Image classification

RAFAŁ KLINOWSKI, JAKUB CISOWSKI

BIOLOGICALLY INSPIRED ARTIFICIAL INTELLIGENCE, SEM. 6

What is image classification?

- Categorizing images
- Tries to answer the question: „What is in the picture?”
- Based on pixels or vectors within an image
- Tries to recognize patterns within an image to make predictions
- We explore „single-label classification” which only assigns one value to an image

Roses

Tulips

Dog

Cat

Real life examples of image recognition

- Medicine – eg. to help doctors detect abnormalities such as cancer
- Self-driving cars
- Face ID and other facial recognition systems
- Retail – eg. the ability to try on a piece of clothing online

Our project

Distinction between cat and dog breeds

- One step further than just saying „dog or cat”
- Input: an image
- Output: name of dog or cat breed as well as certainty (in %)

98% Beagle

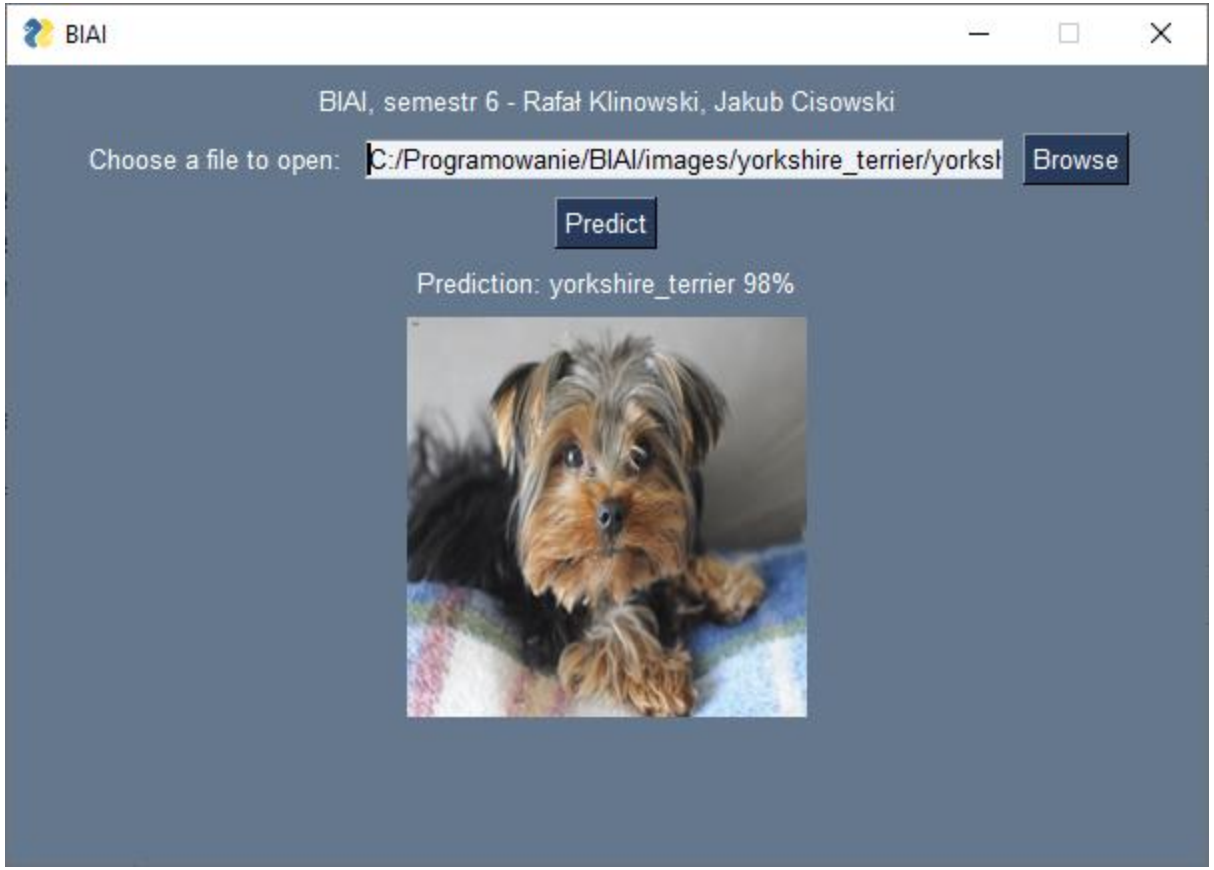
84% Birman

Dataset used for training

- Courtesy of The University of Oxford
- JPG image files
- 37 cat or dog races
- About 200 images for each breed
- Over 7390 images in total
- Various sizes (have to be standardized before training)

Description of technology

- Programming language: Python
- TensorFlow – used to create, train, save and load model
- Keras – used to simplify data loading and dividing into different categories
- MatPlotLib – used to generate plots describing model performance, accuracy
- Algorithm:
  - Divide images into groups based on breed
  - Calculate minimum, maximum and average image size (used when creating model)
  - Load data, split into two groups: training, validation
  - Create, compile, train and evaluate model
  - Save model as a group of files


BIAI
BIAI, semestr 6 - Rafał Klinowski, Jakub Cisowski
Choose a file to open:
C:/Programowanie/BIAI/images/yorkshire_terrier/yorksh
Browse
Predict
Prediction: yorkshire_terrier 98%

Research

We built and trained our model multiple times with different parameters to test their impact on performance

Key parts of the code

target_size, input_shape

- represents the size to which all images will be resized when first loaded
- larger values require significantly more memory and longer training time
- smaller values mean there are less trainable parameters – model will be less accurate
- common values include: 96x96, 160x160, 224x224

```
target_size = (160, 160)
input_shape = (target_size[0], target_size[1], 3)
```

