Building an AI

Slides

Code

AI, mathematically

Part 1

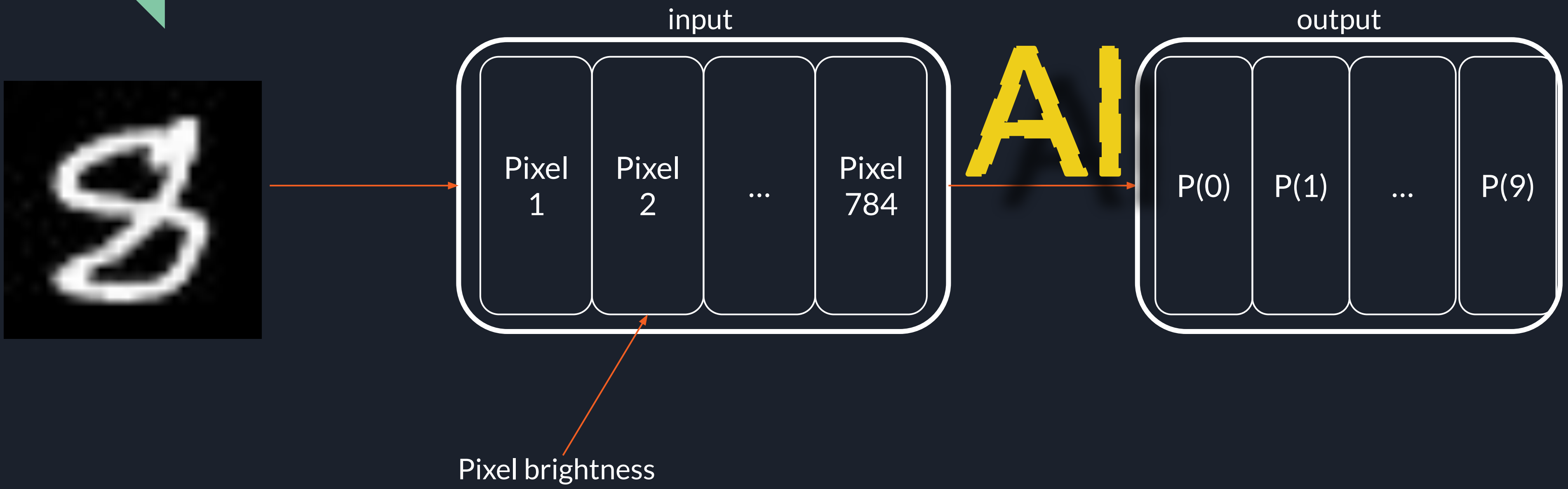
Artificial Intelligence

- Definition from *Britannica:*
  - Ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings.
  - Vague! But ok…

- Why use an AI?
  - Information we give to the computer is always different (e.g. speech recognition: different accents and emotions) -> Intelligence to know how to understand it.
  - A lot of work to do by hand. AI can be trained, like a human!

