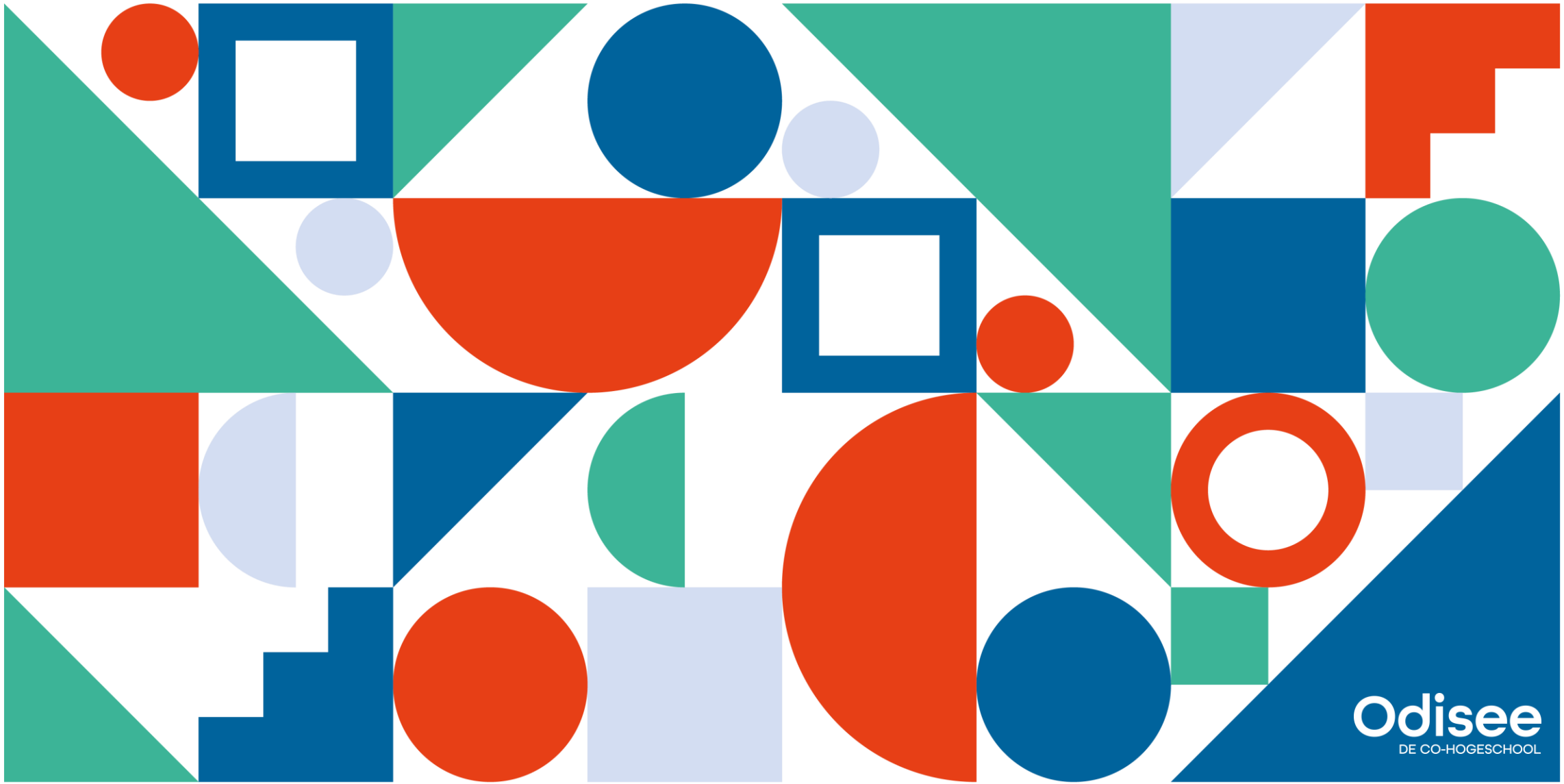

Odisee
DE CO-HOGESCHOOL

Preprocessing en computer visie

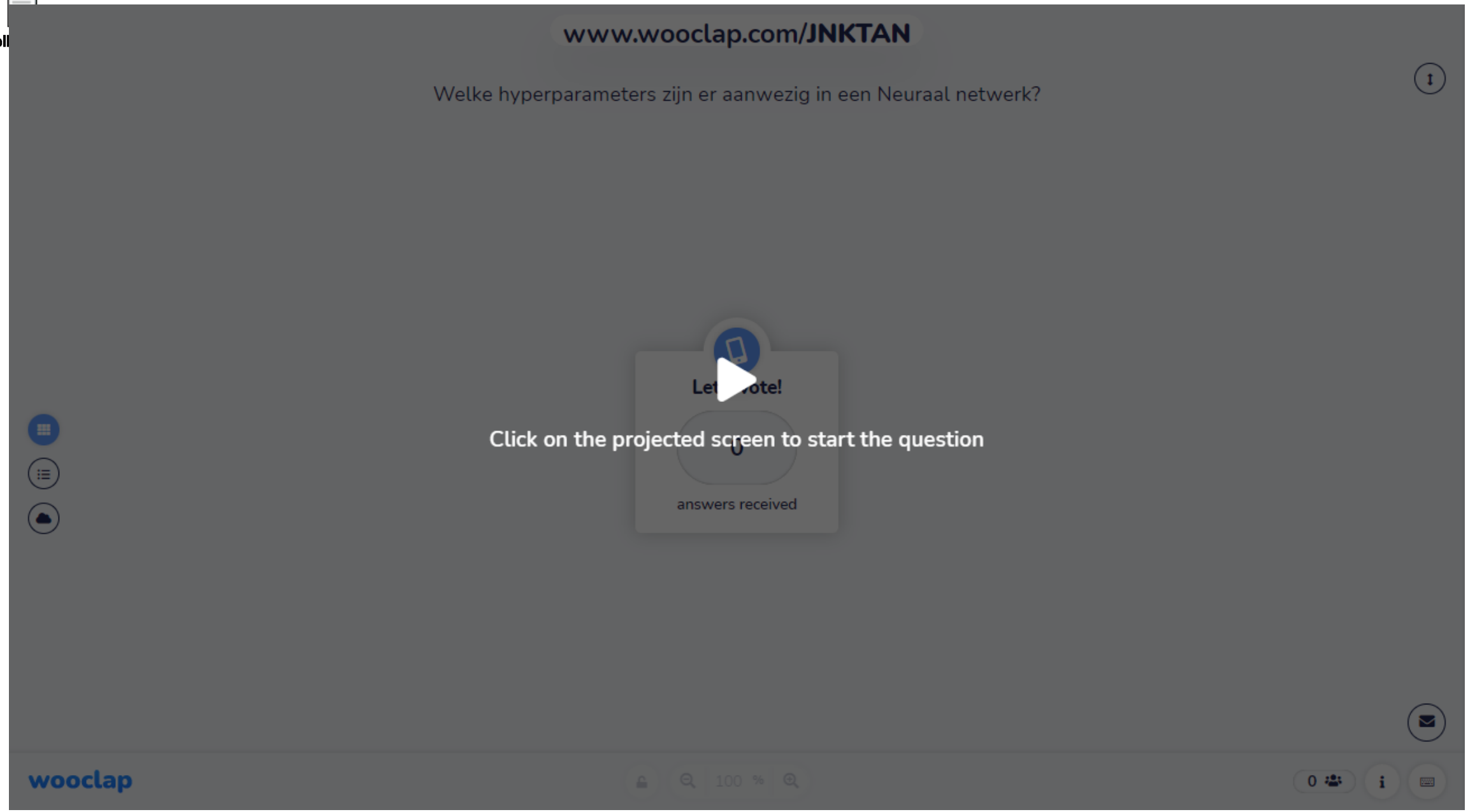
Jens Baetens

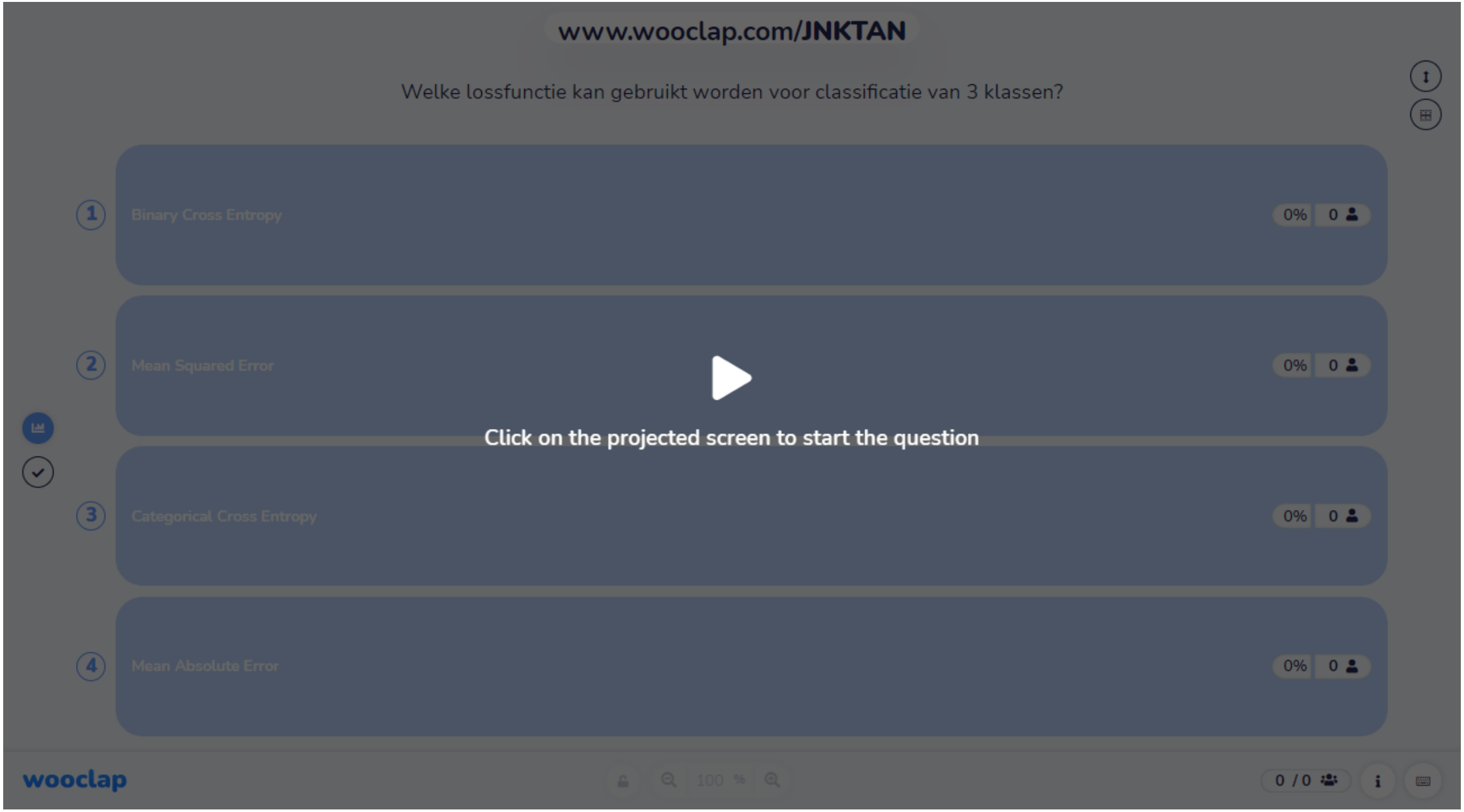
How to participate?

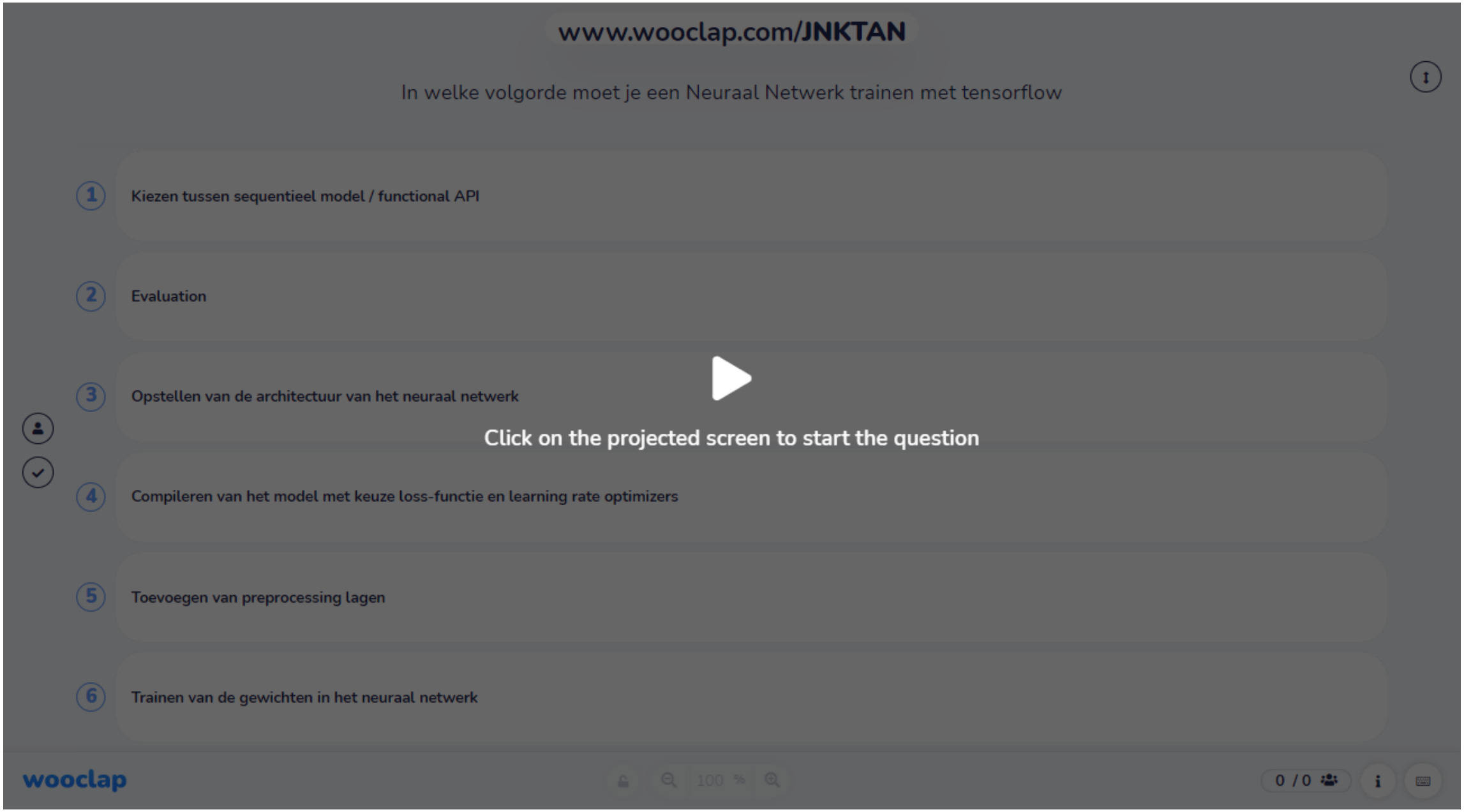
1. Connect to www.wooclap.com/JNKTAN
2. You can participate

1. Not yet connected? Send @JNKTAN to 0460 200 711
2. You can participate

DataColl


www.wooclap.com/JNKTAN
Welke hyperparameters zijn er aanwezig in een Neuraal netwerk?
Click on the projected screen to start the question
answers received
wooclap
100 %
0


www.wooclap.com/JNKTAN
Welke lossfunctie kan gebruikt worden voor classificatie van 3 klassen?
1
Binary Cross Entropy
0%
0
2
Mean Squared Error
0%
0
Click on the projected screen to start the question
3
Categorical Cross Entropy
0%
0
4
Mean Absolute Error
0%
0
wooclap
100 %
0 / 0


www.wooclap.com/JNKTAN
In welke volgorde moet je een Neuraal Netwerk trainen met tensorflow
1 Kiezen tussen sequentieel model / functional API
2 Evaluation
3 Opstellen van de architectuur van het neuraal netwerk
Click on the projected screen to start the question
4 Compileren van het model met keuze loss-functie en learning rate optimizers
5 Toevoegen van preprocessing lagen
6 Trainen van de gewichten in het neuraal netwerk
wooclap
100 %
0 / 0

Preprocessing

Beschikbare lagen

- https://www.tensorflow.org/guide/keras/preprocessing_layers
- Tekstverwerking
  - TextVectorization -> string to tensor
- Getallen
  - Normalization of Discretization
- Categorieën
  - Category encoding, Hashing, StringLookup, IntegerLookup
- Images
  - Resizing, Rescaling, CenterCrop

Adapt()

- Sommige van de lagen worden getrained in de .fit()
- Sommige niet en moeten voor de fit aangepast worden aan de trainingsdata met de adapt() functie voor de fit()
  - TextVectorization -> map tussen strings en integers
  - StringLookup en IntegerLookup -> map tussen inputs en integers
  - Normalization -> mean en standard deviation
  - Discretization -> bucket/bin boundaries
- Dit komt omdat alle data gekend moet zijn terwijl de fit batch per batch verwerkt

