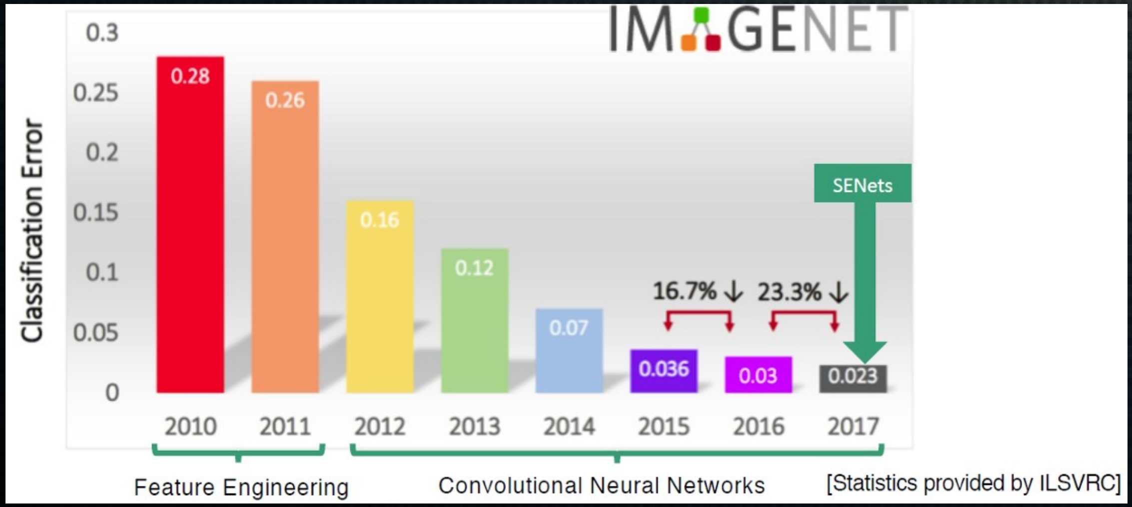
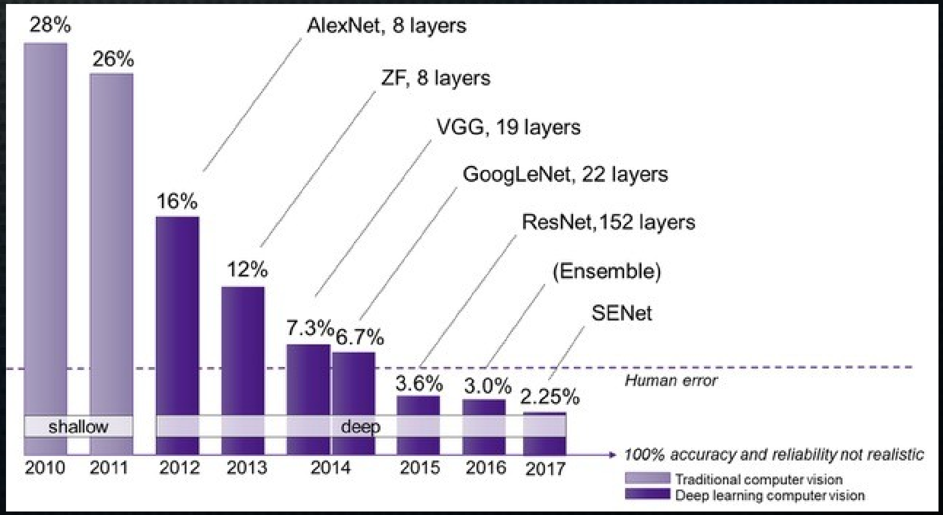
Convolutional neural networks

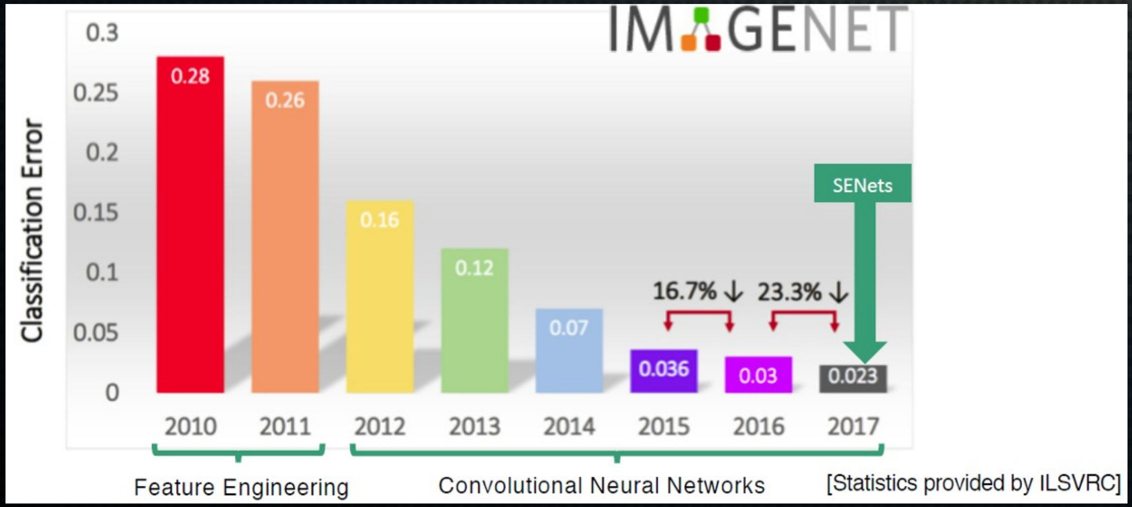
- These dense neural networks are not what made the huge strides in deep learning over the last few years.

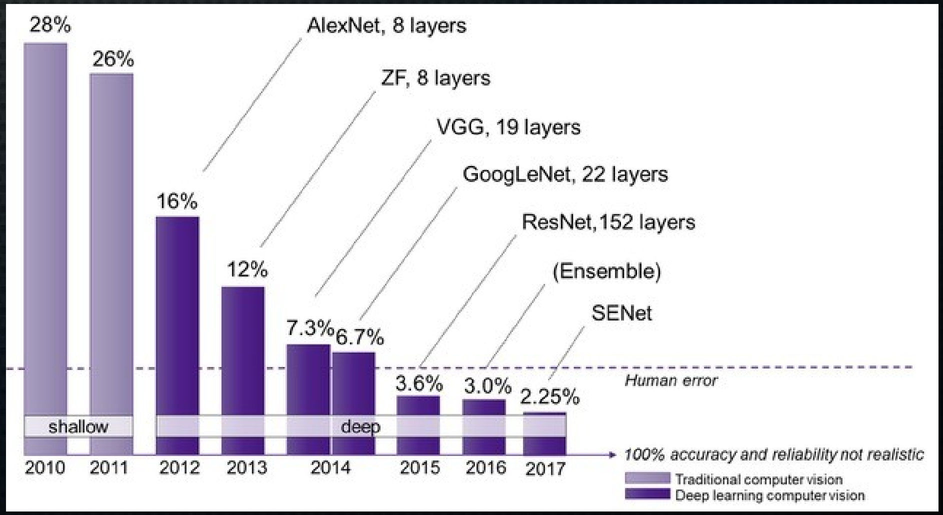




Convolutional neural networks

- These dense neural networks are not what made the huge strides in deep learning over the last few years.
- Instead, those are deep convolutional neural networks



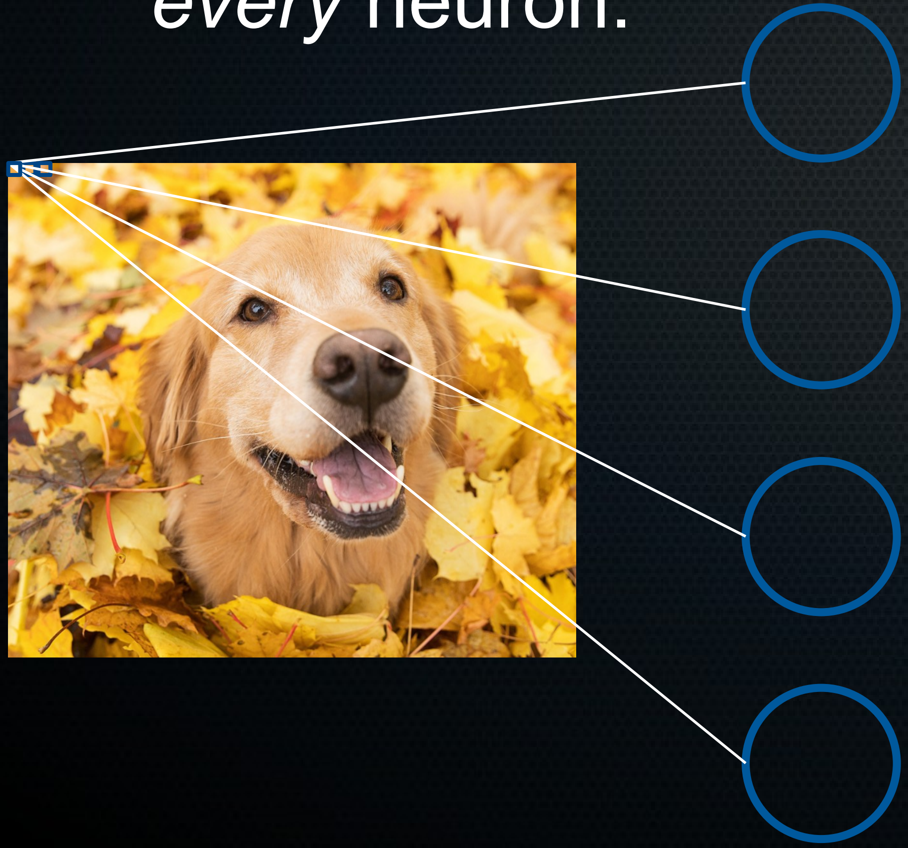


Convo-what now?