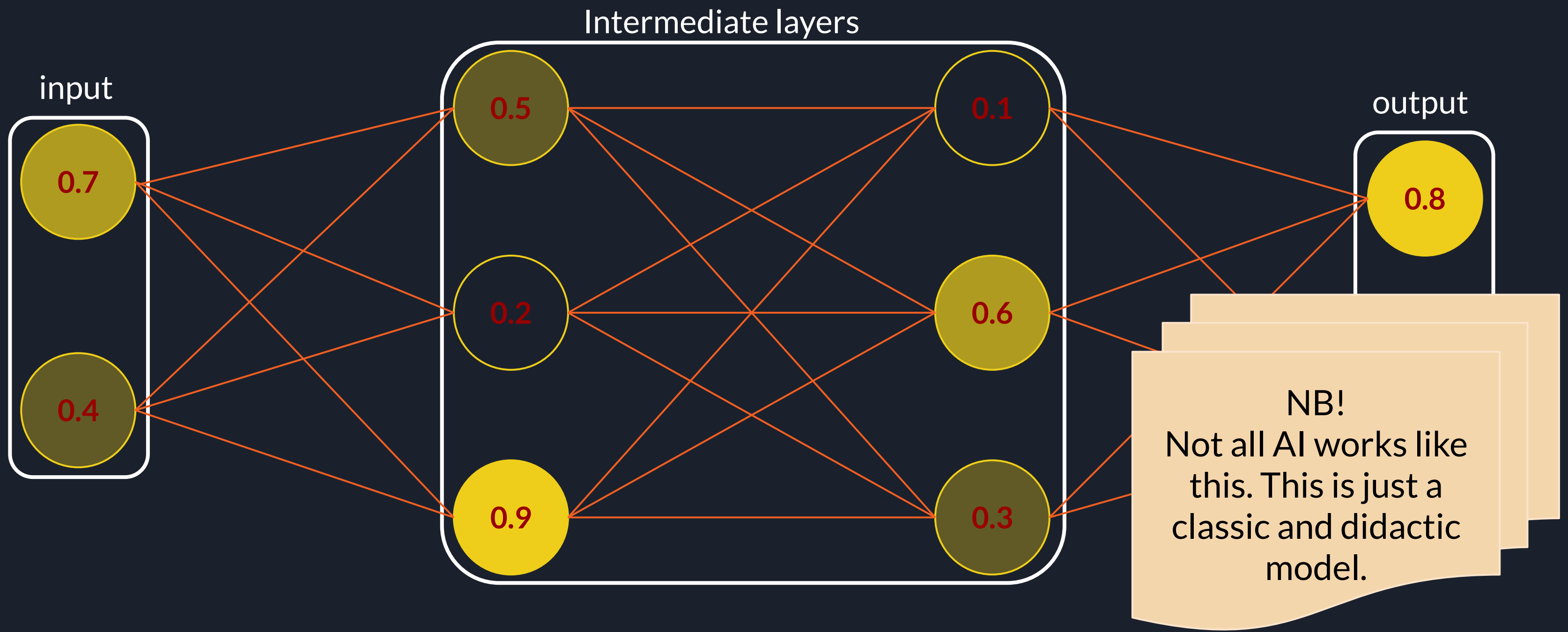
An AI model

- We want to know which number is in the image

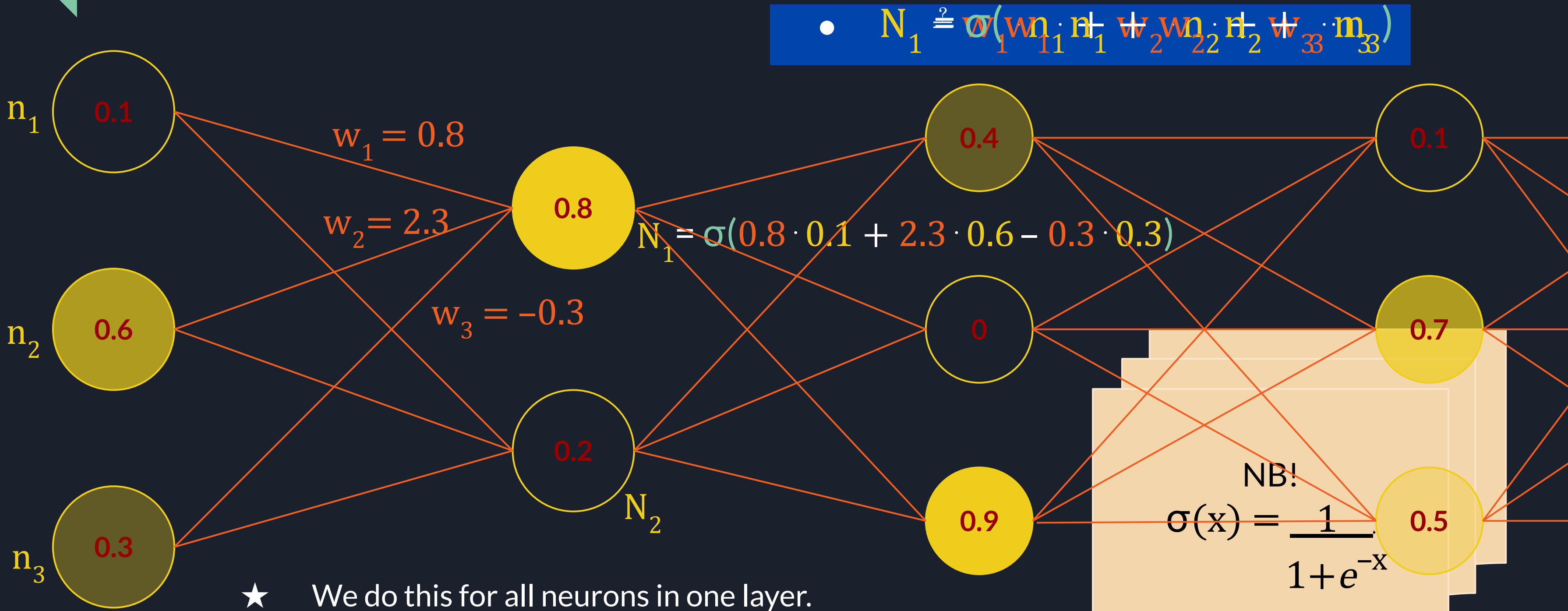


The neuron

- Functional unit of AI. Like a neuron for our intelligence.
  - Previous layers determine the values of the subsequent ones!



How are values decided?

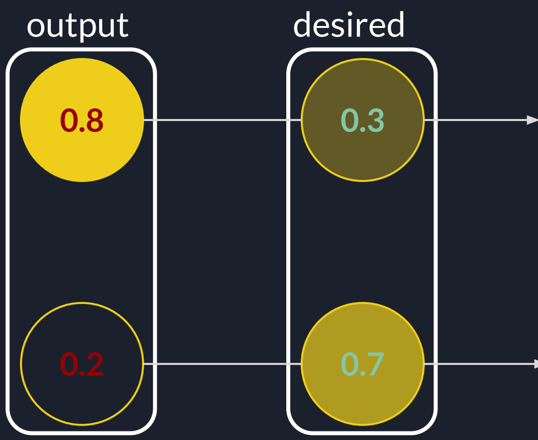


★ We do this for all neurons in one layer.

★ Then, we move on to the next one, until we reach the output

The cost function

- We need a function that evaluates the accuracy of our AI so we can train it.
- We call this function the *cost function* $C = C(w_1, w_2, w_3, \dots, w_{n-1}, w_n)$



$(0.8 - 0.3)^2$

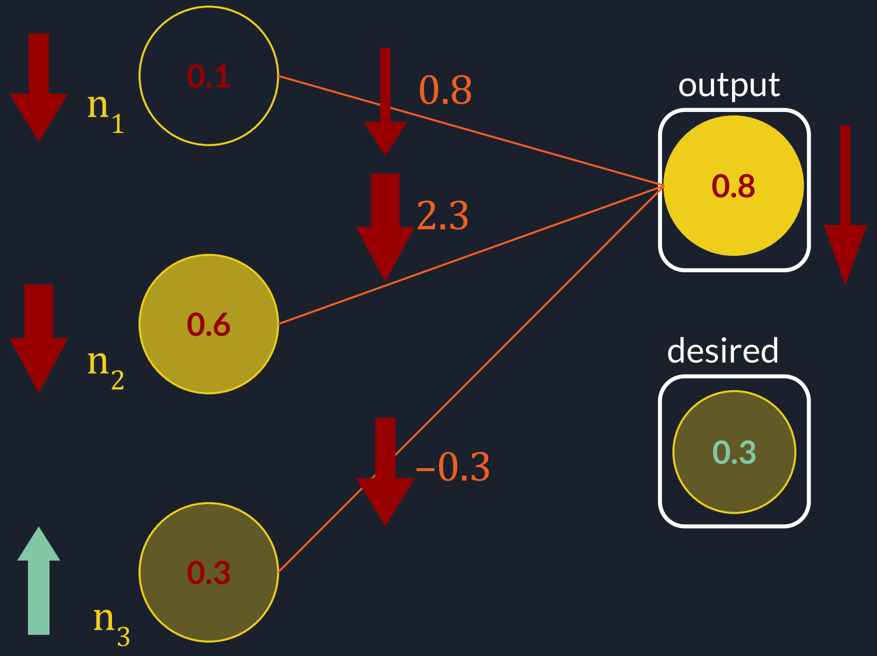
$(0.2 - 0.7)^2$

$$C(\{w_i\}) = (0.8 - 0.3)^2 + (0.2 - 0.7)^2$$

- Goal: Find the weights that minimize the cost function!