Preprocessing voor het model

- Voer de taken uit op de tf.data.Dataset

```
# dit wordt typisch asynchroon uitgevoerd op de CPU en gebufferd en dan in batches doorgegeven aan de trainingsloop
dataset = dataset.map(lambda x, y: (preprocessing_layer(x), y))
# voeg deze lijn toe om de preprocessing in parallel met de training uit te voeren (optioneel)
dataset = dataset.prefetch(tf.data.AUTOTUNE)
model.fit(dataset, ...)
```

- Wanneer je traint op een TPU/GPU probeer zoveel mogelijk preprocessing in tf.data pipeline uit te voeren.
  - Enkel Normalization en Rescaling werkt er goed op

Preprocessing als deel van het model

```
inputs = keras.Input(shape=input_shape)
x = preprocessing_layer(inputs)
outputs = rest_of_the_model(x)
model = keras.Model(inputs, outputs)
```

- Preprocessing synchroon met het model
- Wordt op hetzelfde toestel uitgevoerd als de training
  - Kan dus ook profiteren van de GPU indien aanwezig
  - Veruit de beste optie als je fit op GPU of gebruik maakt van beelden
- Verbeterd de flexibiliteit van je model
  - Wanneer je het model opslaat, worden ook de preprocessing lagen bewaart
  - Kleiner verschil tussen development en productie
  - Het model is gebruiksvriendelijker want kennis over de preprocessing is niet vereist

Welke optie hebben we vorige week gebruikt?

```
# ML-model
body = tf.keras.Sequential([
  tf.keras.layers.Dense(10, activation='relu'),
  tf.keras.layers.Dense(10, activation='relu'),
  tf.keras.layers.Dense(1)
])

x = preprocessor(inputs)
result = body(x)

model = tf.keras.Model(inputs, result)

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

Wat met verschillende soorten inputs

- Soorten inputs: bvb numerieke/categoriek?
- Maak gebruik van Functional API ipv Sequential model

Normalization

```
# Load some data
(x_train, y_train), _ = keras.datasets.cifar10.load_data()
x_train = x_train.reshape((len(x_train), -1))
input_shape = x_train.shape[1:]
classes = 10

# Create a Normalization layer and set its internal state using the training data
normalizer = layers.Normalization()
normalizer.adapt(x_train)

# Create a model that include the normalization layer
inputs = keras.Input(shape=input_shape)
x = normalizer(inputs)
outputs = layers.Dense(classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)

# Train the model
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
model.fit(x_train, y_train)
```

Normalization

- Tf.keras.layers.Normalization
  - Werkt global, volledige data
- Speciale Soorten
  - Source: https://towardsdatascience.com/different-types-of-normalization-in-tensorflow-c
  - Batch
  - Group
  - Instance
  - Layer
  - Weight

```
# normalizer
normalizer = tf.keras.layers.Normalization()
normalizer.adapt(X_train_flat)
```

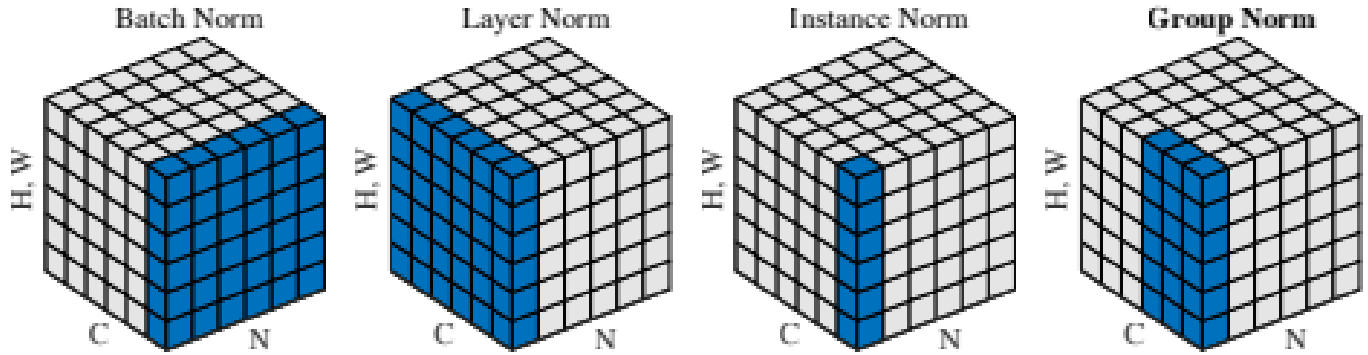


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Normalisation: Batch Normalisation

- Heel goed voor performantie
  - Reden nog niet volledig gekend
- Normalisatie binnen een batch van sample
- Werkt vooral goed bij grote batches
- Te gebruiken na elke laag met een activatiefunctie

```
#Batch Normalization
model.add(tf.keras.layers.BatchNormalization())
```

Normalisation: Group normalisation

- Vooral gebruikt bij computer visie
- Normalisation over groepen van kanalen ipv voorbeelden
  - Betere performantie bij kleine batches

- Werkt beter dan Layer en instance normalisation
  - Combineert de goede elementen van beide

```
#Group Normalization
model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3),
activation='relu'))
model.add(tfa.layers.GroupNormalization(groups=8, axis=3))
```

One-hot encoding

```
# Define some toy data
data = tf.constant([["a"], ["b"], ["c"], ["b"], ["c"], ["a"]])

# Use StringLookup to build an index of the feature values and encode output.
lookup = layers.StringLookup(output_mode="one_hot")
lookup.adapt(data)

# Convert new test data (which includes unknown feature values)
test_data = tf.constant([["a"], ["b"], ["c"], ["d"], ["e"], [""]])
encoded_data = lookup(test_data)
print(encoded_data)
```

```
# Define some toy data
data = tf.constant([[10], [20], [20], [10], [30], [0]])

# Use IntegerLookup to build an index of the feature values and encode output.
lookup = layers.IntegerLookup(output_mode="one_hot")
lookup.adapt(data)

# Convert new test data (which includes unknown feature values)
test_data = tf.constant([[10], [10], [20], [50], [60], [0]])
encoded_data = lookup(test_data)
print(encoded_data)
```

One-hot encoding: hashing trick

- Woordenboek kan heel groot zijn (duizenden) maar veel woorden komen maar zelden voor
  - Inefficiënt om deze woorden een kolom toe te kennen
  - Hash de waarden naar een vector van een bepaalde grootte
  - Verkleint het aantal features en zorgt ervoor dat er geen expliciete indexing meer nodig is

```
# Sample data: 10,000 random integers with values between 0 and 100,000
data = np.random.randint(0, 100000, size=(10000, 1))

# Use the Hashing layer to hash the values to the range [0, 64]
hasher = layers.Hashing(num_bins=64, salt=1337)

# Use the CategoryEncoding layer to multi-hot encode the hashed values
encoder = layers.CategoryEncoding(num_tokens=64, output_mode="multi_hot")
encoded_data = encoder(hasher(data))
print(encoded_data.shape)
```

Ngrams en multi-hot

- N-gram
  - Een sequentie van N opeenvolgende woorden
- Multi-hot encoding
  - In een n-gram staan meer woorden
  - Deze staan op 1 bij multi-hot

```
# Define some text data to adapt the layer
adapt_data = tf.constant(
    [
        "The Brain is wider than the Sky",
        "For put them side by side",
        "The one the other will contain",
        "With ease and You beside",
    ]
)
# Instantiate TextVectorization with "multi_hot" output_mode
# and ngrams=2 (index all bigrams)
text_vectorizer = layers.TextVectorization(output_mode="multi_hot", ngrams=2)
# Index the bigrams via `adapt()`
text_vectorizer.adapt(adapt_data)

# Try out the layer
print(
    "Encoded text:\n", text_vectorizer(["The Brain is deeper than the sea"]).numpy(),
)
```

Tf-idf

- Term frequency – inverse document frequency

```
# Instantiate TextVectorization with "tf-idf" output_mode
# (multi-hot with TF-IDF weighting) and ngrams=2 (index all bigrams)
text_vectorizer = layers.TextVectorization(output_mode="tf-idf", ngrams=2)
# Index the bigrams and learn the TF-IDF weights via `adapt()`
```

Tf-idf

- Term frequency – inverse document frequency
  - Geef een lager gewicht aan termen die in heel veel rijen, entries, teksten voorkomen
  - Woorden die in slechts een aantal documenten of teksten voorkomen krijgen een hoger gewicht
    - “Nigerian prince” is belangrijker om spam te detecteren dan “Kind regards”