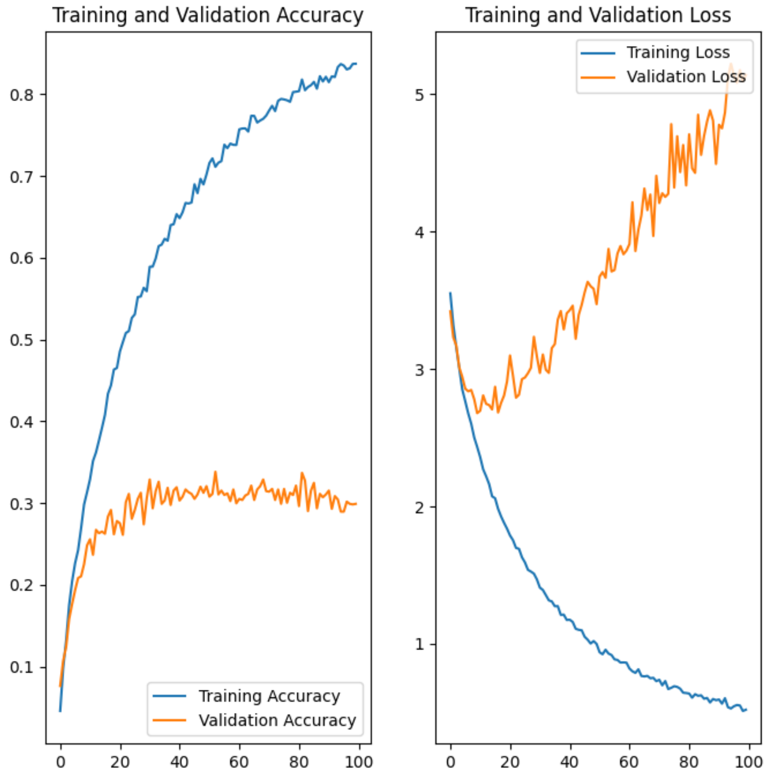
target_size=(224,224)

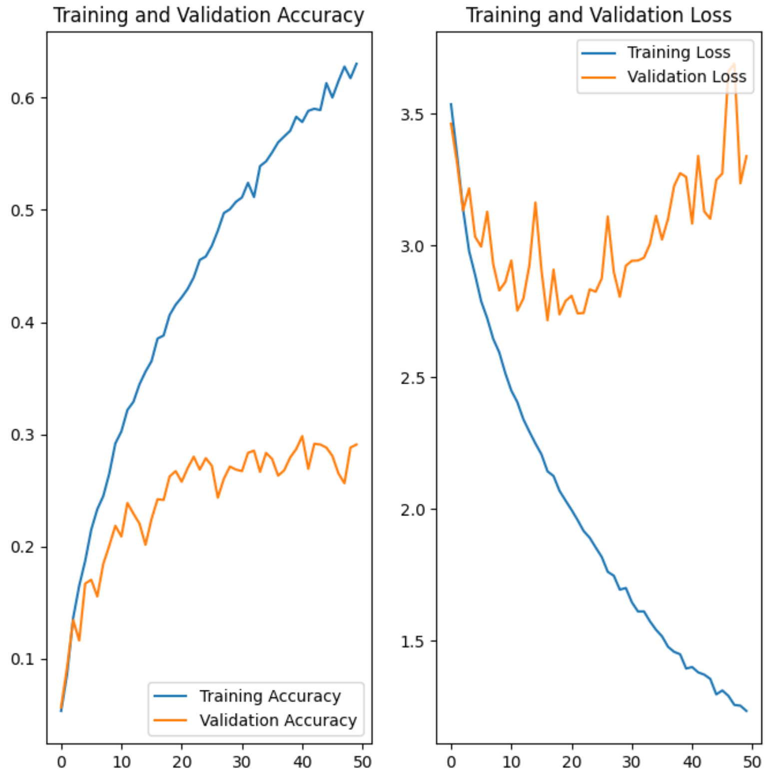
(note: different scale)

target_size=(96,96)