Training an AI

- For artificial intelligence to be truly intelligent, it must know how to learn from its own mistakes!
- As we saw, this is the same thing as minimizing the *cost function* → minimize the differences between the desired and the output



$$N_1 = \sigma(w_1 \cdot n_1 + w_2 \cdot n_2 + w_3 \cdot n_3)$$

- To change a previous neuron, we need to compute the "opinion" of all other neurons of the layer before.

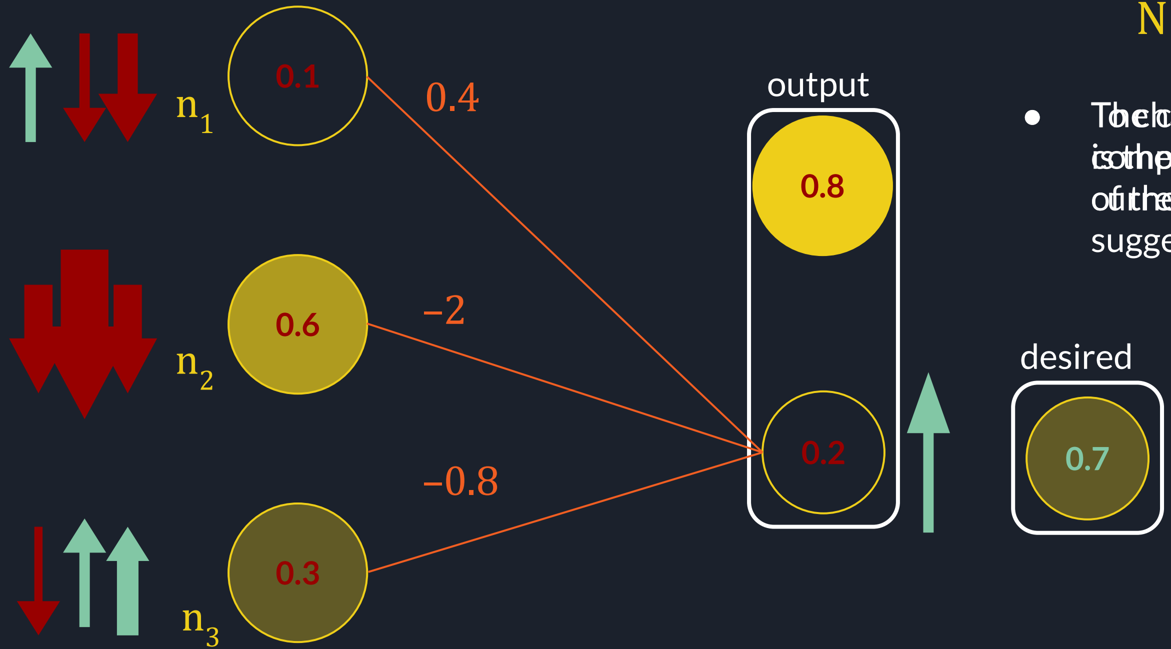
Manners:

1. Decrease the weights proportionally to the previous neurons
2. Decrease the previous neurons proportionally to the weights
   a. But we can't do this directly

Training an AI

- For artificial intelligence to be truly intelligent, it must know how to learn from its own mistakes!
- As we saw, this is the same thing as minimizing the cost function → minimize the differences between the desired and the output

$$N_1 = \sigma(w_1 \cdot n_1 + w_2 \cdot n_2 + w_3 \cdot n_3)$$



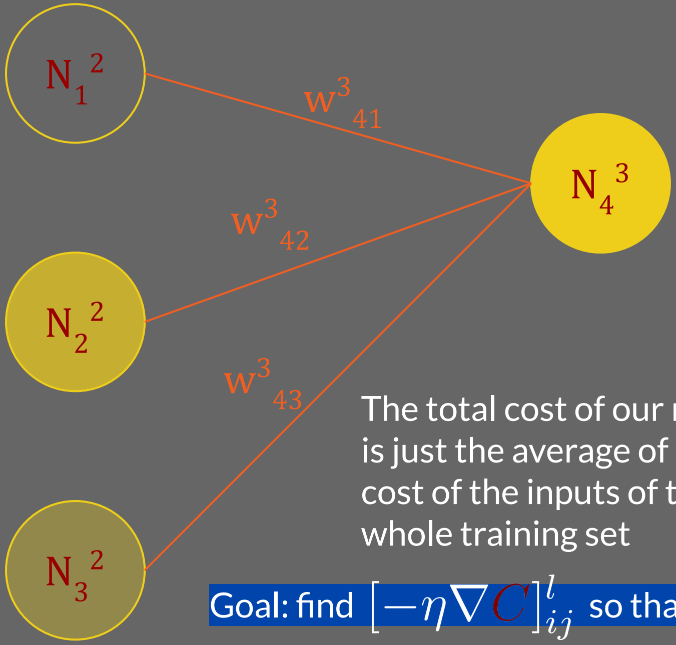
- The change that the previous neurons will want is the composition of the changes that the neurons of the layer before (in this case, those of the output) suggested.
- We proceed recursively, going back layer by layer to satisfy the wishes of the previous neurons!

The following slides are extremely math-heavy, with a lot of index chasing. If you feel overwhelmed, don't worry! We will only write with numbers what we have seen in the previous slides.

It is not really necessary for the coding part.

