GPU's and Deep Learning (aka Neural Networks)

A very cursory glance at the relationship between GPU's, CUDA and Deep Learning

Jonathan Appavoo

GPU's and Deep Learning (aka Neural Networks)

A very cursory glance at the relationship between GPU's, CUDA and Deep Learning

Jonathan Appavoo

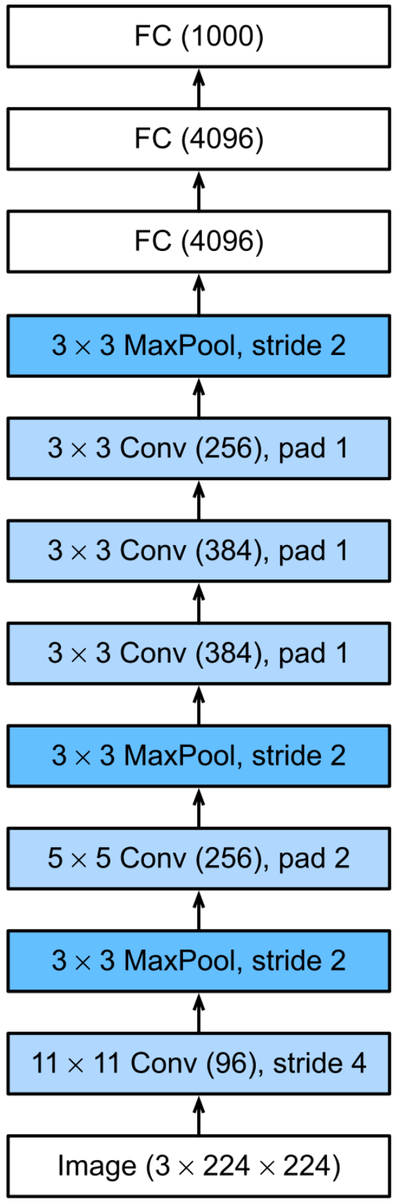
A good intro to CNN can be found here:

https://cs231n.github.io/convolutional-networks/

Deep Learning: Convolution Neural Networks

AlexNet 2012: A GPU powered break through

- 2008 State-of-the-art assumption: Computer Vision requires hand-crafted human engineering
- 1980s excitement around the success of "back-propagation" on various tasks
  - train networks of many layers, for more complex tasks, fails to produce results
  - Many incorrectly conclude 'not possible to learn weights of a deep network'
- Paper demonstrates that it can if scaled far beyond what most anticipated
- Convolutional Neural Networks (CNNs)
  - Leverage GPUs to scale CNN arch
  - Sept 30, 2012: win's ImageNet Large Scale Visual Recognition Challenge



https://dl.acm.org/doi/pdf/10.1145/3065386
https://en.wikipedia.org/wiki/AlexNet

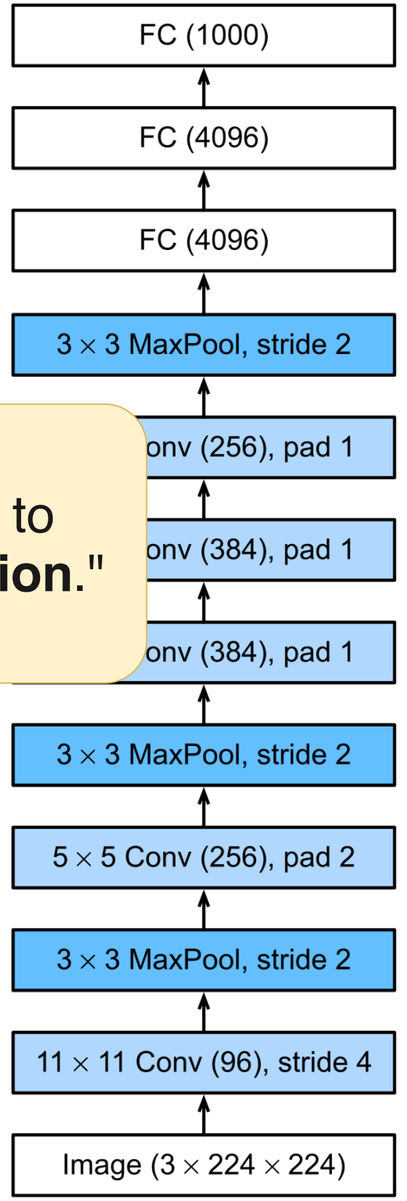
"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: Convolution Neural Networks

AlexNet 2012: A GPU powered break through

- 2008 State-of-the-art assumption: Computer Vision requires hand-crafted human engineering
- 1980s excitement around the success of "back-propagation" on various tasks
  - trai
  - Ma
    weights of a deep network'
- Paper demonstrates that it can if scaled far beyond what most anticipated
- Convolutional Neural Networks (CNNs)
  - Leverage GPUs to scale CNN arch
  - Sept 30, 2012: win's ImageNet Large Scale Visual Recognition Challenge

"Twenty years later, we know what went wrong: for deep neural networks to shine, **they needed far more labeled data and hugely more computation**."



https://dl.acm.org/doi/pdf/10.1145/3065386
https://en.wikipedia.org/wiki/AlexNet

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

CNN Success: Scale up Data and Compute

AlexNet 2012: Back-propogation + CUDA/GPU

"Luckily, current GPUs, paired with a highly optimized implementation of 2D convolution"

hugely more computation

Large digital labeled image data sets become a reality
ImageNet: "15 million labeled high-resolution images in over 22,000 categories"

far more labeled data

CNN's : Less computationally expensive than standard feedforward fully connected NN. (Fewer connections and parameters)
"still been **prohibitively expensive** to apply in large scale to high-resolution images."

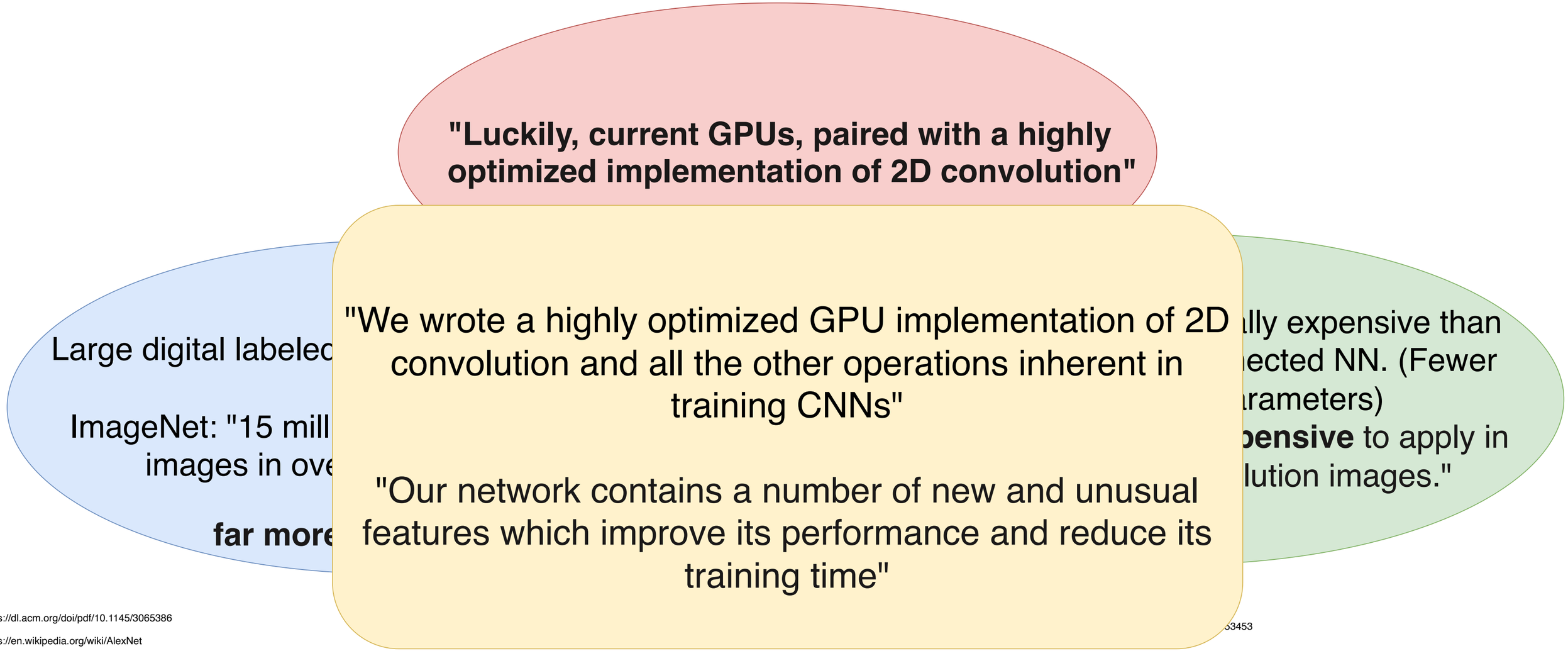
https://dl.acm.org/doi/pdf/10.1145/3065386

