

# Neural Networks Lecture 2

Data Science Toolbox Week 7

# Last time's Learning Objectives

- Learn what a perceptron is
- Learn what a multilayer perceptron is
- Learn how to measure loss for a neural network
- Learn how gradient descent and back propagation work

# Learning Objectives

- Discuss some of the challenges of applying neural networks to computer vision tasks and how they can be addressed with Convolutional Neural Networks.
- Discuss some of the challenges of applying neural networks to language tasks and how they can be addressed with Transformers.

# Image specific problems

1. If we represent an image with a flattened vector, a network doesn't know that pixels 10 and 11 are next to each other – how can it learn spatial patterns like edges, corners or textures?
2. If you take an image of a cat and shift every pixel one to the left, the input layer changes massively but the network still has to classify “catness” – how might it do that?
3. A 128x128 RGB image has 49152 values associated with it, how can we practically process images with networks without massively overparameterizing?
4. A local pattern like a vertical edge might appear anywhere on the image, but a standard NN would have to independently learn the same pattern in every location – how might we adjust for this?

# Convolutional Neural Networks (CNNs)

CNNs typically rely on a process of applying:

1. Convolutional Layers
2. Pooling Layers
3. Flattening vectors and putting them through regular linear layers (with non-linear activation functions)

*Every layer in a neural network is trying to compress data from higher dimensional space to lower dimensional space.*

