Lecture 16: Deep Learning

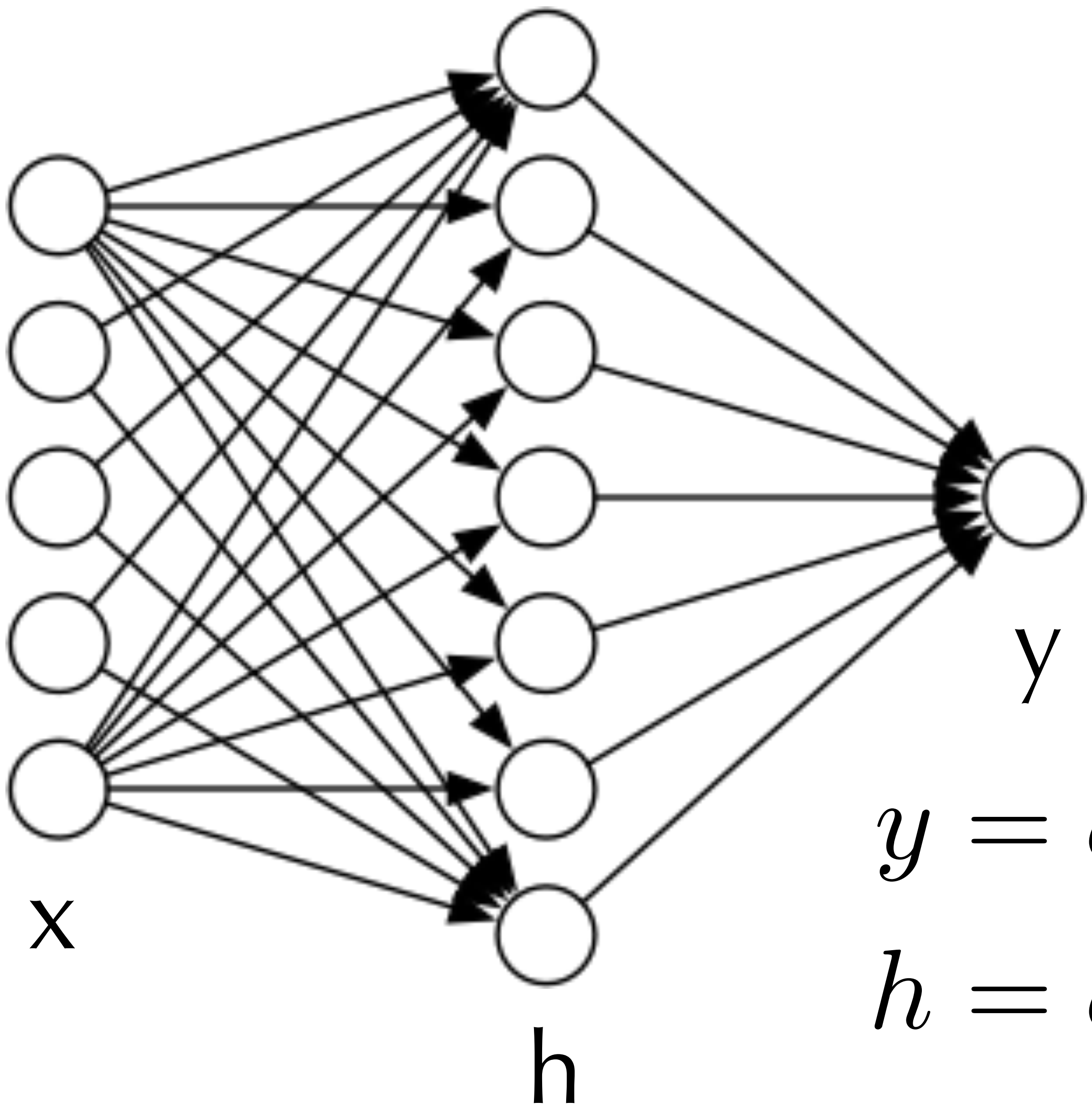
Statistical Machine Learning

Tom Rainforth

12/03/20

UNIVERSITY OF OXFORD

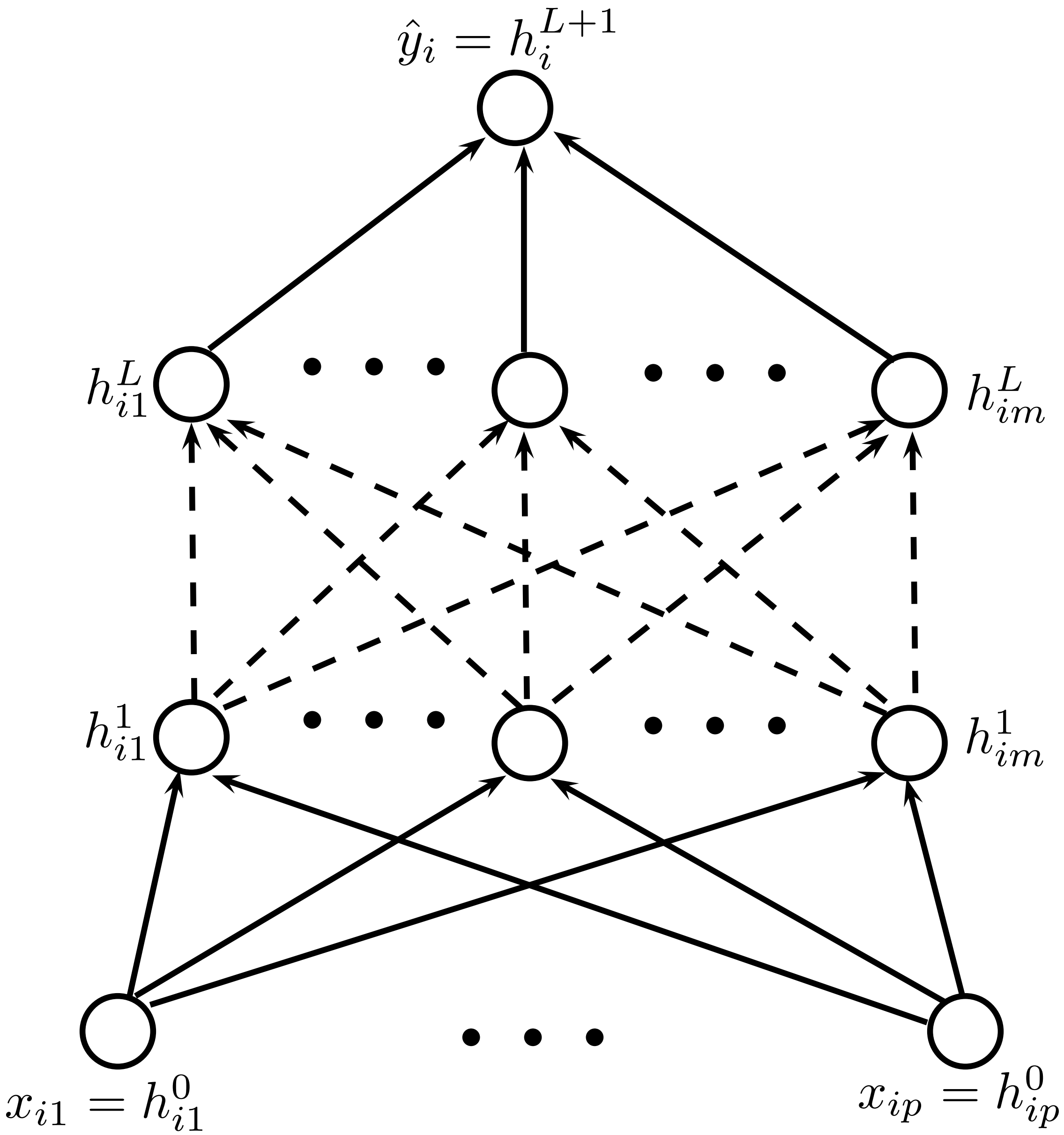
Recap: Neural Networks



$$y = \sigma(W_2 h + b_2)$$

$$h = \sigma(W_1 x + b_1)$$

Recap: Multiple Layers



$$h_i^{\ell+1} = \underline{s}\left(W^{\ell+1} h_i^\ell\right)$$

- $W^{\ell+1} = \left(w_{jk}^\ell\right)_{jk}$: weight matrix at the $(\ell+1)$-th layer, weight w_{jk}^ℓ on the edge between $h_{ik}^{\ell-1}$ and h_{ij}^ℓ
- $\underline{s}$: entrywise (logistic) transfer function

$$\hat{y}_i = \underline{s}\left(W^{L+1}\underline{s}\left(W^L\left(\cdots\underline{s}\left(W^1 x_i\right)\right)\right)\right)$$

- **Many** hidden layers can be used: they are usually thought of as forming a hierarchy from low-level to high-level features.

Deep Learning

=

Fancy Neural Networks

Deep Learning = Fancy Neural Nets

- All the neural networks seen so far have been of a particular basic type: multi-layer perceptrons (MLPs)
- High-level idea of using a series of differentiable linear and nonlinear mappings is much more general that this: can come up with complex computational structures known as **architectures**
  - Deep learning is just neural networks with fancy architectures
  - Advanced models like CNNs, RNNs, etc are all just particular computational structures with the same general idea of alternating between linear mappings and applying local operations / nonlinearities
  - In general, we can think of deep learning as "**differentiable programming**": we write down a, very flexible, parameterised differentiable program from inputs to outputs, then use gradient descent to learn its parameters

Yann LeCun

January 5 at 10:13pm ·

OK, Deep Learning has outlived its usefulness as a buzz-phrase.
Deep Learning est mort. Vive Differentiable Programming!

Yeah, Differentiable Programming is little more than a rebranding of the modern collection Deep Learning techniques, the same way Deep Learning was a rebranding of the modern incarnations of neural nets with more than two layers.

But the important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization.