https://en.wikipedia.org/wiki/AlexNet

https://dl.acm.org/doi/10.1145/1553374.1553453

CNN Success: Scale up Data and Compute

AlexNet 2012: Back-propogation + CUDA/GPU



https://dl.acm.org/doi/pdf/10.1145/3065386

https://en.wikipedia.org/wiki/AlexNet

53453

Deep Learning: Ah the magic words

AlexNet 2012: A GPU powered break through

"In the end, the network's size is limited mainly by the amount of memory available on current GPUs and by the amount of training time that we are willing to tolerate. Our network takes between 5 and 6 days to train on **two GTX 580 3GB** GPUs. ***All of our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available***."

https://www.nvidia.com/docs/IO/100940/GeForce_GTX_580_Datasheet.pdf

Compute Capability 2.0
CUDA 4 (I think)

512 stream processors, grouped in 16 stream multiprocessors clusters (each with 32 CUDA cores)
~$500-$600 USD release price

https://dl.acm.org/doi/pdf/10.1145/3065386

https://en.wikipedia.org/wiki/AlexNet

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: How GPU's were used

AlexNet 2012: A GPU powered break through

- "A single GTX 580 GPU has only 3GB of memory, which limits the maximum size of the networks that can be trained on it.
- 1.2 million training examples are enough to train networks which are too big to fit on one GPU
- "spread the net across two GPUs"
- "read from and write to one another’s memory directly, without going through host machine memory"*

\

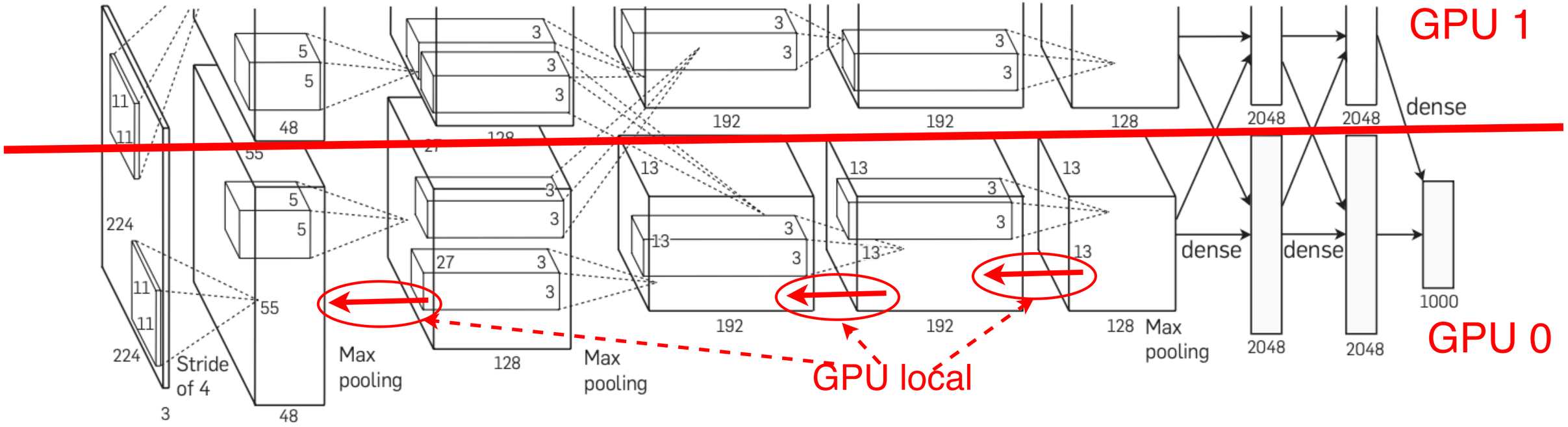
https://dl.acm.org/doi/pdf/10.1145/3065386

https://en.wikipedia.org/wiki/AlexNet

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: How GPU's were used

AlexNet 2012: A GPU powered break through



- "half of the kernels (or neurons) on each GPU"
- "trick: the GPUs communicate only in certain layers."
  - precisely tune the amount of communication until it is an acceptable fraction of the amount of computation

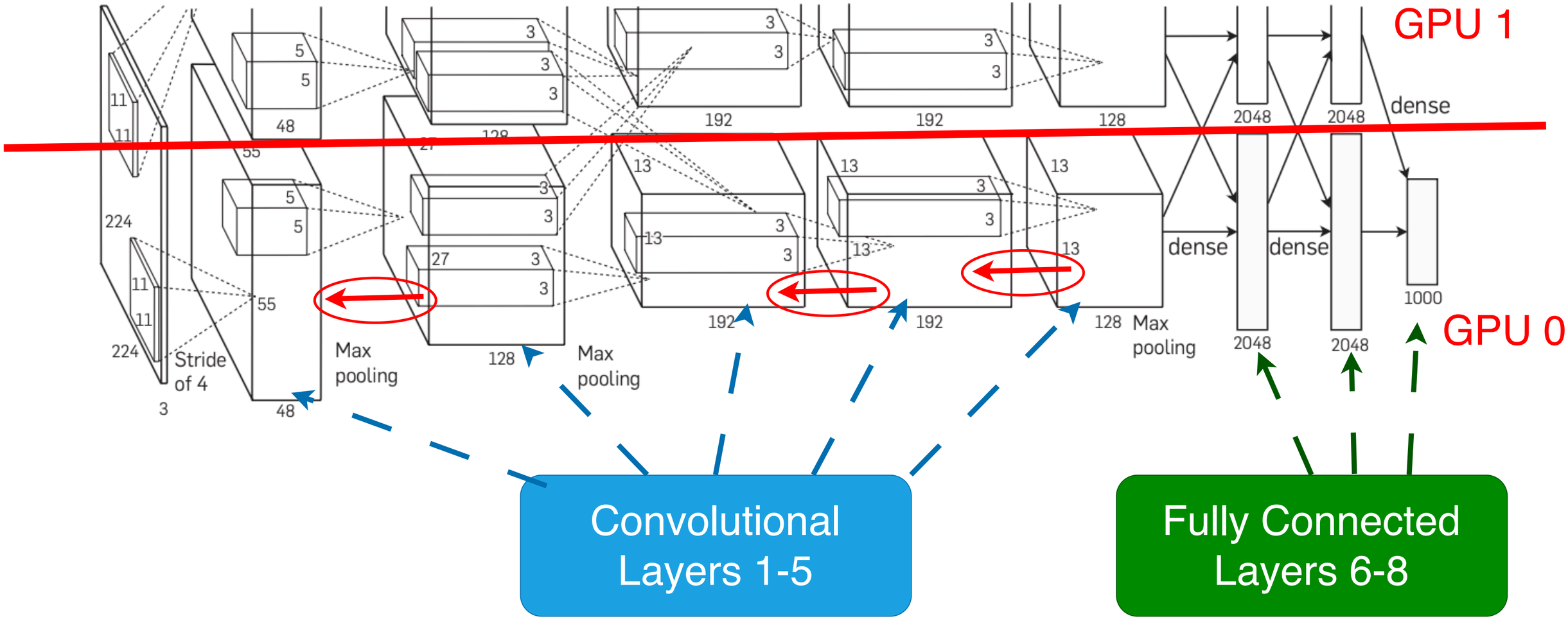
https://dl.acm.org/doi/pdf/10.1145/3065386