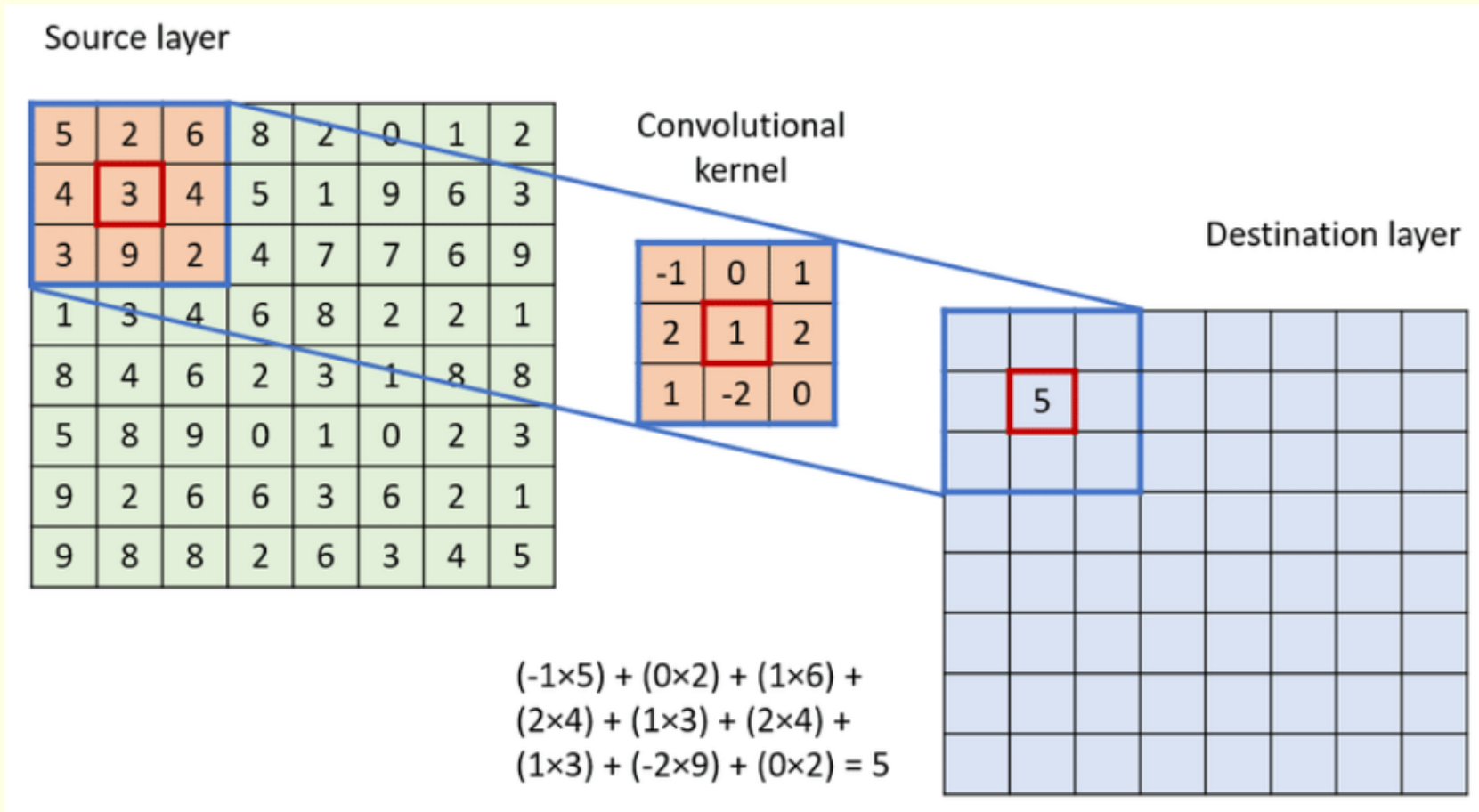
# Convolutional Kernel Layers – Part 1

- Pattern extraction layers, looking for abstract patterns like vertical edges or corners in earlier parts of the network as well as more concrete patterns in deeper parts of the network such a face or a wheel depending on context.
- Each kernel is a small matrix that you slide (or “convolve”) over the whole image, right to left, top to bottom. At each position, you perform the dot product for each number the kernel covers and sums the result.
