGMENTATION DEMONSTRATION
#Later, we will be using data augmentation on the fly during training. This part
#is a demo to visualize how changes in parameters produce new images.
#
#As a rule of thumb, for this particular dataset it's a bad idea to do large rotations,
#as this adds more noise than it does useful training data.
```

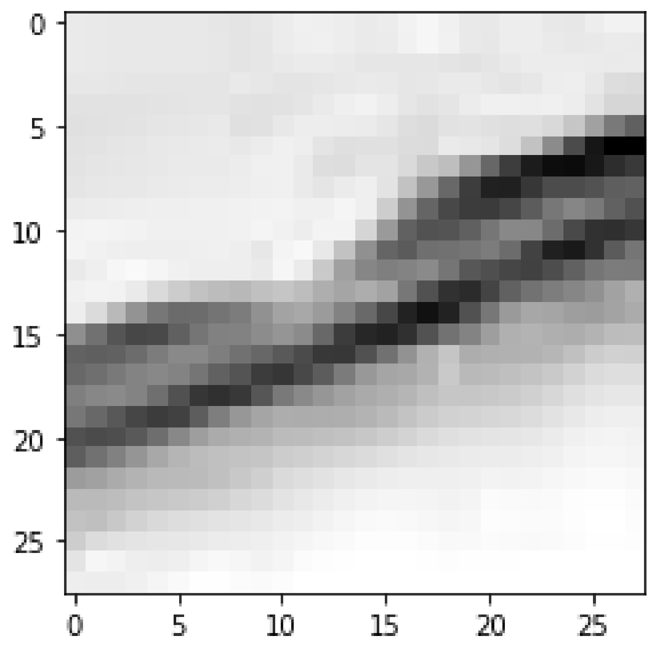
```python
#This creates a data generator object that transforms images through rotation,␣
↪shifting height/width, shearing, zooming,
#or horizontal flipping
datagen = ImageDataGenerator(
rotation_range=10,
width_shift_range=0.1,
height_shift_range=0.1,
shear_range=0.1,
zoom_range=0.2,
horizontal_flip=True,
fill_mode='nearest')

#Display examples of transformations:

# pick an image to transform
img_id = 220
img = train_images[img_id]
plt.figure(0)
plt.imshow(img, cmap=plt.cm.binary) #display original image first
plt.xlabel("original image")
img = img.reshape((1,) + img.shape)
i = 1

#Do transformations on the image and show new generated images
for batch in datagen.flow(img, save_prefix='test', save_format='jpeg'):
    plt.figure(i)
    plot = plt.imshow(image.img_to_array(batch[0]), cmap=plt.cm.binary)␣
↪#display "augmented" data below original image
    plt.xlabel("generated image")
    i += 1
    if i > 3:  # show 3 images
        break

plt.show()
```



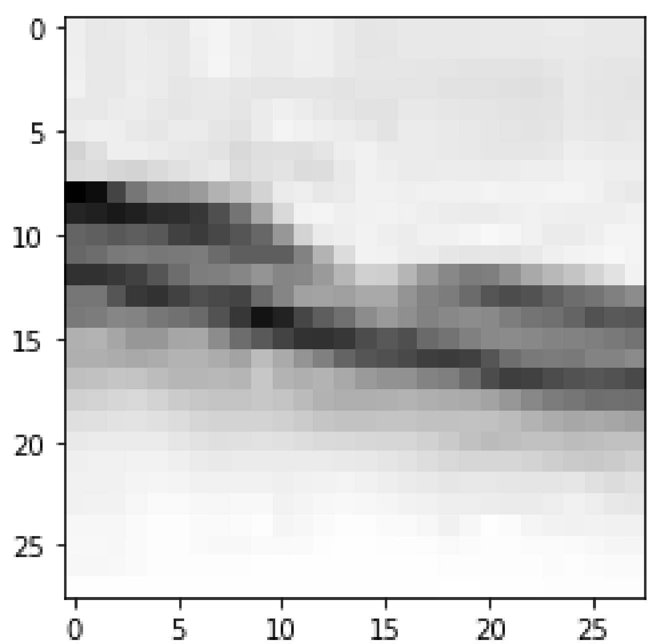
original image



generated image



generated image



generated image

7 Model architectures

8 Model architectures; Small

The model below is adapted from the simplest model we used for classification tasks. We first created this to solve the Fashion MNIST task as one of our first experiments with deep learning. None of its iterations have been able to beat the benchmark or even come close to it. Networks with this architecture tend to hover around the mid 60%. We have tested out several variations of this network and the results can be found under the header Results .

This model has the advantage of being very fast to train, with about 5 minutes per training session it is quite easy to experiment with data augmentation, epochs, and optimisers on this network.

[9]:
```
#This uses data augmentation in order to increase the amount of images it can␣
↪work with
datagen = ImageDataGenerator(
rotation_range=3,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.001,
zoom_range=0.1,
horizontal_flip=False,
fill_mode='nearest')