https://en.wikipedia.org/wiki/AlexNet

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: The layers

AlexNet 2012: A GPU powered break through



https://dl.acm.org/doi/pdf/10.1145/3065386

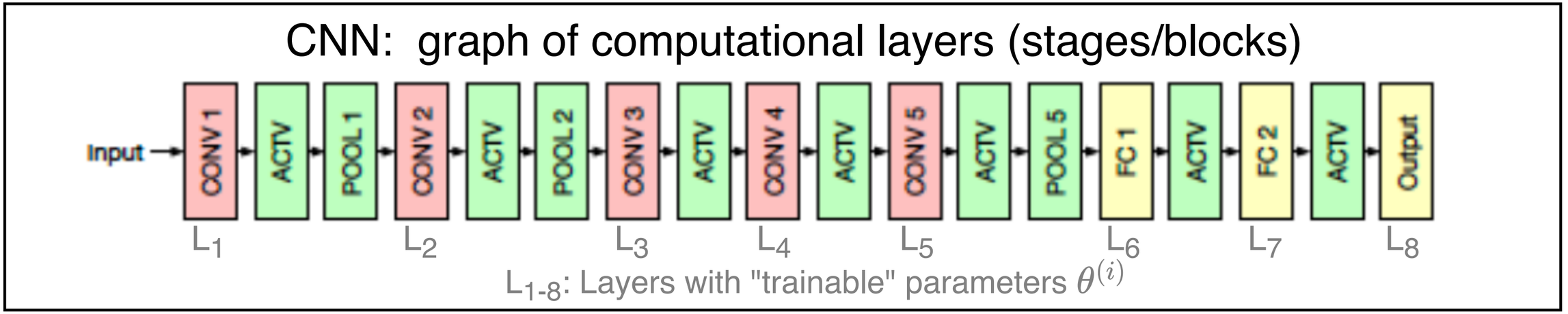
https://en.wikipedia.org/wiki/AlexNet

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: CNN architecture

AlexNet 2012: Layers of parallel computations

- type of Deep Neural Network (DNN) Machine Learning algorithm
- used for image classification, action recognition, etc.



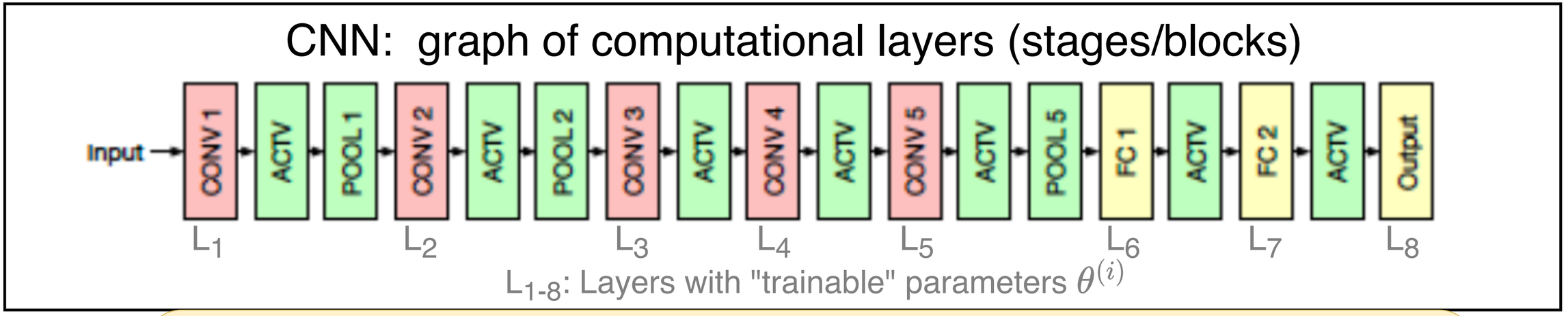
- Example: AlexNet shown with 17 "layers" :
  - each performs a math operation: $x^{(i)} = f^{(i)}(x^{(i-1)}, \theta^{(i)})$
  - $\theta^{(i)} = \emptyset$ for layers without trainable parameters
  - layer input is output from prior layer with $x^{(0)}$ the input to network: x and $x^{(8)} = y$ output of network (layer output is also called "feature maps"

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: CNN architecture

AlexNet 2012: Layers of parallel computations

- type of Deep Neural Network (DNN) Machine Learning algorithm
- used for image classification, action recognition, etc.



- Examp
  - each
  - $\theta^{(i)}$ =
  - layer
    $x^{(8)} = y$ output of network (layer output is also called "feature maps"

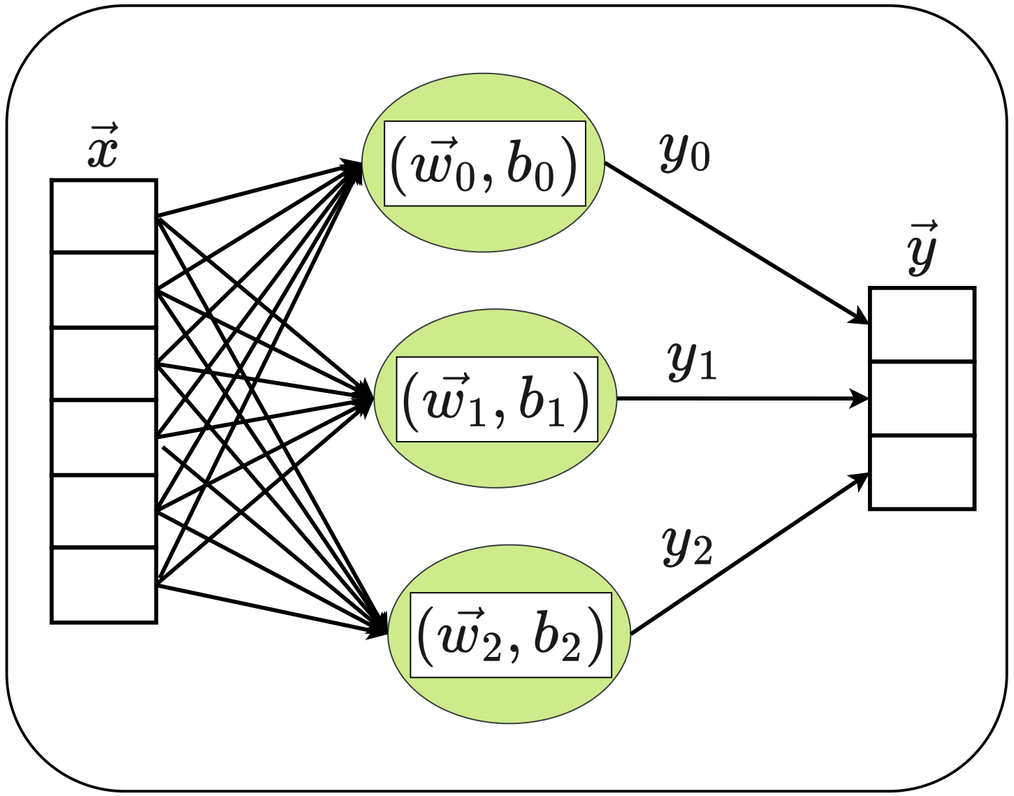
Don't forget at any given layer there can be many "neurons" composing the layer: parallelism within a layer

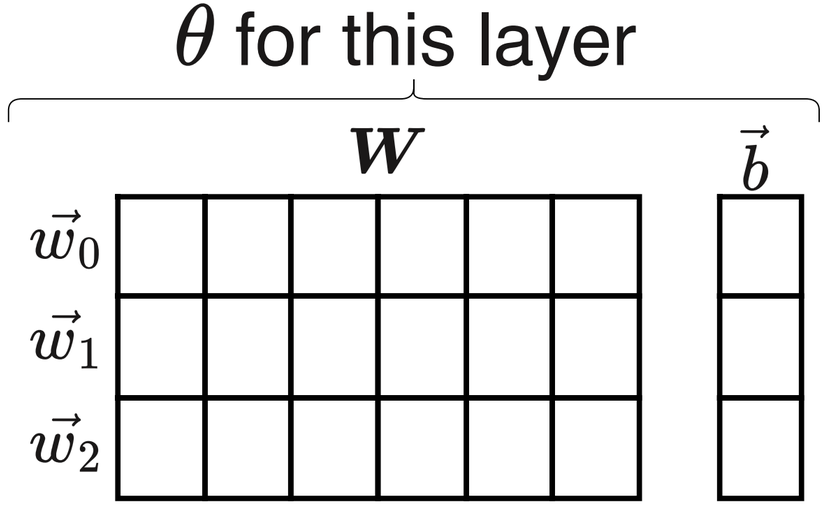
"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: Layer Types: fully connected