- Each result is one value in the output feature map. The whole feature map tells you where your vertical edge/wheel appears.

# Convolutional Kernel Layers – Part 2

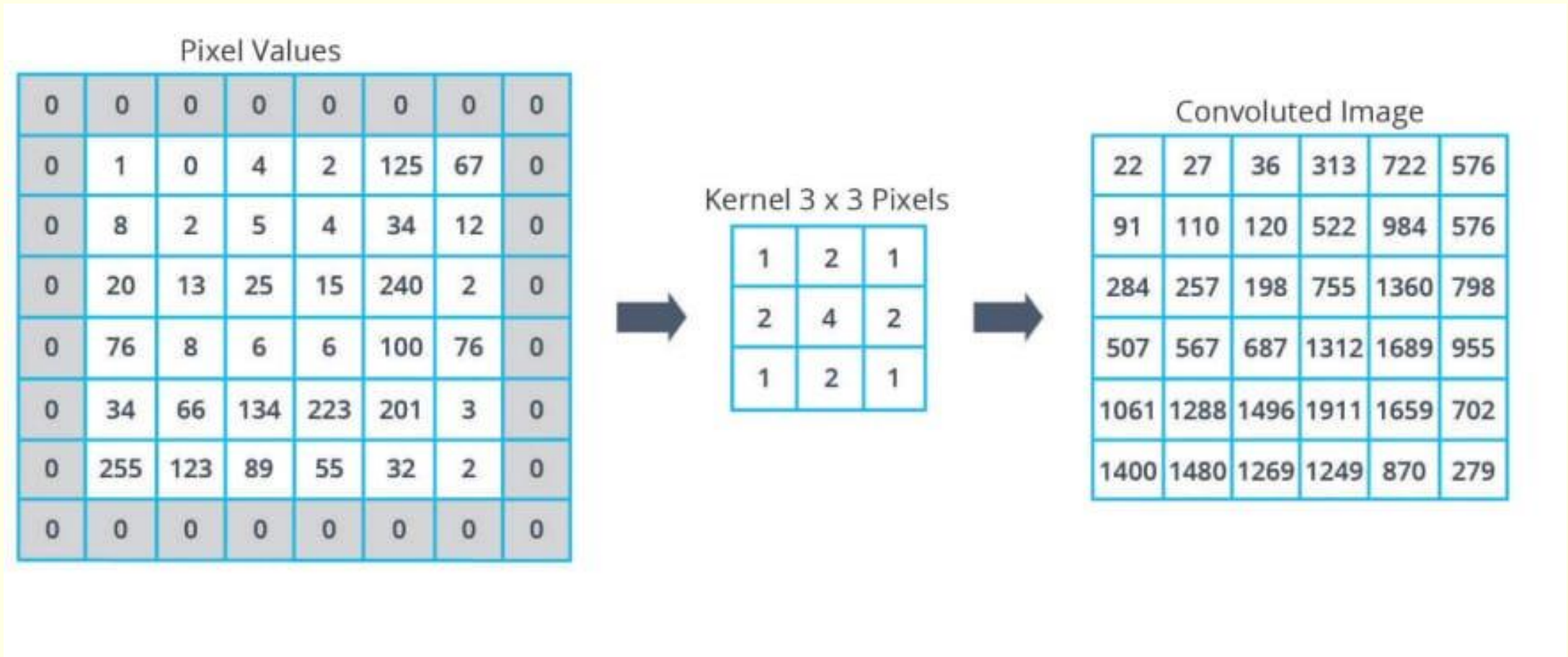
- A single convolutional kernel layer typically learns multiple features at once (e.g. 32 or 64, similar size to your hidden layers) so your output of the convolutional kernel layer will tell you about lots of different patterns.
- The numbers in the kernel layer are learnable parameters, just like the weights and biases of a regular linear layer.
- Kernels have hyperparameters like size, stride and padding.