Deep learning models are built up from simple differentiable component

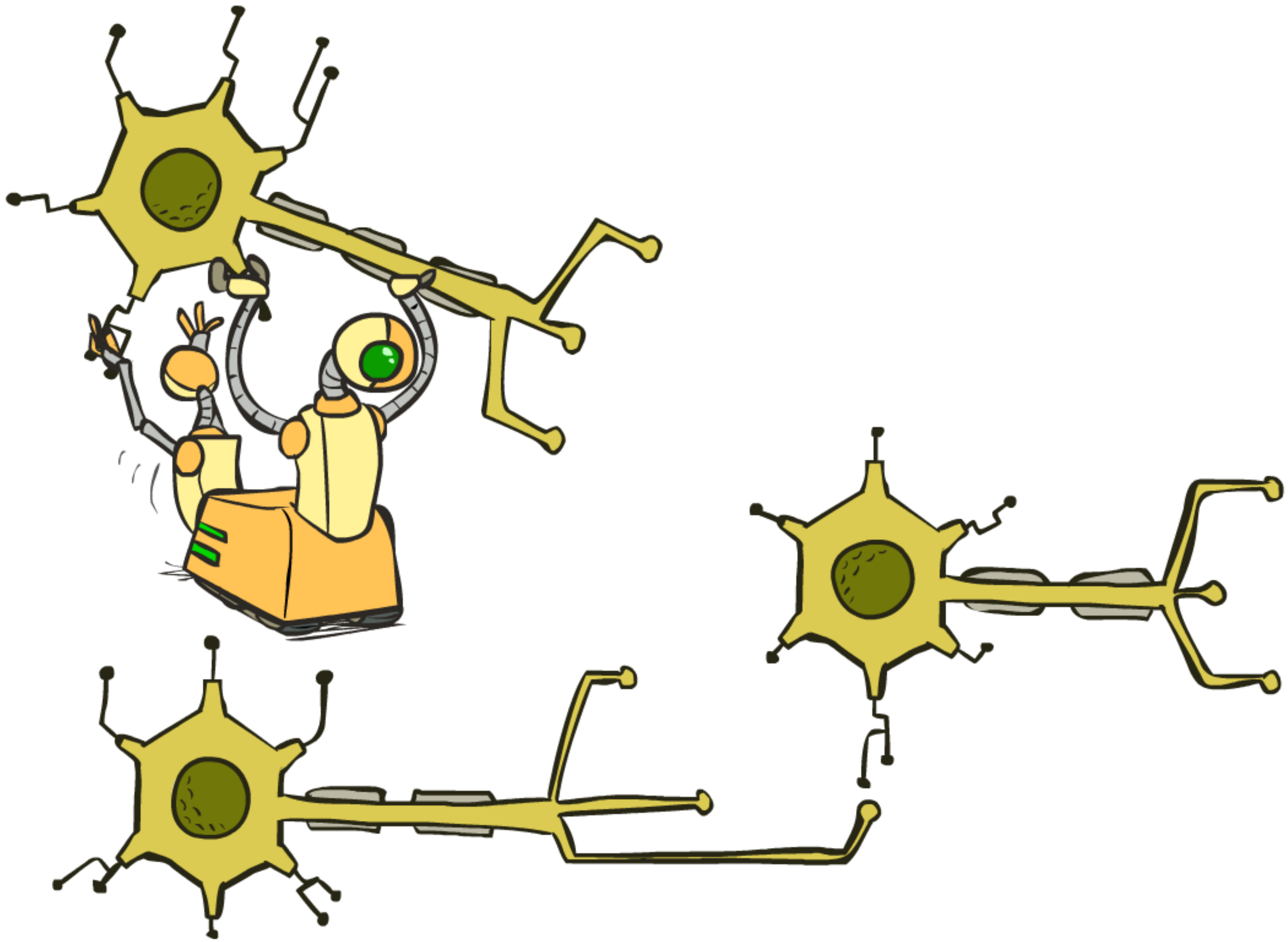


Image Credit: Dan Klein and Pieter Abbeel

Building Blocks

- Linear/fully-connected/dense

$$x \mapsto Wx + b$$

- sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

- tanh

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

- relu

$$\mathrm{relu}(x) = \max(0, x)$$

- softmax

$$\mathrm{softmax}(x_1, \ldots, x_n)$$

$$= \left(\frac{\exp(x_1)}{\sum_i \exp(x_i)}, \ldots, \frac{\exp(x_n)}{\sum_i \exp(x_i)} \right)$$

- Losses

$$\mathrm{CrossEntropy}(t, y) = \sum_i t_i \log y_i$$

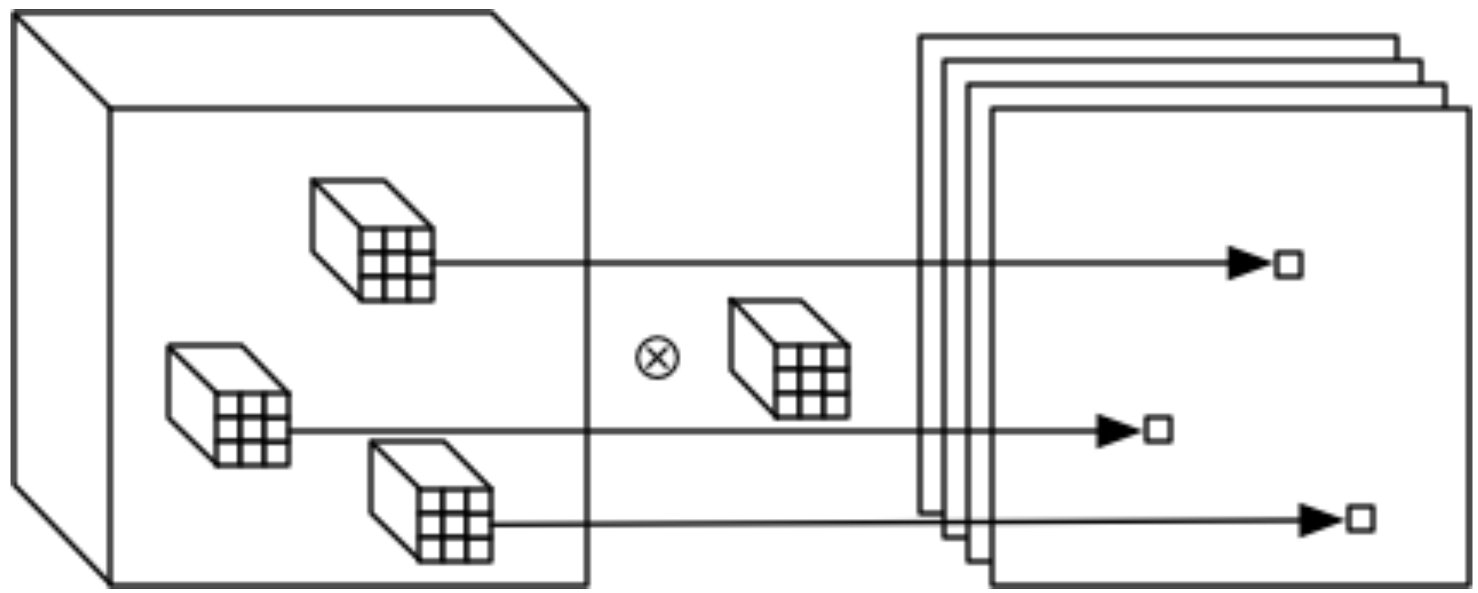
$$\mathrm{Square}(t, y) = \|t - y\|_2^2$$