AlexNet 2012: Small reusable set of layer functions (scalable design)

- fully connected layer: $\vec{y} = \boldsymbol{W}\vec{x} + \vec{b}$ where $y_i = \vec{w}_i{}^{\mathsf{T}}\vec{x} + b_i$



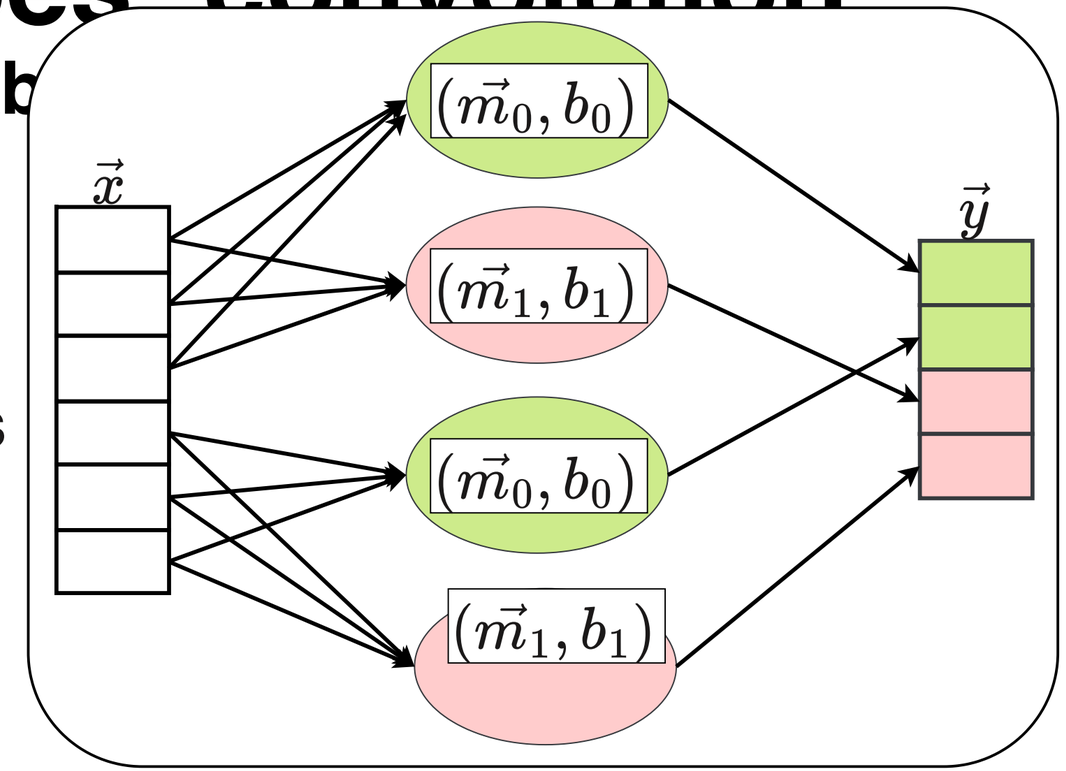


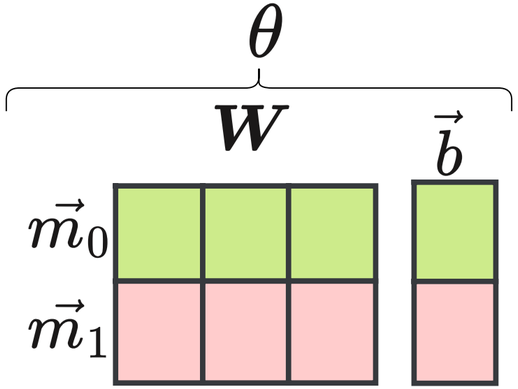
"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: Layer Types: convolution

AlexNet 2012: Small reusable set (scalab

- Convolution layer:
  - Output is formed from several independent convolutions masks (filters/kernels) and bias
  - A layers "hyper-parameters"
    - mask dimension (eg, 3x3)
    - depth (number of filter types)
    - stride (step) (eg 1x1, 3x3)
    - zero-padding (number of zeros)
  - Each filter type has one mask and bias for each "channel" of input (not illustrated here)
- Masks/filters are usually called features

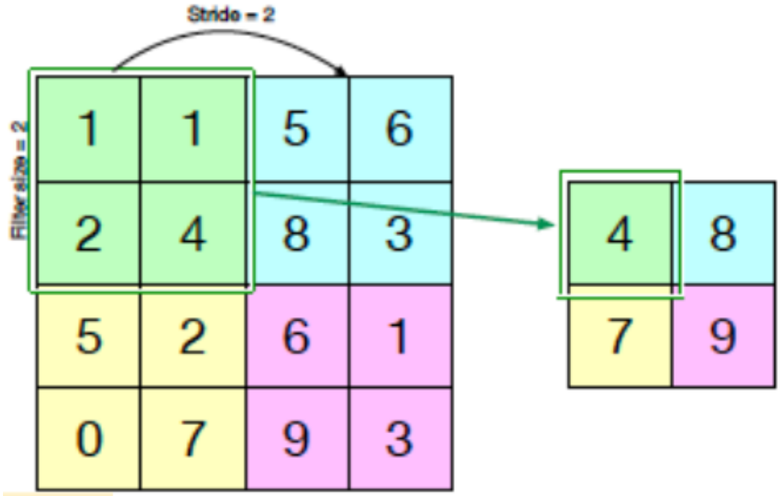




Deep Learning: activation, pooling & dropout

AlexNet 2012: Small reusable set (scalable design)

- Activation: $\vec{y} = \sigma(\vec{x})$ nonlinear σ applied to every element to produce output element; e.g. ReLU = $\max(0, \vec{x})$



- Pooling: down samples input reduces output size

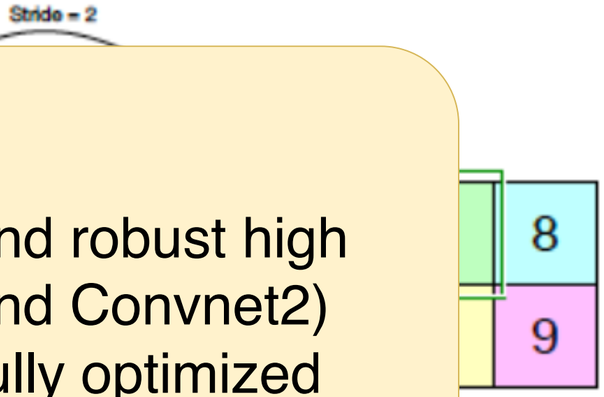
- Drop out a simple layer that used with first two fully connected layers to probabilistically set outputs to zero's for some neurons (helps with overfitting)

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: activation, pooling & dropout

AlexNet 2012: Small reusable set (scalable design)

- Activation: $y = \sigma(x)$ nonlinear σ applied to every element to produce output element; e.g. ReLU = $\max(0, x)$
- Pooling:



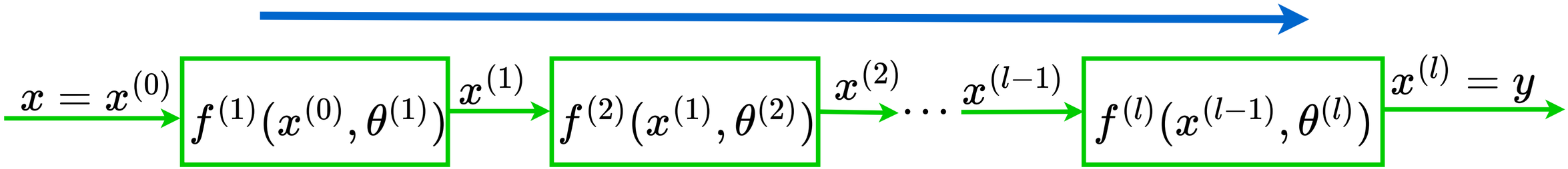
Note all operations are parallelizable -- but portable and robust high performance on GPU still takes work (see CUDNN and Convnet2) and of course memory usage and I/O must be carefully optimized

- Drop out a simple layer that used with first two fully connected layers to probabilistically set outputs to zero's for some neurons (helps with overfitting)

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al. 2012

Deep Learning: CNN Inference

Inference is forward propagation through network



Mathematically a composition of functions:

$$y = F(x, \theta) = f^{(l)} \circ f^{(l-1)} \circ \ldots \circ f^{(1)}(x, \theta)$$

where $\theta = \{\theta^{(i)} | i = 1 \ldots l\}$ are trainable parameters.