```
# Instantiate TextVectorization with "tf-idf" output_mode
# (multi-hot with TF-IDF weighting) and ngrams=2 (index all bigrams)
text_vectorizer = layers.TextVectorization(output_mode="tf-idf", ngrams=2)
# Index the bigrams and learn the TF-IDF weights via `adapt()`
```

Werken met meerdere in/outputs

- Lees en bestudeer de code op deze link:
  - https://www.tensorflow.org/guide/keras/functional#manipulate_complex_graph_topologies

Data generation

Data generators

- Meer informatie vind je hier: https://www.tensorflow.org/guide/data
- Vooral gebruikt als de data niet eenvoudig in het geheugen kan ingeladen worden
  - Beelden, figuren of images
  - Tekstbestanden
  - Heel grote gestructureerde datasets (beperkt en beter zelf in batches verdelen)
- Door het gebruik van een generator kan deze data batch per batch ingeladen worden en door de pipeline sturen
  - Kan natuurlijk ook gedaan worden puur in python ipv tensorflow code te gebruiken

Beelden inladen

- Bron: https://www.tensorflow.org/guide/keras/train_and_evaluate#using_a_kerasutilssequence_object_as_input
- keras.utils.Sequence
  - Python data generator
  - Werkt met multiprocessing
  - Data kan geshuffled worden
- Twee methoden moeten geimplementeerd worden
  - __getitem__ : return een batch die getraind/verwerkt moet worden
  - __len__ : De lengte van de dataset
- Indien je de dataset wil veranderen tussen epochs door
  - on_epoch_end functie

Beelden inladen: voorbeeld

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np

# Here, `filenames` is list of path to the images
# and `labels` are the associated labels.

class CIFAR10Sequence(Sequence):
    def __init__(self, filenames, labels, batch_size):
        self.filenames, self.labels = filenames, labels
        self.batch_size = batch_size

    def __len__(self):
        return int(np.ceil(len(self.filenames) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch_x = self.filenames[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.labels[idx * self.batch_size:(idx + 1) * self.batch_size]
        return np.array([
            resize(imread(filename), (200, 200))
               for filename in batch_x]), np.array(batch_y)

sequence = CIFAR10Sequence(filenames, labels, batch_size)
model.fit(sequence, epochs=10)
```

Gestructureerde data / tekstverwerking

- Hetzelfde patroon kan uitgevoerd worden voor
  - Gestructureerde data
    - Haal de dataset in batches op ipv volledig in te laden
  - Tekstverwerking
    - Lees ipv de juiste figuren, de juiste lijnen/bestanden tekst in

Data augmentation

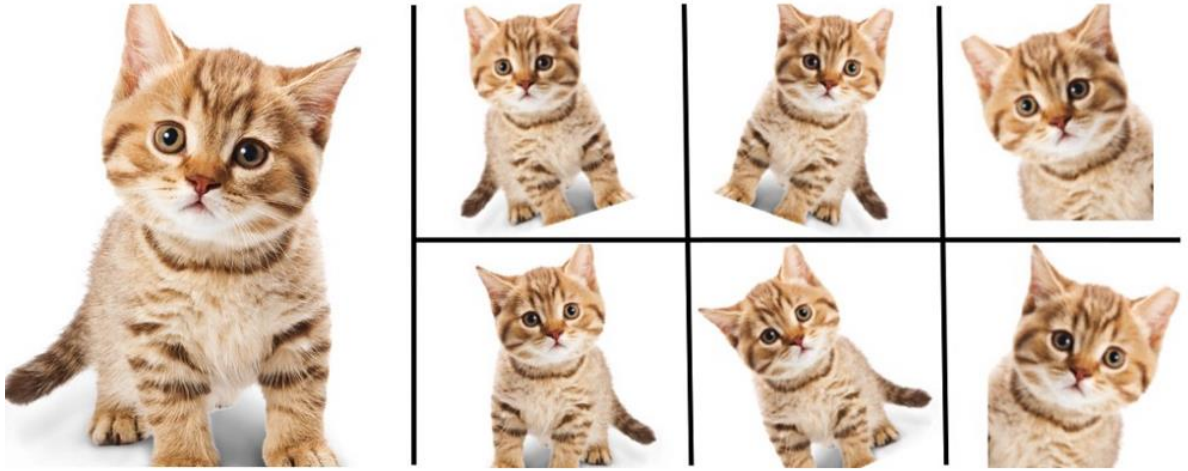
Data augmentation

- Vaak gebruikt in Computer Visie
  - Kan ook in audio
- Genereer extra data door kleine wijzigingen toe te passen
  - Random crop, resize, rotate, spiegelen horizontaal of verticaal, …
  - Aanpassen van de kleuren (elk kleur apart of combinaties)
  - Toevoegen van ruis
  - Versnellen of vertragen van audio
  - Verhogen of verlagen van de stemtoon
  - …

Data augmentation: voorbeeld

```
from tensorflow import keras
from tensorflow.keras import layers

# Create a data augmentation stage with horizontal flipping, rotations, zooms
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.1),
    ]
)
```



Data augmentation: Live demo

- Zie notebook voor demo

- Lagen die hiervoor gebruikt kunnen worden:

  `tf.keras.layers.RandomCrop`

  `tf.keras.layers.RandomFlip`

  `tf.keras.layers.RandomTranslation`

  `tf.keras.layers.RandomRotation`

  `tf.keras.layers.RandomZoom`

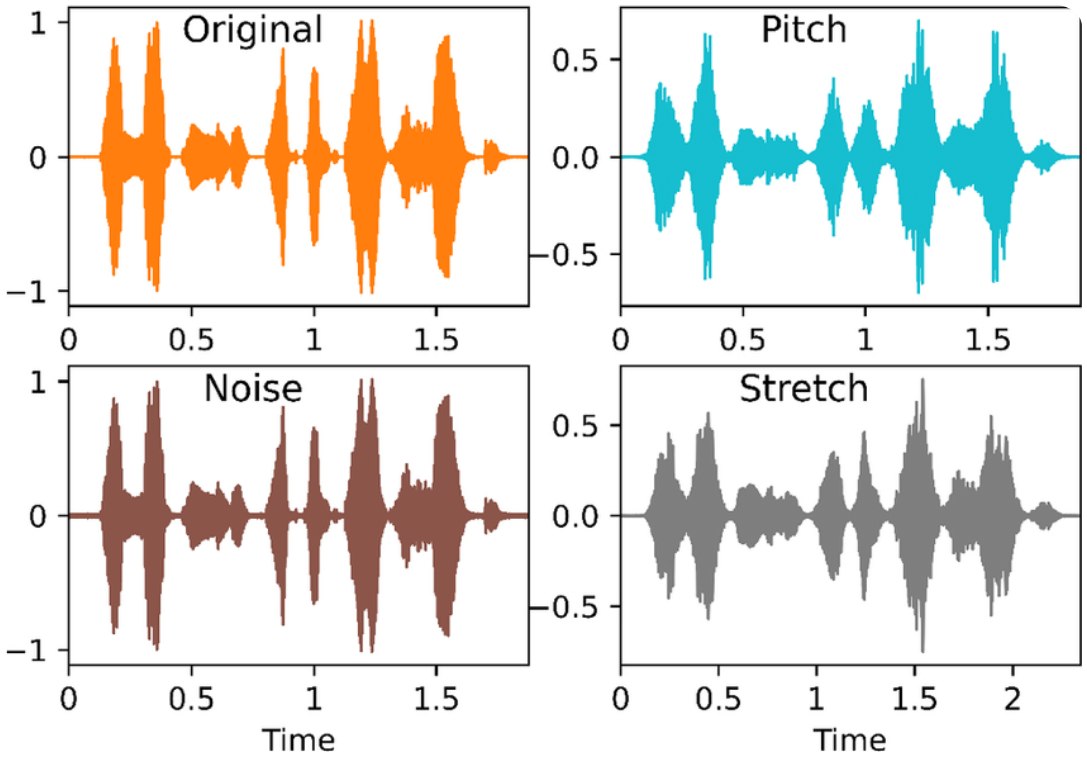
  `tf.keras.layers.RandomHeight`

  `tf.keras.layers.RandomWidth`

  `tf.keras.layers.RandomContrast`

Enkel voor images?

- Audio
  - Ruis toevoegen
  - Toon wijzigen
  - Versnellen/vertragen
- Video
  - Reeks van beelden
  - Zelfde transformatie nodig voor alle beelden



Wat je zelf doet, doe je (soms) beter!