$$\mathrm{Hinge}(t, y) = \max(0, 1 - t \cdot y)$$

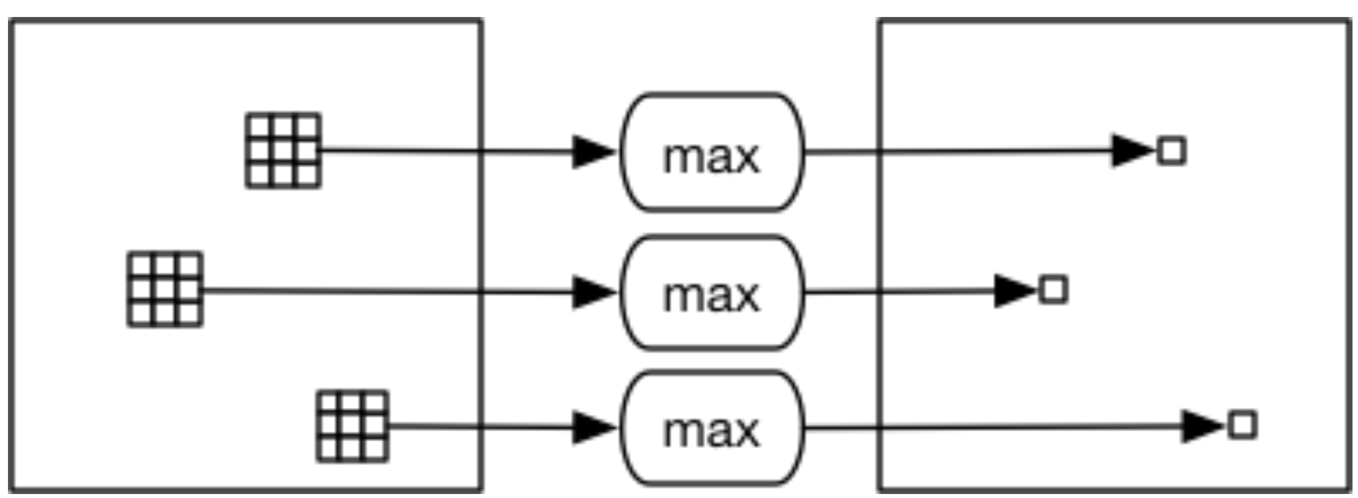
Slide Credit: Yee Whye Teh

Building Blocks

- Convolution



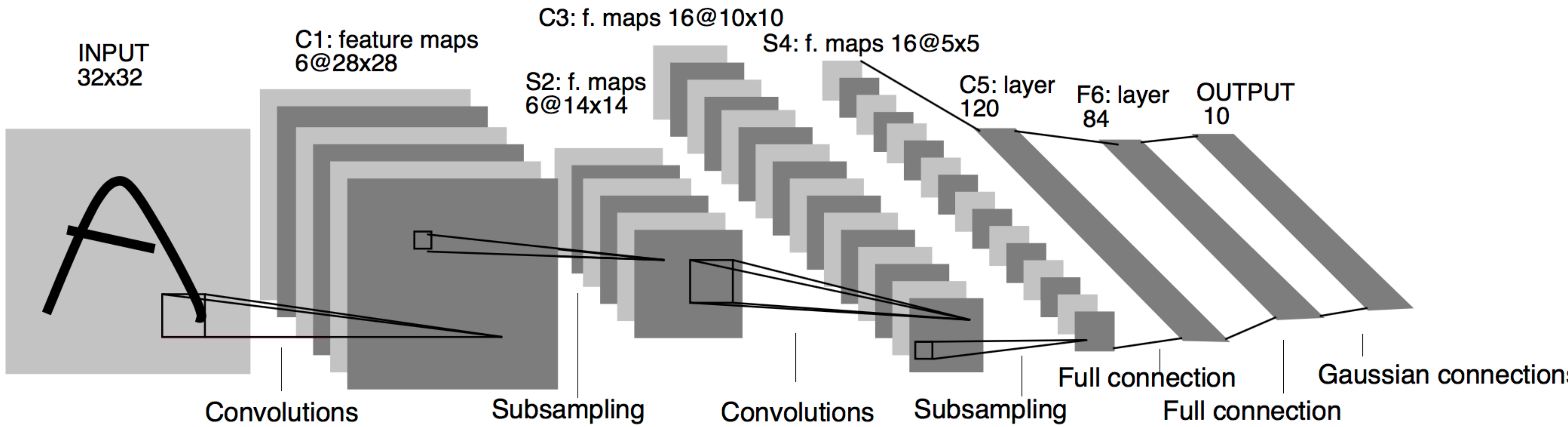
- max pooling



Slide Credit: Yee Whye Teh

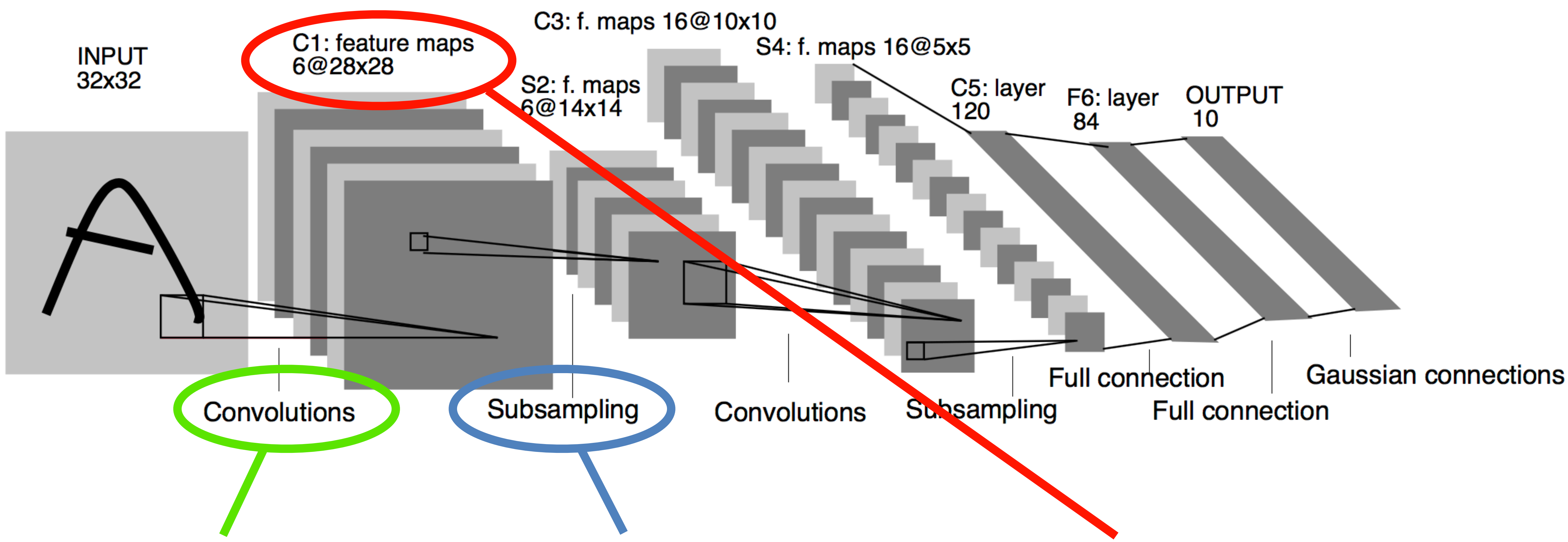
