EECS 247
Mar 2021

UCI

Image Classification with Convolutional Neural Networks

An Example with Furry Friends

prepared by
Arya Daroui

1. Abstract

Image classification has been a topic of interest in the field of machine learning since its inception, with applications including self-driving vehicles, media filtering, and reverse image search. In this report, we will cover the basics of image classification with convolutional neural networks (CNNs), and provide an example of binary classification model for images of cats and dogs with 80% accuracy.

2. Introduction

To model organic neurological activity, neural networks are typically designed as a series of interconnected layers of nodes where connections describe a linear combination. The equation for this linear combination is wrapped by an activation function that simulates the intensity of a neuron firing, and it ultimately maps our input stimulus to our output.

$$y(\vec{x}) = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

$$\text{node}(\vec{x}) = \text{activation}(y(\vec{x}))$$

In contrast to the typical method of designing an equation to provide a solution for a problem, given a sufficiently large and well architected neural network model, the many weights (parameters) that compose the linear combination equation provide a flexible way to adjust parameters such that we can achieve our desire output. However, manually making intelligent decisions on which way to adjust weights is very difficult because each weight's effect on each other and the overall model is not easily characterizable. Additionally, the number of parameters can easily extend into the millions for even small systems, so to do it by hand is not a realistic task.

The solution is to determine a desired output, and then calculate backward (backpropagate) through each node to find what parameter values are needed. This is done by computers, so the method of showing the machine what we want, and then letting it do the rest of the problem solving for us lends itself to coming up with good models for fairly abstract problems that we would not be able to design otherwise. But to achieve this abstract usability, we need to feed significant amounts of data to the model before it starts to become accurate at the level we desire; otherwise it may only be accurate with the specific example data given to it.

2.1 Pitfalls and Hyperparameters

This high level of problem solving comes with its own set of high-level problems. The backpropagation calculation is very computationally expensive, and the cost quickly grows with more complex models and data. The calculation works by traversing a many-dimensional field, and trying to find the minimum loss between a given input and desired output. Because the algorithm traverses the field by calculating gradients that yield multiplying ratios, subequent gradients can be iteratively multiplied toward zero or blow up toward infinity in what is known as the Vanishing Gradient Problem. Additionally, given that it searches locally to minimize loss, it may get stuck in a local minima and not find a better solution even if it is very close. On the other side, the model may become *too* accurate for the given training data (overfitting), and become less accurate for the general inputs given outside the training dataset.

All of thse issues have possible solutions, but it is not always clear what answer is the most effective or correct. It is a delicate balancing act of choosiing hyperparameters to make the model work as accurately as possible without overfitting or making the cost of computation or data requirement too high. Some of these methods are stochastic in nature, and some of them work in specific, abstract cases, but not others, so it can often turn into a game of trial and error.[1]

3. Methodology

Convolutional neural networks are a common choice for machine learning projects involving images. Convolutional neural networks will contain at least one convolutional layer that is not fully connceted like a typical neural network. Instead, it has a kernel, a small square of pixels that slides over an image and operates on the center pixel beneath; this is known as 2D convolution, and it is very common in image processing algorithms. The key here is that the network trains the weights of the pixels in the kernel. Additionally, the output of this filter on its layer generates a feature map, or a small representation of the pattern the layer picks up. This very helpful in many CNN models, but we will not dive into them in this project.

Our main concern is how to choose what layers, how many, and which hyperpameters for our model. A common architecture for an image classification model is convolutional layers followed by fully connected (dense) layers, as popularized by LeNet [1] model in the 90s. I tried a few different variations of this architecture, most of them not achieving accuracy beyond random choice (50%). Although I could not verify, I attribute this to vanishing gradients. The main model I used with has three convolutional layers followed by three dense layers. Its design is shown in Figure 5 in the Appendix, along with the Python TensorFlow code. An additional model I used with one fewer convolutional and dense layer achieved siimilar accuracy. The model was trained on a Windows machine with CUDA acceleration on a GTX 1080 GPU with 8 GB VRAM, 16 GB RAM, and an i7 6700k CPU. The weights and model were saved afterward for easy reuse of the model on weaker machines. They are available upon request.

[1]A point on this heuristic nature that is not mentioned enough about neural networks and machine learning; these models are not absolute solutions. While the general, abstract output may be the same at scale, the choices made for these hyperparameters, datasets, design, still only generate a stochastic model. The utmost care should be taken in how these non-deterministic, trial-and-error choices can affect individual human lives at the abstract level and the scale they are deployed at, because it can be catastrophic, and incorrectly perceived as a blameless artifact of designing these black-box systems.