Given that we know what the functions we also know their partial derivatives:

$$\frac{\partial f^{(i)}}{\partial x^{(i)}} \text{and} \frac{\partial f^{(i)}}{\partial \theta^{(i)}}$$

(with care we can derive closed form partial derivatives for each layer functions)

Deep Learning: Training I: Back-propagating errors

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams https://www.nature.com/articles/323533a0

- Inference can only work if network properly trained
- Training: calibrate weights using inference results on n training data pairs:
- Use Error (loss/cost) function: $E(y, y^*)$ e.g. MSE, $E = \frac{1}{n}\sum_{k=0}^{n}(y_k - y_k^*)^2$
  - where y_k^* is correct output for input x_k in training data ("data label")
- Want to find network parameters θ that minimize E
- Common strategy: stochastic gradient decent -- iterative update passes of θ as follows:

$$\theta_{new} = \theta_{old} - \eta \nabla E$$

Where ∇E is the gradient of the Error function calculated "over" the training data. The goal is to adjust θ such that error of the network on the training data is minimized.

Deep Learning: Training I: Back-propagating errors

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams https://www.nature.com/articles/323533a0

- Inference can only work if network properly trained
- Training: calibrate weights using inference results on n training data pairs:
- Use Error (loss/cost) function: $E(y, y^*)$ e.g. MSE, $E = \frac{1}{n}\sum_{k=0}^{n}\left(y_k - y_k^*\right)^2$
  - where y_k^* is correct output for input x_k in training data ("data label")
- Want to find network parameters θ that minimize E
- Common strategy: stochastic gradient decent -- iterative update passes of θ as follows:

Learning Rate

$$\theta_{new} = \theta_{old} - \eta \nabla E$$

Where ∇E is the gradient of the Error function calculated "over" the training data. The goal is to adjust θ such that error of the network on the training data is minimized.

Deep Learning: Training II: Gradient

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams https://www.nature.com/articles/323533a0

$$\theta_{new} = \theta_{old} - \eta \nabla E$$

What is it and how to compute it?

∇E is a vector whose entries tell us how the error function will change with respect to a parameter of the network (eg a particular weight).

Which formally are the partial derivatives of E with respect to each parameter **over the training data**

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

Deep Learning: Training II: Gradient

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams https://www.nature.com/articles/323533a0

$$\theta_{new} = \theta_{old} - \eta \nabla E$$

∇E is a vector whose entries tell us how the error function will change with respect to a parameter of the network (eg a particular weight).

Which formally are the partial derivatives of E with respect to each parameter **over the training data**

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

Tells direction and magnitude that will maximize Error (steepest ascent)

Deep Learning: Training III: Gradient

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams https://www.nature.com/articles/323533a0

$$\theta_{new} = \theta_{old} - \eta \nabla E \quad \nabla E = \begin{bmatrix} \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

Assuming we have a way to concretely compute these partial derivatives evaluated over the training data, we can update our network parameters using it (subtract scaled by η) :

$$\begin{bmatrix} \theta_{new}^{(1)} \\ \vdots \\ \theta_{new}^{(l)} \end{bmatrix} = \begin{bmatrix} \theta_{old}^{(1)} \\ \vdots \\ \theta_{old}^{(l)} \end{bmatrix} - \begin{bmatrix} \eta \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \eta \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

Deep Learning: Training III: Gradient

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams

https://www.nature.com/articles/323533a0

$$\theta_{new} = \theta_{old} - \eta \nabla E \quad \nabla E = \begin{bmatrix} \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \end{bmatrix}$$

It is essential to remember that, for our purposes, we are trying to consider how this computation can be implemented on the GPU:
Memory & compute requirements, parallelism, etc.

Remember, each entry is really a row of numbers that must be calculated after the training data has been "passed forward."

Even AlexNet has 60 Million Parameters that need to be updated.

Ass
eval
usin

$$\begin{bmatrix} \vdots \\ \theta_{new}^{(l)} \end{bmatrix} = \begin{bmatrix} \vdots \\ \theta_{old}^{(l)} \end{bmatrix} - \begin{bmatrix} \vdots \\ \eta \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

https://cs231n.github.io/convolutional-networks/

Deep Learning: Training III: Calculus

https://www.nature.com/articles/323533a0

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams

Given that the network is really just a composition of functions, we can derive the gradient values for a layer as the multiplication of the derivatives using Chain Rule from calculus

$$y = F(x, \theta) = f^{(l)} \circ f^{(l-1)} \circ \ldots \circ f^{(1)}(x, \theta)$$

where $\theta = \{\theta^{(i)} | i = 1 \ldots l\}$ are trainable parameters.

not shown: 1) see paper, 2) paper I suspect is good linked at the bottom, and 3) lots of tutorials and explanations on the net

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

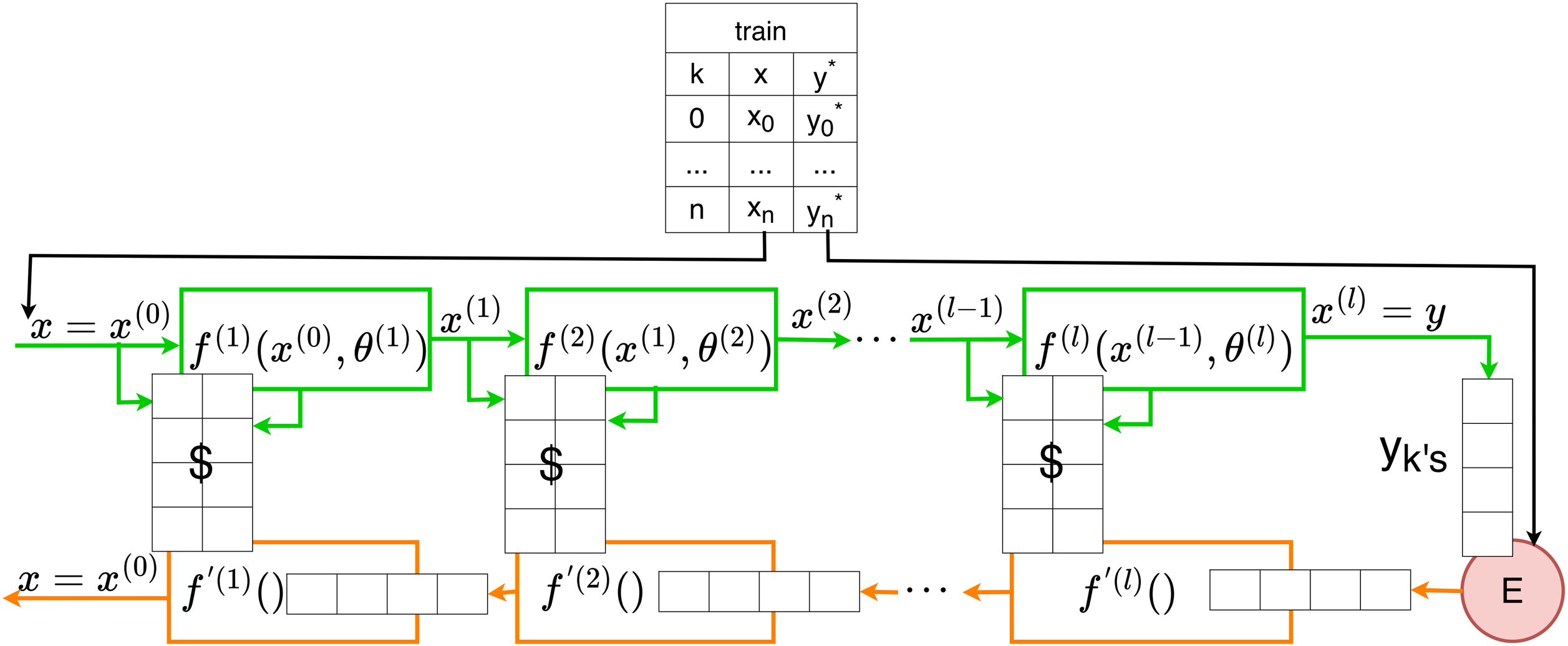
This means we can do a layer-by-layer update of the network parameters in the reverse direction