training time ≈ 20 hours

training time ≈ 45 minutes





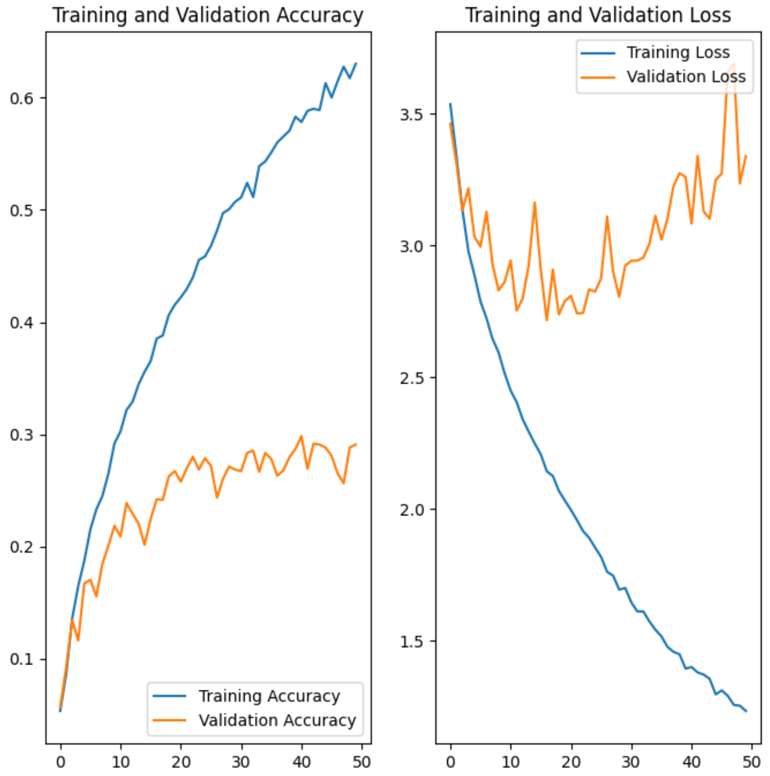
batch_size

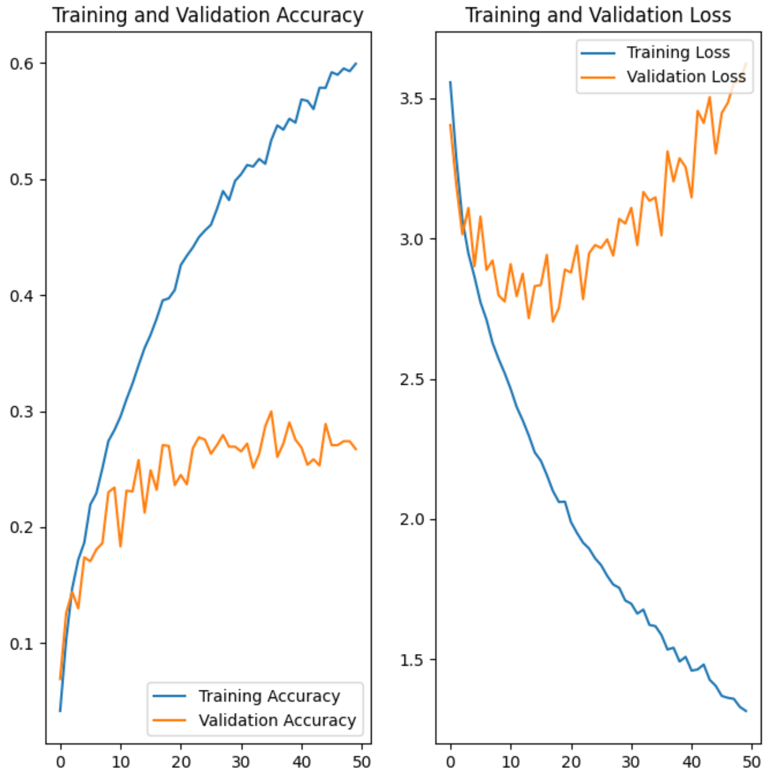
- the number of images in a batch, used to perform one step of training
- if too small, then we end up approximating the function with too few examples that might not be representative of all training data
- if too large, requires a lot of memory and can even run out of it during training which is then interrupted
- common values are 16, 32, 64

```
batch_size = 16
```

batch_size=16 (note: not enough memory to run 32) batch_size=8

training time ≈ 45 minutes training time ≈ 3 hours





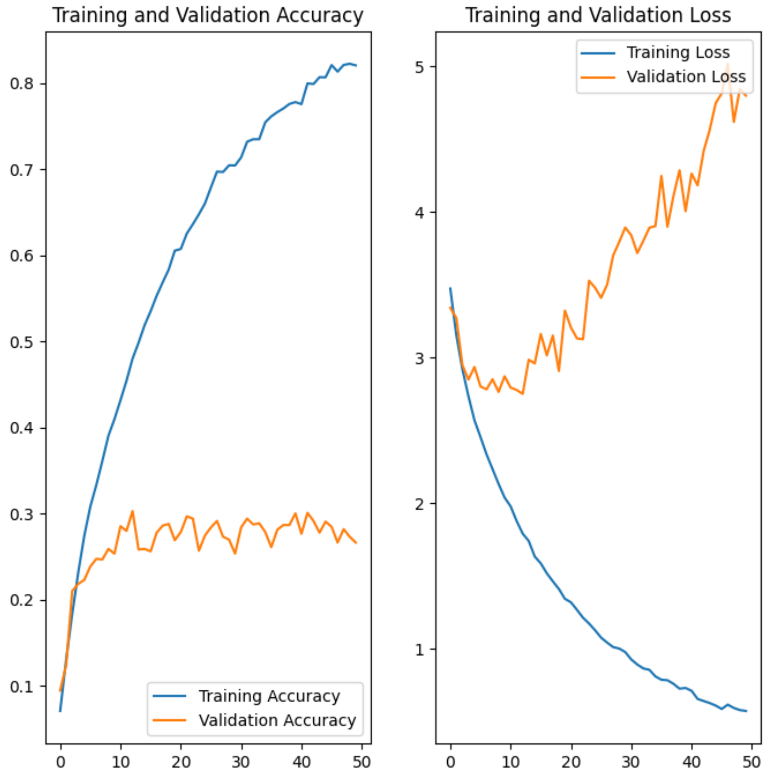
validation_split

- the percentage of training images used for validation
- the rest of images is used for training only
- the dataset we used did not provide a separate set for validation – thus we had to use split
- common value is 20%

```
validation_split=0.2
```