#Create the model with two convolutional layers (32 3x3 filters each), and two␣
↪dense layers with 64 and respectively 4 neurons
model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape = ( 28, 28,␣
↪1) ))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(32, (3,3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(4))

#Compile the model
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])

#Train the model
model.fit(datagen.flow(train_images, train_labels), epochs=6)
```

```
Epoch 1/6
3047/3047 [==============================] - 42s 14ms/step - loss: 0.7720 -
accuracy: 0.7209
Epoch 2/6
```

```
3047/3047 [==============================] - 40s 13ms/step - loss: 0.5939 -
accuracy: 0.7931
Epoch 3/6
3047/3047 [==============================] - 39s 13ms/step - loss: 0.5397 -
accuracy: 0.8123
Epoch 4/6
3047/3047 [==============================] - 40s 13ms/step - loss: 0.5086 -
accuracy: 0.8227
Epoch 5/6
3047/3047 [==============================] - 39s 13ms/step - loss: 0.4825 -
accuracy: 0.8322
Epoch 6/6
3047/3047 [==============================] - 40s 13ms/step - loss: 0.4651 -
accuracy: 0.8385
```

[9]:

```
<tensorflow.python.keras.callbacks.History at 0x7f2ed00faa00>
```

[10]:

```
model.summary ( )
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 32)        9248
_________________________________________________________________
flatten (Flatten)            (None, 3872)              0
_________________________________________________________________
dense (Dense)                (None, 64)                247872
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 260
=================================================================
Total params: 257,700
Trainable params: 257,700
Non-trainable params: 0
_________________________________________________________________
```

[11]:

```
#Model evaluation
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=1)
print("Test accuracy:")
print(test_acc)
```

```
32/32 [==============================] - 0s 3ms/step - loss: 0.7486 - accuracy:
0.6960
Test accuracy:
```

```
0.6959999799728394
```

[12]:
```
#Export the trained model for future use
model.save ( ' Project_Model_Basic.h5')
```

9 Model architectures; Medium

Below is the set-up (neural network architecture + data augmentation) that we beat the benchmark with. It has the performance advantage over the fashion model, but takes significantly more time to train (>1hour on an average mass-market CPU).

Later on in the results section, this model type will be referred to as complex in contrast to the basic fashion model.

[ ]:
```
#Create the ImageDataGenerator object responsible for data augmentation
datagen = ImageDataGenerator(
rotation_range=3,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.001,
zoom_range=0.1,
horizontal_flip=False,
fill_mode='nearest')

#Create the neural net with 3 convolutional layers and three dense layers for␣
↪classification
model2 = models.Sequential()
model2.add(layers.Conv2D(128, (3,3), activation='relu', input_shape = ( 28, 28,␣
↪1 ) ))
model2.add(layers.MaxPooling2D((2,2)))
model2.add(layers.Conv2D(512, (3,3), activation='relu'))
model2.add(layers.MaxPooling2D((2,2)))
model2.add(layers.Conv2D(1024, (3,3), activation='relu'))
model2.add(layers.Flatten())
model2.add(layers.Dense(256, activation='relu'))
model2.add(layers.Dense(128, activation='relu'))
model2.add(layers.Dense(4))

#Compile the model
model2.compile(optimizer='adam',
              loss=tf.keras.losses.
↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