$$\begin{bmatrix} \theta^{(1)}_{new} \\ \vdots \\ \theta^{(l)}_{new} \end{bmatrix} = \begin{bmatrix} \theta^{(1)}_{old} \\ \vdots \\ \theta^{(l)}_{old} \end{bmatrix} - \begin{bmatrix} \eta \frac{\partial E}{\partial \theta^{(1)}} \\ \vdots \\ \eta \frac{\partial E}{\partial \theta^{(l)}} \end{bmatrix}$$

http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Deep Learning: Training IV: Forward & Backward

https://www.nature.com/articles/323533a0

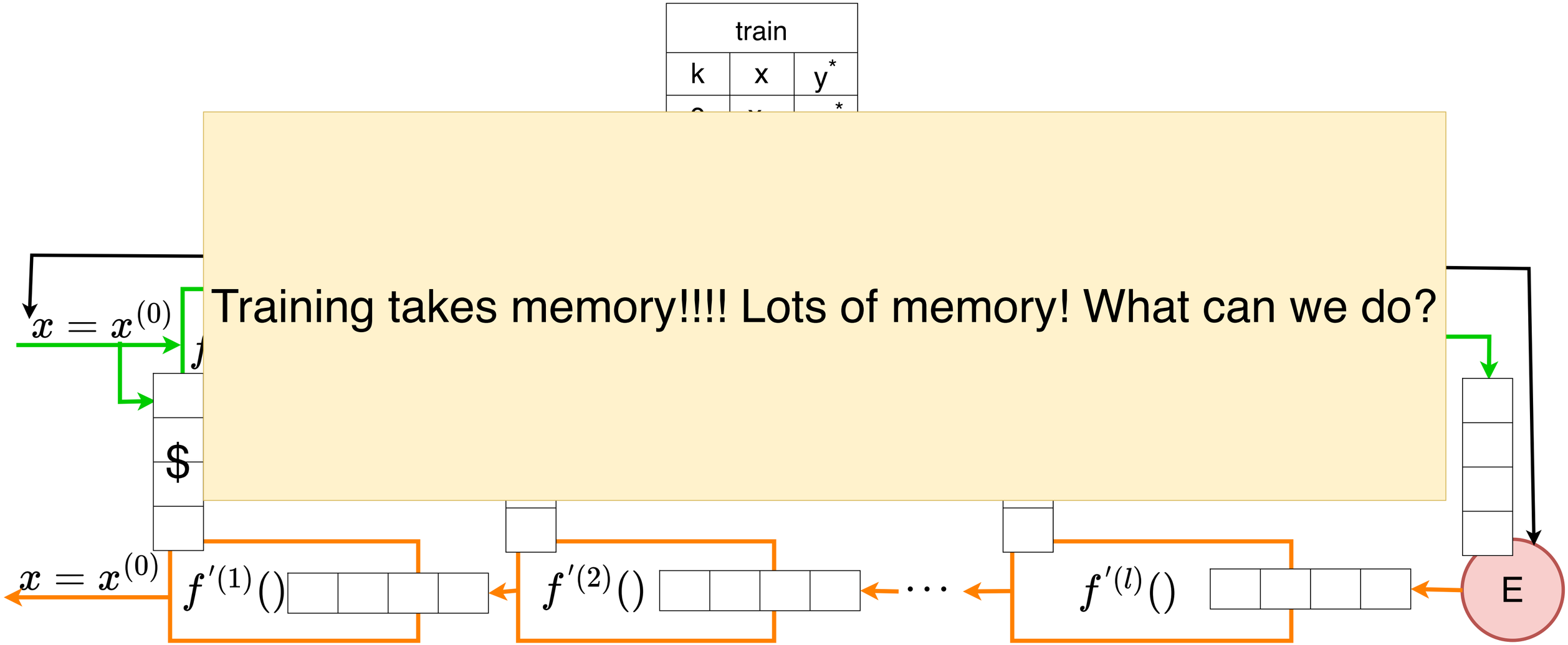


Calc gradient using cached & incoming values, pass new values along and update $\theta^{(i)}$

Note exact values cached depends on the layer type

Deep Learning: Training IV: Forward & Backward

https://www.nature.com/articles/323533a0



Deep Learning: Training V: Caveat

"Learning representations by back-propagating errors", Seminal paper by Rumelhart, Hinton and Williams https://www.nature.com/articles/323533a0

I have been somewhat sloppy with notation and skipped various details. Main point is the flavor of the computation and implications on memory and communications.

Remember despite my notation everything is being done on vectors, matrices and tensors. See for a review Vector, Matrix and Tensor derivatives

https://cs231n.stanford.edu/vecDerivs.pdf

Deep Learning: CUDNN

https://developer.nvidia.com/cudnn

- 2014 NVIDIA releases their own CUDA DNN library
  - "straightforward" implementation of standard deep learning functions
  - "The convolution is not as obvious"
  - Lower convolution into matrix multiple
    - efficient since matmul is so highly optimized ;-)
    - Avoids "materializing" lowered matrices in memory
      - by lazily materializing in on-chip memory