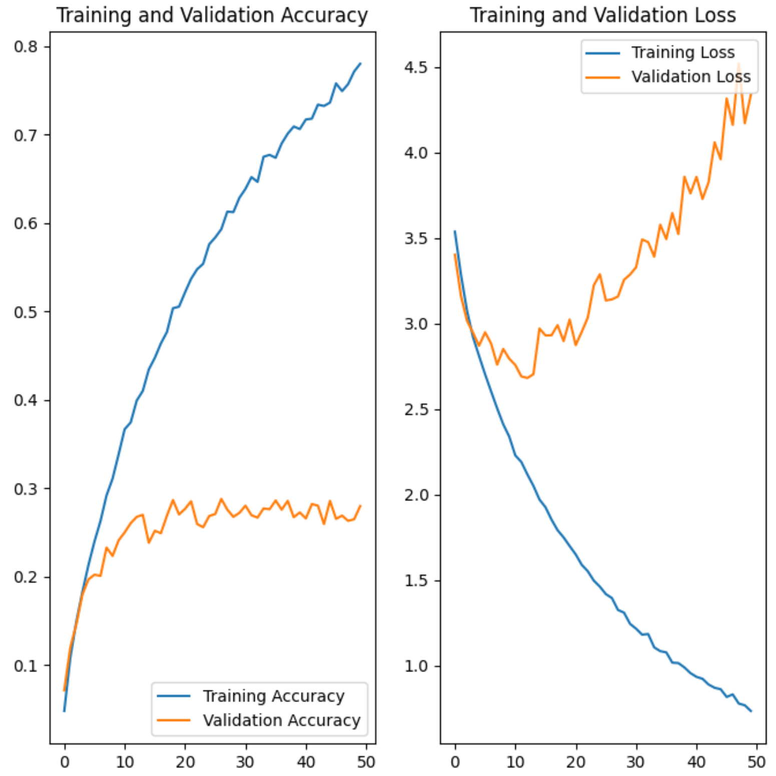
validation_split=0.2

training time ≈ 45 minutes



validation_split=0.3

training time ≈ 45 minutes



shuffle

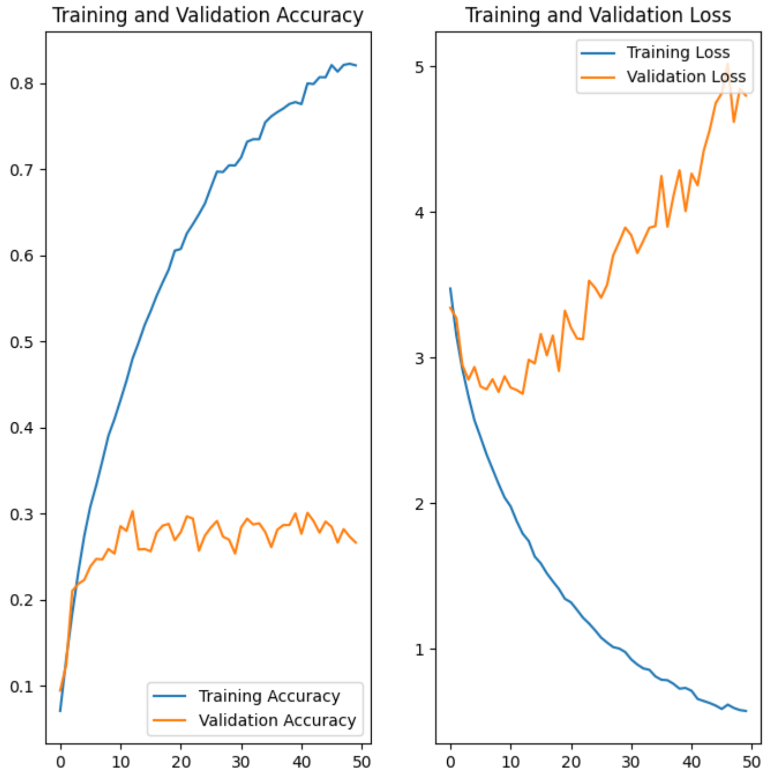
- random ordering of images in training set
- commonly used to improve training accuracy
- prevents the model from learning the order of images
- helps with training time as well due to „prefetch” (some images are ready to be used in memory)
- shuffle parameter tells how many points to keep in memory for random drawing

```
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="training",
    seed=6798,
    image_size=target_size,
    batch_size=batch_size)

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
```

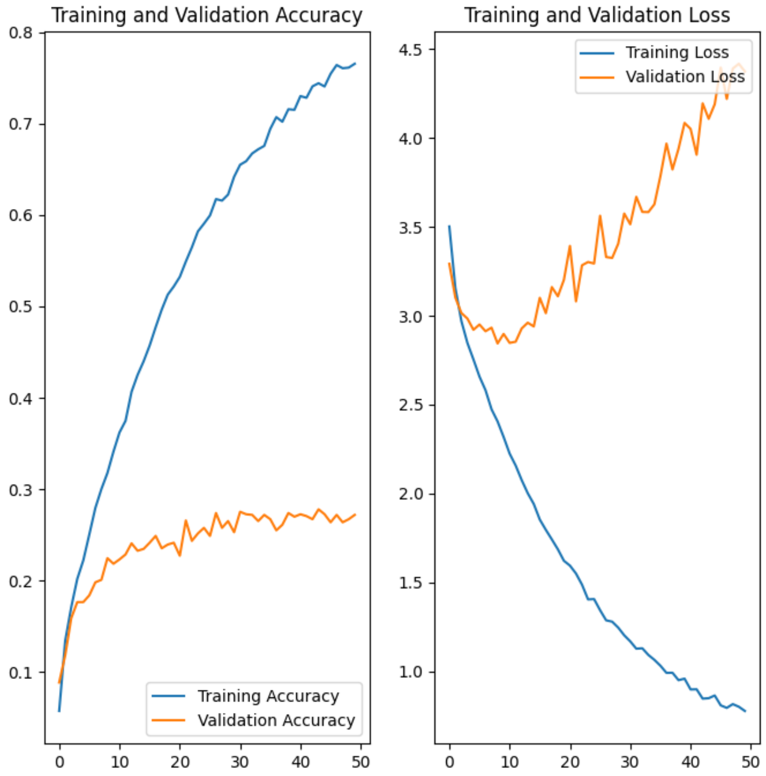
shuffle(1000)

training time ≈ 45 minutes



no shuffle

training time ≈ 50 minutes



data_augmentation

- helps diversify data
- improves accuracy
- we used three functions: RandomFlip (flips some of the images), RandomRotation (rotates some of the images slightly) and RandomZoom (zooms in or out on some of the images slightly)
- tested values: 0.1 and 0.2