#Set training parameters
BATCH_SIZE = 32
```

```
EPOCHS = 5

#Train the model; it does not receive the dataset directly, but rather receives␣
↪through the
#dataset.flow function a slightly modified dataset at each epoch because each␣
↪individual image
#is randomly slightly modified, thereby artificially creating a little more␣
↪variation and
#avoiding overfitting. Data augmentation is only done for the training set. The␣
↪validation data
#is fed into the model as is.
#
#The model.fit command does a lot of work here: it defines the number of times␣
↪the model
#goes over the entire dataset to train; it shuffles the dataset at each epoch␣
↪and
#groups it into batches to avoid overfitting; each image in a batch is slightly␣
↪modified
#randomly using the datagen object (data augmentation), right before it is used
#for training. Therefore, while the model might see the same image 5 times if␣
↪it is
#trained for 5 epochs, it will see the image slightly differently each time
#(e.g. a little zoomed in, a little tilted etc.).
model2.fit(datagen.flow(train_images, train_labels, batch_size=BATCH_SIZE),
          validation_data=(val_images, val_labels), shuffle=True, verbose=1,
          epochs=EPOCHS, steps_per_epoch=len(train_images) // BATCH_SIZE )
```

```
Epoch 1/5
3046/3046 [==============================] - 1052s 345ms/step - loss: 0.8205 -
accuracy: 0.6944 - val_loss: 0.6273 - val_accuracy: 0.7755
Epoch 2/5
3046/3046 [==============================] - 1031s 338ms/step - loss: 0.6283 -
accuracy: 0.7793 - val_loss: 0.5634 - val_accuracy: 0.8080
Epoch 3/5
3046/3046 [==============================] - 1044s 343ms/step - loss: 0.5524 -
accuracy: 0.8078 - val_loss: 0.5009 - val_accuracy: 0.8285
Epoch 4/5
3046/3046 [==============================] - 1055s 346ms/step - loss: 0.5168 -
accuracy: 0.8194 - val_loss: 0.5170 - val_accuracy: 0.8320
Epoch 5/5
3046/3046 [==============================] - 1056s 347ms/step - loss: 0.4920 -
accuracy: 0.8278 - val_loss: 0.5460 - val_accuracy: 0.8177
```

[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x1e7545217f0>
```

[ ]:
```
model2.summary ( )
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 26, 26, 128)       1280
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 13, 13, 128)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 11, 11, 512)       590336
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 5, 5, 512)         0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 3, 3, 1024)        4719616
_________________________________________________________________
flatten_4 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_9 (Dense)              (None, 256)               2359552
_________________________________________________________________
dense_10 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_11 (Dense)             (None, 4)                 516
=================================================================
Total params: 7,704,196
Trainable params: 7,704,196
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
#Evaluate the model
test_loss, test_acc = model2.evaluate(test_images, test_labels, verbose=1)
print("Test accuracy:")
print(test_acc)
```

```
32/32 [==============================] - 1s 34ms/step - loss: 0.7658 - accuracy:
0.6910
Test accuracy:
0.6909999847412109
```

[ ]:
```
#Save the model
model2.save ( ' Project_Model_Middle.h5')
```

10 Model architectures; Large

This last model architecture we experimented only briefly with because it was too bulky. It seemed to overfit much more quickly than the previous model type, so we decided against it.