- Niet alle mogelijke augmentaties bestaan
- Je kan er zelf toevoegen
  - Op basis van een lambda laag met een functie

```
def random_invert_img(x, p=0.5):
  if  tf.random.uniform([]) < p:
    x = (255-x)
  else:
    x
  return x
```

```
def random_invert(factor=0.5):
  return layers.Lambda(lambda x: random_invert_img(x, factor))

random_invert = random_invert()
```

  - Overerving van een Layer

```
class RandomInvert(layers.Layer):
  def __init__(self, factor=0.5, **kwargs):
    super().__init__(**kwargs)
    self.factor = factor

  def call(self, x):
    return random_invert_img(x)
```

Wanneer voer je data augmentatie uit?

- Voor de training
  - Asynchroon (non-blocking) op de CPU
  - Meer opslag nodig om de verschillende varianten bij te houden
  - Lagen niet standaard mee geëxporteerd maar kunnen wel toegevoegd worden
- Voeg de preprocessing lagen toe aan het model
  - Augmentaties uitgevoerd synchroon met de andere lagen
  - Kan uitgevoerd worden op de GPU
  - Mee geëxporteerd bij bewaren model
    - Standaardisaties worden automatisch uitgevoerd (Resizing, Cropping, …)
    - Augmentaties (Random-Rotation, …) enkel uitgevoerd bij .fit()

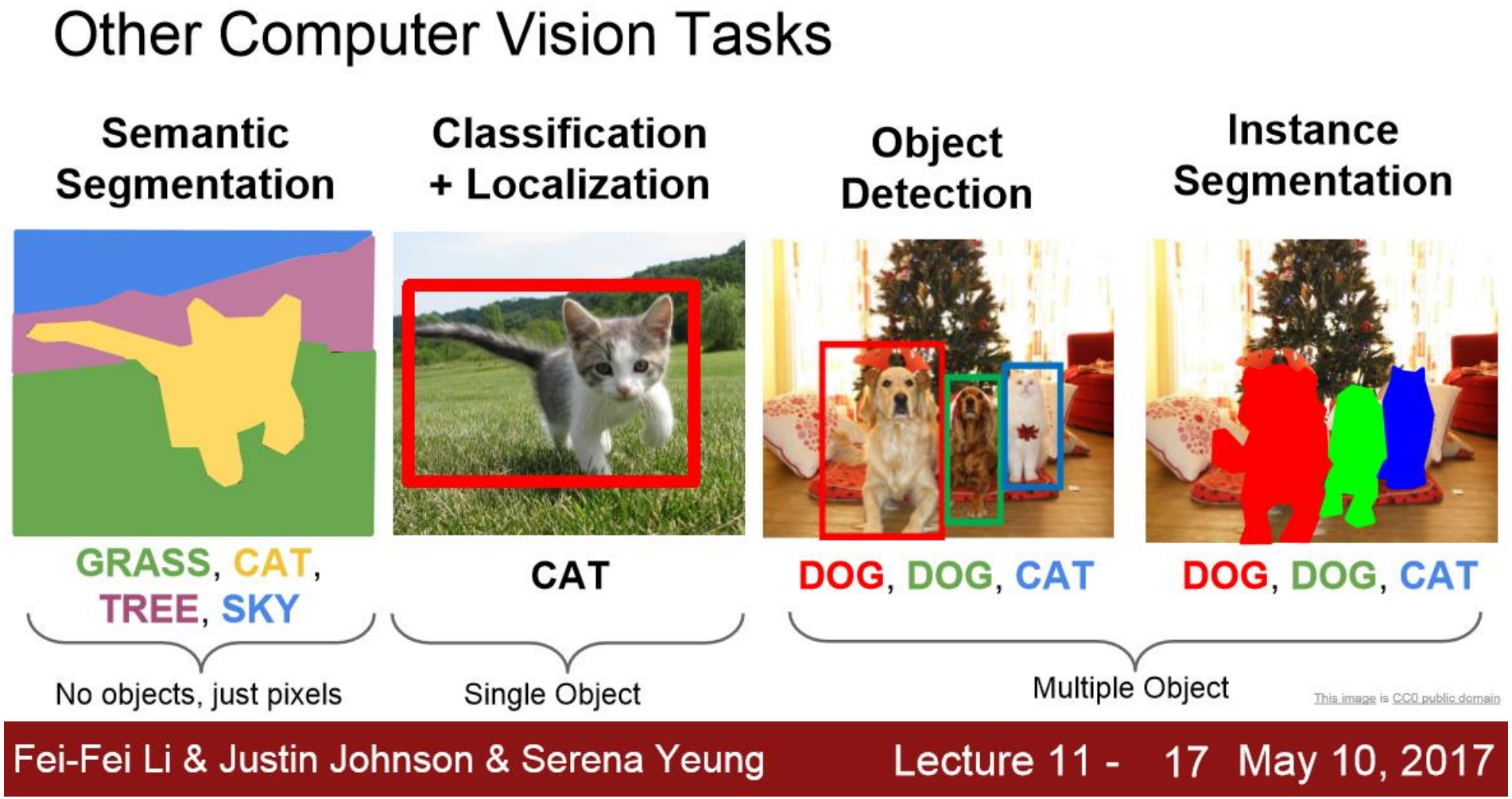
Computer visie

Wat zijn de grootste problemen bij computervisie?

Wat zijn de grootste problemen bij computervisie?

- Algoritmes zien individuele pixels ipv deelfiguren
  - Eigenlijk nog erger want ze zien maar 1 kleur
- Resolutie van huidige figuren is heel groot
  - Heel veel parameters/gewichten die getrained moeten worden
  - 1024 * 1024 figuur heeft per neuron en miljoen gewichten
- Heel veel data nodig om al deze gewichten te trainen

Toepassingen:



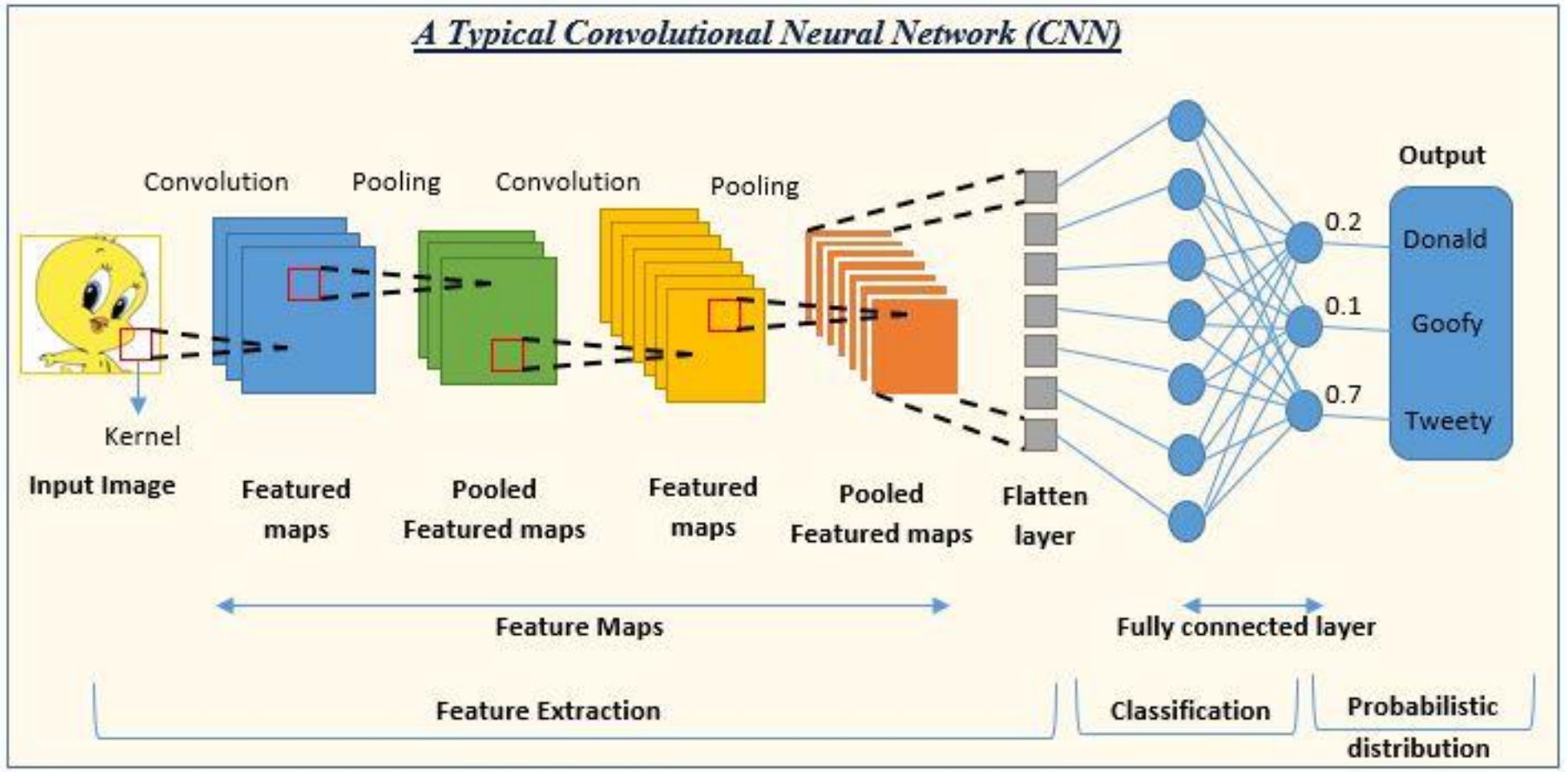
Convolutioneel neuraal netwerk of CNN

- Poging om computers beelden te laten zien zoals mensen het doen
  - Kijken naar nabijgelegen pixels om kleine objecten te herkennen (ogen, neus, oren, …)
  - Kijken naar deze nabijgelegen objecten om grotere zaken te herkennen (gezicht)
  - Herhaal het vorige
- Het aantal parameters in het neuraal netwerk reduceren
  - Efficiënter om te trainen
  - Minder data nodig voor een goed model op te bouwen