Slide Credit: Yee Whye Teh

Example: LeNet



[LeCun et al 1998]

Example: LeNet



This is just a particular (sparse) way of setting up the weight matrix

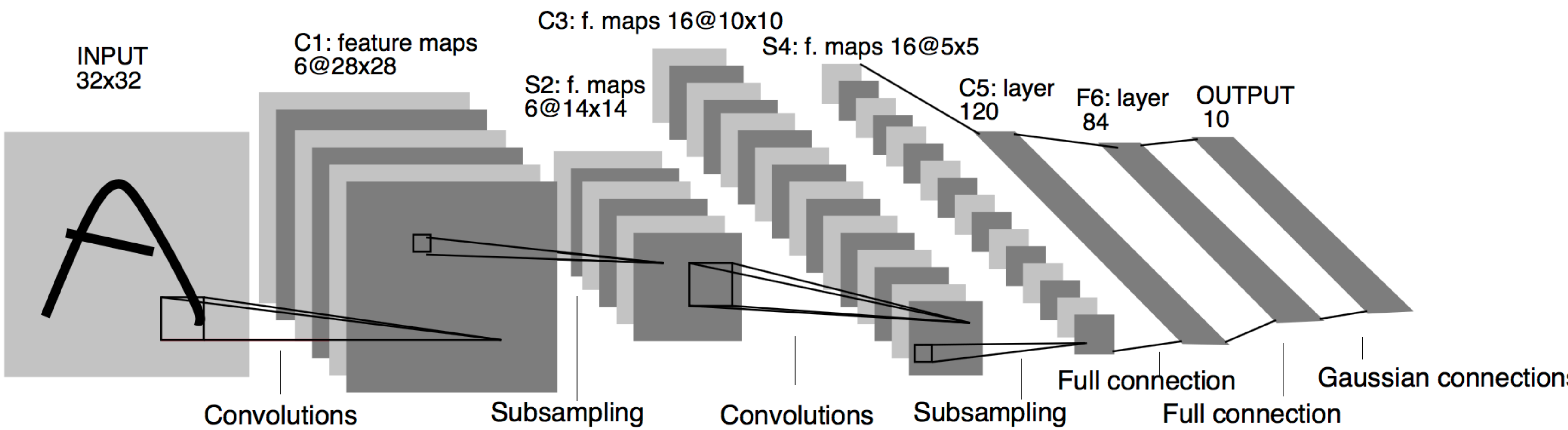
This is a slightly different local operation than we have seen so far