Training an AI: the calculus

- We want to know the quickest path to minimize the *cost function*.
- Opposite to the quickest path to maximize it → opposite to the gradient of the cost function!



$$N_i^l = \sigma\left(\sum_{j=0}^{\text{all}} w_{ij}^l N_j^{l-1}\right)$$

$$C(\{w_{ij}^l\}) = \frac{1}{n}\sum_{k=1}^{n} C_k(\{w_{ij}^l\}, \{N_i^l\})$$

The total cost of our model is just the average of the cost of the inputs of the whole training set

Each training set consists o
n sets of inputs

NB!
Ideally, our training set = whole train data. However, that is costly, so we train the AI using random batches of smaller training sets

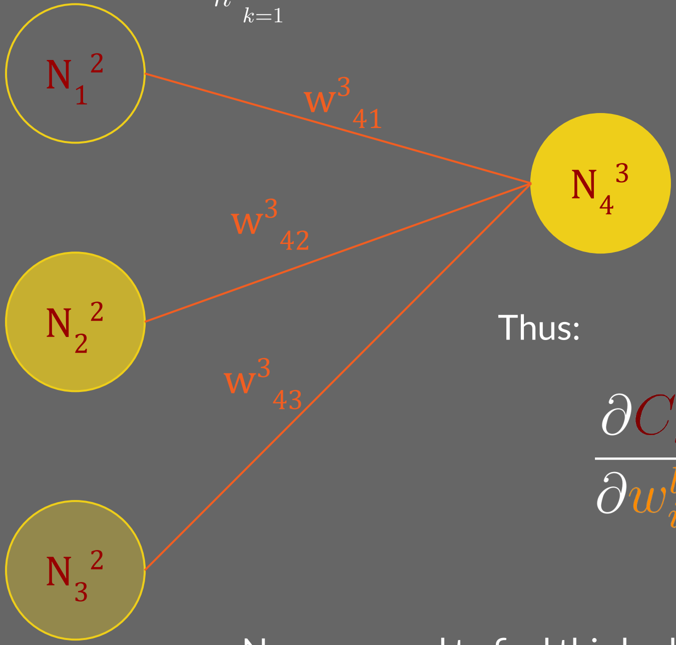
Goal: find $[-\eta\nabla C]_{ij}^l$ so that we can set $w_{ij}^l \rightarrow w_{ij}^l + [-\eta$

Because the gradient is linear, we need only care about C_k

Training an AI: calculating the gradient

$$N_i^l = \sigma\left(\sum_{j=0}^{\text{all}} w_{ij}^l N_j^{l-1}\right) = \sigma(\blacksquare)$$

$$C(\{w_{ij}^l\}) = \frac{1}{n}\sum_{k=1}^{n} C_k(\{w_{ij}^l\}, \{N_i^l\})$$



$$[-\eta\nabla C_k]_{ij}^l = \frac{\partial C_k}{\partial w_{ij}^l} = \frac{\partial C_k}{\partial N_i^l}\frac{\partial N_i^l}{\partial w_{ij}^l}$$

But

$$\frac{\partial N_i^l}{\partial w_{ij}^l} = \frac{\partial \sigma(\blacksquare)}{\partial \blacksquare}\frac{\partial \blacksquare}{\partial w_{ij}^l} = \sigma'(\blacksquare) N_j^{l-1}$$

Thus:

$$\frac{\partial C_k}{\partial w_{ij}^l} = \frac{\partial C_k}{\partial N_i^l}\sigma'\left(\sum_{j=0}^{\text{all}} w_{ij}^l N_j^{l-1}\right) N_i^{l-1}$$

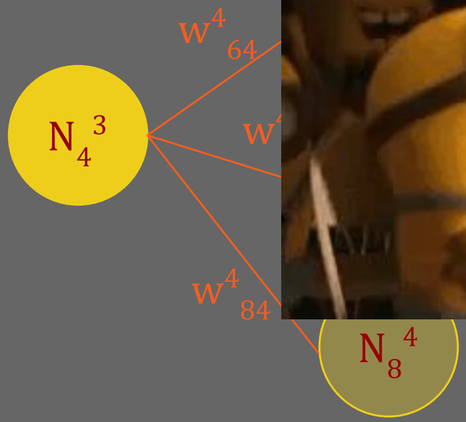
Now we need to find this bad boy; then we will be done!

Training an AI: calculating the gradient

$$N_i^l = \sigma\Big(\sum_{j=0}^{\text{all}} w_{ij}^l N_j^{l-1}\Big) = \sigma(\blacksquare)$$

$$C(\{w_{ij}^l\}) = \frac{1}{n}\sum_{k=1}^{n} C_k(\{w_{ij}^l\}, \{N_i^l\})$$

$$\frac{\partial C_k}{\partial N_i^l} = \sum^{\text{all}} \frac{\partial C_k}{\partial N_j^{l+1}} \frac{\partial N_j^{l+1}}{\partial N_i^l}$$



$$\blacksquare \quad \ldots N_i^l = \sigma'(\blacksquare) w_{ji}^{l+1}$$

$$\ldots \partial N_i^l = \sum_{j=1} \ldots \partial N_j^{l+1} \; \sigma'\Big(\sum_{k=0} w_{jk}^{l+1} N_k^l\Big) w_{ji}^{l+1}$$

$$\frac{\partial C_k}{\partial w_{ij}^l} = \frac{\partial C_k}{\partial N_i^l}\sigma'\Big(\sum_{j=0}^{\text{all}} w_{ij}^l N_j^{l-1}\Big) N_i^{l-1}$$

It is a recursive formula!

We set $\partial C_k/\partial N_j^L = 2(N_{\text{actual, j}} - N_{\text{expected, j}})$ as our base case.

A few remarks regarding the math

1. It can be written in a much better way in matrix form:

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{(BP4)}$$

2. Because of that, it is highly parallelizable (i.e. the computer can theoretically do much of the multiplication and addition simultaneously) → GPUs are extremely useful!

3. Nevertheless, it remains quite processing-intensive (number of parameters can skyrocket!)

Luckily, we don't really need to know all the math in order to use AI!