Convolutioneel neuraal netwerk of CNN

Nieuwe lagen: Convolutionele en Pooling

Standaard Neuraal netwerk



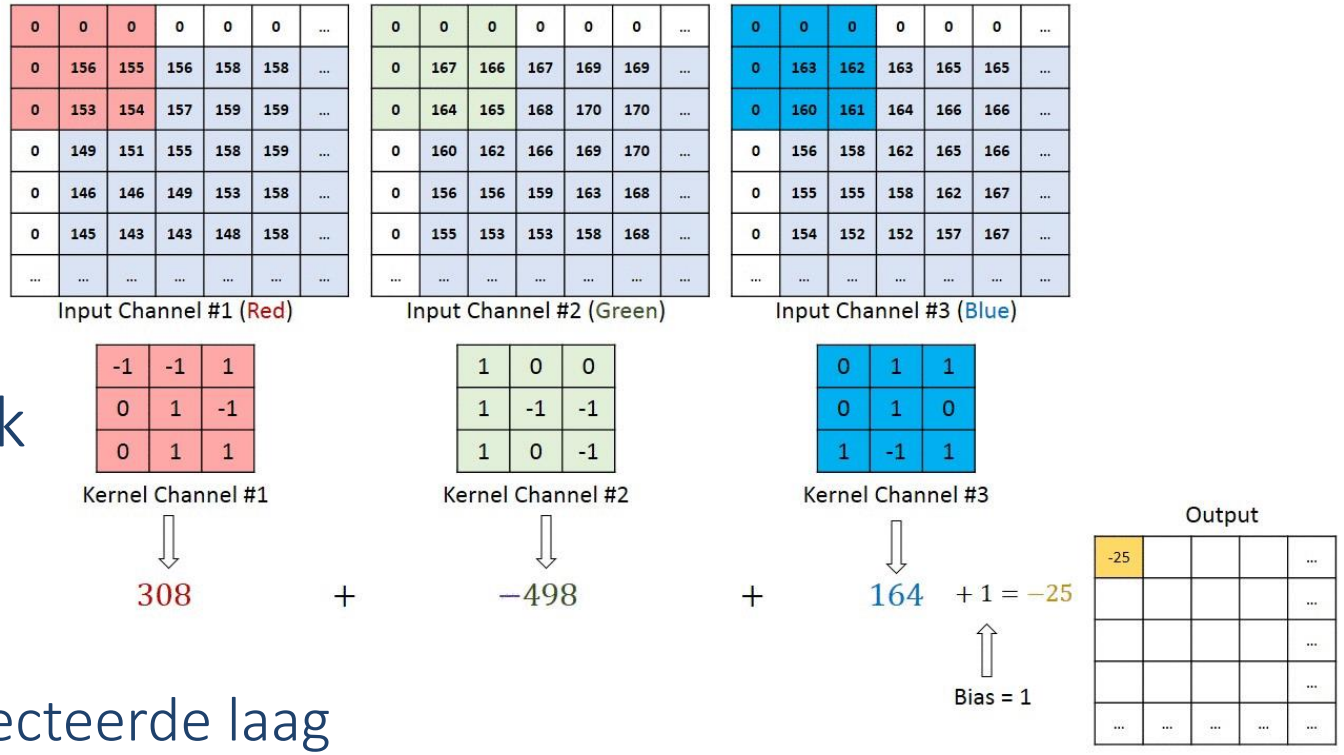
Convolutioneel neuraal netwerk of CNN

- Convolutionele laag
  - Zoek verband tussen nabijgelegen pixels
- Pooling laag
  - Reduceer de dimensies
- Standaard Neuraal netwerk
  - Kan 1 of meerdere lagen bevatten

- Goede bron met animaties: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

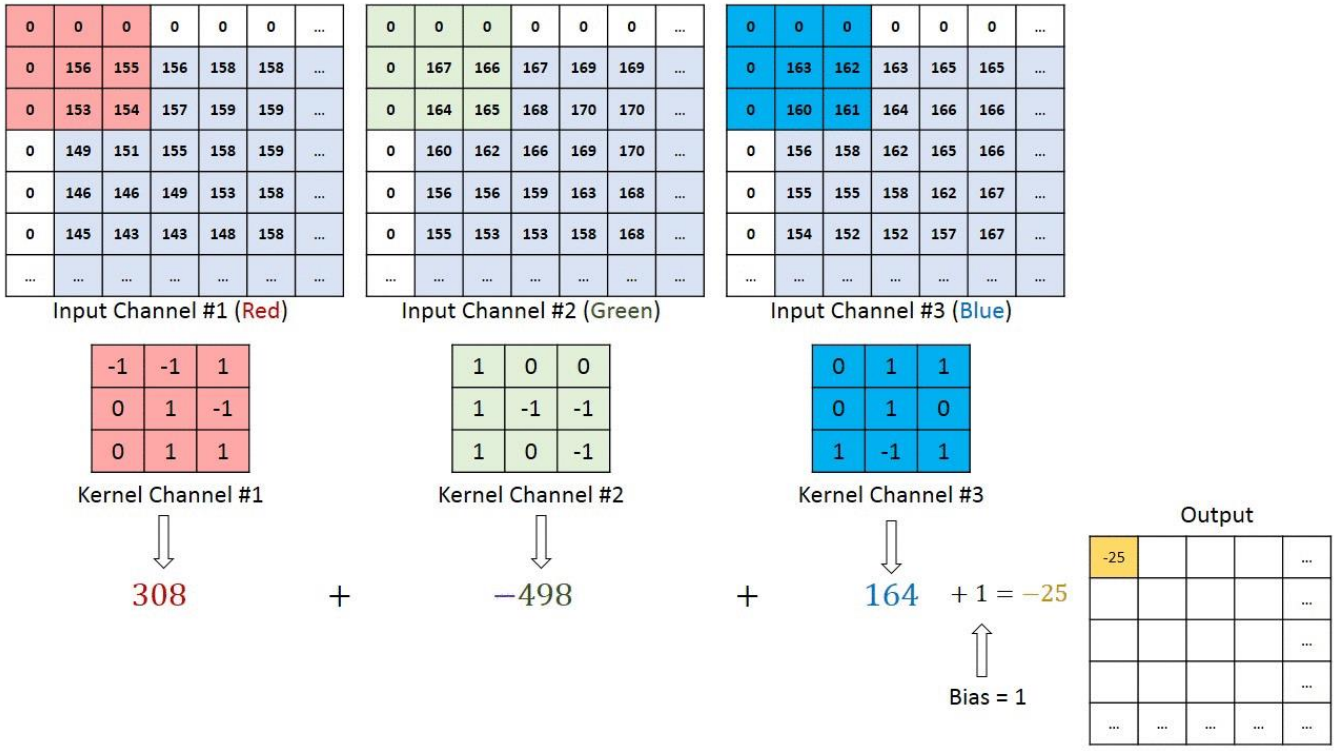
Convolutionele laag

- Pixel-waarden naar hogere niveau features omzetten
- Zet elke pixel om naar een combinatie van de omliggende pixels
  - 3x3 of 5x5 meestal
- Kernel = de matrix met gewichten
  - Worden getraind
  - Alle pixels met dezelfde gewichten
- Meerdere kernels per laag mogelijk
- In geval van 3x3
  - 27 gewichten * aantal kernels
  - Veel minder dan een volledig geconnecteerde laag