- Let’s look at an image

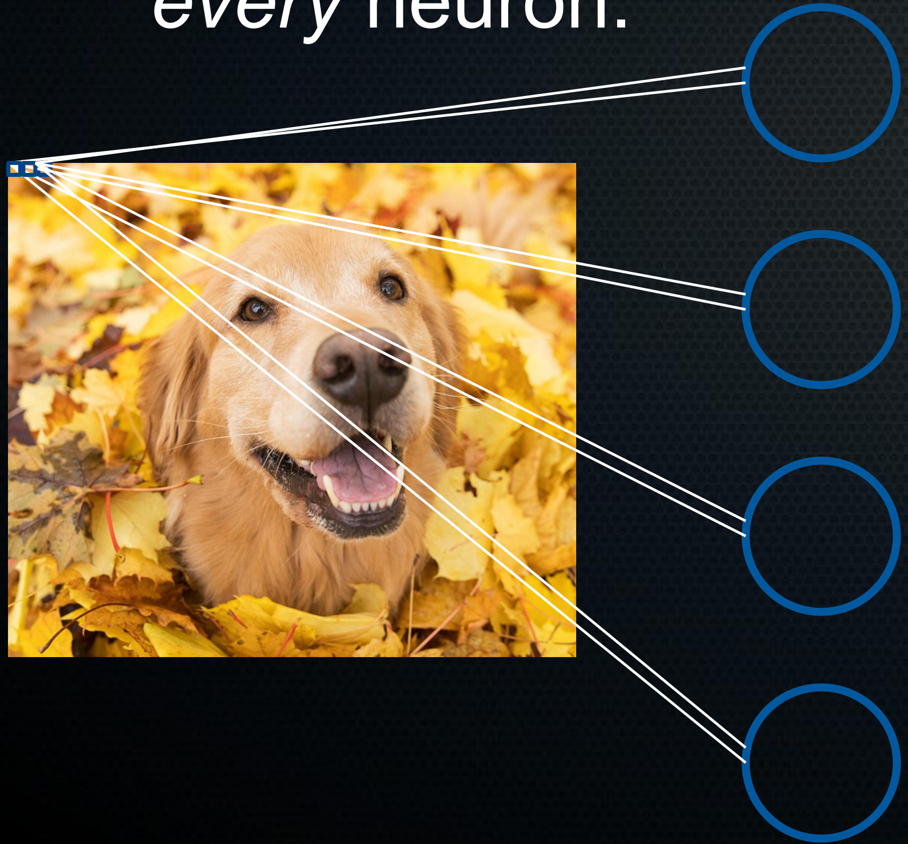
Convo-what now?

- Let’s look at an image
- In a dense architecture, every pixel value is connected to *every* neuron.



Convo-what now?

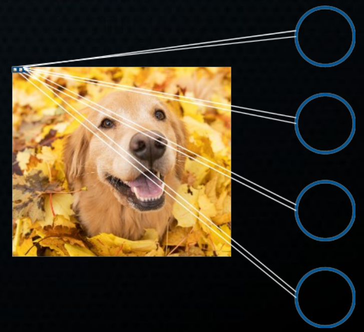
- Let’s look at an image
- In a dense architecture, every pixel value is connected to *every* neuron.



Etc.

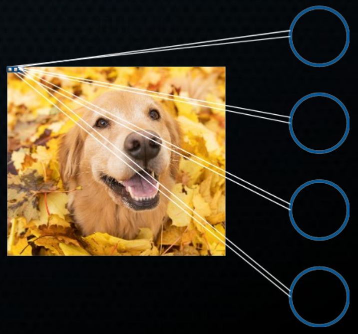
Convo-what now?

- Let's look at an image
- In a dense architecture, every pixel value is connected to *every* neuron.
- This gives problems:
  - You get an *insane* amount of parameters to optimise. 250*250 pixels * 20 input neurons = 1,250,020 weights and biases. You can forget about any sort of findable or achievable (global) optimum.
  - There is no locality: if you want your network to know whether or not there is a dog in an image, all these parameters must be optimised so that you can recognise the dog anywhere.



Convo-what now?

This is madness!



Convo-what now?

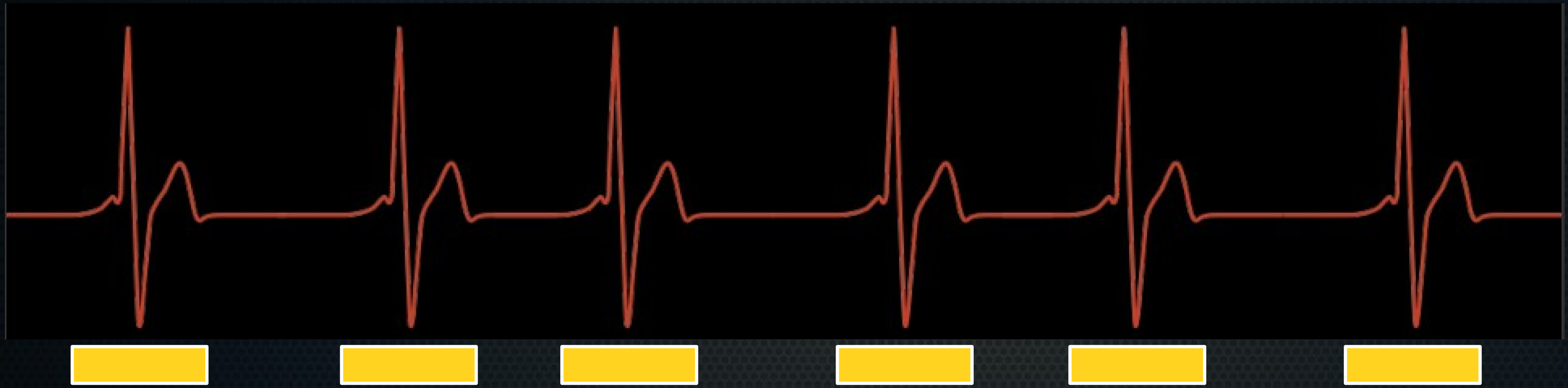
- The answer: convolution. Let’s look at a 1D example!

- When is the heart beating?

These examples were gladly ~~stolen~~ borrowed from Marc

Convo-what now?

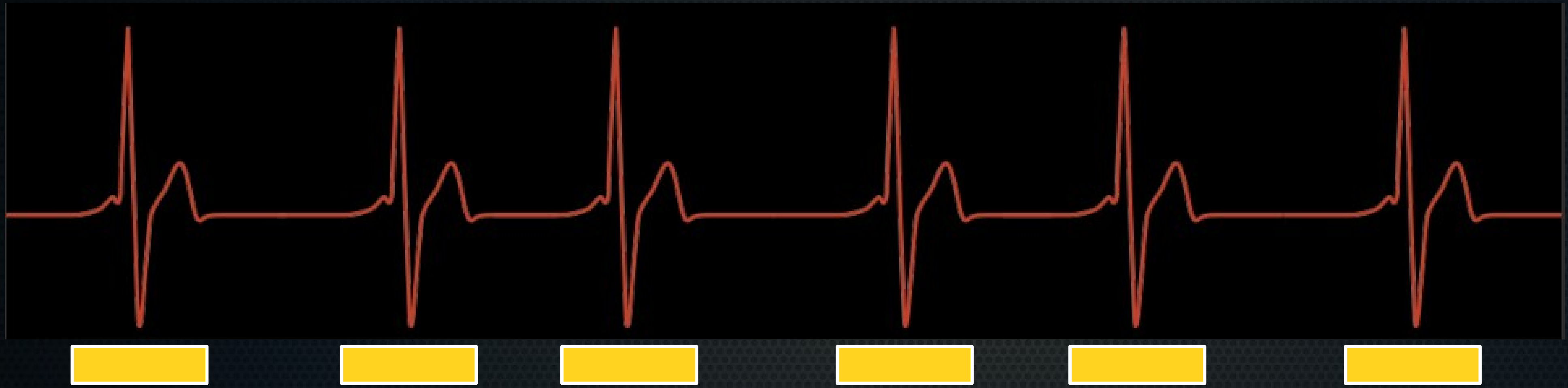
- The answer: convolution. Let’s look at a 1D example!



- When is the heart beating?

Convo-what now?

- The answer: convolution. Let's look at a 1D example!

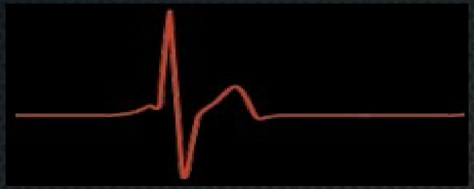
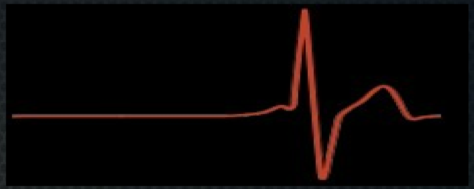


$$\text{signal} = x = \begin{bmatrix} 0 & 0 & 0 & 1 & -1 & 0.5 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{label} = y = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

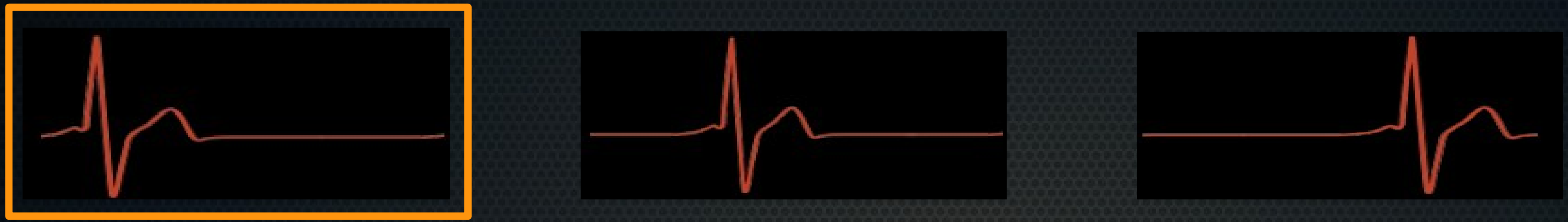
Convo-what now?

- Signal can be at different positions in the sequence:

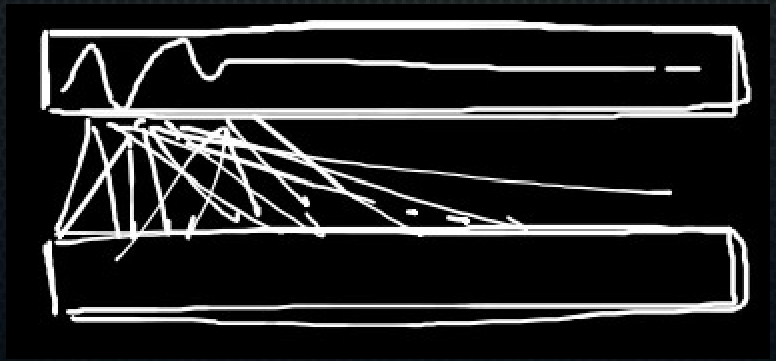




Convo-what now?