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal",
                          input_shape=input_shape),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.1),
    ]
)
```

Creating the model

- we could use many different layers in different orders or with different parameters
- most room for experimenting

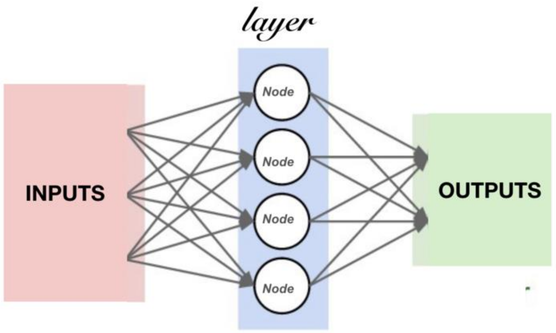
```
model = Sequential([
    data_augmentation,
    layers.Rescaling(1. / 255, input_shape=input_shape),     # Normalize colors
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Dropout(0.2),     # Dropout 20% of the nodes to increase validation accuracy
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(num_classes)
])
```

Layers used in the final model

- The model used is Sequential, which means the layers are applied in order
- data_augmentation – explained before
- Rescaling – normalizes pixel values for images (from 0 – 255 to 0.0 – 1.0)
- Conv2D – Convolution Layer, performs element-wise multiplication on some parts of the input matrix at a time, outputs a different matrix
- MaxPooling2D – downsamples the input for each channel of input by choosing max value from each feature vector patch
- Dropout – randomly drops out (sets to zero) some inputs
- Flatten – flattens the input (reduces multidimensional vectors to a single dimension array)

What is a layer?

- A layer is a structured used to pass information
- Takes one or more inputs, applies a transformation, and has one or more outputs
- Input itself is a layer too
- Transformations are mostly non-linear functions



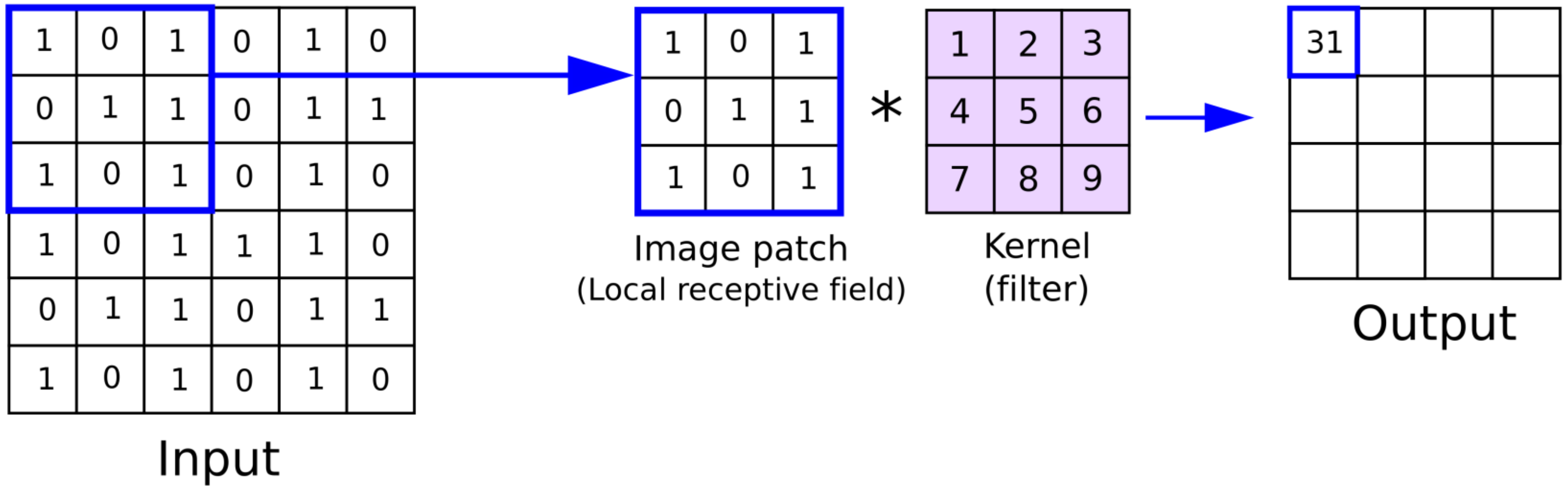
Usually uniform containing one or more types of functions (mostly non-linear).

’Rescaling’ layer

- It’s necessary to normalize input data
- Makes all layers and features work on the same level of scale (0.0 – 1.0) instead of having different scales (0-1, 0-255, etc.)

'Conv2D' layer

- Performs element-wise multiplication on some parts of the input matrix at a time
- Outputs a different matrix

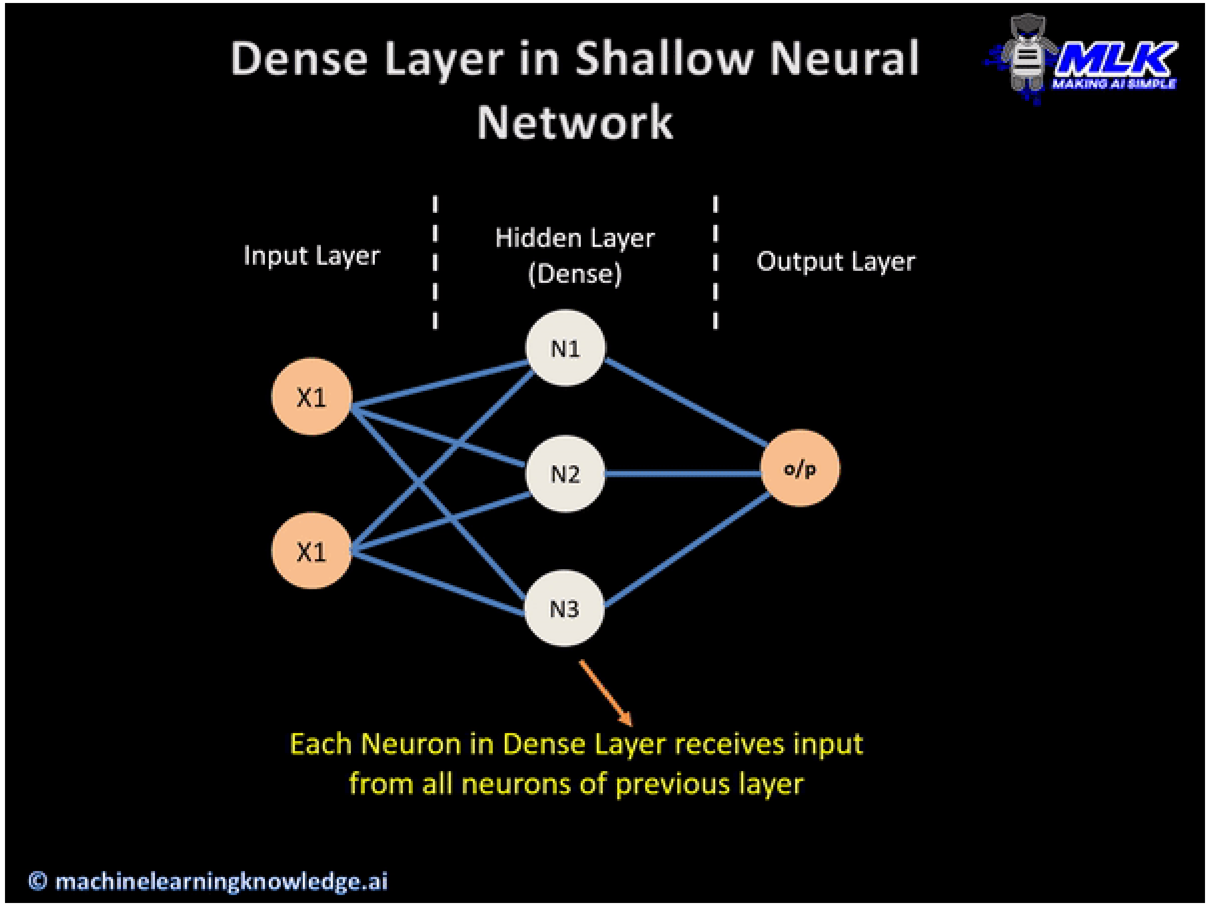


’Dense’ layer

- applies a non-linear transform on data
- each neuron in Dense layer receives input from all neurons from previous layer
- activation function passed as parameter
- for activation function we used ReLU – returns input if it’s positive, otherwise returns zero; it’s fast, simple and universally works well
- parameter – number of outputs
- second Dense layer – output size is equal to the amount of classes we have (37)

