## Beskrivelse

#### Skrive omkring det problem som I forsøger at løse/forudse.

Jeg har valgt at lave et neuralt netværk der ud fra et billede af en kat, skal kunne fortælle dens race. Inspirationen kommer af 2 grunde:

* At jeg selv har en kat der lige nu ikke har en bestemt race, så jeg vil forsøge at få en computer til at give mig svaret
* Det klassiske *Er det en kat eller er det en hund?* eksempel

## Datasættet

#### Hvor kommer datasættet fra og hvad i datasættet kan bruges til at løse problemet og forudse løsning.

### Cat breeds

<https://www.kaggle.com/datasets/ma7555/cat-breeds-dataset>

På [kaggle.com](https://www.kaggle.com/) findes der mange datasæt at vælge i mellem, i forskellige typer. Det jeg startede med at vælge at arbejde med, hedder *Cat Breeds* og indeholder 67 katteracer fordelt ud på 127.000 billeder. Datasættet kommer med billeder, der er sorteret i mapper der er navngivet med race navne og en CSV fil der indeholder en liste over alle billeder, med beskrivelse af katten på billede så som: race, alder, køn, størrelse og farve.

En udfordring jeg mødte med dette datasæt var dog, at det indeholdte korrupte filer der konstant forstyrrede model træningen samt billeder der slet ikke var relevante, da vedkommende der lavede datasættet har indsamlet billeder på diverse adoptions sider, sandsynligvis via webscraping hvilket også har betydet at kvaliteten af datasættet har været tvivlsomt.

### Cat breeds (Cleared)

<https://www.kaggle.com/datasets/denispotapov/cat-breeds-dataset-cleared>

Efter at have arbejdet med det første datasæt i noget tid, valgte jeg at kigge efter et nyt. Her fandt jeg samme datasæt som før, men hvor en anden bruger har haft frasorteret alt det irrelevante. Her kom datasættet ned til 119.000 billeder.

Dog har jeg valgt at skifte datasæt sent, så hvor vidt det er mærkbart bedre kan jeg endnu ikke konkludere.

## Algoritmen

#### Neuralt netværk – Deep learning

<https://www.tensorflow.org/>

<https://www.tensorflow.org/tutorials/load_data/images>

Neuralt netværk til billedgenkendelse, da det er godt til at genkende mønstre og før opgaven, lavede vi en øvelse om håndskrift genkendelse der brugte samme framework og algoritme, så der var problemstillingen allerede lignende og da havde jeg allerede fået afprøvet én måde at gøre det på.

<https://www.ibm.com/think/topics/machine-learning-types>

<https://cloud.google.com/discover/deep-learning-vs-machine-learning>

Datasættet bliver opdelt i 2:

1. Trænings sæt, som udgør 80% af datasættet.
2. Validerings sæt, som udgør 20% af datasættet.

## Konklusion

#### Kunne problemet løses bedre/nemmere uden ML?

#### Er problemet for komplekst?

#### Hvordan kunne man forbedre modellen?

## Rescaling Layer

Omskriver en pixels farveskala fra den normale **[0, 255]** til **[0, 1]**, for effektivisering af træningen da denne kan blive mere ustabil af større værdier.

## Conv2D Layer

Tilføjer et filter til hvert billede, med en lille udskæring der typisk er **3x3** eller **5x5** pixels, for at fokusere på det område ad gangen og leder efter mønstre der senere kan bruges.

## MaxPooling Layer

## Dropout Layer

## Flatten Layer

## Dense Layer

## Output DenseLayer

Certainly! Here’s a quick overview of the layers used in your model:

1. **Rescaling Layer** (tf.keras.layers.Rescaling(1./255)):  
   This layer scales pixel values from [0, 255] to [0, 1]. The parameter 1./255 specifies the scaling factor, normalizing images to improve model convergence.
2. **Conv2D Layer** (tf.keras.layers.Conv2D(filters, kernel\_size, activation='relu')):  
   This layer applies convolution filters to the input, extracting local patterns (edges, textures).
   * filters: Number of output channels (e.g., 32, 64).
   * kernel\_size: Size of the convolution window (e.g., 3 for a 3x3 window).
   * activation='relu': Applies ReLU, turning negative values to zero, which helps learn non-linear patterns.
3. **MaxPooling2D Layer** (tf.keras.layers.MaxPooling2D(pool\_size)):  
   Downsamples by selecting the maximum value in a window, reducing the spatial dimensions.
   * pool\_size: Size of the pooling window, often (2, 2), which halves width and height, reducing computation.
4. **Dropout Layer** (tf.keras.layers.Dropout(rate)):  
   Randomly sets a fraction of inputs to zero during training, preventing overfitting.
   * rate: Fraction of input to drop, e.g., 0.2 for 20%.
5. **Flatten Layer** (tf.keras.layers.Flatten()):  
   Flattens the 2D spatial data (height and width) into a 1D vector, preparing it for dense layers.
6. **Dense Layer** (tf.keras.layers.Dense(units, activation='relu')):  
   Fully connected layer, where each input node connects to each output node.
   * units: Number of neurons in the layer (e.g., 128).
   * activation='relu': Non-linear activation to introduce learning complexity.
7. **Output Dense Layer** (tf.keras.layers.Dense(num\_classes)):  
   Final dense layer with output units equal to the number of classes, producing raw class scores. No activation is used here if from\_logits=True in SparseCategoricalCrossentropy, which applies softmax during loss calculation.

Let me know if you’d like further clarification on any specific layer!

Great questions! Let’s go over each layer with further clarification:

1. **Rescaling Layer**:  
   Yes, by "pixel values," we mean the color intensity values, which usually range from 0 to 255 for each color channel (Red, Green, Blue). Rescaling divides these values by 255 to convert them into a range between 0 and 1. This smaller range helps the model learn more efficiently, as high values can cause instability during training.
2. **Conv2D Layer**:  
   A Conv2D layer applies a small filter (or "window") that slides across the image, focusing on a tiny part at a time (like a 3x3 or 5x5 area of pixels). For each window, it performs mathematical operations to detect patterns, such as edges or textures, in that local area.
   * filters (e.g., 32, 64) is the number of filters (or pattern detectors), which essentially controls how many different types of patterns it can detect.
   * kernel\_size (e.g., 3) is the width and height of the filter, which defines the window size for each pattern detection.
3. **MaxPooling2D Layer**:  
   If you didn’t set pool\_size, it defaults to (2, 2), which means it downsamples by a factor of 2 in both dimensions (width and height).  
   Max pooling takes a small window (like 2x2) and selects the maximum pixel value within that window, which means it keeps the most prominent feature in each region and discards the rest. This helps to reduce the image’s spatial size while preserving the essential features, making the model faster and less prone to overfitting.
4. **Dropout Layer**:  
   Dropout randomly ignores (sets to zero) a fraction of the neurons in the layer during each training step. This doesn’t block parts of the image directly but rather blocks out certain neurons. By forcing the model to rely on different neurons each time, dropout encourages it to learn more robust patterns across all features, rather than relying too heavily on specific ones.
5. **Dense Layer**:  
   Yes, dense layers represent the classic "fully connected" structure often shown in neural network diagrams. In a dense layer, every neuron in one layer is connected to every neuron in the next layer. This structure makes dense layers excellent for combining the features detected in the earlier convolutional layers to make a final classification decision.

Each of these layers plays a unique role in transforming the image data, detecting patterns, and making predictions. Let me know if you’d like more details on any of these!