Pneumonia Project

DM873 - Deep Learning

Tolnai, Balázs András
batol20@sdu.student.dk

Goldapp, Christian Tim Michael
chgol20@sdu.student.dk

Ma, Qianhua
qima18@sdu.student.dk

Henriksen, Rasmus Aasø
rhenr17@sdu.student.dk

Bach, Camilla Marie
cabac19@sdu.student.dk

18. December 2020

1 Layers

The Dense Layer
The Dense layer implementation is quite simple. The layer inherits the layer class and takes as arguments units and an activation function. The unit parameter sets the amount of biases and weights created, as every unit is supposed to have one of each. The activation function is set as relu as default, since this is a good all-round activation function and commonly used. The activation function can be altered if needed.
The weights are initialized with a random normal function to ensure that the weights have different values and do not start out in a symmetrical configuration which might collapse the mapping space. The biases are initialized with zeros to avoid adding further complexity and interfering with the weights. The layer outputs the dot-product of the input and the weights with the bias added. If an activation function is given as an argument, then this is applied to the results before it is outputted.

The Convolutional Layer
The convolutional layer takes as arguments filters, kernel_size, strides, padding, activation and dilation_rate. The filters determines how many kernels is applied in each layer and is 32 by default. This value is important for the user to tune as a hyperparameter. The kernel_size determines the size of each kernel to run over the input and is 3 by 3 as default. The parameter strides is initialized to 1 by 1, and does the convolution operation without skipping pixels. The user can set activation function to be applied to the results after convolution and before output. The parameter padding is set to 'valid', but the user can specify if they want another type of padding at the edges of the image.. The dilation_rate is set to one to not do any dilution.

The MaxPooling Layer
The MaxPooling layer takes as arguments the pool size, the strides, and the padding mode. The pool size defaults to $(2, 2)$ and if no strides are given, the strides are set to the pool size, resulting in non-overlapping pools. The padding mode is by default "valid", which is fine for the default case that the pool size is significantly smaller than the input. MaxPooling implements a 2-Dimensional pooling operation using the backend primitive "max_pool2d". MaxPooling divides the 2D input data into pools of a given size and moving window distance and takes the largest value of every pool as the output. Intuitively, this can be understood as selecting the "most important" feature of a small area as a way to downsample data. MaxPooling does not have trainable weights; instead it is used for extracting features and downsampling them. The pool size acts like a tweakable meta-variable.

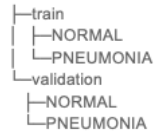
2 Model

The Net
Our model is a generally quite simple model. It consists of convolution layers, followed by maxpooling layers. The filter sizes are exponentially increasing, so first we can detect larger features (like edges), and as the sizes of the feature maps get smaller, more filters can detect smaller, more detailed features. We use Batch normalization, to speed up the training time, and, perhaps reduce overfitting. We also utilised spatial dropout. Opposed to normal dropout, which is not very effective in convolutional nets, rather than dropping out individual elements, it drops out entire feature maps. This helps promote independence between feature maps. As usual, the net's head is a fairly large dense layer.

Training
The training takes two iterations. First we trained the model on the training data, using validation. wet set the number of epochs to 1000, and an early stopping callback got created, to stop the training at the right moment, with a certain patience. After this, the program reads the number of epochs from history and retrains the model on the whole dataset, without validation, hopefully with better results.

3 Evaluation

Due to a misunderstanding, the custom generator for the evaluation-part, started out as a custom generator for the training and validation data. Since a lot of work was put into this, it will be briefly described as well. The directories was split into training and validation data by creating a new database, that contained two subsets; A training set with 1072 samples of each class and a validation set with 269 samples of each class. The structure is as follows:



Before beginning the datagenerator for the training data, there was made a label and partition dictionary with the paths for the images. The images' labels was encoded as 0 for normal and 1 for pneumonia. The datagenerator has a function, that loads the images in grayscale format to change the colorchannels from 3 to 1. The datagenerator also rescales the values of every pixel to between 0 and 1, since this is often improving performance of a neural net.