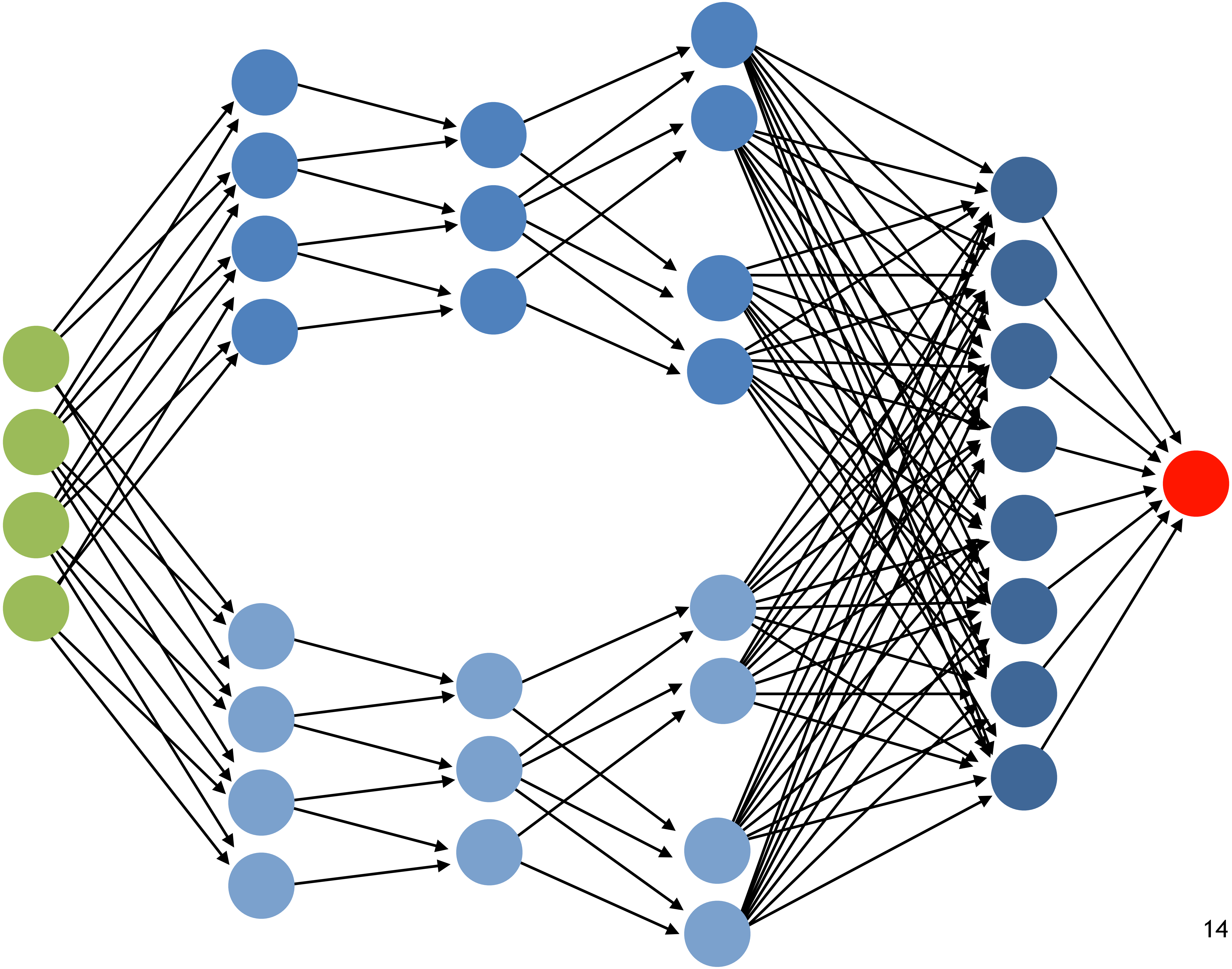
This represents 6x28x28=4704 hidden units structured in a particular way