- Signal can be at different positions in the sequence:



- Dense network needs to optimise such that different weights somehow cause the network to output 1 for different positions of the signal:

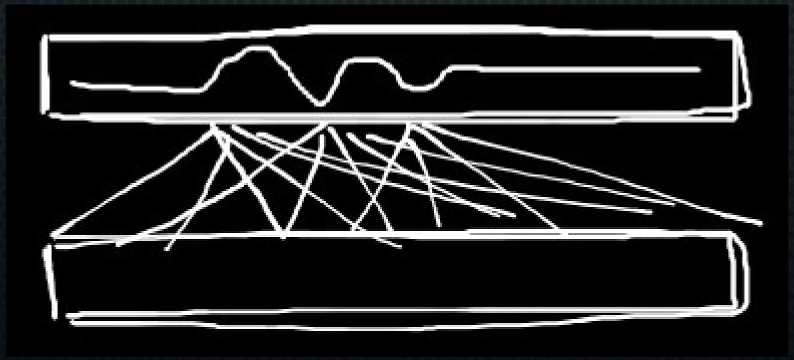


$$\text{label} = y = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Convo-what now?

- Signal can be at different positions in the sequence:

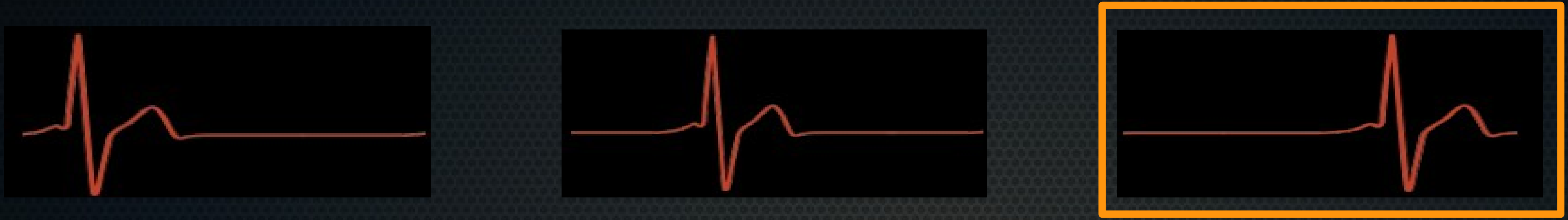
- Dense network needs to optimise such that different weights somehow cause the network to output 1 for different positions of the signal:



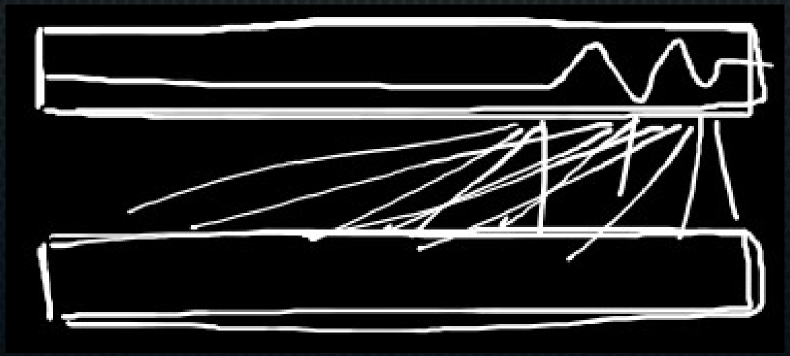
$$\text{label} = y = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Convo-what now?

- Signal can be at different positions in the sequence:



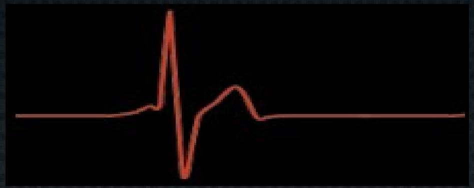
- Dense network needs to optimise such that different weights somehow cause the network to output 1 for different positions of the signal:



$$\text{label} = y = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1]$$

Convo-what now?

- Convolution:



Input

0	0	0	1	-1	0.5	0	0	0	0

Kernel

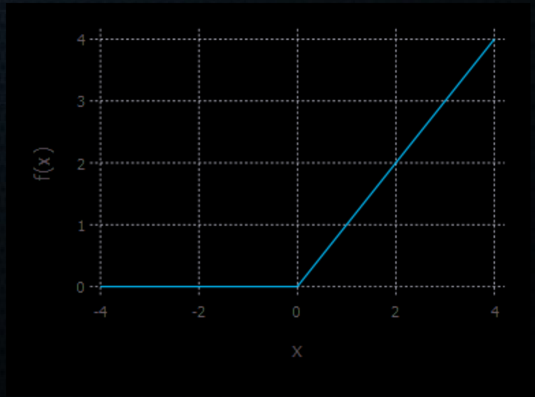
2

Kernel output

0									

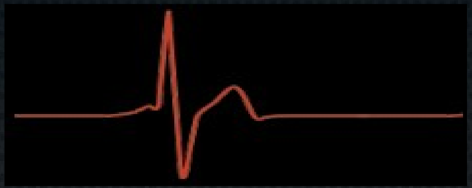
ReLu output

0	0	0	1	-1	0.5	0	0	0	0



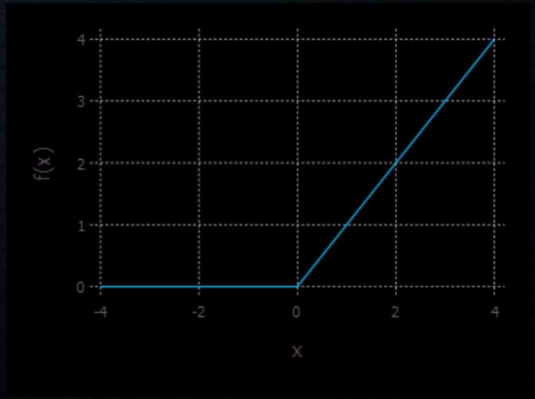
Convo-what now?

- Convolution:



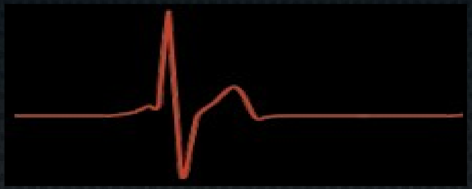
Input	0	0	0	1	-1	0.5	0	0	0	0
	*									
Kernel	2									
Kernel output	0									
ReLu output	0									

→ Convolve (move) the kernel over the sequence



Convo-what now?

- Convolution:



Input

0	0	0	1	-1	0.5	0	0	0	0

*

Kernel

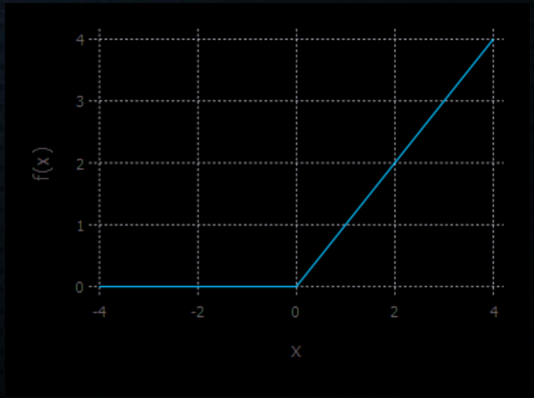
2

Kernel output

0	0								

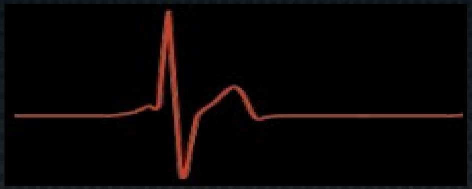
ReLu output

0	0								

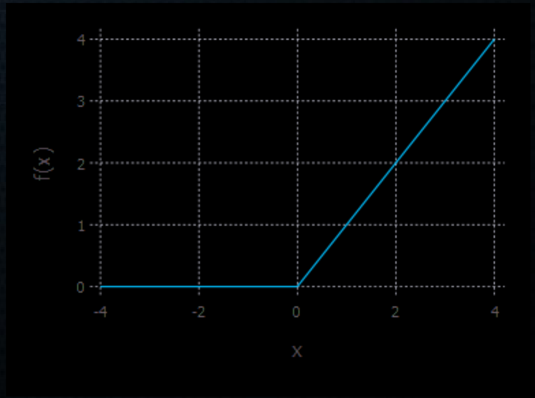


Convo-what now?

- Convolution:

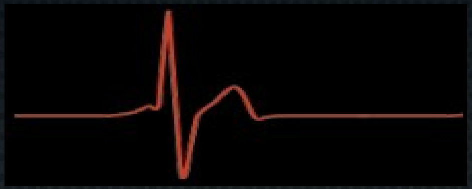


Input	0	0	0	1	-1	0.5	0	0	0	0
			*							
Kernel			2							
Kernel output	0	0	0							
ReLu output	0	0	0							

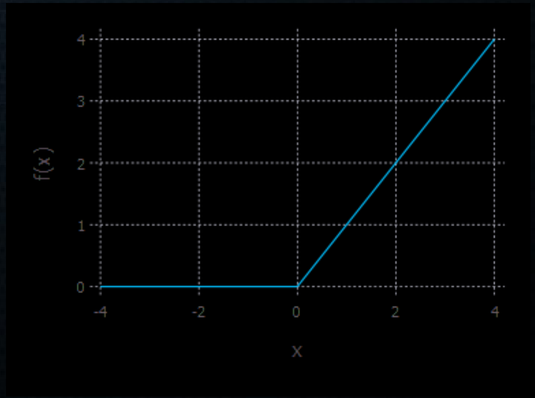


Convo-what now?

- Convolution:

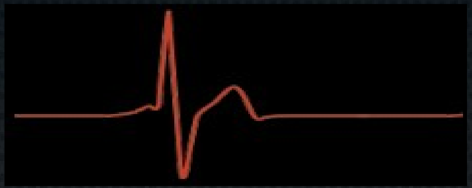


Input	0	0	0	1	-1	0.5	0	0	0	0
				*						
Kernel				2						
Kernel output	0	0	0	2						
ReLu output	0	0	0	1						

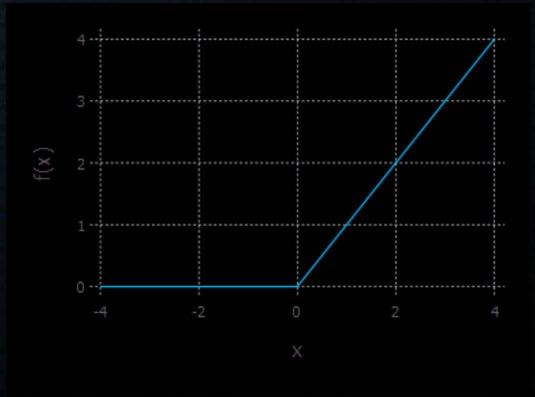


Convo-what now?