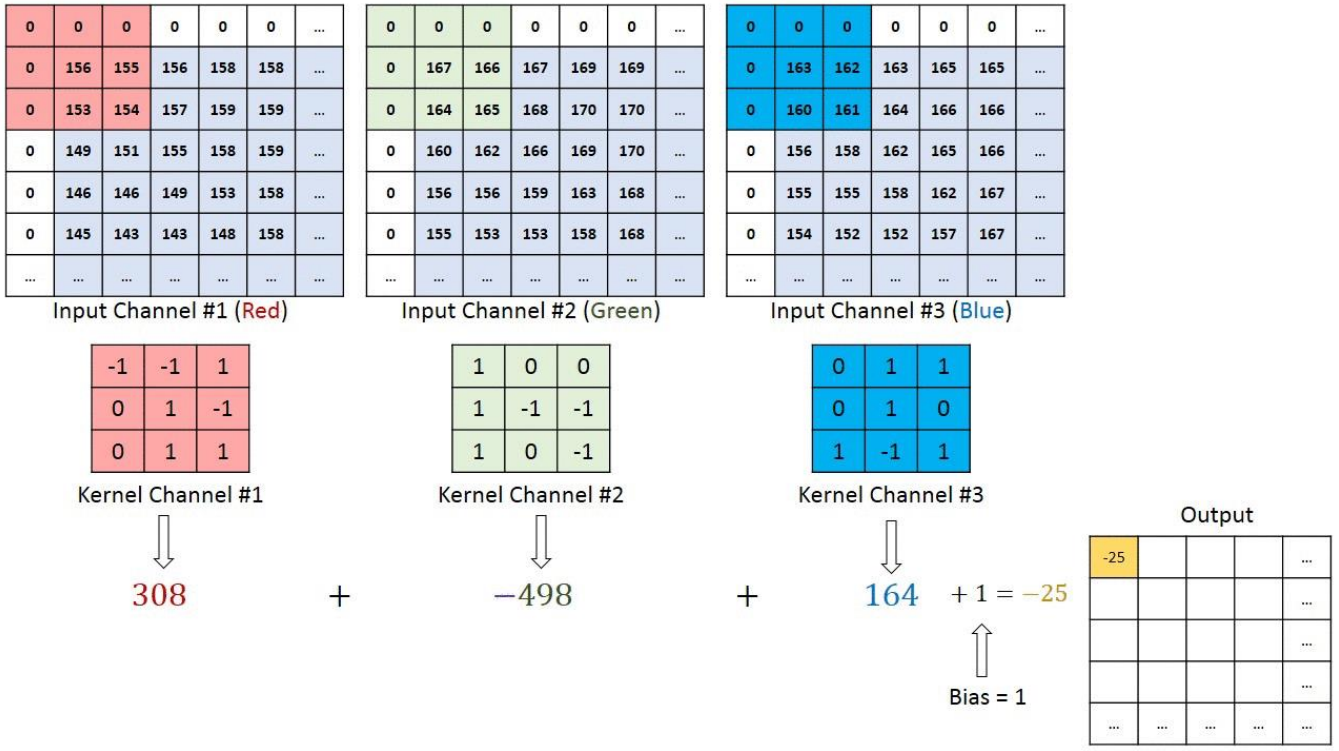
Convolutionele laag

- Twee mogelijke manieren van padding mogelijk
  - Padding: wat je moet doen aan de randen aangezien de pixels niet bestaan
  - Valid padding
    - Laat de randen weg
    - Resulteert in lagere dimensie
  - Same padding of zero padding
    - Voeg nullen toe (zoals hiernaast)
    - Dimensies blijven hetzelfde (same)



Convolutionele laag

- Stride
  - Het aantal pixels dat de kernel opschuift elke stap
  - Stapgrootte
    - Typisch 1 maar soms groter
  - Stride groter dan 1 resulteer in
    - Kleinere dimensie van outputs



Convolutionele laag: hyperparameters

- Dimensies van de kernels
- Type van padding
- Aantal kernels
- Stapgrootte - stride

Convolutionele laag: eindresultaat

- Convolutionele laag zoekt naar hogere level features
- Meedere convolutionele lagen kunnen gebruikt worden
- Meerdere kernels per laag nodig
- Dimensies van de figuren blijven gelijk
  - Aantal gewichten van het laatste neurale network neemt sterk toe
- Vergeet geen activatiefunctie uit te voeren na elke convolutionele laag
  - Vaak ReLu gebruikt: alle negatieve waarden worden 0

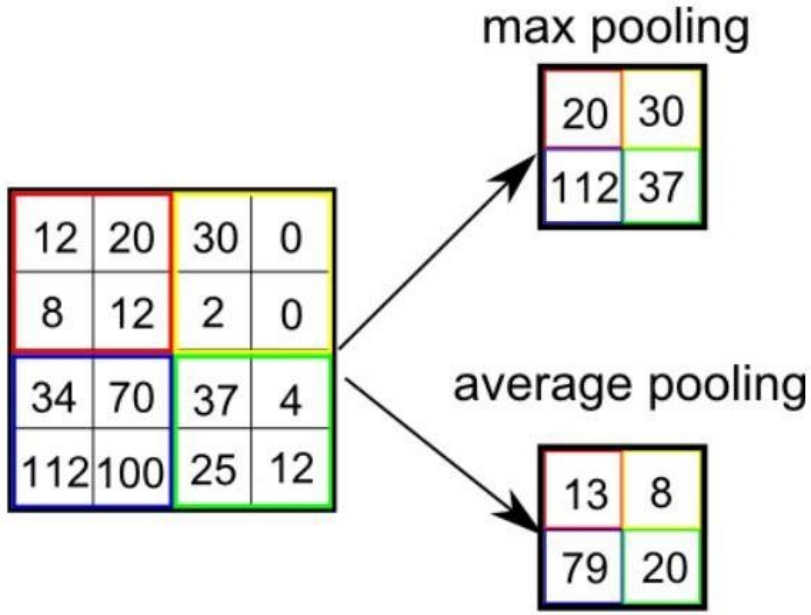
Oefening convolutionele laag

- Input: Grijsbeelden van 48 x 48

- Wat zijn de outputdimensies indien de kernel
  - 3x3, zero padding, stride 1 ->
  - 3x3, zero padding, stride 2 ->
  - 5x5, valid padding, stride 1 ->
  - 7x7, valid padding, stride 3 ->

Pooling layer

- Dimensionality reduction
  - Performantie kost van convolutionele laag compenseren
  - Bepaald door de “stride” van de laag
- Dominante features detecteren
- Hierbij wordt gebruik gemaakt van een filter/matrix
  - Geen gewichten om te trainen