https://arxiv.org/abs/1410.0759

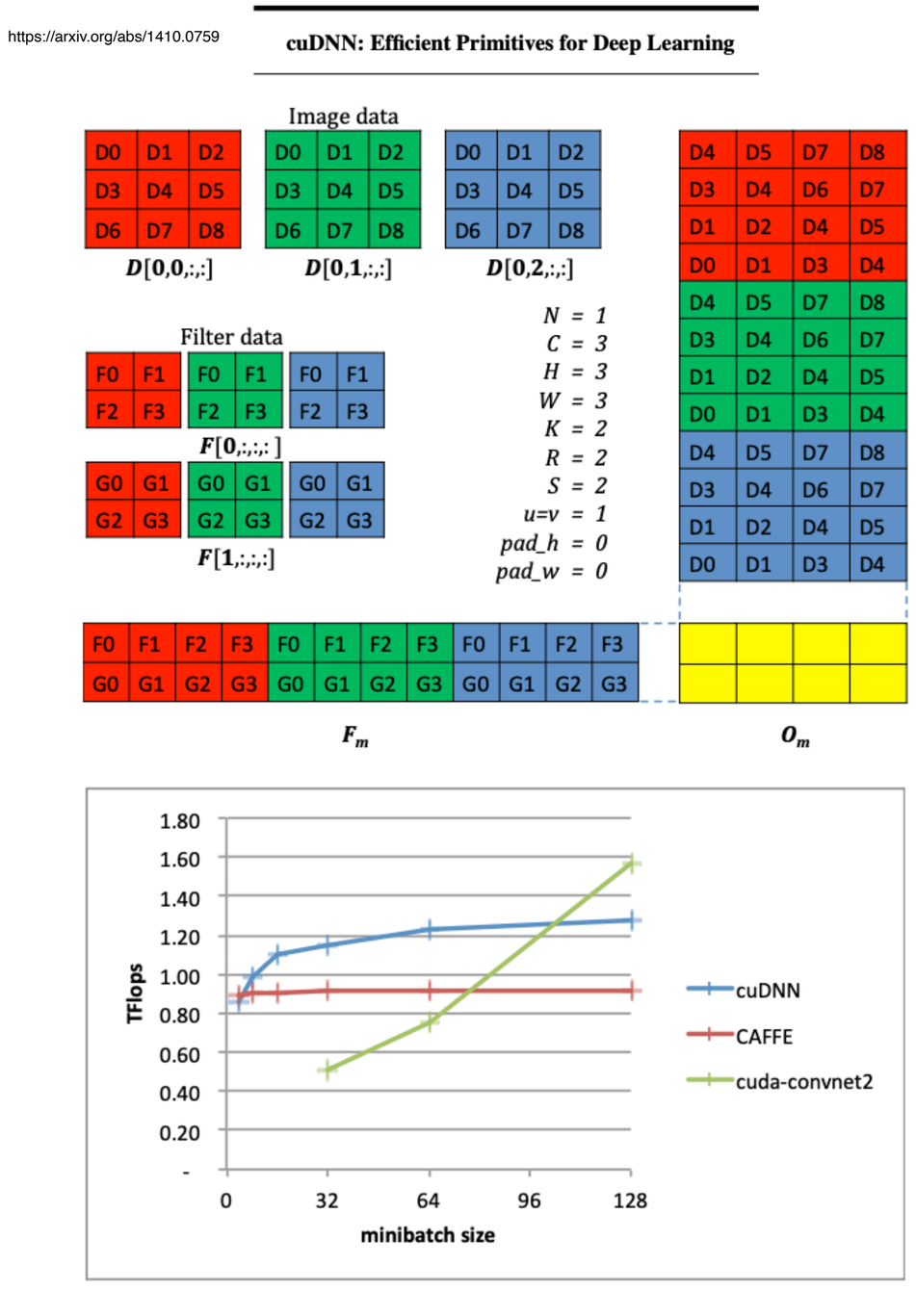


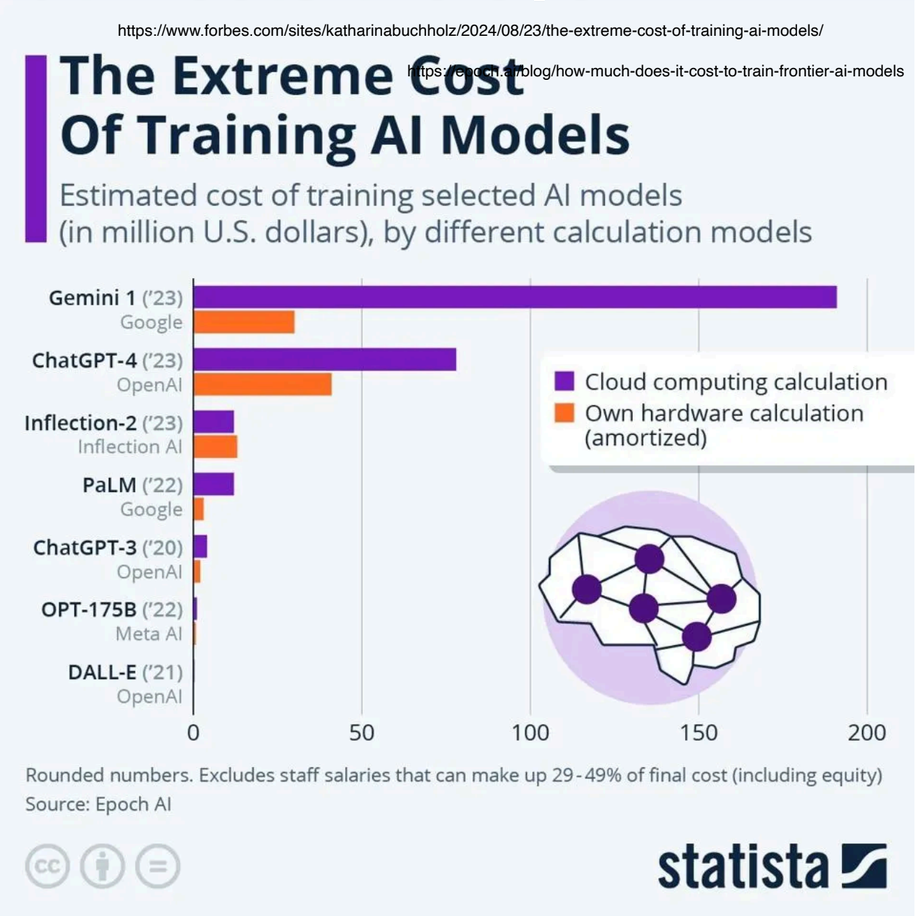
Figure 2: Comparative Performance

Deep Learning: LLMs: Transformers

More that meets the eve

https://www.forbes.com/sites/katharinabuchholz/2024/08/23/the-extreme-cost-of-training-ai-models/