[ ]:
```
# creates a data generator object that transforms images
datagen = ImageDataGenerator(
rotation_range=3,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.001,
zoom_range=0.1,
horizontal_flip=False,
fill_mode='nearest')

model3 = models.Sequential()

model3.add(layers.Conv2D(128, (3,3), activation='relu', input_shape = ( 28, 28,␣
↪1)))
model3.add(layers.MaxPooling2D((2,2)))
model3.add(layers.Conv2D(512, (3,3), activation='relu'))
model3.add(layers.MaxPooling2D((2,2)))
model3.add(layers.Conv2D(1024, (3,3), activation='relu'))
model3.add(layers.Flatten())
model3.add(layers.Dense(256, activation='relu'))
model3.add(layers.Dense(128, activation='relu'))
model3.add(layers.Dense(4))

model3.compile(optimizer='adam',
              loss=tf.keras.losses.
↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

BATCH_SIZE = 32
EPOCHS = 5

#Train the model; it does not receive the dataset directly, but rather receives␣
↪through the
#dataset.flow function a slightly modified dataset at each epoch because each␣
↪individual image
#is randomly slightly modified, thereby artificially creating a little more␣
↪variation and
#avoiding overfitting. Data augmentation is only done for the training set. The␣
↪validation data
#is fed into the model as is.
#
#The model.fit command does a lot of work here: it defines the number of times␣
↪the model
#goes over the entire dataset to train; it shuffles the dataset at each epoch␣
↪and
```

```
#groups it into batches to avoid overfitting; each image in a batch is slightly␣
↪modified
#randomly using the datagen object (data augmentation), right before it is used
#for training. Therefore, while the model might see the same image 5 times if␣
↪it is
#trained for 5 epochs, it will see the image slightly differently each time
#(e.g. a little zoomed in, a little tilted etc.).
model3.fit(datagen.flow(train_images, train_labels, batch_size=BATCH_SIZE),
          validation_data=(val_images, val_labels), shuffle=True, verbose=1,
          epochs=EPOCHS, steps_per_epoch=len(train_images) // BATCH_SIZE )
```

```
Epoch 1/5
3046/3046 [==============================] - 1065s 350ms/step - loss: 0.7468 -
accuracy: 0.7283 - val_loss: 0.5426 - val_accuracy: 0.8179
Epoch 2/5
3046/3046 [==============================] - 1062s 349ms/step - loss: 0.5274 -
accuracy: 0.8181 - val_loss: 0.4544 - val_accuracy: 0.8458
Epoch 3/5
3046/3046 [==============================] - 1074s 353ms/step - loss: 0.4745 -
accuracy: 0.8355 - val_loss: 0.4865 - val_accuracy: 0.8323
Epoch 4/5
3046/3046 [==============================] - 1073s 352ms/step - loss: 0.4447 -
accuracy: 0.8449 - val_loss: 0.3740 - val_accuracy: 0.8670
Epoch 5/5
3046/3046 [==============================] - 1065s 350ms/step - loss: 0.4239 -
accuracy: 0.8510 - val_loss: 0.4584 - val_accuracy: 0.8444
```

[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x1e7010c5a30>
```

[ ]:
```
model3.summary ( )
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 26, 26, 128)       1280
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 128)       0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 11, 11, 512)       590336
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 512)         0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 3, 3, 1024)        4719616
_________________________________________________________________
flatten_5 (Flatten)          (None, 9216)              0
_________________________________________________________________
```

```
dense_12 (Dense)             (None, 256)              2359552
_________________________________________________________________
dense_13 (Dense)             (None, 128)              32896
_________________________________________________________________
dense_14 (Dense)             (None, 4)                516
=================================================================
Total params: 7,704,196
Trainable params: 7,704,196
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
test_loss, test_acc = model3.evaluate(test_images, test_labels, verbose=1)
print("Test accuracy:")
print(test_acc)
```

```
32/32 [==============================] - 1s 34ms/step - loss: 0.8046 - accuracy:
0.6980
Test accuracy:
0.6980000138282776
```

[ ]:
```
model3.save ( ' Project_Model_Large.h5')
```

11 The Winner

This is the model that we achieved our highest result with. The key difference from the above code is the use of the nadam optimizer.

[ ]:
```
# This is the original code we used.
datagen = ImageDataGenerator(
rotation_range=3,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.001,
zoom_range=0.1,
horizontal_flip=False,
fill_mode='nearest')

modelNA5 = models.Sequential()

modelNA5.add(layers.Conv2D(128, (3,3), activation='relu', input_shape = ( 28,␣
↪28, 1 ) ))
modelNA5.add(layers.MaxPooling2D((2,2)))
modelNA5.add(layers.Conv2D(512, (3,3), activation='relu'))
modelNA5.add(layers.MaxPooling2D((2,2)))
modelNA5.add(layers.Conv2D(1024, (3,3), activation='relu'))
modelNA5.add(layers.Flatten())
```

```
modelNA5.add(layers.Dense(256, activation='relu'))
modelNA5.add(layers.Dense(128, activation='relu'))
modelNA5.add(layers.Dense(4))

modelNA5.compile(optimizer='nadam',
              loss=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

BATCH_SIZE = 32
EPOCHS = 5

#Train the model; it does not receive the dataset directly, but rather receives␣
 ↪through the
#dataset.flow function a slightly modified dataset at each epoch because each␣
 ↪individual image
#is randomly slightly modified, thereby artificially creating a little more␣
 ↪variation and
#avoiding overfitting. Data augmentation is only done for the training set. The␣
 ↪validation data
#is fed into the model as is.