- Convolution:

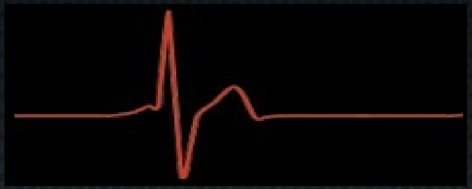


Input	0	0	0	1	-1	0.5	0	0	0	0
										*
Kernel										2
Kernel output	0	0	0	2	-2	1	0	0	0	0
ReLu output	0	0	0	2	0	1	0	0	0	0



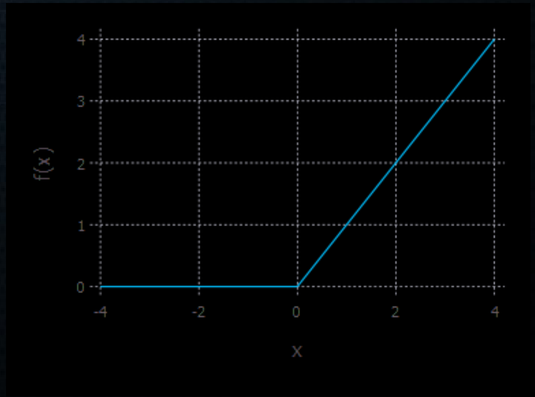
Convo-what now?

- Convolution:



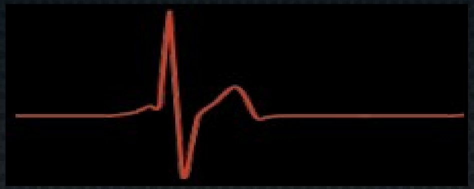
Input	0	0	0	1	-1	0.5	0	0	0	0
										*
Kernel										2
Kernel output	0	0	0	2	-2	1	0	0	0	0
ReLu output	0	0	0	2	0	1	0	0	0	0

This is a kernel that detects positive numbers



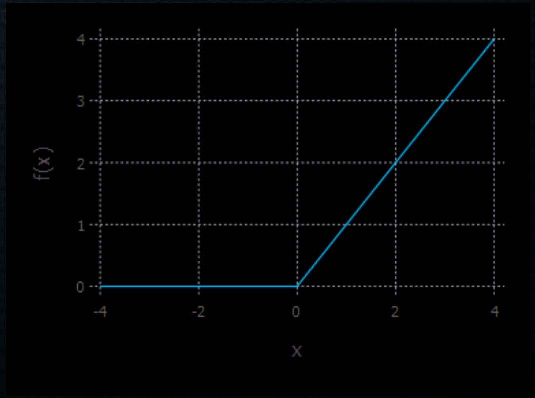
Convo-what now?

- Convolution:



Input	0	0	0	1	-1	0.5	0	0	0	0
										*
Kernel										2
Kernel output	0	0	0	2	-2	1	0	0	0	0
ReLu output	0	0	0	2	0	1	0	0	0	0

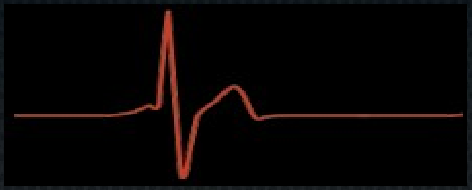
Just like a neuron has weights, this kernel has a trainable weight (2)



This is a kernel that detects positive numbers

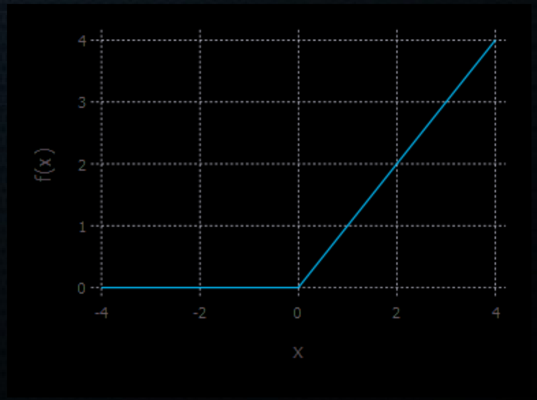
Convo-what now?

- Convolution:



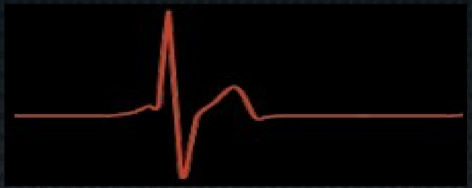
Input	0	0	0	1	-1	0.5	0	0	0	0
	*	*	*							
Kernel	0	2	1							
		+								
Kernel output	-	0								-
ReLu output	-	0								-

→ A kernel can have a size >1



Convo-what now?

- Convolution:



Input

0	0	0	1	-1	0.5	0	0	0	0

* * *

Kernel

0	2	1

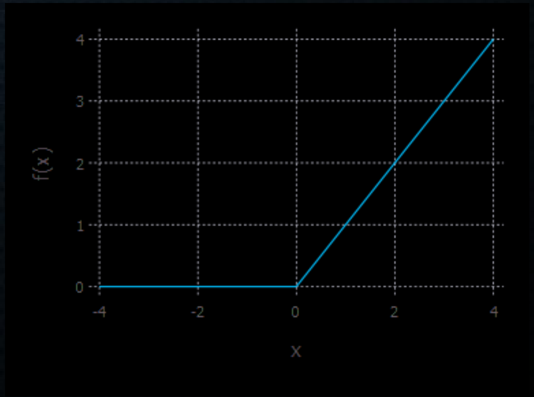
+

Kernel output

-	0	1							-

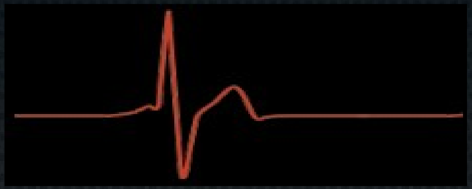
ReLu output

-	0	1							-



Convo-what now?

- Convolution:



Input

0	0	0	1	-1	0.5	0	0	0	0

* * *

Kernel

0	2	1

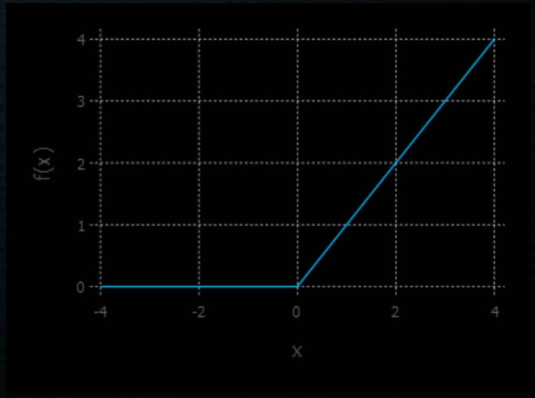
+

Kernel output

-	0	1	1						-

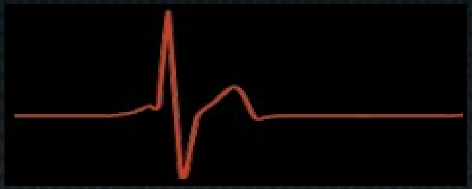
ReLu output

-	0	1	1						-



Convo-what now?

- Convolution:



Input

0	0	0	1	-1	0.5	0	0	0	0

* * *

Kernel

0	2	1

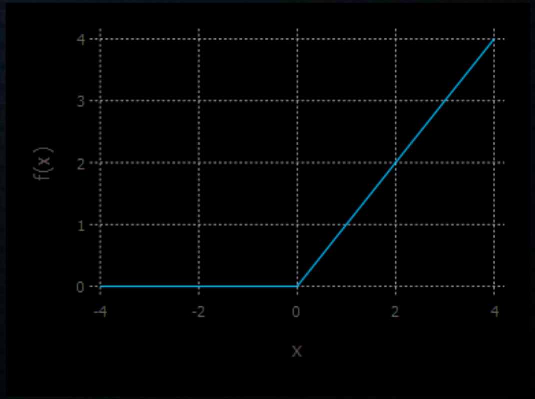
+

Kernel output

-	0	1	1	-1.5					-

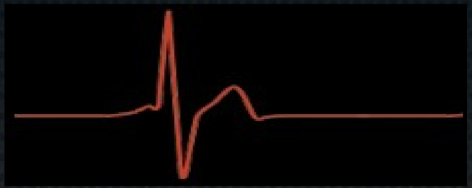
ReLu output

-	0	1	1						-



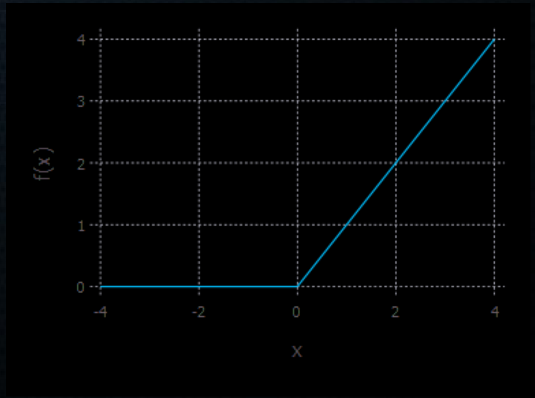
Convo-what now?

- Convolution:



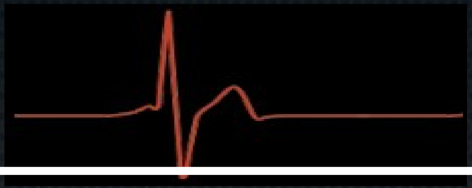
Input	0	0	0	1	-1	0.5	0	0	0	0
								*	*	*
Kernel								0	2	1
									+	
Kernel output	-	0	1	1	-1.5	1	0	0	0	-
ReLu output	-	0	1	1	0	1	0	0	0	-

Note shrinkage due to edge effects



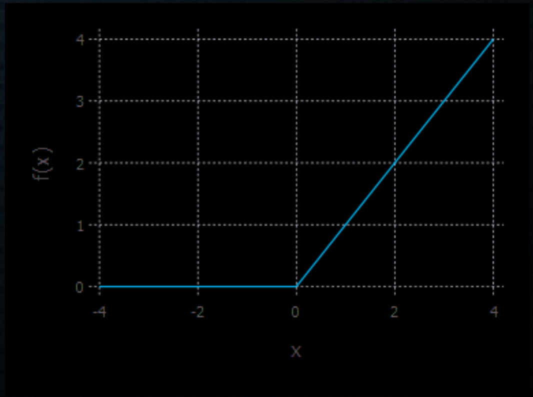
Convo-what now?