https://epoch.ai/blog/how-much-does-it-cost-to-train-frontier-ai-models



Attention Is All You Need
Vaswani et al. 2017

https://arxiv.org/abs/1706.03762

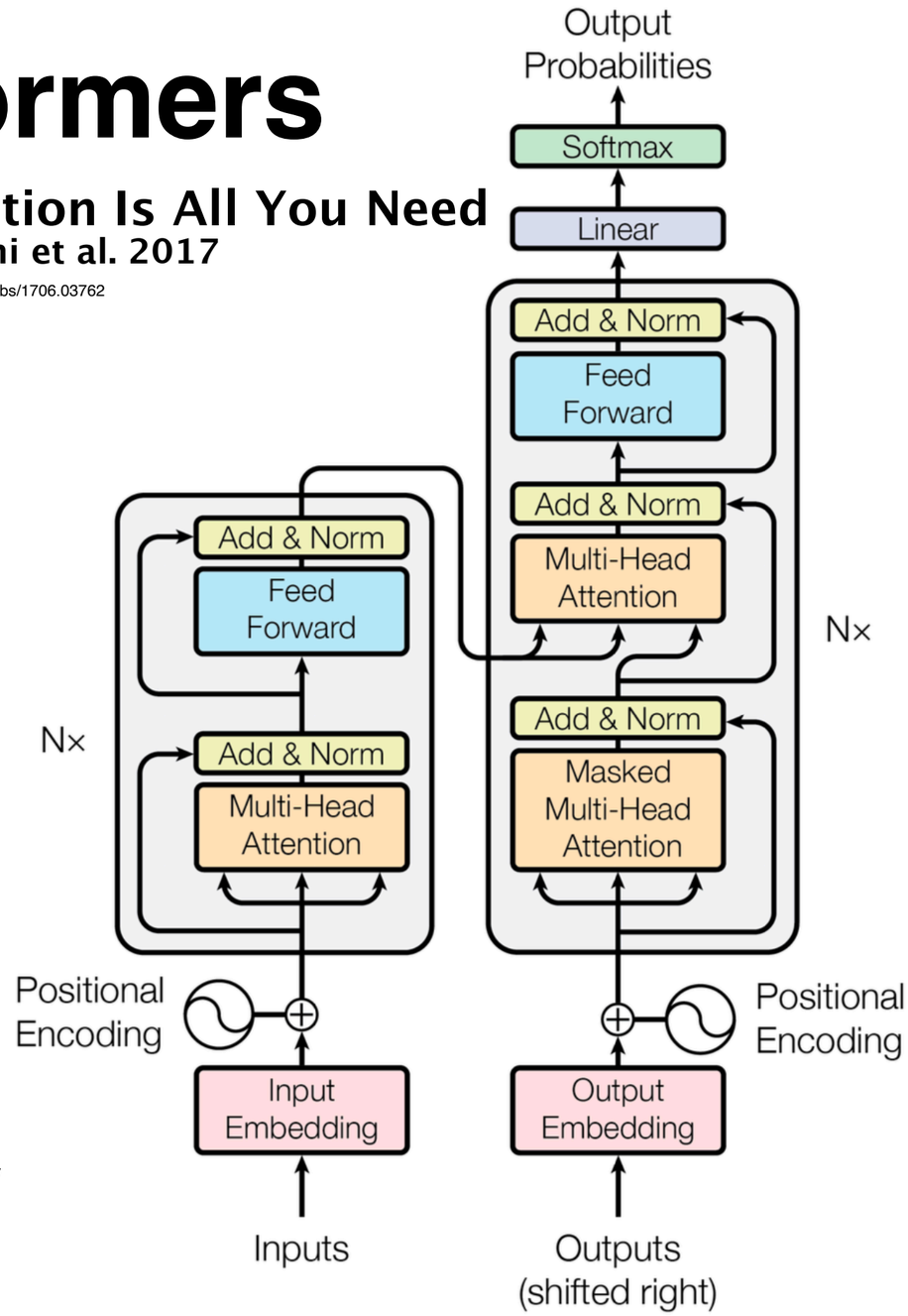
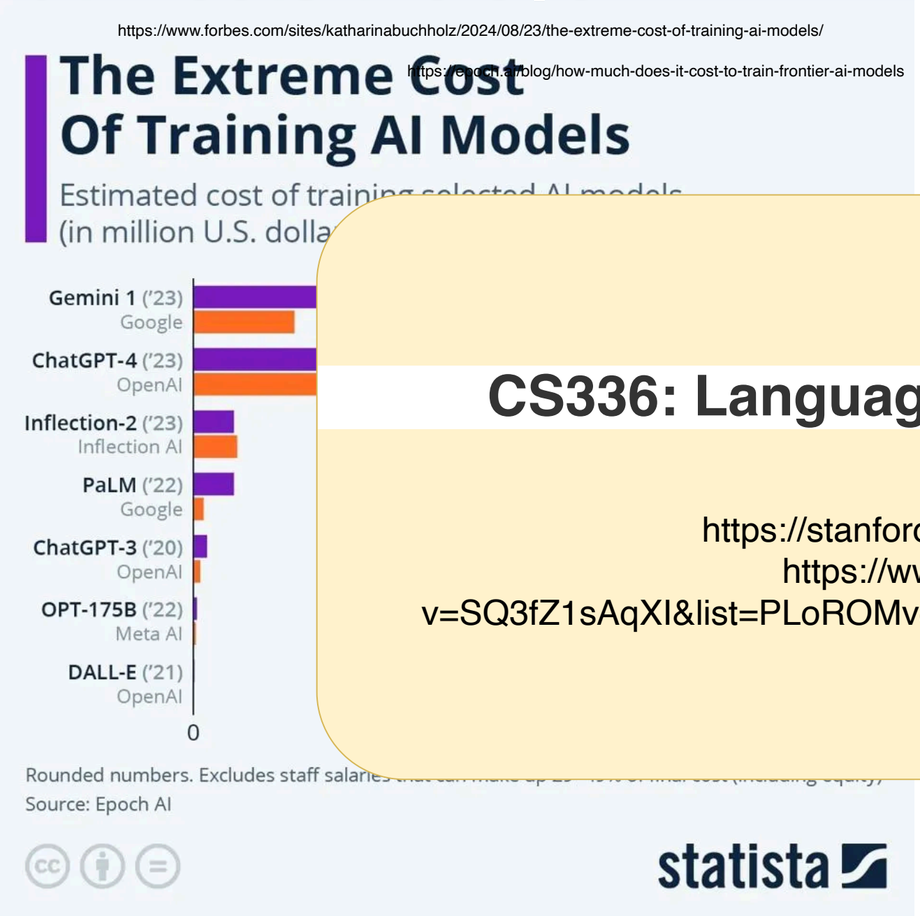


Figure 1: The Transformer - model architecture.

- https://developer.nvidia.com/blog/accelerating-transformers-with-nvidia-cudnn-9/
- llm.c:
  - https://github.com/karpathy/llm.c
  - https://www.youtube.com/watch?v=aR6CzM0x-g0

Deep Learning: LLMs: Transformers

More that meets the eve



CS336: Language Modeling from Scratch

https://stanford-cs336.github.io/spring2025/
https://www.youtube.com/watch?v=SQ3fZ1sAqXI&list=PLoROMvodv4rOY23Y0BoGoBGgQ1zmU_MT_&index=2

Attention Is All You Need
Vaswani et al. 2017
https://arxiv.org/abs/1706.03762

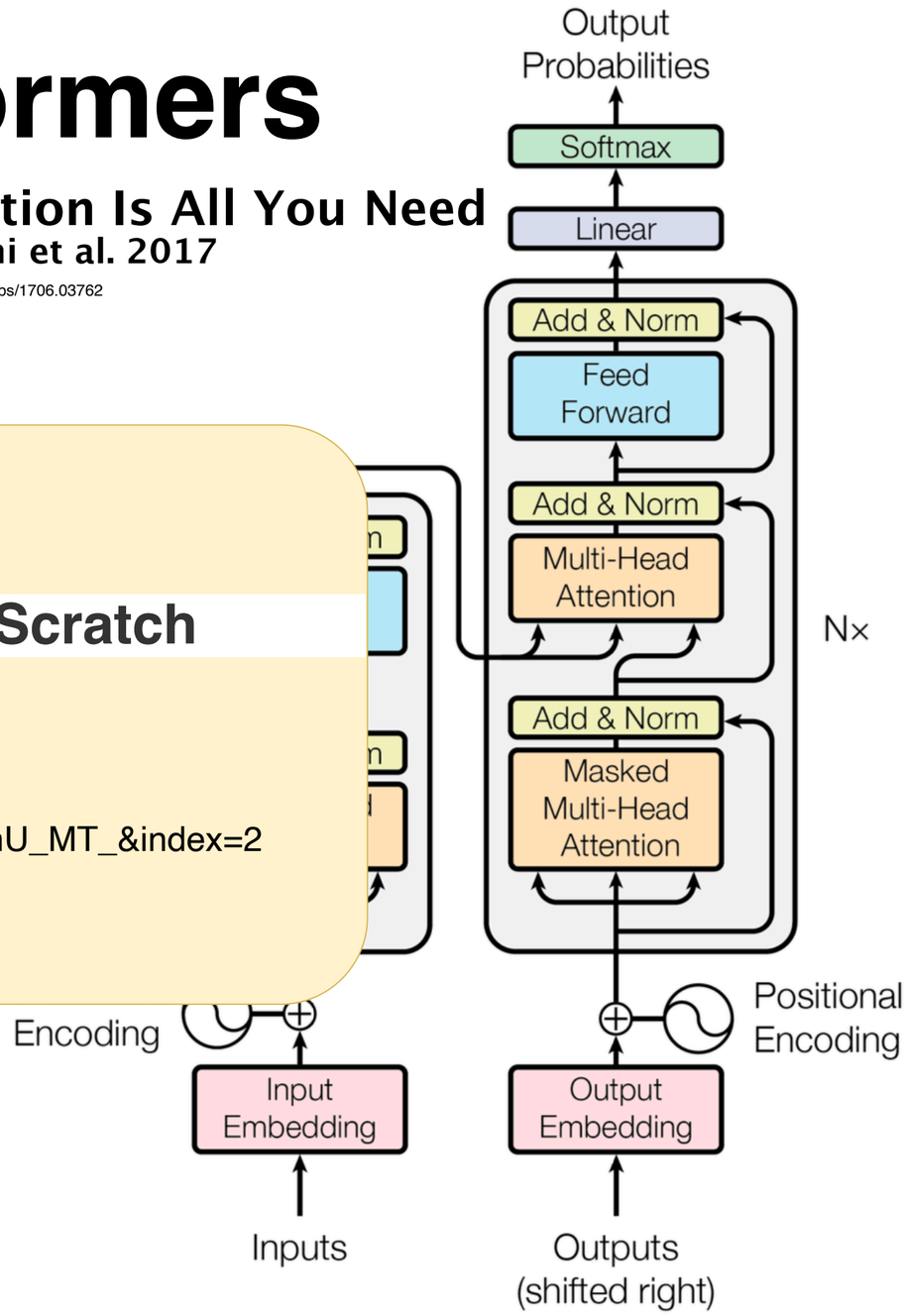


Figure 1: The Transformer - model architecture.

- https://developer.nvidia.com/blog/accelerating-transformers-with-nvidia-cudnn-9/
- llm.c:
  - https://github.com/karpathy/llm.c
  - https://www.youtube.com/watch?v=aR6CzM0x-g0