3.1 Design choices

As mentioned, tthe main problem I ran into was the accuracy not leaving random choice, even across epochs. This was solved for using `relu` for all activation functions but the output layer. There were other small issues like matching the final dense layer shape with the number of classes. Bizarrely, when the class data generator is selected as `binary`, the `binary_crossentropy` loss function makes an incorrectly shaped final dense layer for binary classification. I ended up choosing `sparse_categorical_crossentropy` instead, and which provided the added benefit of easier expandability of classes if the dataset evolves.

Which, in regard to the dataset and what we are trying to achieve, the overall goal of this model is to input an image of a cat or dog, and output which one it is. The dataset was composed of 10,000 pictures of cats and dogs obtained from Kaggle [2]. There was some preprocessing done such as flipping, rotating, and zooming images, but when changing these hyperparameters, I did not find much variation in the accuracy or overfitting. Other hyperparameters I toyed with but did not find much effect from were the final activation function and dropout rates. I adjusted the size and number of filters (kernels), and found significant differences in processing time and accuracy. I found a good balance at 32 filters and a 3x3 kernel size. The model was trained with a batch size of 8 over 16 epochs, which was around where I found the accuracy would plateau, as shown in Figure 1 below. The accompanying loss graph is shown in Figure 3 in the Appendix.

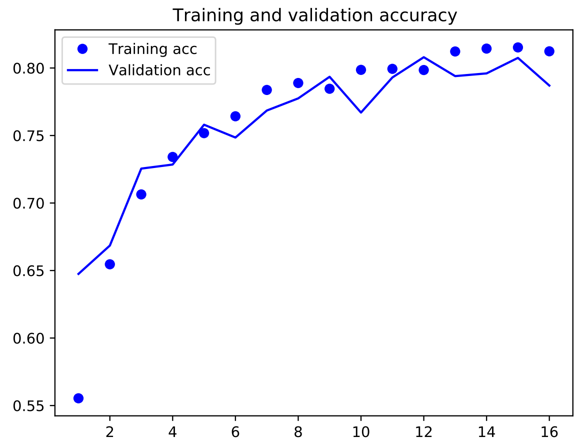


Figure 1. The accuracy of the model over each training epoch.

4. Data and Discussion

The main model achieved an accuracy of 80% against the validation data, shown in Figure 4 in the Appendix. The model with one fewer convolutional and dense layer achieved 77%. The console output shown in the block below is for images `cat4796`, `cat4846`, `dog4157`, and `dog4357`, respectively, which I chose randomly; these input images are shown in Figure 2.

Although the model is fairly accurate above random chance, it appears to favor clasification toward cats. This could be resolved by more training data, or even better, by expanding the number of classes. Since this is a mutually exlusive decision system, more class choices (assuming similar accuracy) would lead to better overall model accuracy because it would steer choices away from random chance. For this particular model, this can be easily accomplished by adding new folders within the dataset with the appropriate images, and changes the data generator to `categorical`.

Console testing output

```
dog confidence: 1.19e-24, cat confidence: 1.0
probably cat
dog confidence: 0.0, cat confidence: 1.0
probably cat
dog confidence: 6.63e-15, cat confidence: 1.0
probably cat
dog confidence: 1.0, cat confidence: 0.0029
probably dog
```

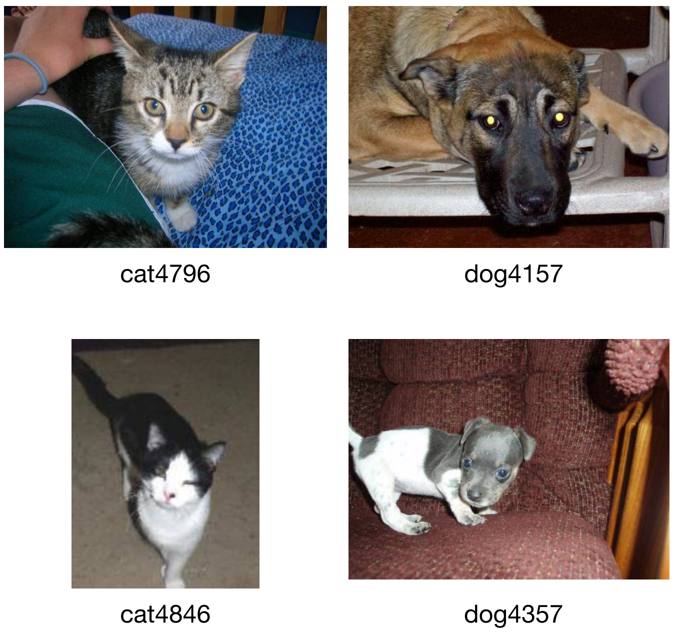


Figure 2. Randomly selected test data to show individual output.