# Kernel Calculation









# Padding for Kernels



# Examples of real Kernels

| <i>Original</i>  | <i>Gaussian Blur</i>  | <i>Sharpen</i>   | <i>Edge Detection</i>  |
|--|---|--|--|
| $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$                | $\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$    | $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$              | $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$          |
|  |  |  |  |

# Pooling Layers

- In order to further reduce the dimensionality of our data, we can use pooling layers where we attempt to summarise our image based on groups of values.
- Given a kernel size and stride, you can select what sort of pooling operation you want (common ones are max pooling, which returns the maximum across your kernel, or average pooling).

# Max Pool Example

|   |   |   |
|---|---|---|
| 1 | 0 | 2 |
| 3 | 1 | 2 |
| 1 | 2 | 3 |

Max = 3

|   |   |   |
|---|---|---|
| 1 | 0 | 2 |
| 3 | 1 | 2 |
| 1 | 2 | 3 |

Max = 2

|   |   |   |
|---|---|---|
| 1 | 0 | 2 |
| 3 | 1 | 2 |
| 1 | 2 | 3 |

Max = 3

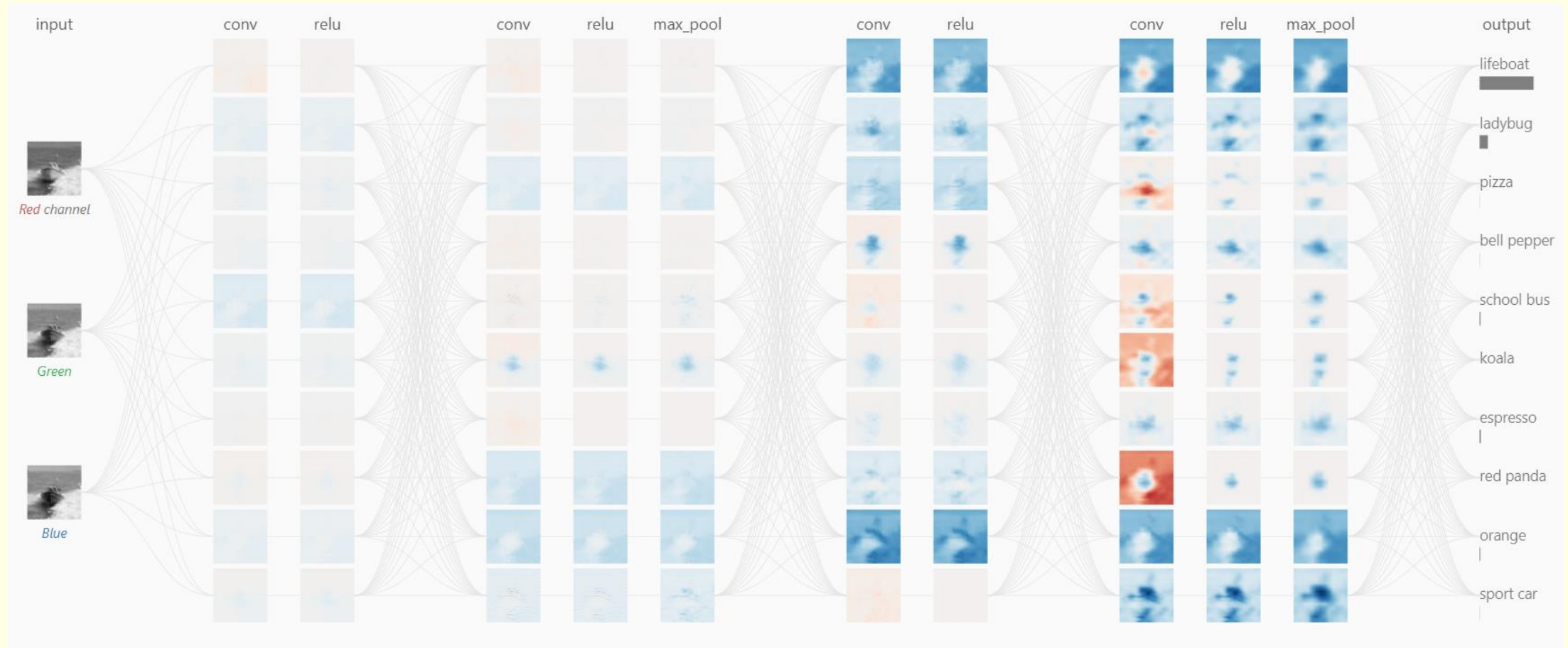
|   |   |   |
|---|---|---|
| 1 | 0 | 2 |
| 3 | 1 | 2 |
| 1 | 2 | 3 |

Max = 3

Output:

|   |   |
|---|---|
| 3 | 2 |
| 3 | 3 |

# The full pipeline:



<https://poloclub.github.io/cnn-explainer/>

# Language specific problems

1. How do you adapt numerical methods of learning to text inputs?
2. How do you capture the interdependency of words within a sentence in a single layer of a neural network?
3. How do you do this in parallel (instead of processing one token at a time)?
4. How do you learn to represent the same word in different contexts (e.g. river bank vs bank account)?