AI,
in practice

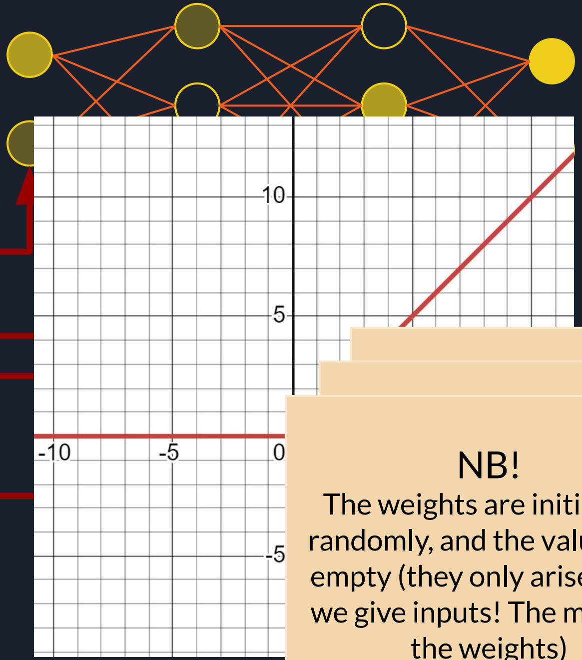
Part 2

Building an AI model: the neurons

- We shall use TensorFlow (Python library) with Keras (its API) to code our model

```
[6]  model = keras.models.Sequential()
     #we shall use sequential layers, as seen in the slides!
```

```
[7]  model.add(keras.layers.Flatten(input_shape= (28, 28)))
     #the images are 28x28 matrices. Here we flatten them and
     model.add(keras.layers.Dense(128, activation= 'relu'))
     model.add(keras.layers.Dense(128, activation= 'relu'))
     #here we set the intermediate layers, each with 128 neuro
     #"dense" means the neurons are arranged and behaved like
     model.add(keras.layers.Dense(10, activation= 'softmax'))
     #last layer, one neuron for each digit probability. softm
```



NB!
The weights are initialised randomly, and the values are empty (they only arise when we give inputs! The model is the weights)

Building an AI model: the training settings

- These settings will be taken into account when we actually train the model

```
model.compile(optimizer= 'adam',
              loss= 'sparse_categorical_crossentropy',
              metrics= ['accuracy'])
```

This sets the backpropagation algorithm. “Adam” is a modified version of the gradient descent that has been shown.

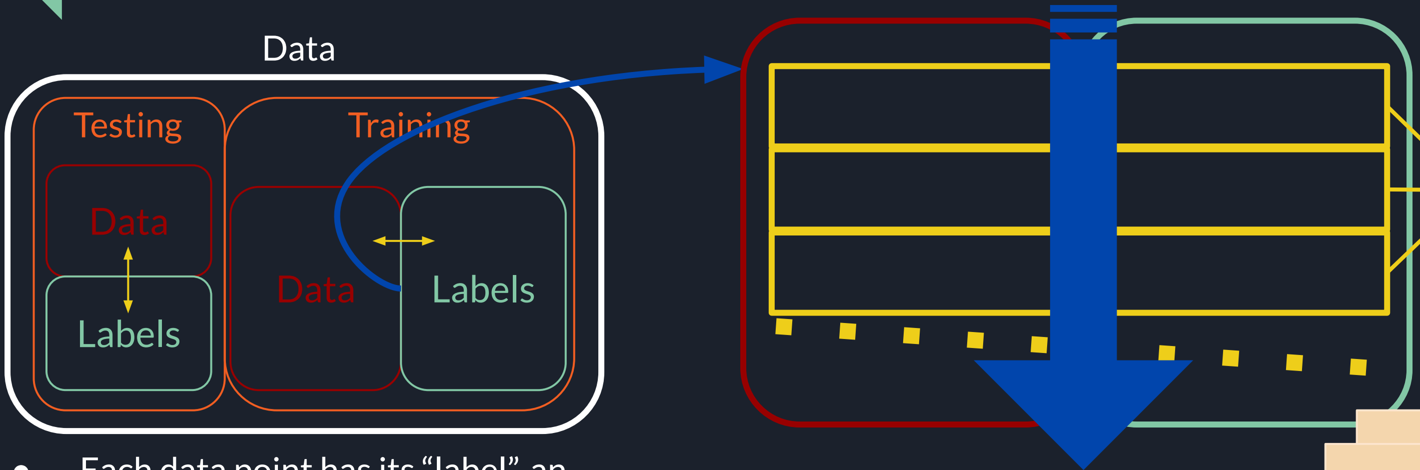
This sets the cost function. Unlike the “sum of squares” we used, this is more appropriate to our kind of data (categorical variables)

This sets the metrics that will be shown to us. Of course, we are interested in the model’s accuracy!

NB!
The metric does not interfere with the training process. It is only shown to us!

Building an AI model: organizing the data

- Our amount of data is huge, so we use fractions of it



We divide our training set into random batches. Our model is updated after each batch.

- Each data point has its “label”, an answer key
- The AI (or the researcher) checks the model’s prediction for the labels against the true labels in order to train (or tune) the model

A run through the entire training dataset is called an epoch. We can have many of them until the model is finishe

NB!
We need to strike a balance between training time, model size/complexity, and accuracy; all while avoiding overfitting!

Building an AI model: getting the data

- We need to format our data in order to have the model use it

```
mnist = keras.datasets.mnist
(train_data, train_labels), (test_data, test_labels) = mnist.load_data()
#MNIST has 70,000 images of digits.
#60,000 of them is to training, 10,000 to testing.
```

Luckily, Keras already has the MNIST dataset formatted!

```
train_data = keras.utils.normalize(train_data, axis= 1)
test_data = keras.utils.normalize(test_data, axis= 1)
#normalizing the data (the axis argument has to do with how the data is s
```

The only thing we need to do is to normalize the pixel brightness values so that they are consistent

Building an AI model: training the model!

```
model.fit(train_data, train_labels, batch_size=20, epochs=3)
#here we train our model
```

(quite the self-explanatory methods)

```
loss, accuracy = model.evaluate(test_data, test_labels)
#evaluating our model
```

The rest of the code is housekeeping and
seeing which images were poorly predicted

Your turn!

- Modify the code to achieve the best possible accuracy! tinyurl.com/sigmaMNIST
    - What happens when you change the number of layers? How about the number of neurons? And the normalizing function?
    - How about changing the optimizer?
    - Can changes in batch size effect changes in the resulting model? Does a model that went through more epochs perform better?
- BEWARE: is training time taking too long? How about evaluation time?
- Which images is the model getting wrong? Do different models have different wrong guesses?
- Would you have guessed better than the model?