5. Conclusion

Neural networks are a powerful way to solve abstract problems. By providing our desired output to our network, we back-calculate adjustements for parameters in the network to achieve our desired output. Although there are many potential pitfalls to this system, most of them are solved problems, and we are provided many hyperparameters we can choose to get around them.

In this report, we specifically looked at how convolutional neural networks can be used to classify images. We took an example of binary classification of cats and dogs, walked through the architecture of the model, its design choices, and trained it to obtain an accuracy of 80% against our validation data. We also took a look at how the model resolves these outputs at the individual scale by looking at single image inputs and observing console output. The next step for this type of project is to expand the categories from binary to many-class, with this same source code with minor modication, and with appropriate training data.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] chetanimravan. Dogs & cats immages. [Online]. Available: https://www.kaggle.com/chetankv/dogs-cats-images

Appendix

Code

train.py

```python
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
from keras.preprocessing.image import ImageDataGenerator


# constants
imgSize = (200, 200)
batchSize = 8
epochs = 16
trainingDirectory = "C:\\Users\\Arya Daroui\\Desktop\\Cnn\\Train\\"
testingDirectory = "C:\\Users\\Arya Daroui\\Desktop\\Cnn\\Test\\"

# data preprocessing
dataGenerator = ImageDataGenerator(rescale=1./255, zoom_range=0.15, horizontal_flip=True, rotation_range=15)
trainingData = dataGenerator.flow_from_directory(directory=trainingDirectory, target_size=imgSize, batch_size
    =batchSize, class_mode='binary')
testingData = dataGenerator.flow_from_directory(directory=testingDirectory, target_size=imgSize, batch_size=
    batchSize, class_mode='binary')

# model generation
model = Sequential()

model.add(Conv2D(filters=16, kernel_size=(3, 3), input_shape=trainingData.image_shape, activation='relu'))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(filters=32, kernel_size=(3, 3), input_shape=trainingData.image_shape, activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D())
model.add(Flatten()) # must flatten for upcoming dense layer

model.add(Dense(units=32, activation='relu'))

model.add(Dense(units=16, activation='relu'))
model.add(Dropout(0.25))

model.add(Dense(len(set(trainingData.classes)), activation='sigmoid'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(trainingData, epochs=epochs, validation_data=testingData)

# save trained model
model.save_weights('cnnweights.h5')
model.save('cnnmodel.h5')


# plot
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

test.py

```
from keras.engine.training import Model
from keras.preprocessing.image import load_img, img_to_array
from numpy import expand_dims
from keras.models import load_model

def Test(pathToImg: str, imgSize: tuple, model: Model):
    image = expand_dims(img_to_array(load_img(pathToImg, target_size= imgSize)), axis=0)
    prediction = model.predict(image)
    outputStr = "dog confidence: {:.3}, cat confidence: {:.3}\n".format(prediction[0][0], prediction[0][1])
    if prediction[0][0] > prediction[0][1]:
        outputStr += 'probably dog'
    else:
        outputStr += 'probably cat'
    return outputStr