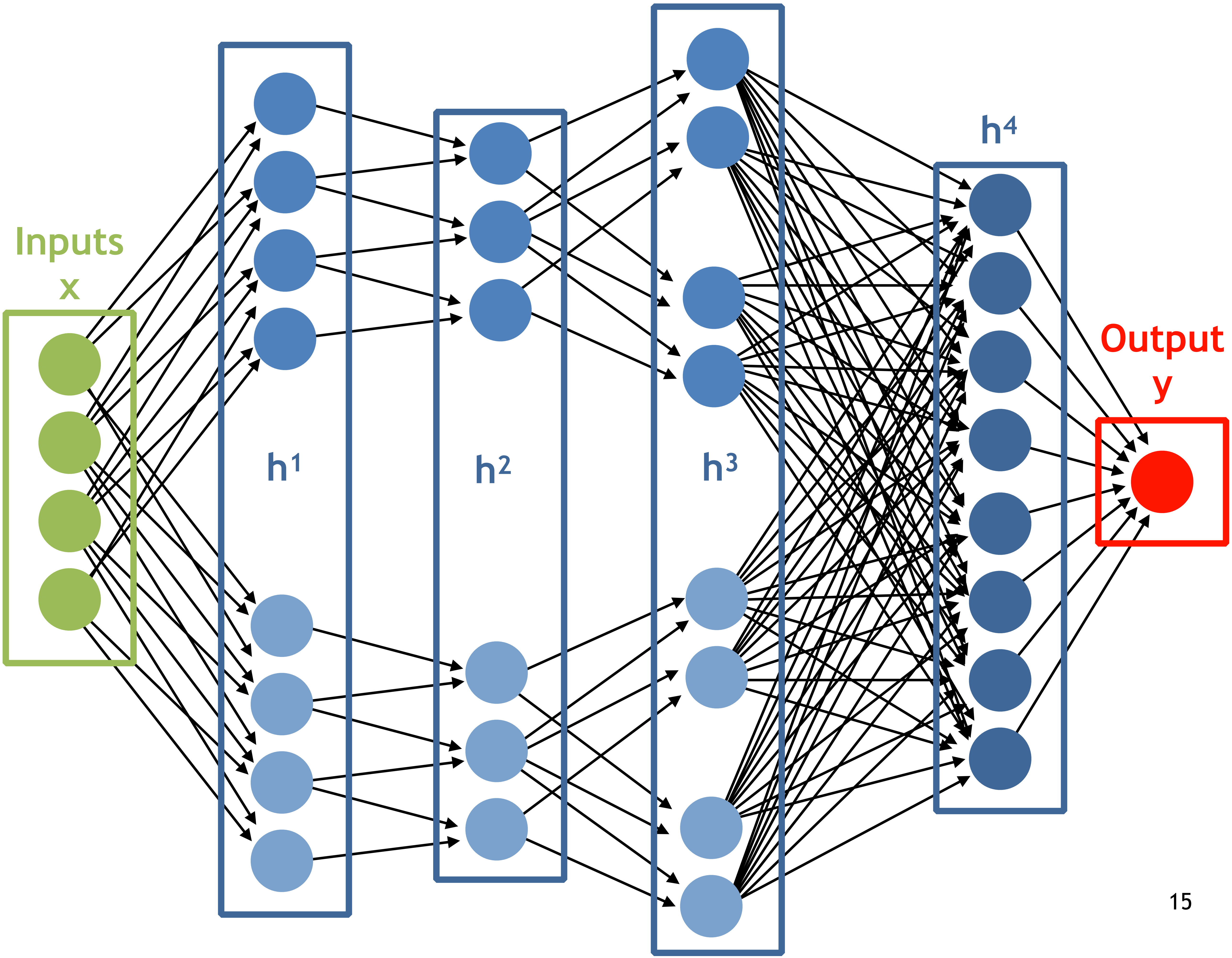
[LeCun et al 1998]