Available optimizers

- SGD
- RMSprop
- Adam
- AdamW
- Adadelta
- Adagrad
- Adamax
- Adafactor
- Nadam
- Ftrl
- Lion
- Loss Scale Optimizer

Some may not work with our model :/

Available activations

relu function

sigmoid function

softmax function

softplus function

softsign function

tanh function

selu function

elu function

exponential function

leaky_relu function

relu6 function

silu function

hard_silu function

gelu function

hard_sigmoid function

linear function

mish function

log_softmax function

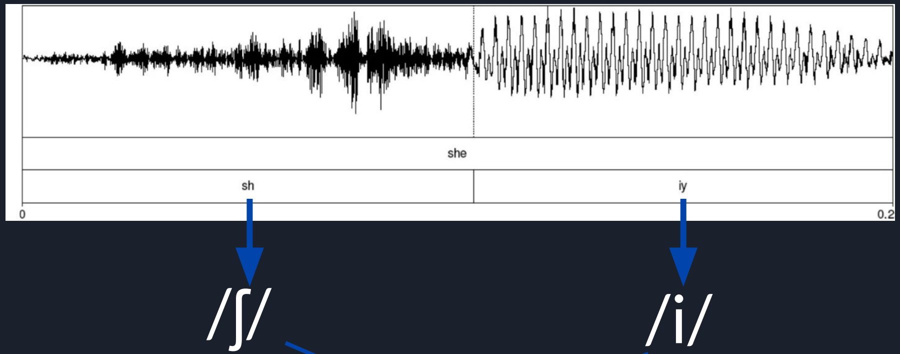
Same AI, different data

Extra section

How we speak

- The building blocks of words are phonemes (little sound pieces)

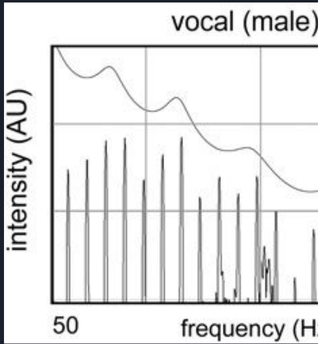
  - Basic idea: identify the phonemes in an audio file → identify the words based on the phonemes

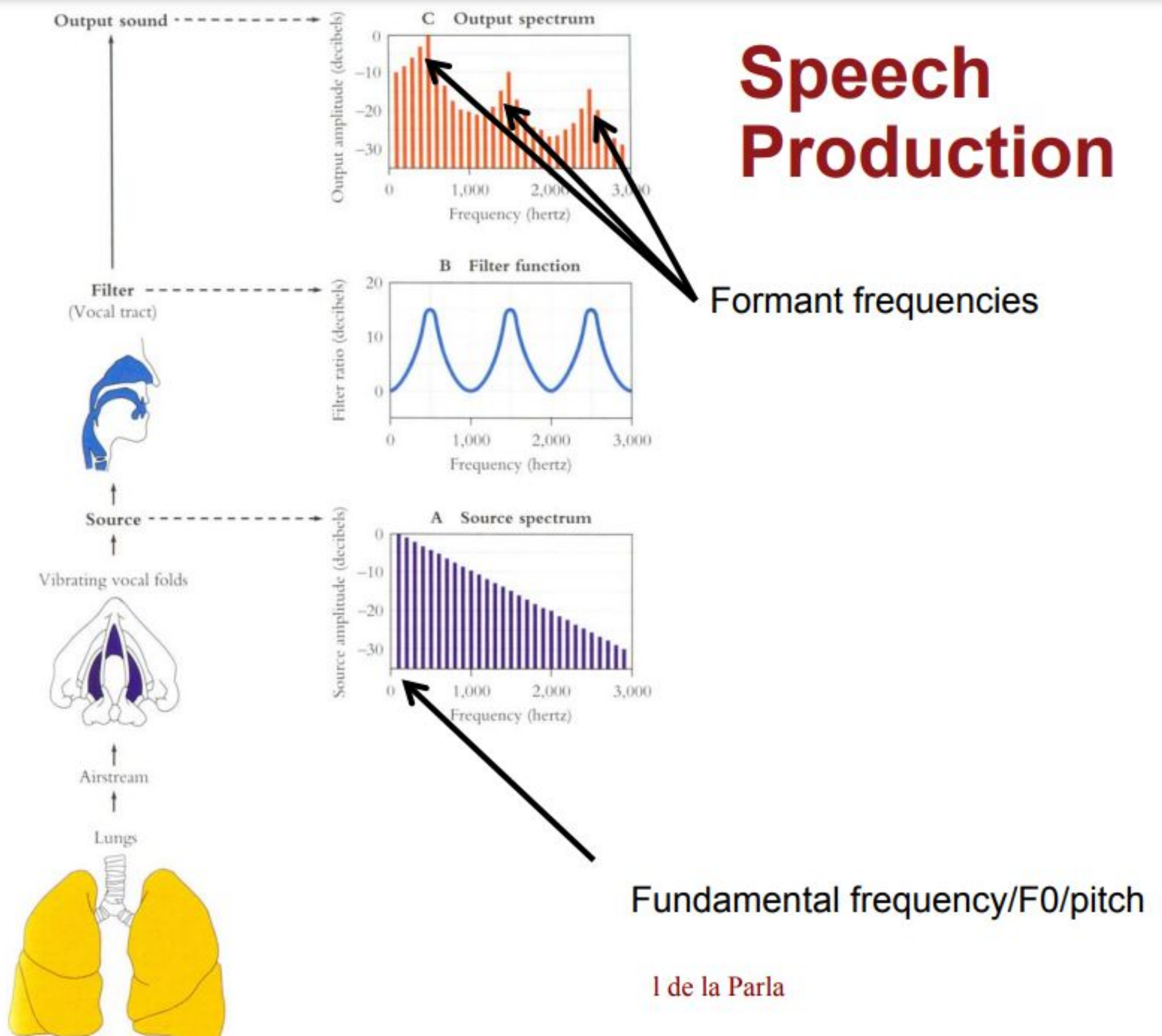


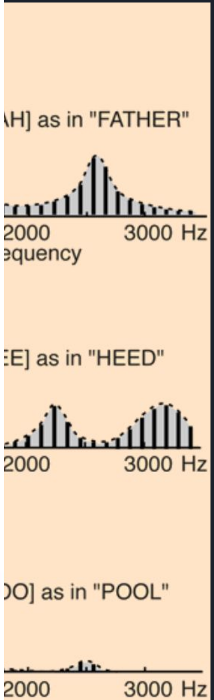
/ˈʃi/ = “she”!