- Convolution:



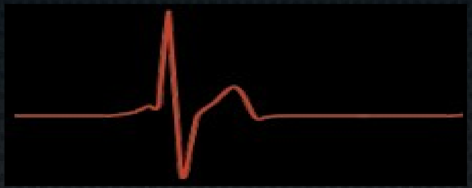
Input	0	0	0	0	1	-1	0.5	0	0	0	0	0
									*	*	*	
Kernel									0	2	1	
										+		
Kernel output		0	0	1	1	-1.5	1	0	0	0	0	
ReLu output		0	0	1	1	0	1	0	0	0	0	

Could fix by *padding* the input



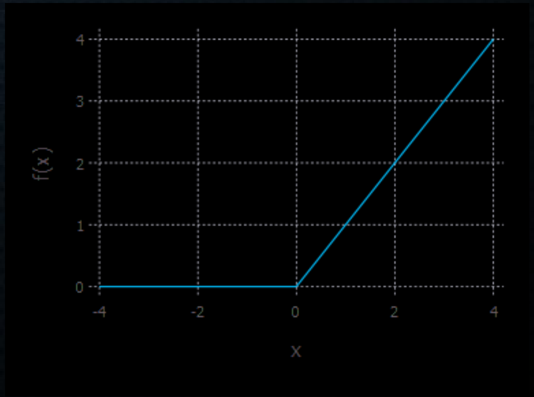
Convo-what now?

- Convolution:



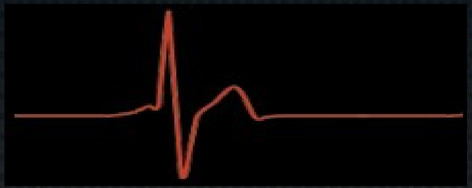
Input	0	0	0	1	-1	0.5	0	0	0	0
	*	*	*							
Kernel	0	2	1							
		+								
Kernel output	-	0								-
ReLu output	-	0								-

Or could compress output size further by changing *stride* (step size). Stride = 3 now.

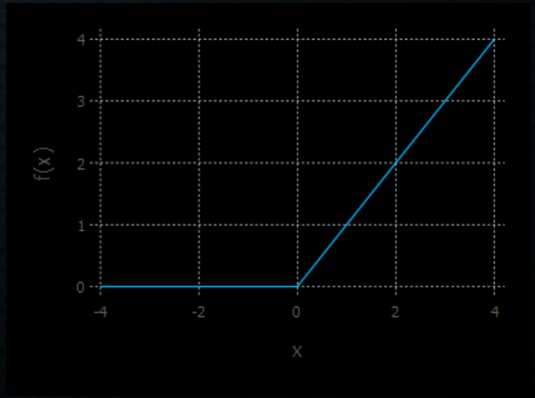


Convo-what now?

- Convolution:

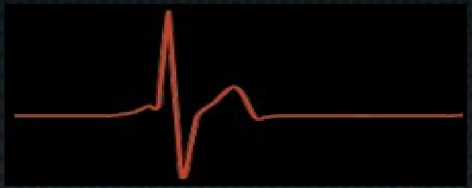


Input	0	0	0	1	-1	0.5	0	0	0	0
				*	*	*				
Kernel				0	2	1				
					+					
Kernel output	-	0	-	-	-1.5					-
ReLu output	-	0	-	-	0					-



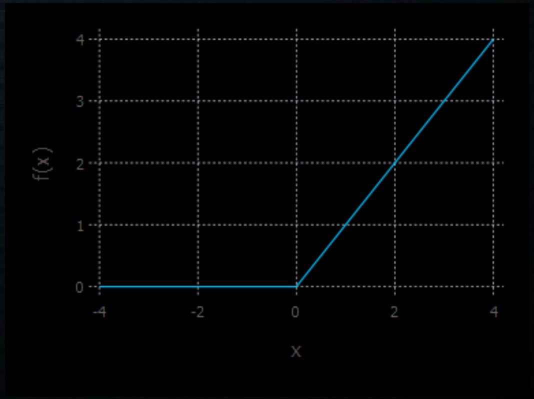
Convo-what now?

- Convolution:



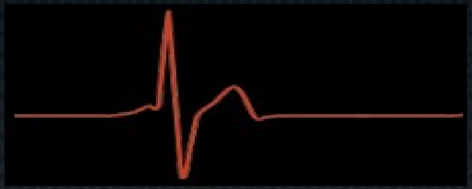
Size = 3
Stride = 3

Input	0	0	0	1	-1	0.5	0	0	0	0
							*	*	*	
Kernel							0	2	1	
								+		
Kernel output	-	0	-	-	-1.5	-	-	0	-	-
ReLu output	-	0	-	-	0	-	-	0	-	-



Convo-what now?

- Convolution:

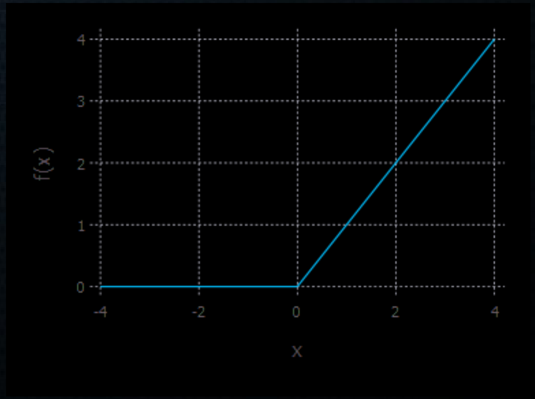


Size = 3
Stride = 3

Input	0	0	0	1	-1	0.5	0	0	0	0
Kernel							0	2	1	

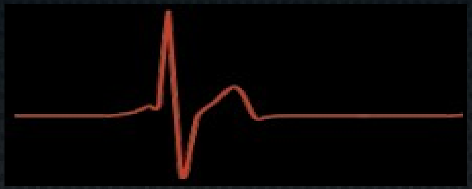
Kernel output	0	-1.5	0
ReLu output	0	0	0

This example is not useful because I randomly picked some weights for the kernel. But normally you can train these weights by backpropagation such that the network works well!



Convo-what now?

- Convolution:



Size = 3
Stride = 1

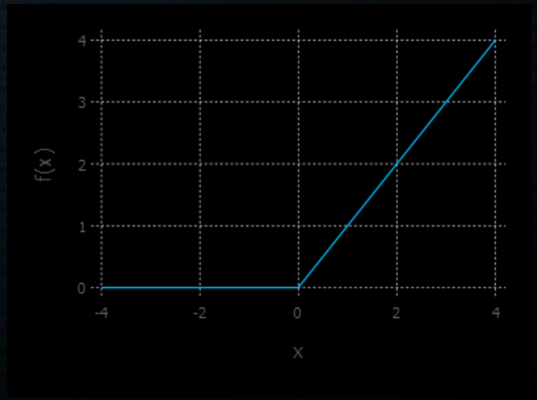
Input	0	0	0	1	-1	0.5	0	0	0	0

	*	*	*
Kernel	0	2	1

$\underbrace{\qquad\qquad}_{+}$

Kernel output	-	0	1	1	-1.5	1	0	0	0	-
ReLu output	-	0	1	1	0	1	0	0	0	-

Note shrinkage due to edge effects

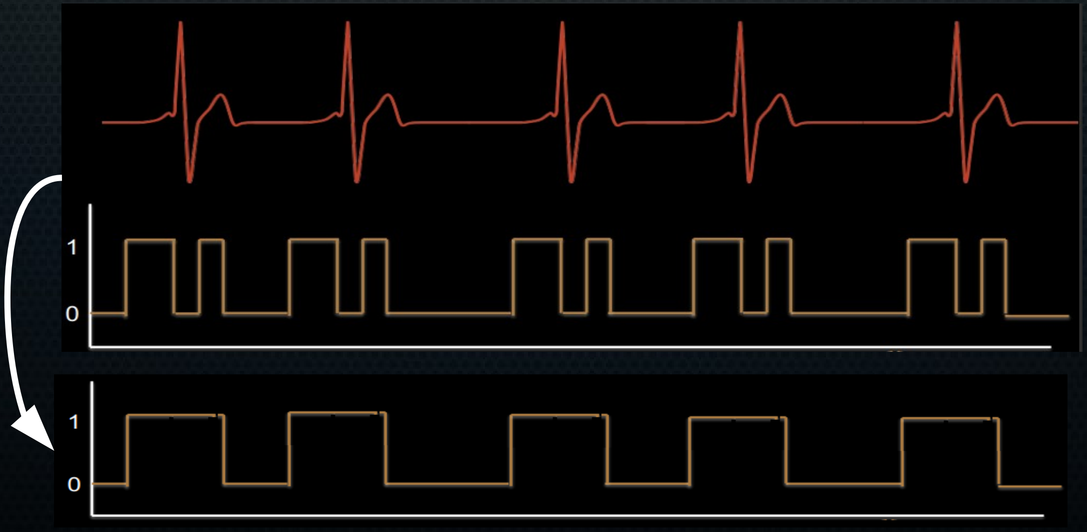


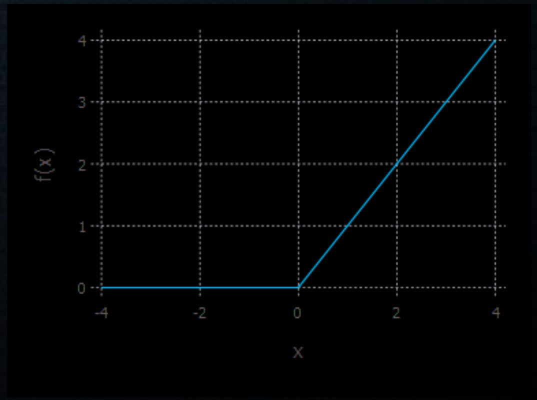
Convo-what now?

Input	0	0	0	1	-1	0.5	0	0	0	0
								*	*	*
Kernel								0	2	1
									+	
Kernel output	-	0	1	1	-1.5	1	0	0	0	-
ReLu output	-	0	1	1	0	1	0	0	0	-

Size = 3
Stride = 1

Not optimal, but by adding another convolution layer, you might get to something like:





2D convolution

- Size = 2*2; stride = 1

0	22	0	1
12	2	3	23
3	34	26	2
0	22	86	3
4	3	1	4

0	2
2	0

68		

2D convolution

- Size = 2*2; stride = 1

0	22	0	1
12	2	3	23
3	34	26	2
0	22	86	3
4	3	1	4

0	2
2	0

68	4	

2D convolution

- Size = 2*2; stride = 1

0	22	0	1
12	2	3	23
3	34	26	2
0	22	86	3
4	3	1	4

0	2
2	0

68	4	8

2D convolution

- Size = 2*2; stride = 1

0	22	0	1
12	2	3	23
3	34	26	2
0	22	86	3
4	3	1	4

0	2
2	0

68	4	8
10		

Etc.