#The model.fit command does a lot of work here: it defines the number of times␣
 ↪the model
#goes over the entire dataset to train; it shuffles the dataset at each epoch␣
 ↪and
#groups it into batches to avoid overfitting; each image in a batch is slightly␣
 ↪modified
#randomly using the datagen object (data augmentation), right before it is used
#for training. Therefore, while the model might see the same image 5 times if␣
 ↪it is
#trained for 5 epochs, it will see the image slightly differently each time
#(e.g. a little zoomed in, a little tilted etc.).
modelNA5.fit(datagen.flow(train_images, train_labels, batch_size=BATCH_SIZE),
          validation_data=(val_images, val_labels), shuffle=True, verbose=1,
          epochs=EPOCHS, steps_per_epoch=len(train_images) // BATCH_SIZE )
```

The model is found under the name Best_model.

12 Results

[30]:
```
#This code block serves to demonstrate the accuracy of our models. We provide␣
 ↪two
#models we have already trained in .h5 format.
```

```
saved_models = [ "Nadam5(1).h5", "OCT-76-1-WITH-INPUT-SHAPE.h5" ]

model_path = saved_models[0] #enter value 0 for the Nadam model (acc 76.6%) or␣
↪value 1 for the Adam model (acc 76.1%)

model = tf.keras.models.load_model(model_path)

#This piece of code evaluates the model on the test data
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=1)
print("Test accuracy:")
print(test_acc)
```

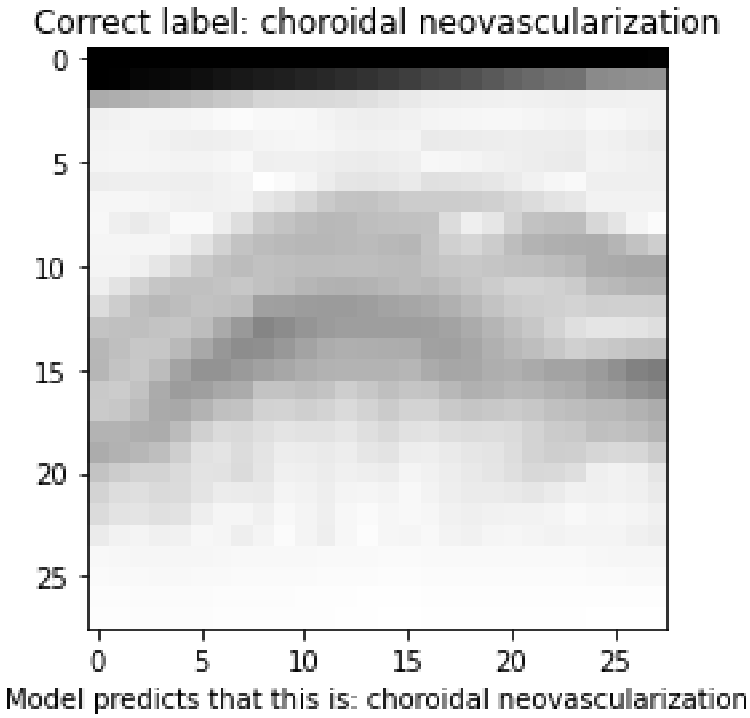
```
32/32 [==============================] - 1s 30ms/step - loss: 0.5922 - accuracy:
0.7660
Test accuracy:
0.765999972820282
```

[28]:

```
#This shows the models performance on a given image

img_index = 923 #pick an image from the test set (1000 images)

prediction = model.predict(np.array([test_images[img_index]]))
predicted_class = class_names[np.argmax(prediction)] #takes the highest␣
↪activation value of the network for that image and \
#chooses that as the predictied class

plt.imshow(test_images[img_index], cmap=plt.cm.binary) #plots the image
plt.title("Correct label: " + class_names[test_labels[img_index]]) #takes the␣
↪label as specified and titles accordingly
plt.xlabel("Model predicts that this is: " + predicted_class) #takes the label␣
↪that the model precits for that image
plt.grid(False)
plt.show
```

[28]:

```
<function matplotlib.pyplot.show(close=None, block=None)>
```



13 MedMNIST Benchmark

The source for the following table is the MedMNIST website (https://medmnist.com/). The values are the result of training neural networks with large pretrained convolutional bases on the same image dataset we used.

[33]:
```
img=mpimg.imread('InkedInkedBenchmark.jpg')
plt.figure ( figsize = ( 20, 20 ) )
imgplot = plt.imshow(img)
plt.axis ( False )
plt.title ( ' Benchmark Accuracies:')
```

[33]:
```
Text(0.5, 1.0, ' Benchmark Accuracies:')
```

Benchmark Accuracies:

Benchmarking

Methods	PathMNIST		ChestMNIST		DermaMNIST		OCTMNIST		PneumoniaMNIST	
	AUC	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC	ACC
ResNet-18 (28)	0.972	0.844	0.706	0.947	0.899	0.721	0.951	**0.758**	0.957	0.843
ResNet-18 (224)	0.978	0.860	0.713	**0.948**	0.896	0.727	0.960	0.752	0.970	0.861
ResNet-50 (28)	0.979	**0.864**	0.692	0.947	0.886	0.710	0.939	0.745	0.949	0.857
ResNet-50 (224)	0.978	0.848	0.706	0.947	0.895	0.719	0.951	0.750	0.968	0.896
auto-sklearn	0.500	0.186	0.647	0.642	0.906	0.734	0.883	0.595	0.947	0.865
AutoKeras	0.979	**0.864**	0.715	0.939	0.921	0.756	0.956	0.736	0.970	0.918
Google AutoML Vision	**0.982**	0.811	**0.718**	0.947	**0.925**	**0.766**	**0.965**	0.732	**0.993**	**0.941**

As can be seen, the networks we provide beat the benchmark set at 75.8% by both achieving accuracy scores of >76%.

14 List of Results

This list is supposed to serve as an overview over all the models that we have trained. We chose to provide only two models in .h5 format - those that beat the benchmark - because of their large filesize.

The conditions are: 1. Optimiser (Adam or Nadam) 2. n[Epochs] (for the Small model 5/10 and for the Medium model 5/7) 3. Type of network (Small model, Medium model) 4. Data Augmentation (True or False)

Conditions: Adam, Epoch = 5, Small, Data Aug. = True
Accuracy: 61.6

Conditions: Nadam, Epoch = 5, Small, Data Aug. = True
Accuracy: 68.2%

Conditions: Adam, Epoch = 10, Small, Data Aug. = True
Accuracy: 66.4%

Conditions: Nadam, Epoch = 10, Small, Data Aug. = True
Accuracy: 67.7%

Conditions: Nadam, Epoch = 5, Small, Data Aug. = False
Accuracy: 62.7%

Conditions: Nadam, Epoch = 10, Small, Data Aug. = False
Accuracy: 68.3%

Conditions: Nadam, Epoch = 5, Medium, Data Aug. = True
Accuracy: 76.6%

Conditions: Adam, Epoch = 7, Medium, Data Aug. = True
Accuracy: 74.7%

Conditions: Nadam, Epoch = 7, Medium, Data Aug. = True
Accuracy: 73.5%

Conditions: Nadam, Epoch = 7, Medium, Data Aug. = False
Accuracy: 72.4%

Conditions: Nadam, Epoch = 5, Medium, Data Aug. = False
Accuracy: 69%

Conditions: Adam, Epoch = 5, Medium, Data Aug. = True
Accuracy: 76.1%

15 Discussion

We have shown that it is possible to build deep convolutional neural networks with a low number of layers (e.g. 3 convolutional and 3 dense layers) that outperform sophisticated available pretrained networks such as ResNet-50 (which has 50 convolutional layers). It is true that the nets benchmarked in the MedMNIST paper are general-purpose, while the ones we built are highly specific to our application. But these results raise two interesting questions for both medical imaging / diagnostic AI, and more generally for using neural networks in image analysis. The first is whether it is worthwhile to build large, general-purpose neural nets, if a specialized tool might do a better job and may be comparatively easy to build. The second question is to what degree a benchmark such as MedMNIST is useful, given that, from our results, it seems reasonable to propose that the best results, or cap on achievable accuracy in the results given by different architectures, may have less to do with the networks themselves and more to do with the limited information that is implicit in using 28x28 images. That may be the case in particular where the images are grayscale, such as in the OCT dataset. Here, the usefulness of data augmentation techniques is also highlighted. In all, our results display an interesting example of "less is more", where it is possible, with low computing resources, to build a useful deep learning program for medical image analysis.

References

Kermany, D. S., Goldbaum, M., Cai, W., Valentim, C. C., Liang, H., Baxter, S. L., … & Zhang, K. (2018). Identifying medical diagnoses and treatable diseases by image-based deep learning. Cell, 172(5), 1122-1131.

MedMNIST Classification Decathlon: A Lightweight AutoML Benchmark for Medical Image Analysis. MedMNIST. (n.d.). https://medmnist.com/.

Nielsen, M. A. (2019, January 1). Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/.

Ruscica, T. (2020). TensorFlow 2.0 Complete Course - Python Neural Networks for Beginners Tutorial. YouTube. https://www.youtube.com/watch?v=tPYj3fFJGjk&t=16741s.

Tensorflow. TensorFlow. (n.d.). https://www.tensorflow.org/.

Yang, J., Shi, R., & Ni, B. (2021, April). Medmnist classification decathlon: A lightweight automl benchmark for medical image analysis. In 2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI) (pp. 191-195). IEEE.

Annex: Mishaps

Because we initially worked with models that do not contain the input_shape specifier in the first convolutional layer, counting on tensorflow to sort that out, we discovered we would be unable to load them from .h5 files due to the format that the Keras load function expects. We had to manually reload the models into ones that do have an input_shape in order to be able to work with them. To do that, we created an empty model with the exact same network topology, but where the first layer has input dimensions (28,28,1) and then loaded the weights of the old model into the newly generated one.

The code below serves to demonstrate how we rectified this issue. We are including it because it was an interesting instance of troubleshooting.

[52]:
```
#This model is an exact replicate of a model we have create and is only used to␣
↪load the weigths of a pre-trained model into
Placeholder_1_model = models.Sequential()