Ph

•

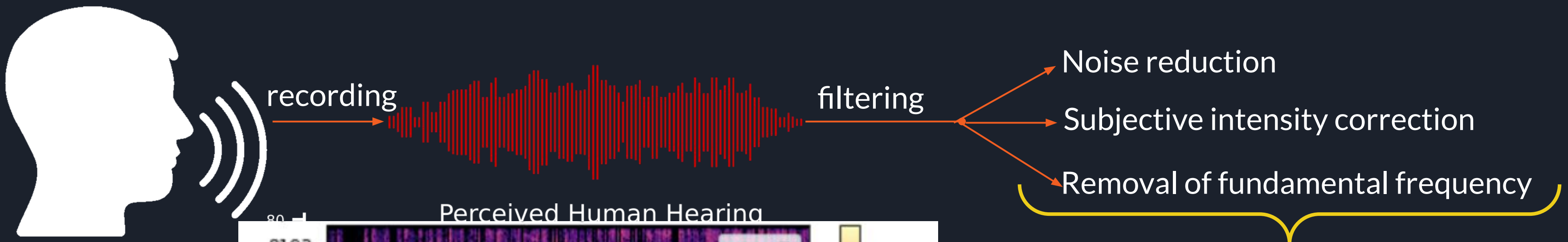






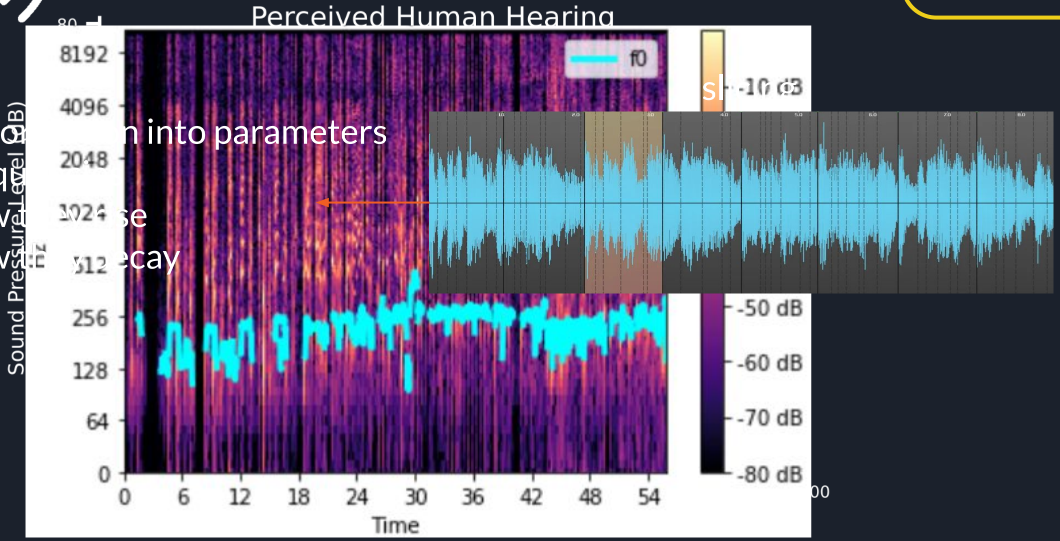
- Key idea: we recog
phonemes in our a

Extracting the audio



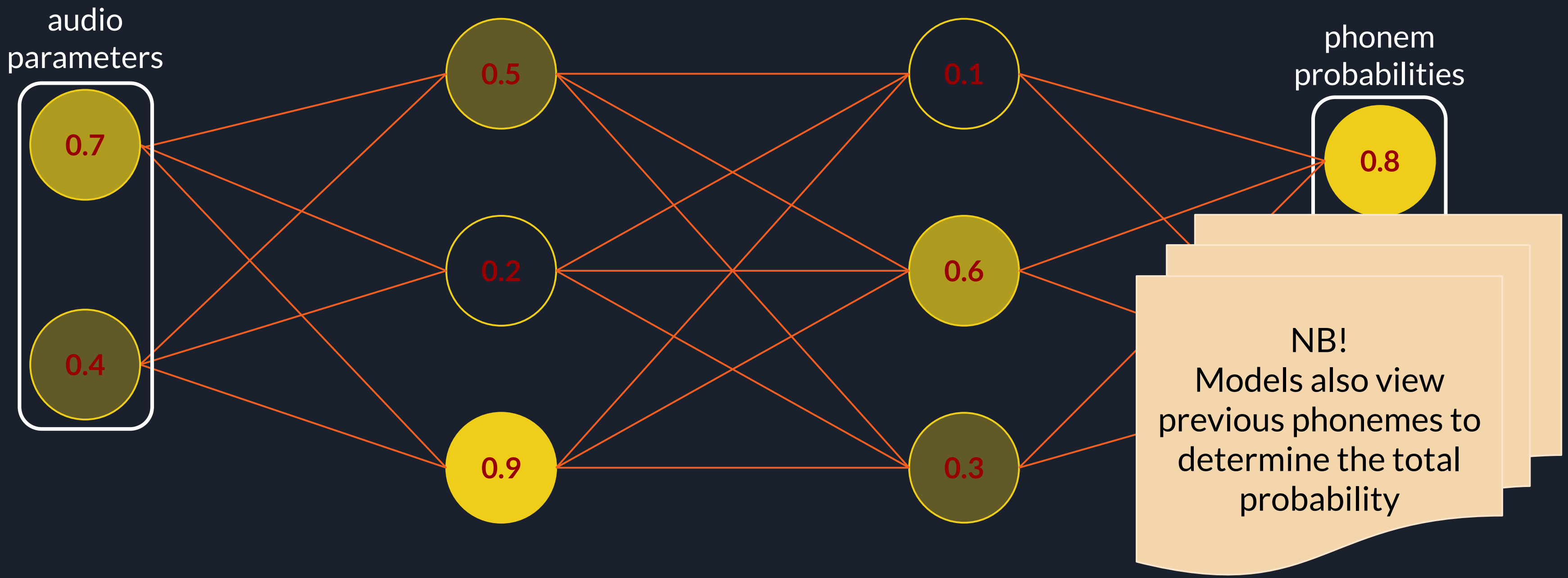
Extract info... n into parameters

- Freq...
- How t... se
- How t... ecay



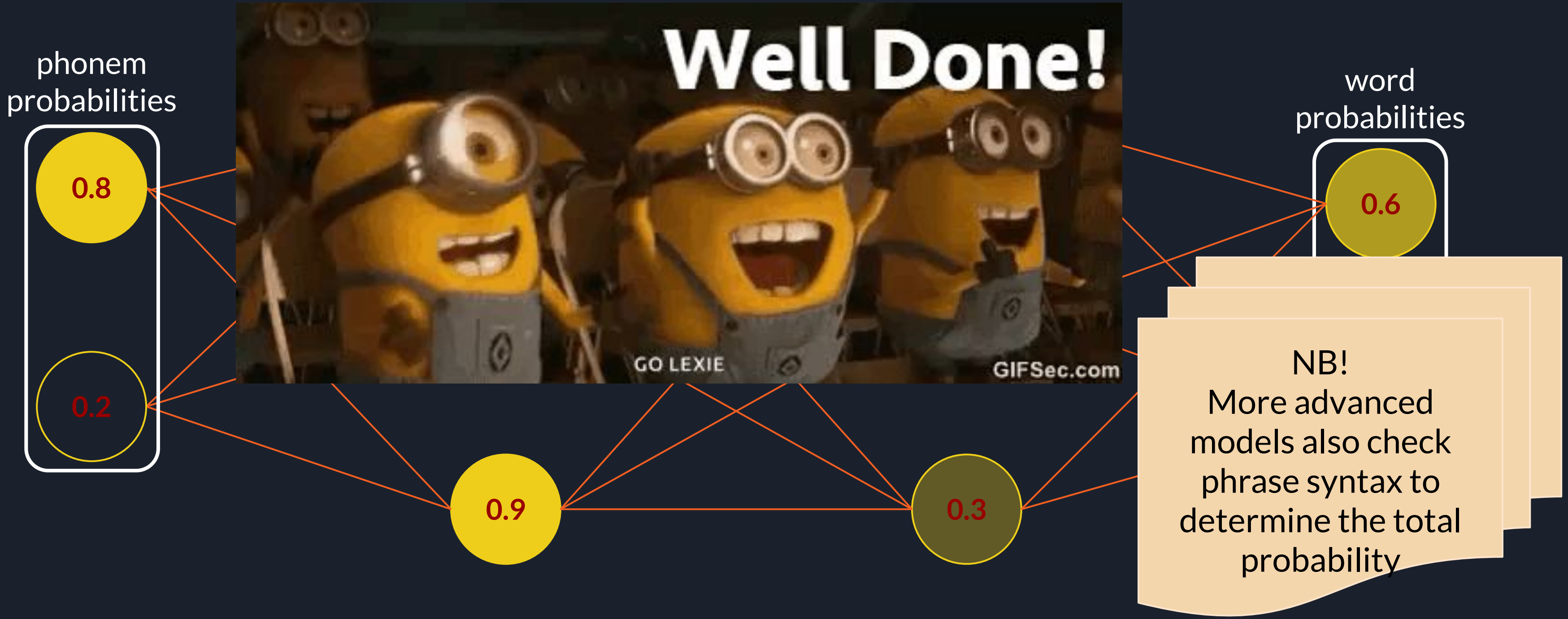
Processing the audio

- We use AI to determine the probability a specific phonem has been detected



Processing the audio

- We can use AI to determine the probability a specific word has been detected



The code

For the extra section

Python

- https://alphacephei.com/vosk/models/vosk-model-small-en-us-0.15.zip
  - Extract and copy into the project

```python
from vosk import Model, KaldiRecognizer
import pyaudio
import serial
ser = serial.Serial('COM11', 115200)
def send_arduino(c):
  ser.write(c)
model = Model('vosk-model-small-en-us-0.15')
recognizer = KaldiRecognizer(model, 16000) #sampling rate
mic = pyaudio.PyAudio()
stream = mic.open(format=pyaudio.paInt16,
          channels=1,
          rate=16000,
          input=True,
          frames_per_buffer=8192)
stream.start_stream()
```

```python
def recognise():
  data = stream.read(4096) #number of bytes
  if recognizer.AcceptWaveform(data): #if the voice recognition is acceptable
    text = recognizer.Result()
    text = text[14:-3] #this string cut is here to remove  a bunch of random characters
    return text #returns the text that was listened