The Custom Data-Generator
A data-generator is conceptually simple. It is a stream of input, through a filter or transformation, to a desired output. For this generator specifically, the goal was to have it read the matrix of values from the .txt correctly and output it in a way, that the model could interpret. The datagenerator takes the directory as an input, together with specifications of the image-size. The generator makes a list of files and creates a list of labels determined by which directory the file comes from. The generator then needed a function to convert the txt-file to a more image-like file for the model to read. The generator has the function convert, that takes a batch of files as input. It starts by creating empty arrays to store the values and labels in.. Then it goes through the batch, reads the values from the txt-file, rescales it to normalized values like in the training-data, reshapes the values and appends the values and labels to the empty arrays. It finally returns the batch of values and labels. Through the testing, it turns out, that for now, the generator only works with a batch-size of 1, but for this purpose it will do. If the generator was supposed to be for the training part, it would need a little more work.

4 Results

For the very first tries we tried to make a large, VGG-16 like network. With 16 convolution layers, from 32 filters to 512, and two Dense layers. We managed to reach incredible validation accuracy.

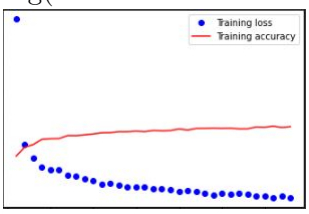
```
Epoch 18/1000
75/75 [==============================] - 35s 471ms/step - loss: 0.0739 - accuracy: 0.9745 - val_loss: 0.0811 - val_accuracy: 0.9833
Epoch 19/1000
75/75 [==============================] - 35s 464ms/step - loss: 0.0486 - accuracy: 0.9867 - val_loss: 0.3714 - val_accuracy: 0.9433
Epoch 20/1000
75/75 [==============================] - 35s 467ms/step - loss: 0.0595 - accuracy: 0.9813 - val_loss: 0.1371 - val_accuracy: 0.9433
Epoch 21/1000
75/75 [==============================] - 35s 462ms/step - loss: 0.0535 - accuracy: 0.9794 - val_loss: 0.1127 - val_accuracy: 0.9767
Epoch 22/1000
75/75 [==============================] - 35s 465ms/step - loss: 0.0513 - accuracy: 0.9838 - val_loss: 0.0674 - val_accuracy: 0.9900
```

```
val_loss: 0.0010 - val_accuracy: 1.0000
```

After evaluation, it turned out that it was probably just overfitting, as the model performed terrible. It is unsure why the validation accuracy kept increasing too, perhaps the validation data was picked unluckily and was too easy to categorise compared to the training data. Later we tried a smaller and smaller convolutional net. We tried a net with 10 layers, double convolution layers before each pooling. This was probably still learning too quickly and start overfitting.

```
Epoch 1/1000
67/67 [==============================] - 54s 764ms/step - loss: 2.7580 - accuracy: 0.5075 - val_loss: 6.7394 - val_accuracy: 0.5000
Epoch 2/1000
67/67 [==============================] - 50s 749ms/step - loss: 1.1922 - accuracy: 0.5319 - val_loss: 1.2952 - val_accuracy: 0.5000
Epoch 3/1000
67/67 [==============================] - 51s 752ms/step - loss: 1.0646 - accuracy: 0.5774 - val_loss: 1.1051 - val_accuracy: 0.5000
Epoch 4/1000
67/67 [==============================] - 50s 749ms/step - loss: 0.9455 - accuracy: 0.6335 - val_loss: 0.7183 - val_accuracy: 0.5130
Epoch 5/1000
67/67 [==============================] - 50s 742ms/step - loss: 0.9664 - accuracy: 0.6091 - val_loss: 0.7198 - val_accuracy: 0.5112
Epoch 6/1000
67/67 [==============================] - 49s 724ms/step - loss: 0.7931 - accuracy: 0.6596 - val_loss: 0.5126 - val_accuracy: 0.8346
Epoch 7/1000
67/67 [==============================] - 49s 731ms/step - loss: 0.7772 - accuracy: 0.6864 - val_loss: 0.5491 - val_accuracy: 0.6747
Epoch 8/1000
67/67 [==============================] - 49s 724ms/step - loss: 0.7199 - accuracy: 0.7202 - val_loss: 3.9789 - val_accuracy: 0.5260
Epoch 9/1000
```

In the end we ended up with a simple model with single convolutional layers (filters form 64 to 1024), which performed reasonably well. The final model's second round of training(there was no validation):



And the final evaluation:

```
51/51 [==============================] - 2s 35ms/step - loss: 0.3691 - accuracy: 0.8431
[0.36914196610450745, 0.843137264251709]
```