Another type of convolution: max pooling

- Size = 2*2; stride = 1; just take the maximum value in the kernel area

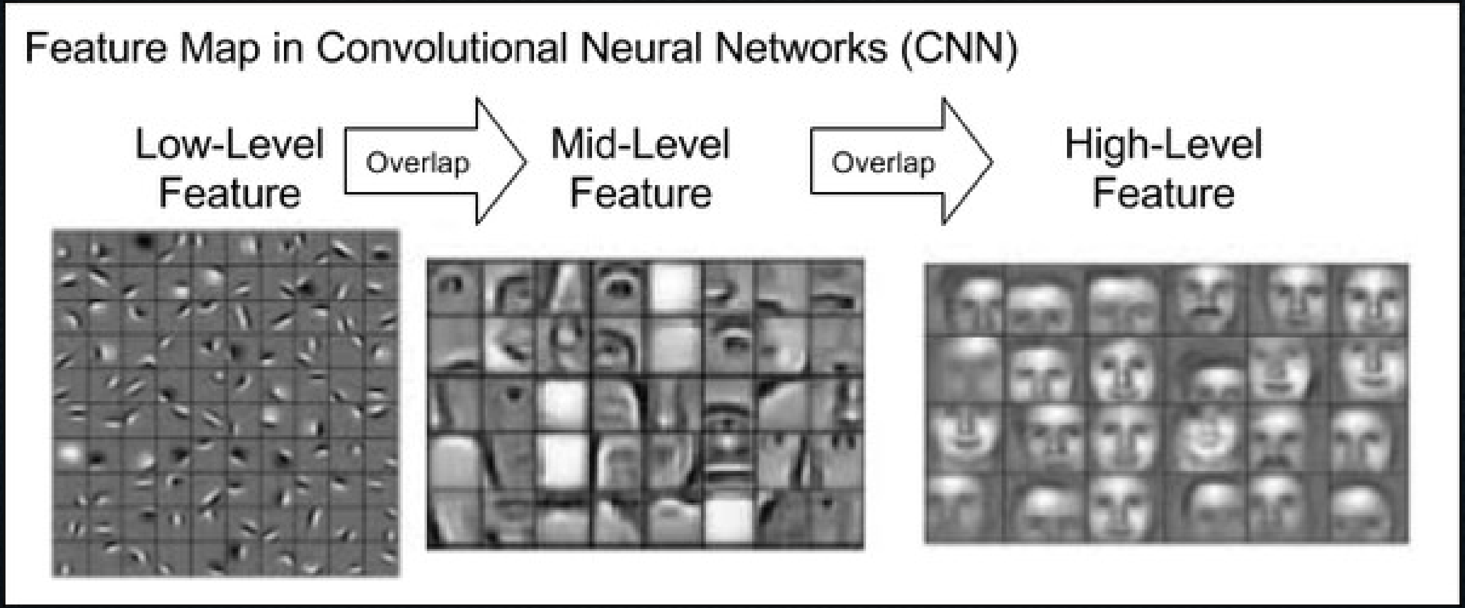
0	22	0	1
12	2	3	23
3	34	26	2
0	22	86	3
4	3	1	4

22	22	23
34	26	26
34	86	86
22	86	86

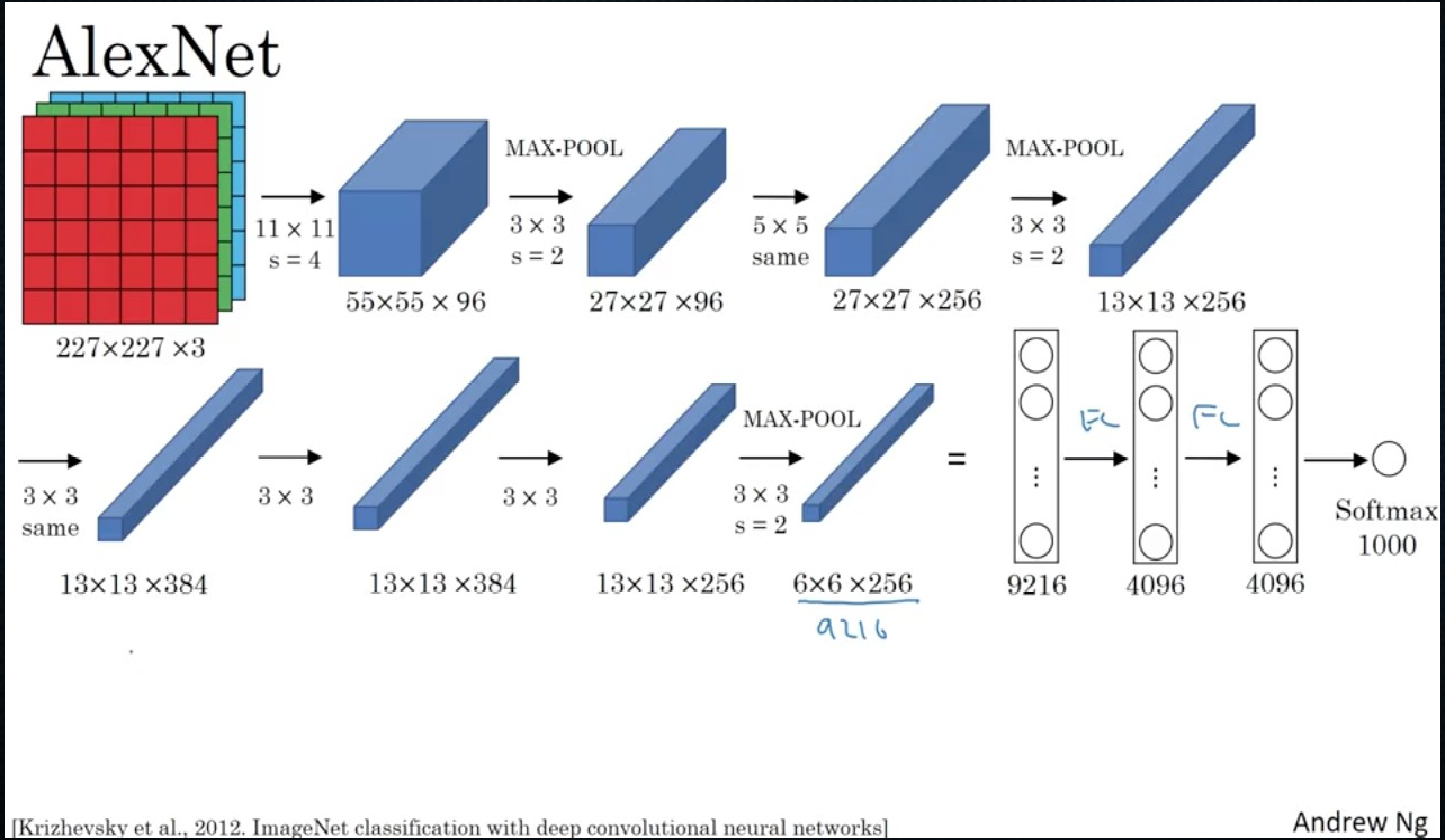
Another type of convolution: pooling/averaging

Use in face detection

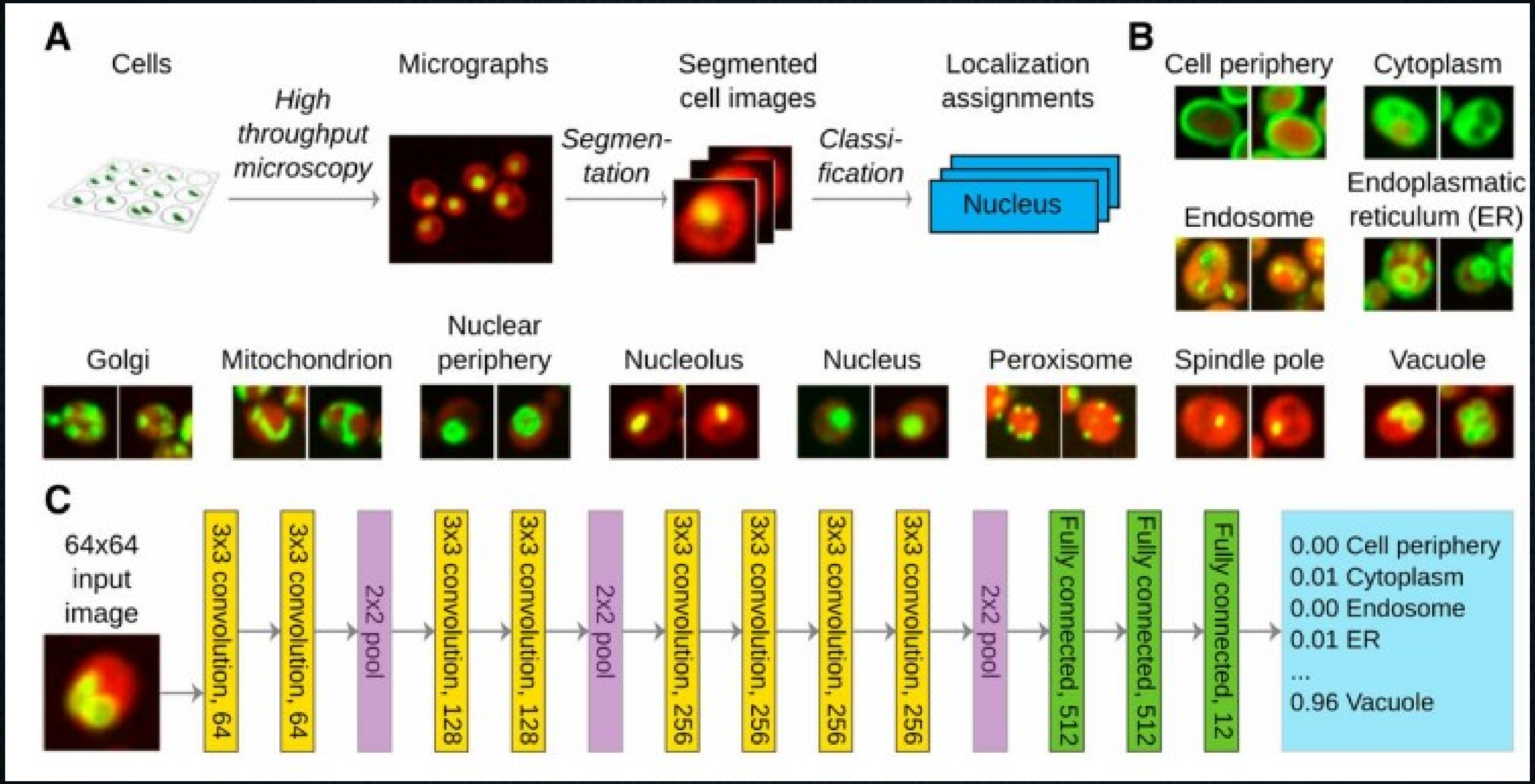
- Since kernels so few parameters: can use *many* of them per layer → each becomes sensitive to different image features