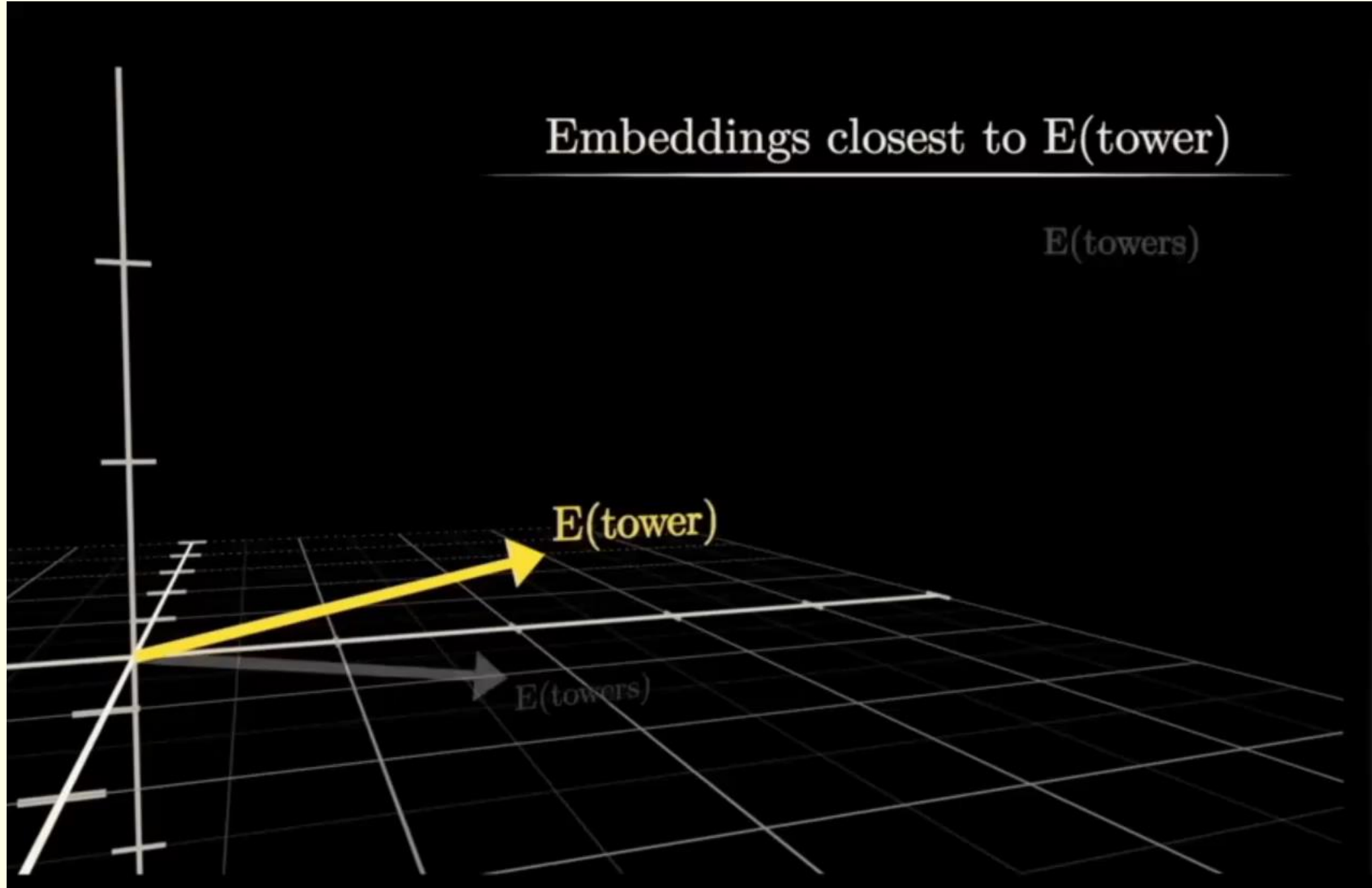
# Attention is about 30% of what you need

## Transformer Architecture:

1. Take an input sentence, split into tokens (words or parts of words) and embed each of the tokens into vectors
2. Feed these embedding vectors into an attention block, which updates embedding vectors based on how tokens relate to each other
3. Feed these updated embeddings into a standard multilayer perceptron which in some sense encodes information\*
4. Feed this back into an attention block and repeat this process many times (GPT3 has 96 blocks for example)