```
layers.Dense(64, activation='relu'),
layers.Dense(num_classes)
```

’Dense’ layer – cont.

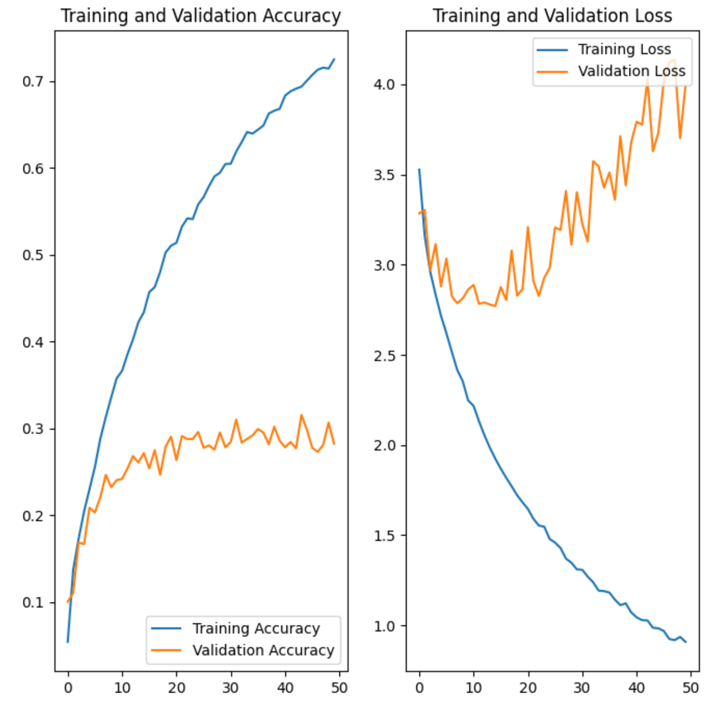


’Dropout’ layer

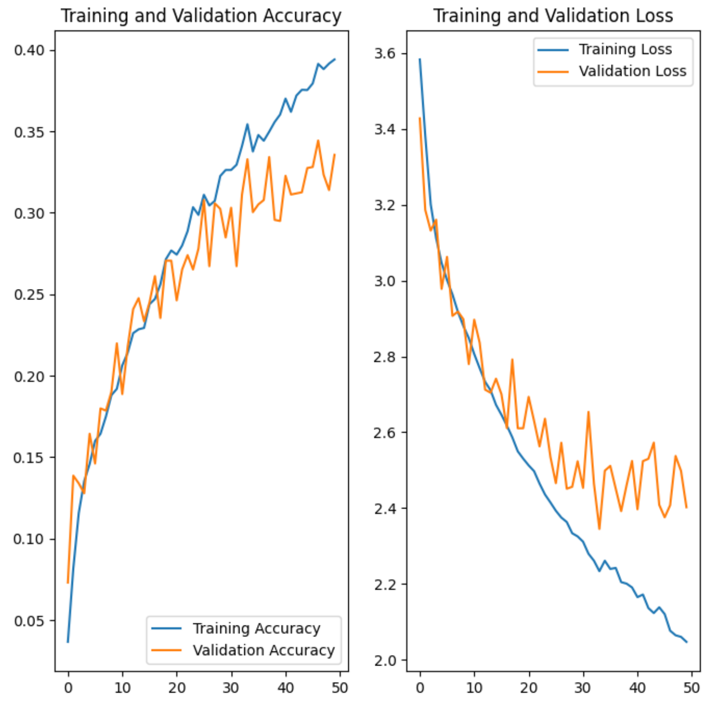
- Used to prevent overfitting
- Overfitting – situation when model predicts „too well” on training data, and therefore is not very accurate at making predictions from outside of training set
- By randomly dropping out some of inputs we get a more diverse, less dependent set