# constants
imgSize = (200, 200)
projDirectory = "C:\\Users\\Arya Daroui\\Desktop\\Cnn\\"

# load trained model
model = load_model(projDirectory + "cnnmodel.h5")

#test
print(Test(projDirectory + "Test\\Cats\\cat.4796.jpg", imgSize, model))
print(Test(projDirectory + "Test\\Cats\\cat.4846.jpg", imgSize, model))
print(Test(projDirectory + "Test\\Dogs\\dog.4157.jpg", imgSize, model))
print(Test(projDirectory + "Test\\Dogs\\dog.4357.jpg", imgSize, model))
```

Additional figures

Training loss

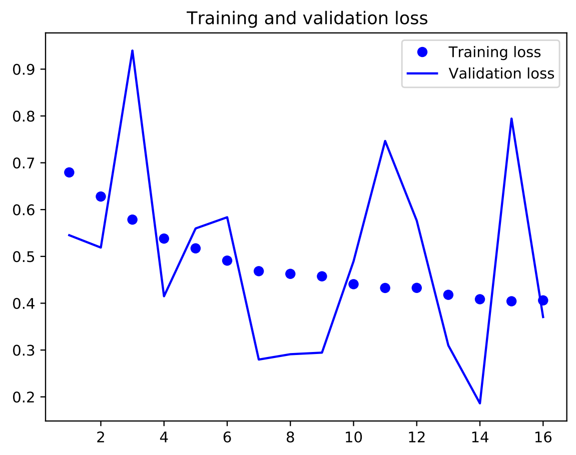
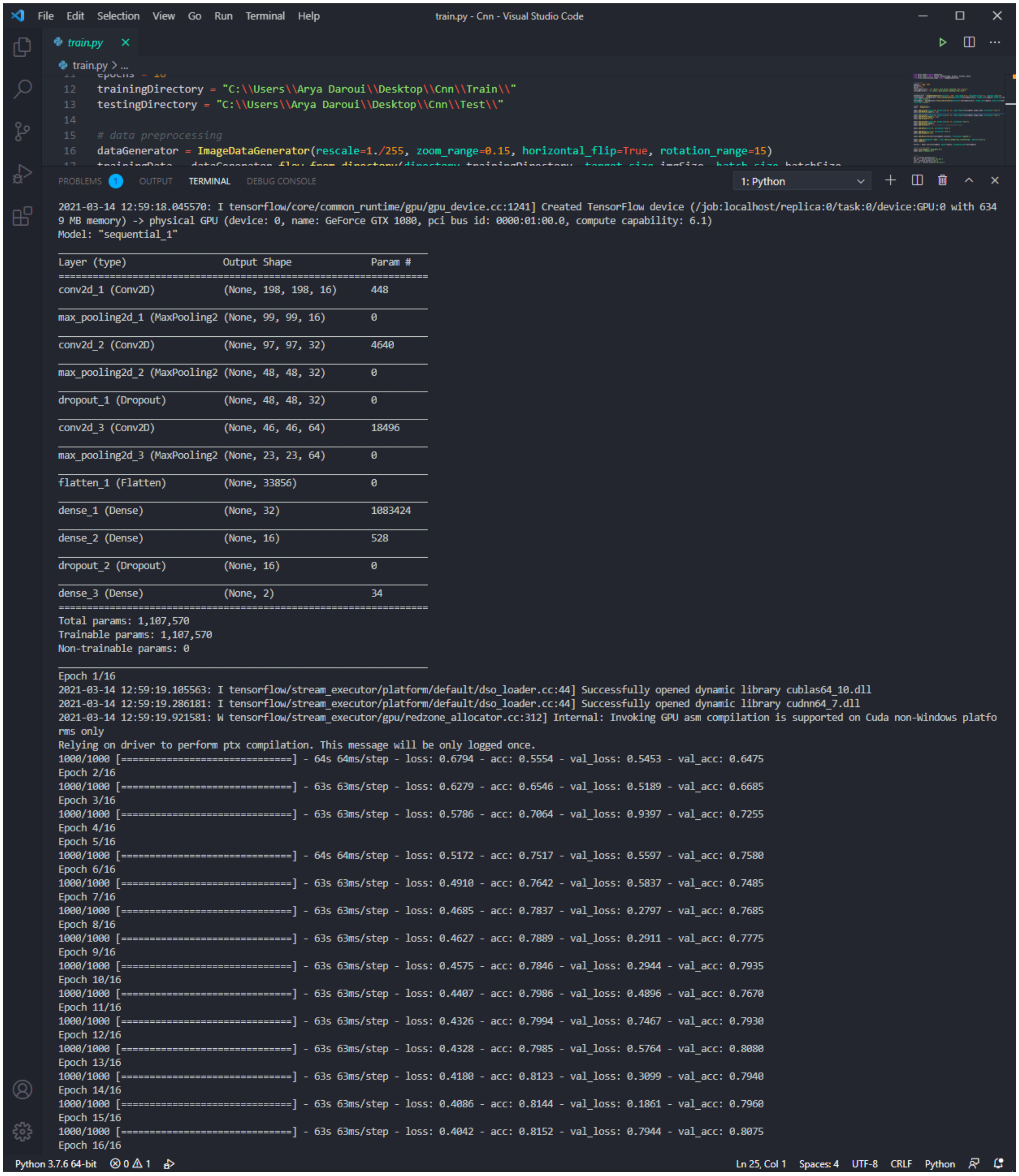


Figure 3. The loss of the model over each training epoch.

Training screenshot


File Edit Selection View Go Run Terminal Help
train.py - Cnn - Visual Studio Code
train.py
train.py > ...
PROBLEMS 1
OUTPUT
TERMINAL
DEBUG CONSOLE
1: Python
Python 3.7.6 64-bit ⊗ 0 ⚠ 1
Ln 25, Col 1 Spaces: 4 UTF-8 CRLF Python