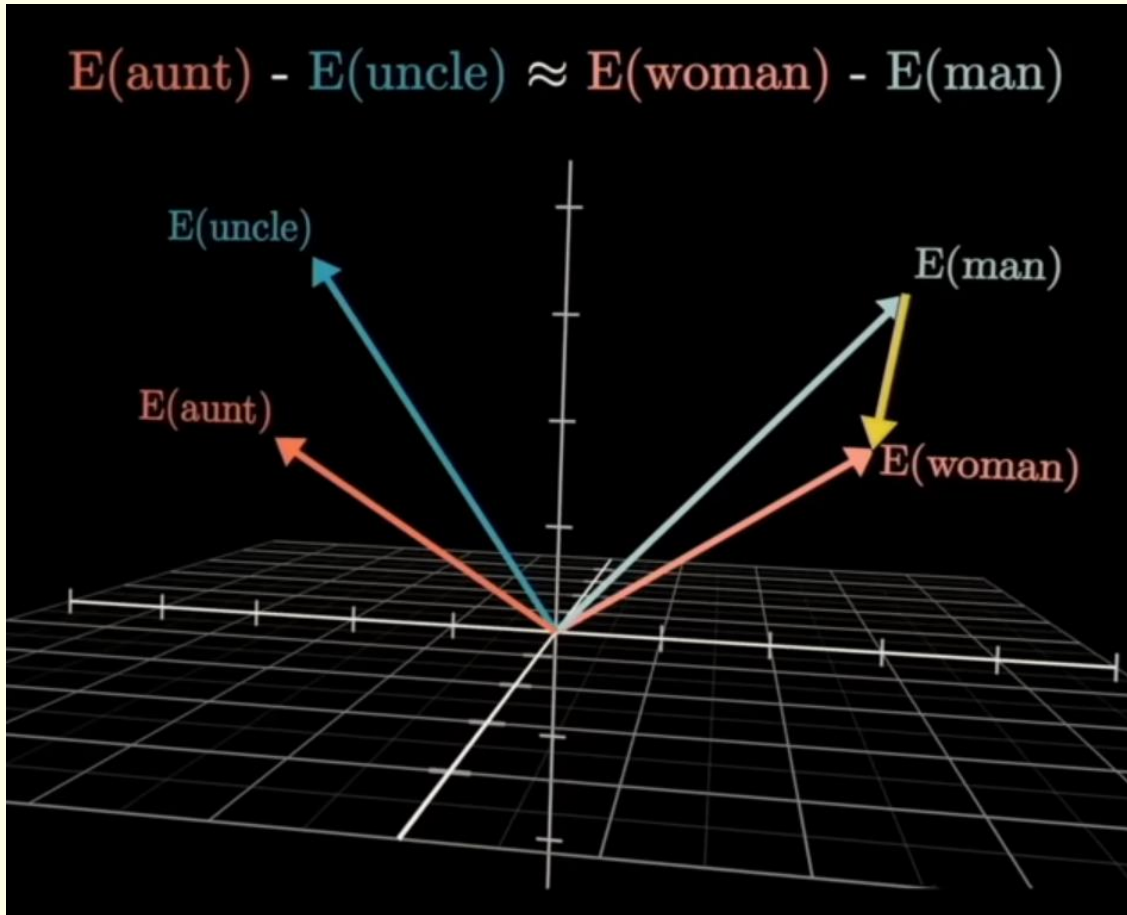


# Word2Vec





# Word2Vec

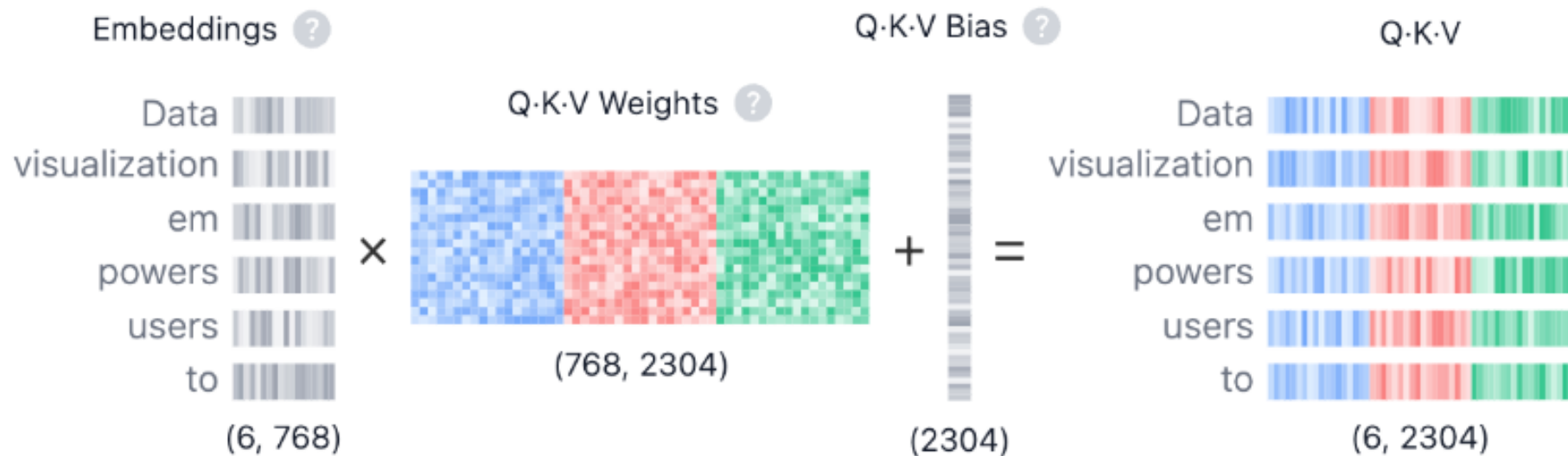


[Efficient Estimation of Word Representations in Vector Space \(2013\)](#)

- A pre-trained model that turns words/tokens into high dimensional embedding vectors.
- Words with similar meanings are embedded near to one another.
- Directions in this space in some sense represent different semantics.
- Useful for NLP tasks on words and short sentences but unable to capture context beyond aggregates.

- For each attention “Head” we learn three weight matrices:
1. The Query “Q” which ‘asks’ something about how certain tokens attend to one another e.g. does this adjective refer to this noun
  2. The Key “K” which is trained with the query to get the certain tokens to ‘attend’ to one another
  3. The Value “V” kicks in once we match the Query with the relevant Key and in essence ‘answers’ the question.