```
layers.Dropout(0.2)
```

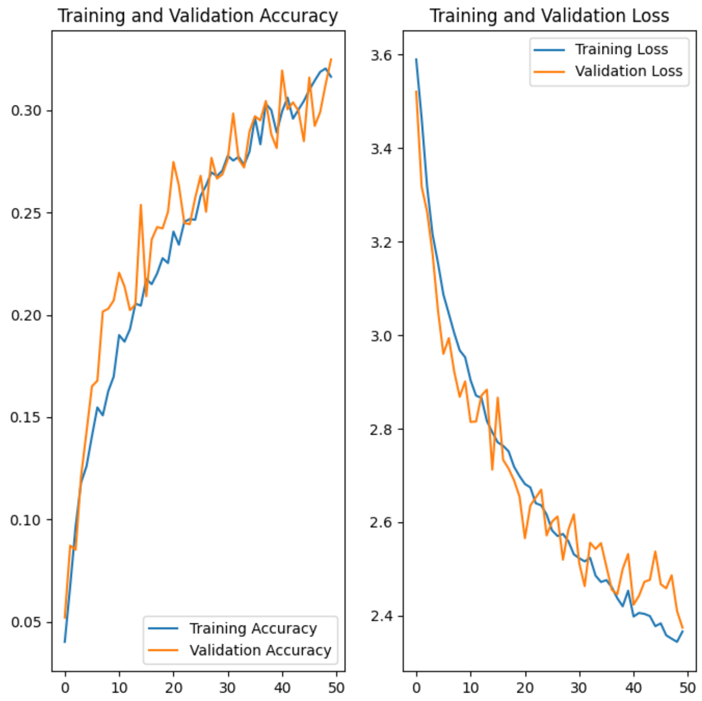
default (dropout 0.2)



dropout 0.2 twice



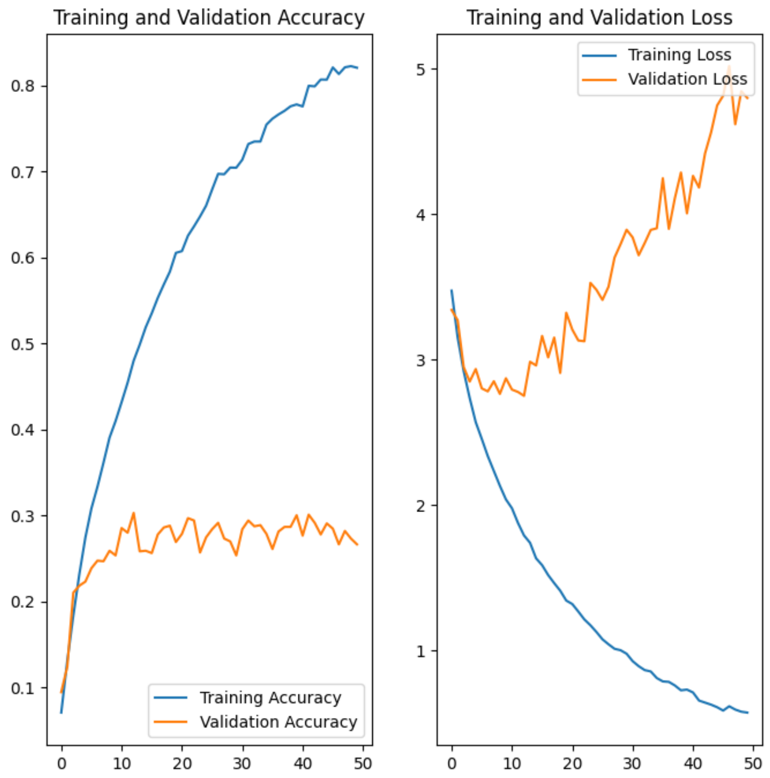
dropout 0.4 twice



Other experiments

- We experimented with many different layer combinations and even some pretrained models
- All experiments are compared with the ’final model’ described previously
- Two most relevant attempts: modifying the sequence (using the same layers but in different order), adding significantly more layers

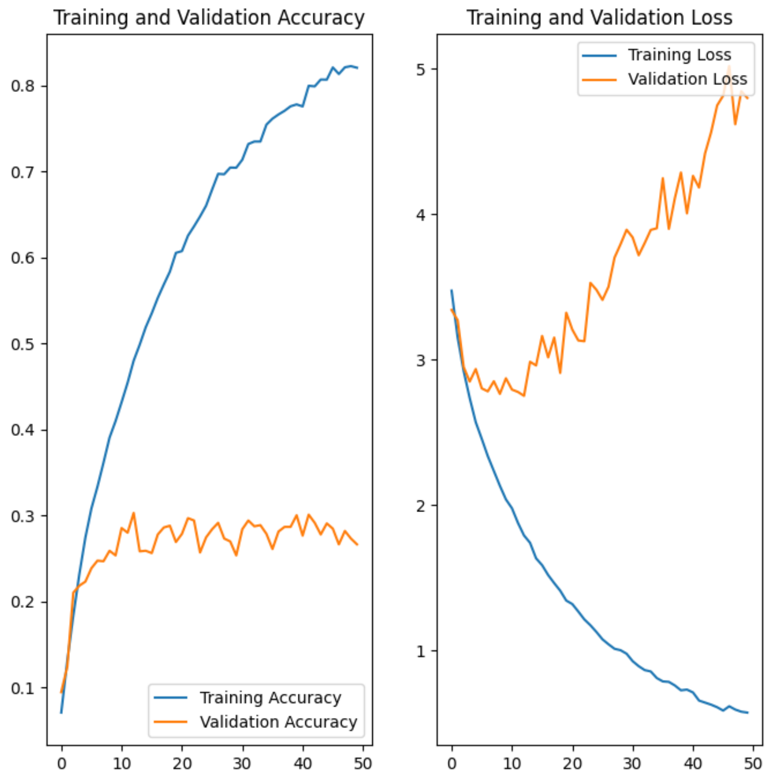
default



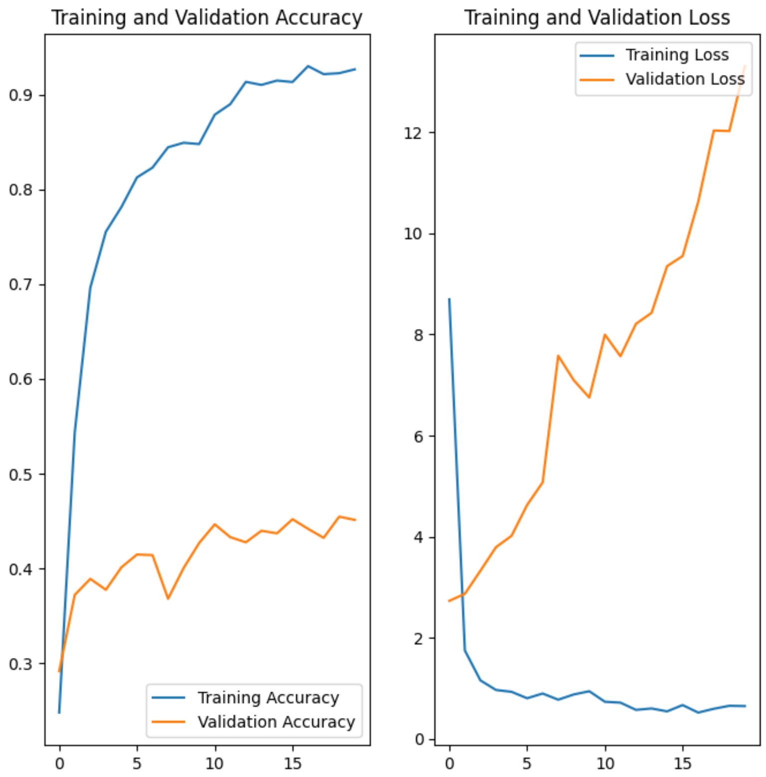
modified sequence

(note: different scale)

default



significantly more layers

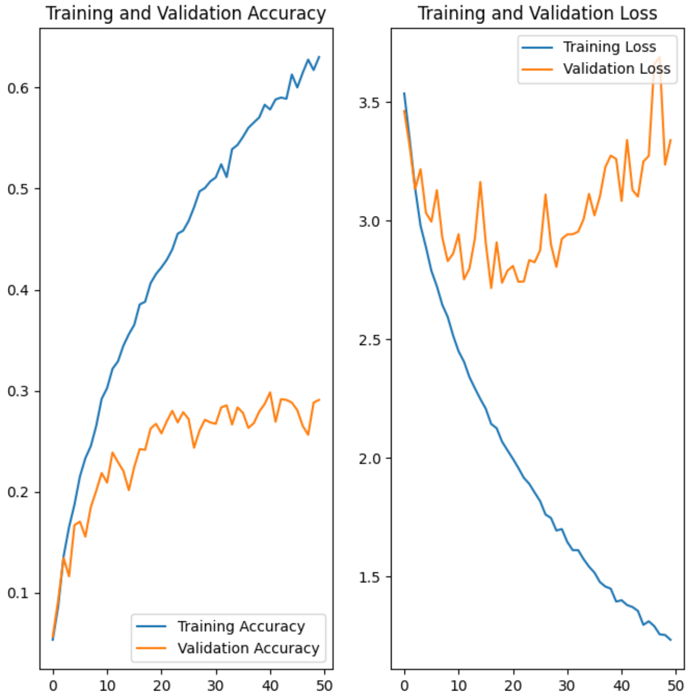


Optimizer

- Specified during model compilation
- Specific algorithms used for model training
- Have different speed and performance based on data and other parameters
- Different parameters: learning rate, decay rate, functions
- Affect loss function – try to reduce losses

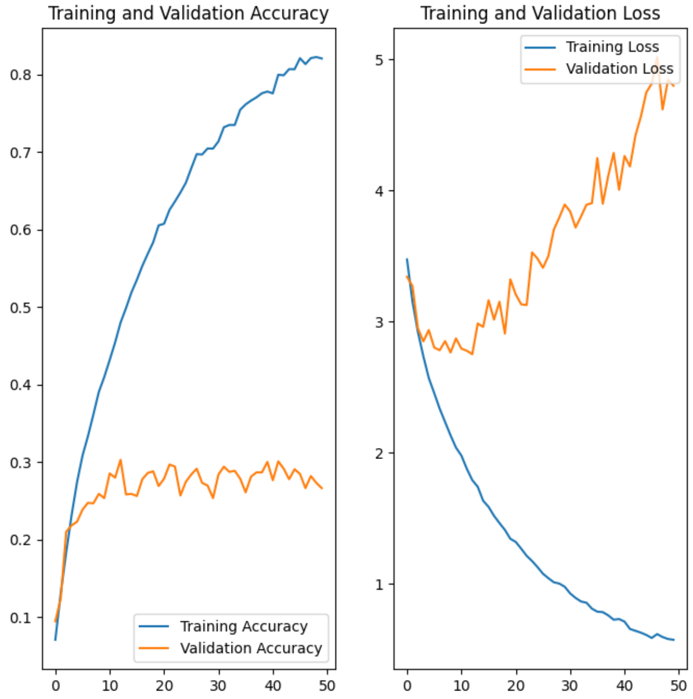
```
model.compile(optimizer='nadam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Adam