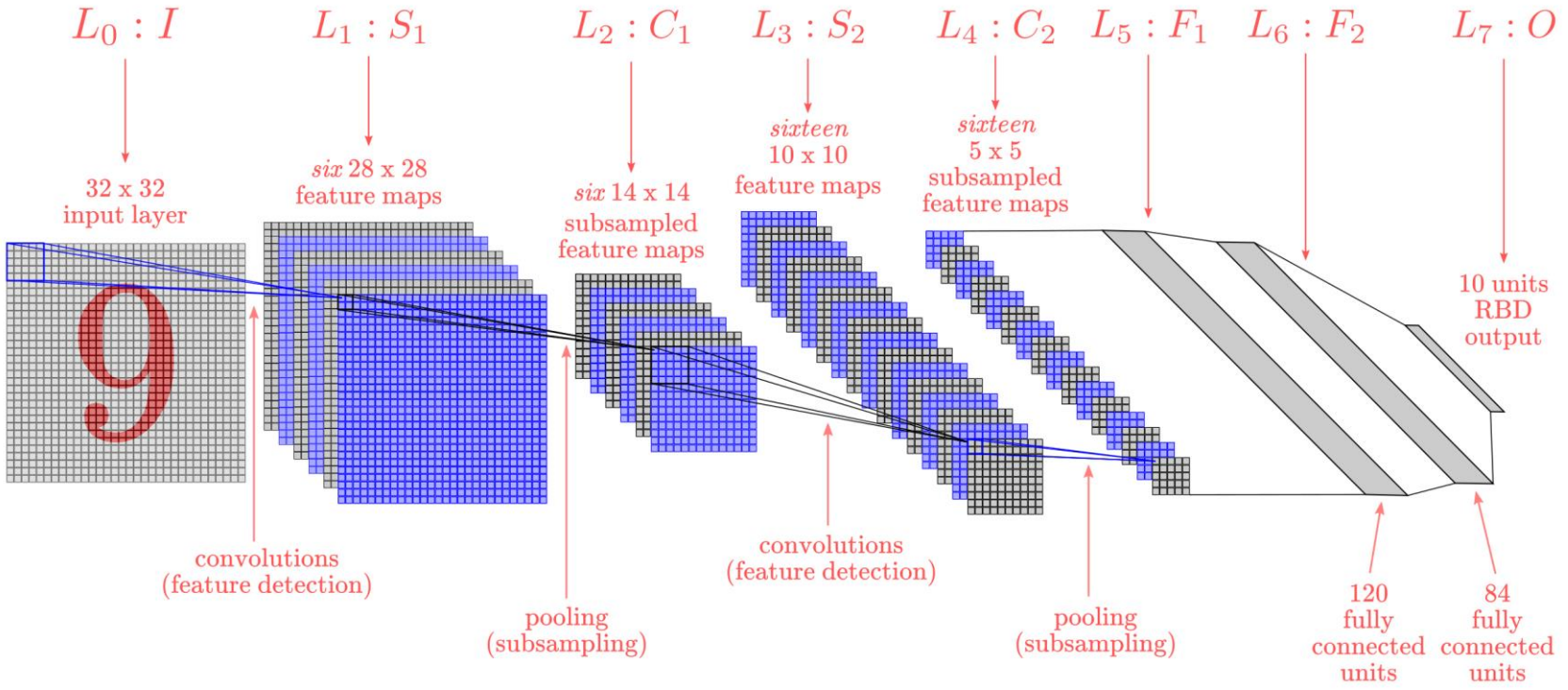
Pooling layer: Hyperparameters

- Dimensies en stride
  - Hoe groot is de filter waarin we gaan kijken en hoe veel pixels schuiven we de filter op elke stap (stride)
  - Bepaalt de reductie in dimensies
  - Typisch: 2x2 met een stride van 2 wat resulteert in een halvering van de dimensies
- Type pooling
  - Max pooling
    - Onderdrukt ruis en typisch beter
  - Average pooling

	Layer Type	Kernel Attribute	Number of Filters	Feature Map Size
	Image Input Layer			115 × 51 × 3
L1	Convolutional Layer	7 × 7 × 3, stride 2, no padding	96	55 × 23 × 96
	ReLU Layer			
	CCN Layer			
	Max Pooling	3 × 3, stride 2, no padding		27 × 11 × 96
L2	Convolutional Layer	5 × 5 × 96, stride 1, 2 × 2 zero padding	128	27 × 11 × 128
	ReLU Layer			
	CCN Layer			
	Max Pooling	3 × 3, stride 2, no padding		13 × 5 × 128
L3	Convolutional Layer	3 × 3 × 128, stride 1, 1 × 1 zero padding	256	13 × 5 × 256
	ReLU Layer			
L4	Convolutional Layer	3 × 3 × 256, stride 1, 1 × 1 zero padding	256	13 × 5 × 256
	ReLU Layer			
L5	Convolutional Layer	3 × 3 × 256, stride 1, 1 × 1 zero padding	128	13 × 5 × 128
	ReLU Layer			
	Max Pooling	3 × 3, stride 2, no padding		6 × 2 × 128
F1	Fully Connected Layer			4096
	ReLU Layer			
F2	Fully Connected Layer			2048
	ReLU Layer			
	Dropout			
F3	Fully Connected Layer			5
	Softmax Layer			

Veel gebruikte CNN's

- LeNet
  - Voorgesteld in 1989
  - Om handgeschreven cijfers te herkennen
    - Toegepast voor postcodes
  - Kernels van 5x5
  - Hoe van 6 naar 16?
    - Kernel is 6x5x5



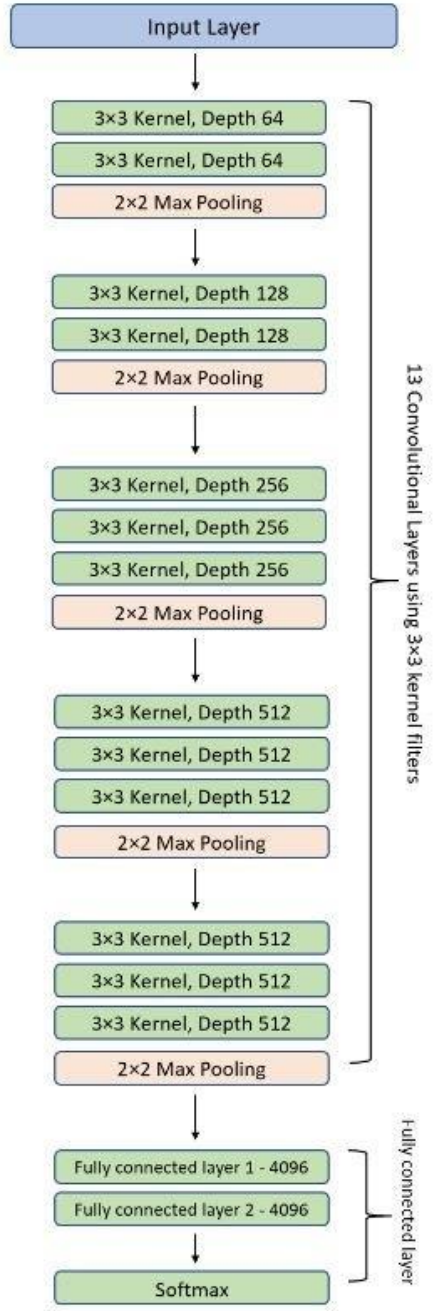
Veel gebruikte CNN's

- AlexNet
  - Heel invloedrijk concept geweest binnen deep learning en impact van GPU's

Veel gebruikte CNN's

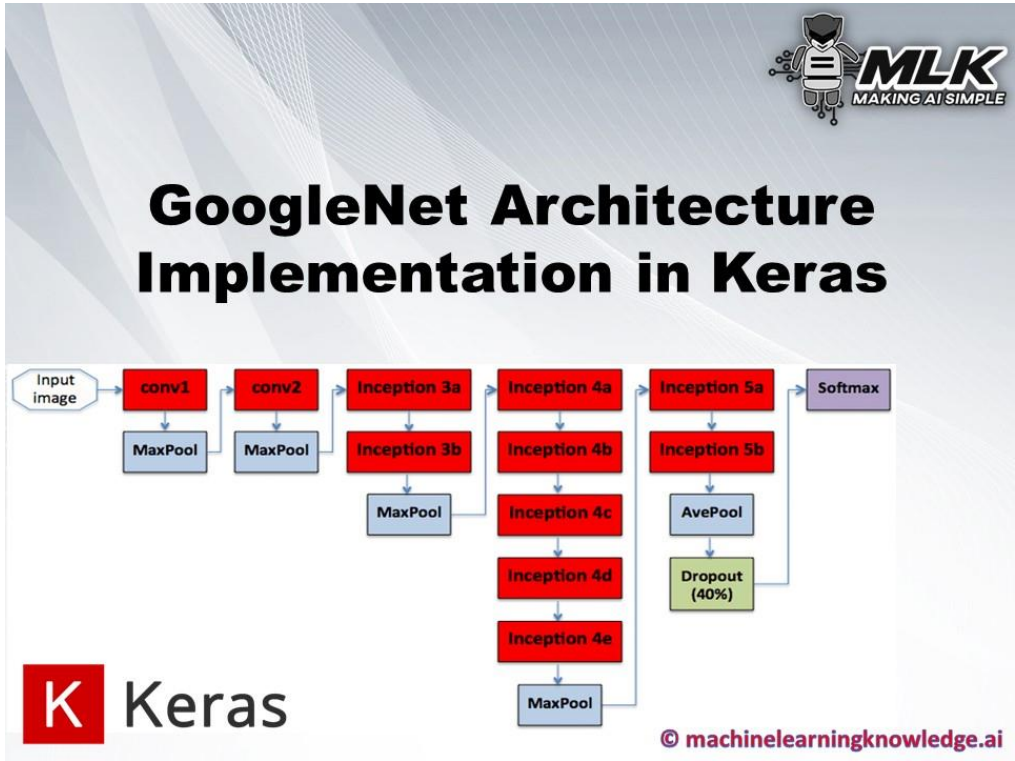
- VGGNet
  - Visual geometry group
  - Deep CNN
    - Deep = veel convolutionele lagen
    - 16 tot 19 lagen
  - Meer lagen resulteert in betere performantie
    - Niet altijd het geval bij standaard neurale netwerken

- Bron: https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/



Veel gebruikte CNN's

- GoogLeNet
  - 27 lagen in het model
  - Heel snel de dimensies reduceren
    - Grote kernel 7x7 om niet te veel informatie te verliezen
  - Inception lagen bevatten ook convolutionele lagen
  - Kan 1000 klassen herkennen in een figuur



Veel gebruikte CNN's

- ResNet
  - Residual neural netwerk
  - Maakt gebruik van identity shortcut connections
    - Laat de output ook een aantal lagen overslaan en geef het opnieuw als input
    - Lost gedeeltelijk het vanishing gradient probleem op
      - Meerdere lagen zouden steeds beter moeten zijn
- Basis voor een reeks state-of-the-art modellen in computervisie