<https://poloclub.github.io/transformer-explainer/>

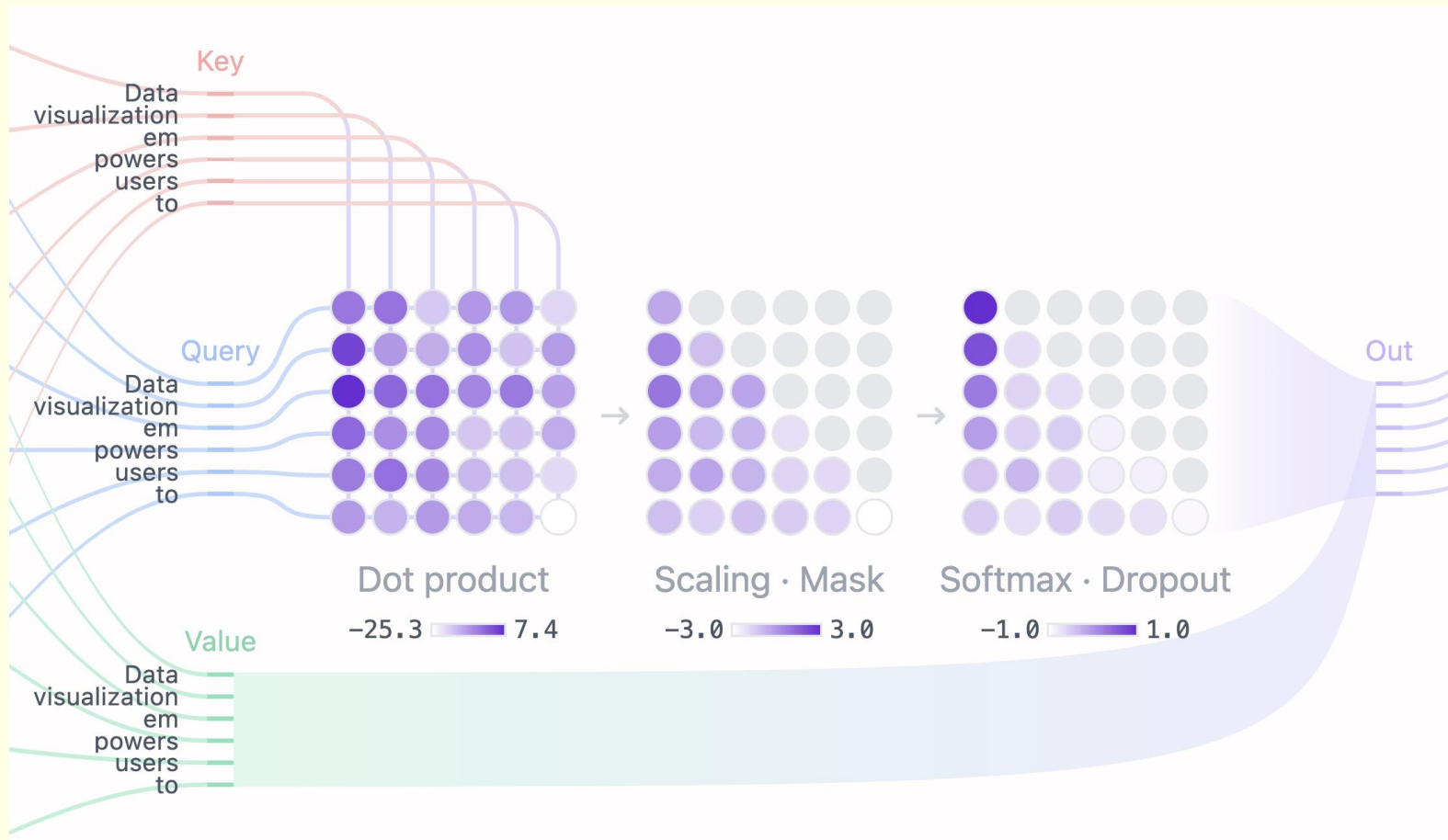


$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

# Multi-headed attention

- Each attention “head” has specific queries on how words in a sentence relate to one another.
- Obvious examples might relate adjectives to nouns, adverbs to verbs, nouns to pronouns or something more abstract which suggests what embedding updates should take place based on what’s there.
- Each head has a distinct key and query matrices which produce distinct attention patterns. These distinct value patterns which produce distinct updates through the key matrix.
- Every update from each attention head gets added up to the original embedding which flowed into the attention block.

# Masked Attention



- For standard attention we take an input sentence, hide the final token and try to predict it.
- We can cover up the last two tokens and try and predict the penultimate etc.
- But with standard attention, our predictor will benefit from knowing the future.
- We can use masked attention to hide information about words after what we're predicting.
- This involves setting future attention outputs to negative infinity before calculating softmax, so their output contribution is 0.

# The last token and over-squashing:

- As you go through multiple blocks you hope all your vectors gain richer and richer meanings to make you better and better at predicting next tokens.
- At the very end you want to look at the vector of the last token of the sentence and softmax it to get some probability of the next word in a sentence, you can calculate your loss here by comparing with actual examples of text.
- In theory, with enough layers of attention and MLPs, the last token is all you need to capture enough semantics about the whole sentence proceeding.
- In practise, this can lead to an effect called over-squashing if your context window is too large!

# Transformer Overview

- Transformers involve a combination of embedding, attention and a standard MLP.
- The way transformers are designed make them extremely efficient to train in parallel which is ideal for training on Graphical Processing Units (GPUs).
- They serve as the backbone for modern large language models, but can be also be used for computer vision tasks (1) and time series applications (2) and can be applied in other methods like graph neural networks (3)

1. [An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale](#)
2. [Transformers in Time Series: A Survey](#)
3. [Graph Attention Networks](#)

# Further reading/resources:

- Go and watch the latter four videos of 3Blue1Brown's videos on transformers and LLMs (especially the video on attention) : [https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1\\_67000Dx\\_ZCJB-3pi&si=gY0nycdjWu2kf80g](https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&si=gY0nycdjWu2kf80g)
- <https://poloclub.github.io/cnn-explainer/>
- <https://poloclub.github.io/transformer-explainer/>
- For the curious, a free one hour course on coding attention in PyTorch: <https://learn.deeplearning.ai/courses/attention-in-transformers-concepts-and-code-in-pytorch/lesson/han2t/introduction>