Example AlexNet (2012)



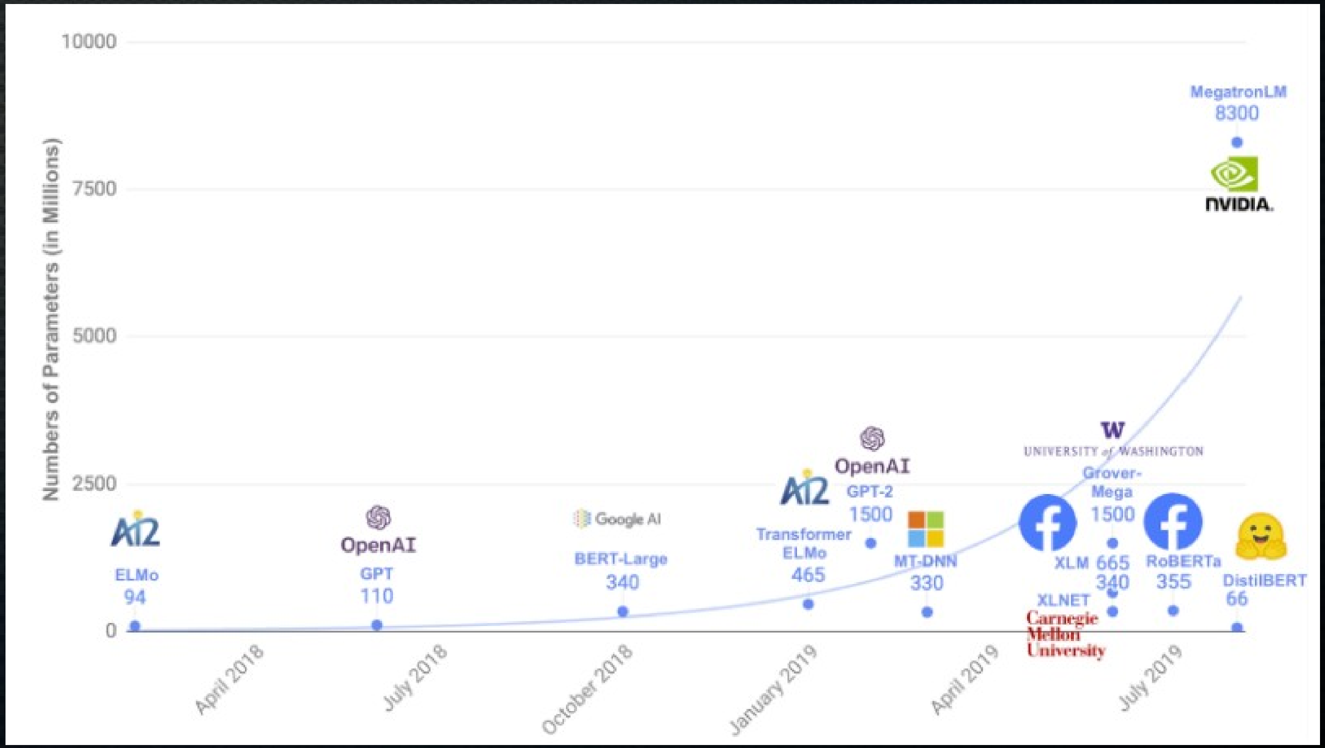
Biological use



Pärnamaa, T., & Parts, L. (2017). Accurate classification of protein subcellular localization from high-throughput microscopy images using deep learning. G3: Genes, Genomes, Genetics, 7(5), 1385-1392.

There's a lot more

- Batch normalisation
- Vanishing gradient problem
- Dropout
- Recurrent neural nets



Implementation

- We are not going to implement convolutional neural networks ourselves: implementing backpropagation properly on a simple dense network is already taxing enough.
- Still, doing that should give you a solid basis for understanding convolutional neural networks, and we’ll introduce the Keras library for building (convolutional) neural networks next Monday.

Praatje biologie + NN?

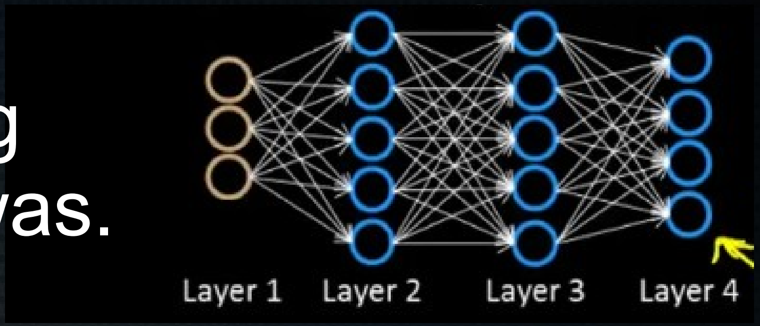
- Liefst kort praatje van iemand die met (conv)NN in de biologie wat doet → hoe is het om dit te vertalen naar biologie? Waar loop je tegenaan?
- Iemand van Jeroen of iemand van Alexander Schönhuth (ALS-classifier?) → of Richard Schenkman, tip van Bas

Afternoon practical

- Implement backpropagation yourself
- Train a dense neural network on the MNIST dataset

HIERNA VOLGEN OUDE SLIDES DIE WAARSCHIJNLIJK WEG KUNNEN

How do we use this cost to update parameters?



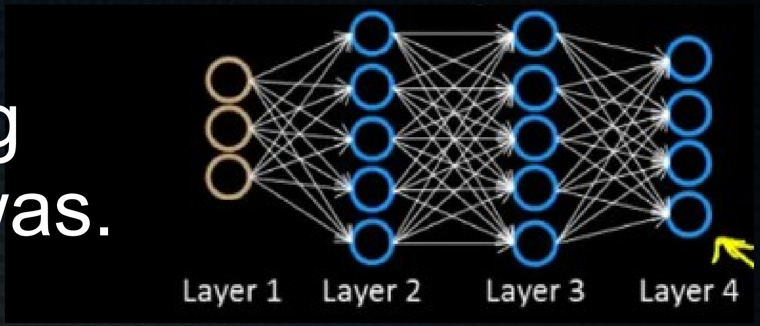
- We want to have a term that says how wrong the activation of each neuron in each layer was.

$$\delta_j^{(l)} = \text{error of node j in layer l} = \text{error of } a_j^{(l)}$$

- For the output layer:

$$\delta_j^{(4)} = \underbrace{a_j^{(4)}}_{h_\theta(x)_j} - y_j$$

How do we use this cost to update parameters?



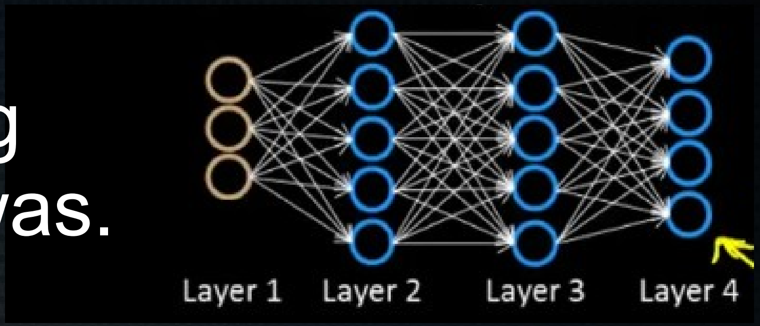
- We want to have a term that says how wrong the activation of each neuron in each layer was.

$$\delta_j^{(l)} = \text{error of node j in layer l} = \text{error of } a_j^{(l)}$$

- For the output layer:

$$\delta_j^{(4)} = \underbrace{a_j^{(4)}}_{h_\theta(x)_j} - y_j \longrightarrow \underbrace{\delta^{(4)} = a^{(4)} - y}_{\text{vectorised}} \longrightarrow \begin{bmatrix} 0.22 \\ 0.8 \\ -0.2 \\ 0.34 \end{bmatrix} = \begin{bmatrix} 0.22 \\ 0.8 \\ 0.8 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

How do we use this cost to update parameters?



- We want to have a term that says how wrong the activation of each neuron in each layer was.

$$\delta_j^{(l)} = \text{error of node j in layer l} = \text{error of } a_j^{(l)}$$

- For the output layer:

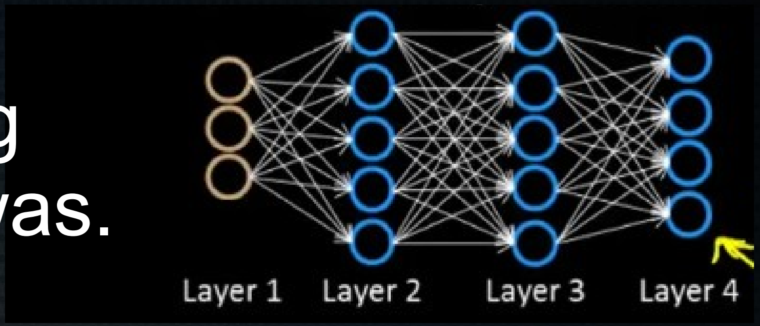
$$\delta_j^{(4)} = a_j^{(4)} - y_j \longrightarrow \delta^{(4)} = a^{(4)} - y$$

- Previous layers:

$$\delta^{(3)} = (\Theta^{(3)})^T \cdot \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \cdot \delta^{(3)} .* g'(z^{(2)})$$

How do we use this cost to update parameters?