  else:
    return "..." #if it doesn't recognize what was said it's ok
while 1:
  word = recognise()
  for i in ["red", "read", "right", "thread"]:
    if i in word:
      send_arduino(b'r')
      break
  for i in ["blue", "bone"]:
    if i in word:
      send_arduino(b'b')
      break
  for i in ["green", "bring"]:
    if i in word:
      send_arduino(b'g')
      break
  print("...")
```

Arduino

```
#define R 2
#define G 4
#define B 6
void setup() {
    Serial.begin(115200);
    pinMode(2, OUTPUT);
    pinMode(4, OUTPUT);
    pinMode(6, OUTPUT);
}
```

```
void loop() {
  if(Serial.available()){
    char c = Serial.read();
    if(c == 'r'){
      digitalWrite(R, 1);
      digitalWrite(G, 0);
      digitalWrite(B, 0);
    }else if(c == 'g'){
      digitalWrite(R, 0);
      digitalWrite(G, 1);
      digitalWrite(B, 0);
    }else if(c == 'b'){
      digitalWrite(R, 0);
      digitalWrite(G, 0);
      digitalWrite(B, 1);
    }
  }
}
```

References (for parts 1 and 2):

- https://www.3blue1brown.com/topics/neural-networks
- http://neuralnetworksanddeeplearning.com/chap2.html
- https://keras.io/
- I actually forgot where I adapted the code from, but it is quite straightforward nevertheless

References (for extra section)

- https://www.scaler.com/topics/artificial-intelligence-tutorial/speech-recognition-in-ai/
- https://jonathan-hui.medium.com/speech-recognition-phonetics-d761ea1710c0