```python
trainingDirectory = "C:\\Users\\Arya Daroui\\Desktop\\Cnn\\Train\\"
testingDirectory = "C:\\Users\\Arya Daroui\\Desktop\\Cnn\\Test\\"

# data preprocessing
dataGenerator = ImageDataGenerator(rescale=1./255, zoom_range=0.15, horizontal_flip=True, rotation_range=15)
```

```
2021-03-14 12:59:18.045570: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1241] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 634
9 MB memory) -> physical GPU (device: 0, name: GeForce GTX 1080, pci bus id: 0000:01:00.0, compute capability: 6.1)
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 198, 198, 16)      448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 99, 99, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 97, 97, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 48, 48, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 48, 48, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 46, 46, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 23, 23, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 33856)             0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1083424
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
dropout_2 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 34
=================================================================
Total params: 1,107,570
Trainable params: 1,107,570
Non-trainable params: 0
_________________________________________________________________
Epoch 1/16
2021-03-14 12:59:19.105563: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cublas64_10.dll
2021-03-14 12:59:19.286181: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cudnn64_7.dll
2021-03-14 12:59:19.921581: W tensorflow/stream_executor/gpu/redzone_allocator.cc:312] Internal: Invoking GPU asm compilation is supported on Cuda non-Windows platfo
rms only
Relying on driver to perform ptx compilation. This message will be only logged once.
1000/1000 [==============================] - 64s 64ms/step - loss: 0.6794 - acc: 0.5554 - val_loss: 0.5453 - val_acc: 0.6475
Epoch 2/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.6279 - acc: 0.6546 - val_loss: 0.5189 - val_acc: 0.6685
Epoch 3/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.5786 - acc: 0.7064 - val_loss: 0.9397 - val_acc: 0.7255
Epoch 4/16
Epoch 5/16
1000/1000 [==============================] - 64s 64ms/step - loss: 0.5172 - acc: 0.7517 - val_loss: 0.5597 - val_acc: 0.7580
Epoch 6/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4910 - acc: 0.7642 - val_loss: 0.5837 - val_acc: 0.7485
Epoch 7/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4685 - acc: 0.7837 - val_loss: 0.2797 - val_acc: 0.7685
Epoch 8/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4627 - acc: 0.7889 - val_loss: 0.2911 - val_acc: 0.7775
Epoch 9/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4575 - acc: 0.7846 - val_loss: 0.2944 - val_acc: 0.7935
Epoch 10/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4407 - acc: 0.7986 - val_loss: 0.4896 - val_acc: 0.7670
Epoch 11/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4326 - acc: 0.7994 - val_loss: 0.7467 - val_acc: 0.7930
Epoch 12/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4328 - acc: 0.7985 - val_loss: 0.5764 - val_acc: 0.8080
Epoch 13/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4180 - acc: 0.8123 - val_loss: 0.3099 - val_acc: 0.7940
Epoch 14/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4086 - acc: 0.8144 - val_loss: 0.1861 - val_acc: 0.7960
Epoch 15/16
1000/1000 [==============================] - 63s 63ms/step - loss: 0.4042 - acc: 0.8152 - val_loss: 0.7944 - val_acc: 0.8075
Epoch 16/16
```

Figure 4. Screenshot of console output after training the main model.

Model architecture

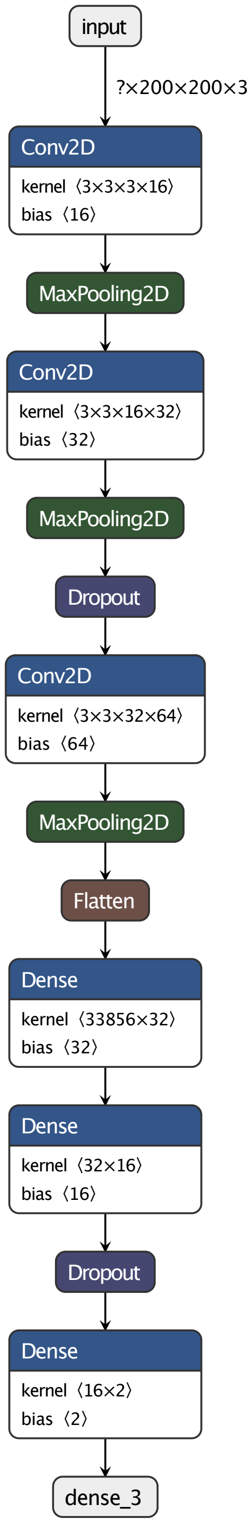


Figure 5. High level overview of the neural network model. Created with Netron.