- We want to have a term that says how wrong the activation of each neuron in each layer was.

$$\delta_j^{(l)} = \text{error of node j in layer l} = \text{error of } a_j^{(l)}$$

- For the output layer:

$$\delta_j^{(4)} = a_j^{(4)} - y_j \longrightarrow \delta^{(4)} = a^{(4)} - y$$

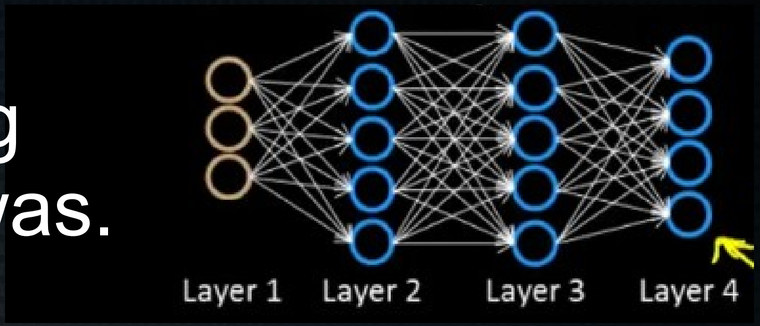
- Previous layers:

$$\delta^{(3)} = (\Theta^{(3)})^T \cdot \delta^{(4)} .* \boxed{g'(z^{(3)})} \longrightarrow$$

Somewhat complicated derivation:
derivative of sigmoid(x)
= sigmoid(x) .* (1-sigmoid(x))
See link in the exercises for more info.

$$\delta^{(2)} = (\Theta^{(2)})^T \cdot \delta^{(3)} .* g'(z^{(2)})$$

How do we use this cost to update parameters?



- We want to have a term that says how wrong the activation of each neuron in each layer was.

$$\delta_j^{(l)} = \text{error of node j in layer l} = \text{error of } a_j^{(l)}$$

- For the output layer:

$$\delta_j^{(4)} = a_j^{(4)} - y_j \longrightarrow \delta^{(4)} = a^{(4)} - y$$

- Previous layers:

$$\delta^{(3)} = (\Theta^{(3)})^T \cdot \delta^{(4)} .* \boxed{g'(z^{(3)})} \longrightarrow$$

Somewhat complicated derivation:
derivative of sigmoid(x)
= sigmoid(x) .* (1-sigmoid(x))
See link in the exercises for more info.

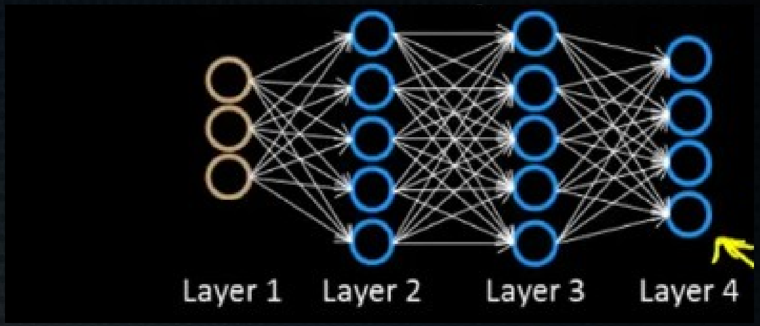
$$\downarrow$$

$$a^{(3)} .* (1 - a^{(3)})$$

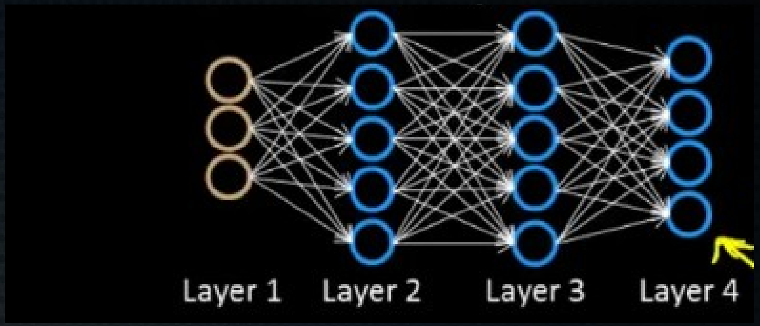
$$\delta^{(2)} = (\Theta^{(2)})^T \cdot \delta^{(3)} .* g'(z^{(2)})$$

How do we use this cost to update parameters?

- That‘s a lot of math. What‘s the intuition?
- In the final layer, we calculate errors with some cost function → how wrong are we?



How do we use this cost to update parameters?



- That's a lot of math. What's the intuition?
- In the final layer, we calculate errors with some cost function → how wrong are we?
- This cost depends on 3 things:

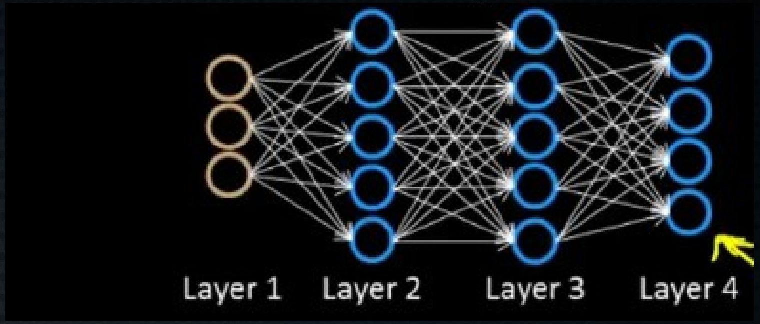
Bias of each unit in the layer

Weights of each unit in the layer

Activation value coming in from previous layer

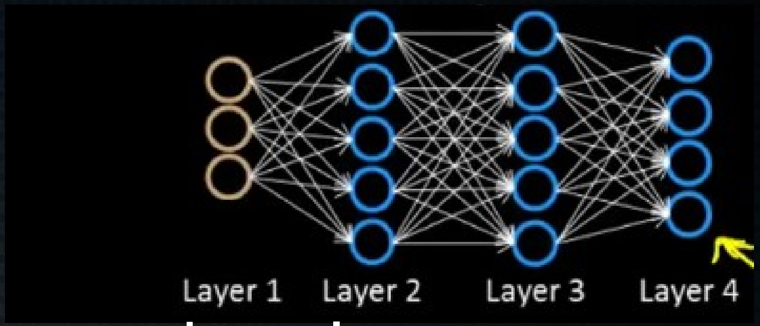
Error for a sample (calculated from output layer)

How do we use this cost to update parameters?

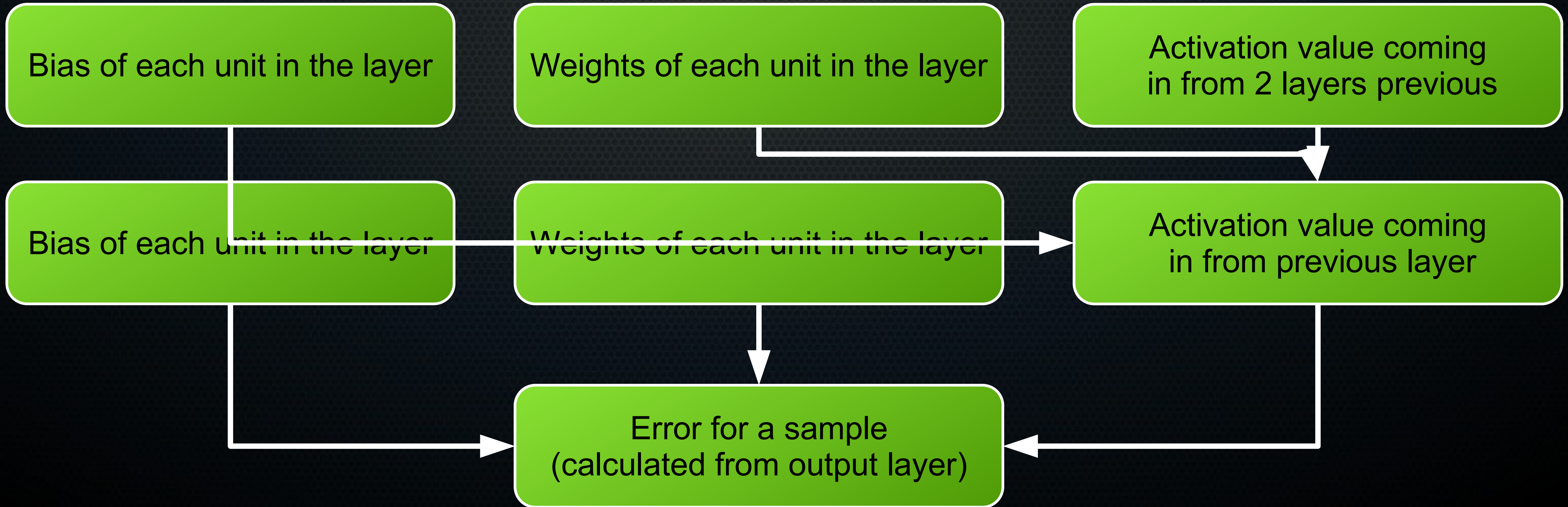


- That‘s a lot of math. What‘s the intuition?
- In the final layer, we calculate errors with some cost function → how wrong are we?
- This cost depends on 3 things : weights, biases, activation previous layer
- We can *change* the weights and biases by using gradient descent on the partial derivatives of the cost function wrt. to them and taking a small step → this will change the output of the final layer the next time, lowering the cost.
- What we are left with is some error that is due to the activation of the previous layer.

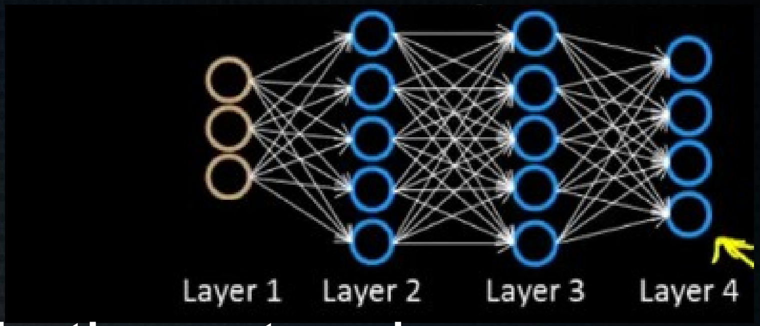
How do we use this cost to update parameters?



- What we are left with is some error that is due to the activation of the previous layer. But of course, those values are due to that layer's weights and biases, and activation of its previous layer:



How do we use this cost to update parameters?



- Via a complex derivation that we won‘t do here, the procedure we just discussed leads to calculating the partial derivatives of the cost function with respect to all the weights and biases in the network.
- **→ HIER NOG INVOEGEN: minuut 10-11 backpropagation video → totale implementatie**
- **NOOT: Ik heb Andrew Ngs uitleg gevolgd, maar ik vind dat het allemaal uit de lucht komt vallen. Ik wil dit zelf ook nog iets beter uitleggen eer de studenten richting de video‘s te wijzen, maar ik krijg die synthese nog niet goed genoeg gedaan.**
  MOETEN HIER DUS NOG ~7 slides met meer uitleg