Placeholder_1_model.add(layers.Conv2D(128, (3,3), activation='relu',␣
↪input_shape = ( 28, 28, 1 ) )) #Contains the crucial argument for being able␣
↪to load the models
Placeholder_1_model.add(layers.MaxPooling2D((2,2)))
Placeholder_1_model.add(layers.Conv2D(512, (3,3), activation='relu'))
Placeholder_1_model.add(layers.MaxPooling2D((2,2)))
Placeholder_1_model.add(layers.Conv2D(1024, (3,3), activation='relu'))
Placeholder_1_model.add(layers.Flatten())
Placeholder_1_model.add(layers.Dense(256, activation='relu'))
Placeholder_1_model.add(layers.Dense(128, activation='relu'))
Placeholder_1_model.add(layers.Dense(4))

Placeholder_1_model.load_weights("Nadam5.h5") #Loads the pre-trained model

Placeholder_1_model.compile(optimizer='nadam',
              loss=tf.keras.losses.
↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

[53]:
```
#Tests the accuracy of the model to see if the loading worked properly
#Benchmark_model_1 = tf.keras.models.load_model("Mishap_model.h5")
test_loss, test_acc = Placeholder_1_model.evaluate(test_images, test_labels,␣
↪verbose=1)
print("Test accuracy:")
print(test_acc)
```

```
32/32 [==============================] - 1s 34ms/step - loss: 0.5922 - accuracy:
```

```
0.7660
Test accuracy:
0.765999972820282
```

[54]:
```
#This model is an exact replicate of a model we have create and is only used to␣
↪load the weigths of a pre-trained model into
Placeholder_2_model = models.Sequential()

Placeholder_2_model.add(layers.Conv2D(128, (3,3), activation='relu',␣
↪input_shape = ( 28, 28, 1 ) )) #Contains the crucial argument for being able␣
↪to load the models
Placeholder_2_model.add(layers.MaxPooling2D((2,2)))
Placeholder_2_model.add(layers.Conv2D(512, (3,3), activation='relu'))
Placeholder_2_model.add(layers.MaxPooling2D((2,2)))
Placeholder_2_model.add(layers.Conv2D(1024, (3,3), activation='relu'))
Placeholder_2_model.add(layers.Flatten())
Placeholder_2_model.add(layers.Dense(256, activation='relu'))
Placeholder_2_model.add(layers.Dense(128, activation='relu'))
Placeholder_2_model.add(layers.Dense(4))

Placeholder_2_model.load_weights("2Best_model.h5") #Loads the pre-trained model

Placeholder_2_model.compile(optimizer='adam',
              loss=tf.keras.losses.
↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

[55]:
```
test_loss, test_acc = Placeholder_2_model.evaluate(test_images, test_labels,␣
↪verbose=1)
print("Test accuracy:")
print(test_acc)
```

```
32/32 [==============================] - 1s 34ms/step - loss: 0.5723 - accuracy:
0.7610
Test accuracy:
0.7609999775886536
```