Adadelta

NAdam

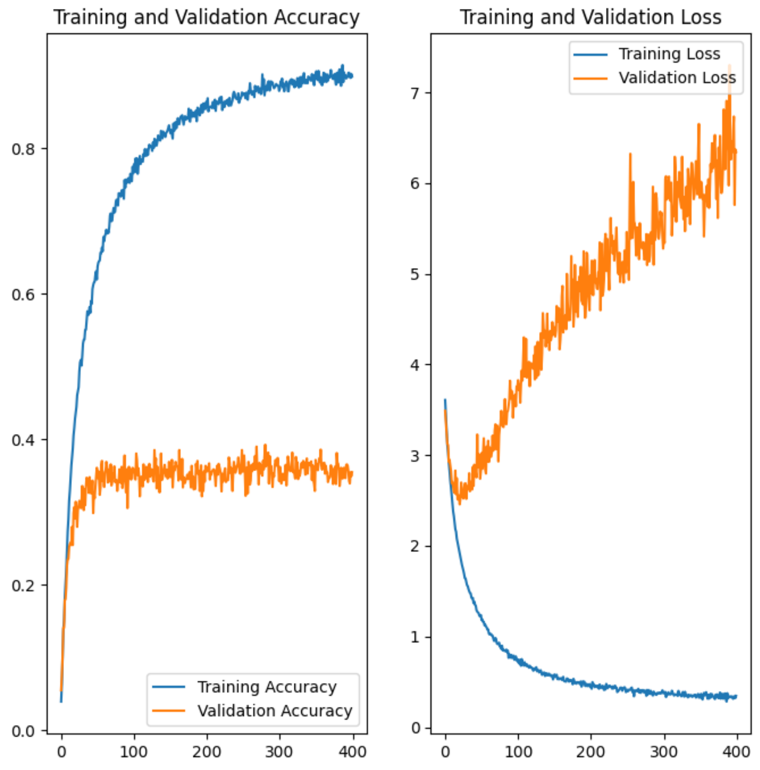


Epochs

- Amount of training iterations
- Each iteration, all samples are passed though the neural network once
- More does not necessarily mean better – after a certain amount of iterations, accuracy stays nearly constant or even goes down
- For research, we mostly used epochs=50 – good middleground between short training time and decent results

```
epochs = 400
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)
```

Final model performance



Conclusions

- The dataset we used was not big enough to fully prevent overfitting, and to keep training accuracy as close as possible to validation accuracy
- We had to settle for some unoptimized parameters (eg. batch_size) – hardware problems
- The final model performs very well on training data

Problems we encountered

- Main problem – because the model is only trained on 37 classes of data, it cannot distinguish other classes (other breeds of dogs or cats)
- When passing an image that does not belong to these 37 classes, we will not get good results

Sources

- https://www.kaggle.com/datasets/zippyz/cats-and-dogs-breeds-classification-oxford-dataset
- https://www.tensorflow.org/
- https://docs.python.org/3/reference/
- https://machinelearningknowledge.ai/
- https://towardsdatascience.com/
- https://anhreynolds.com/blogs/cnn.html
- https://iq.opengenus.org/purpose-of-different-layers-in-ml/