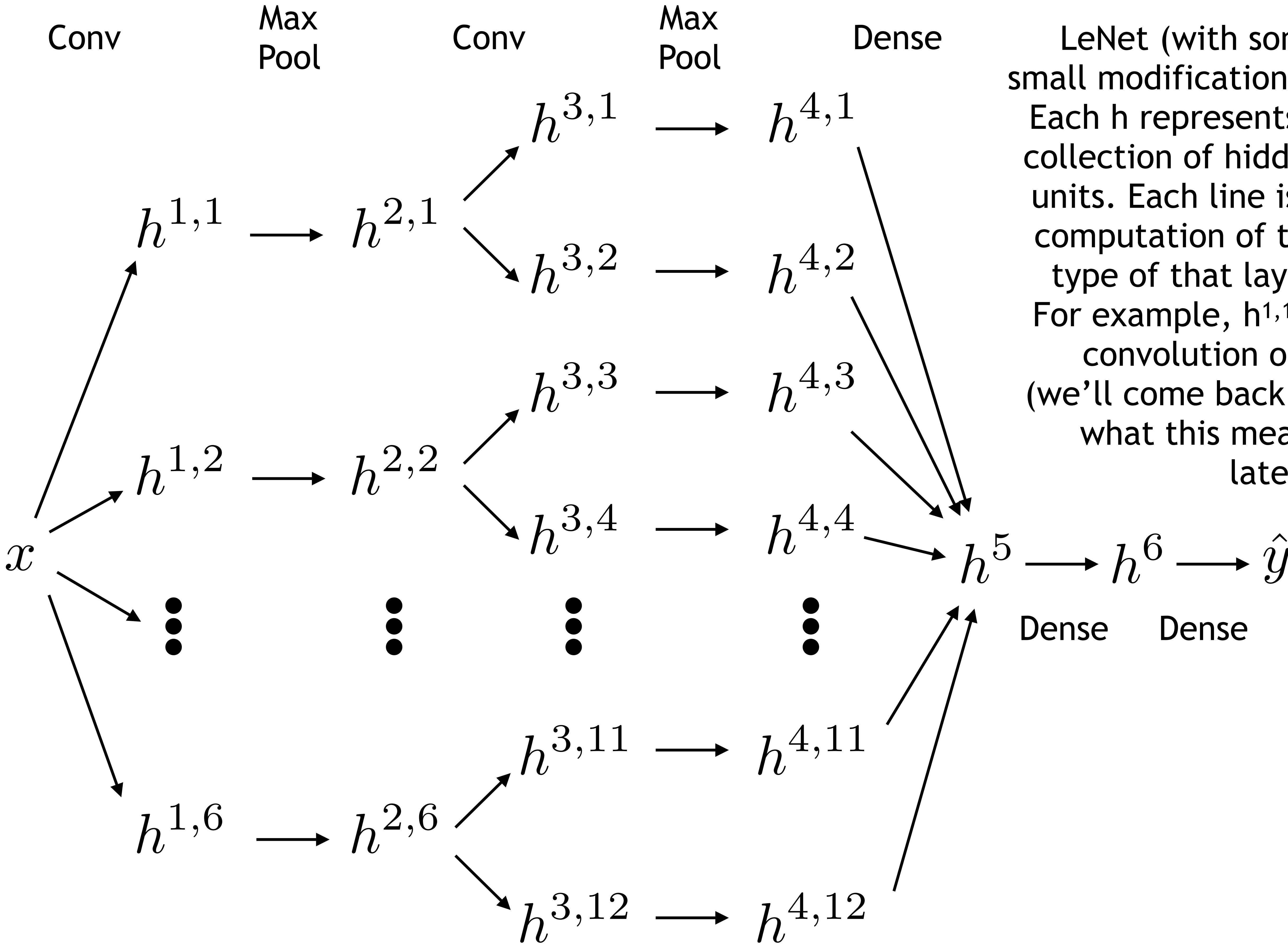
Example: LeNet

Inputs	h^1	h^2	h^3	h^4	h^5	h^6	Output
D=1024	4074 units	1176 units	1600 units	400 units	120 units	120 units	D=10

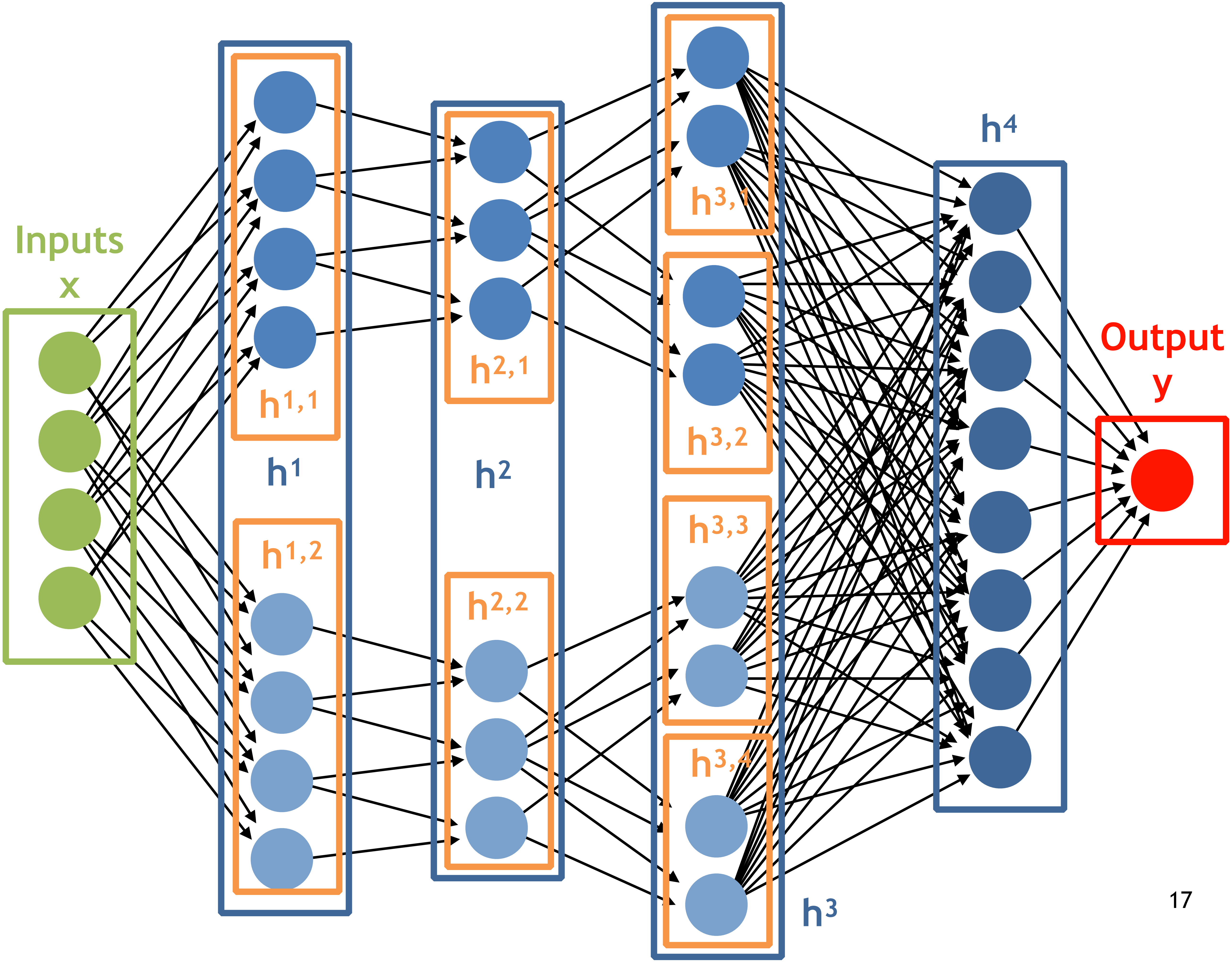





Inputs
x
h¹
h²
h³
h⁴
Output
y



LeNet (with some small modifications). Each h represents a collection of hidden units. Each line is a computation of the type of that layer. For example, $h^{1,1}$ is convolution of x (we'll come back to what this means later).


Inputs
x
h1,1
h1,2
h1
h2,1
h2,2
h2
h3,1
h3,2
h3,3
h3,4
h3
h4
Output
y

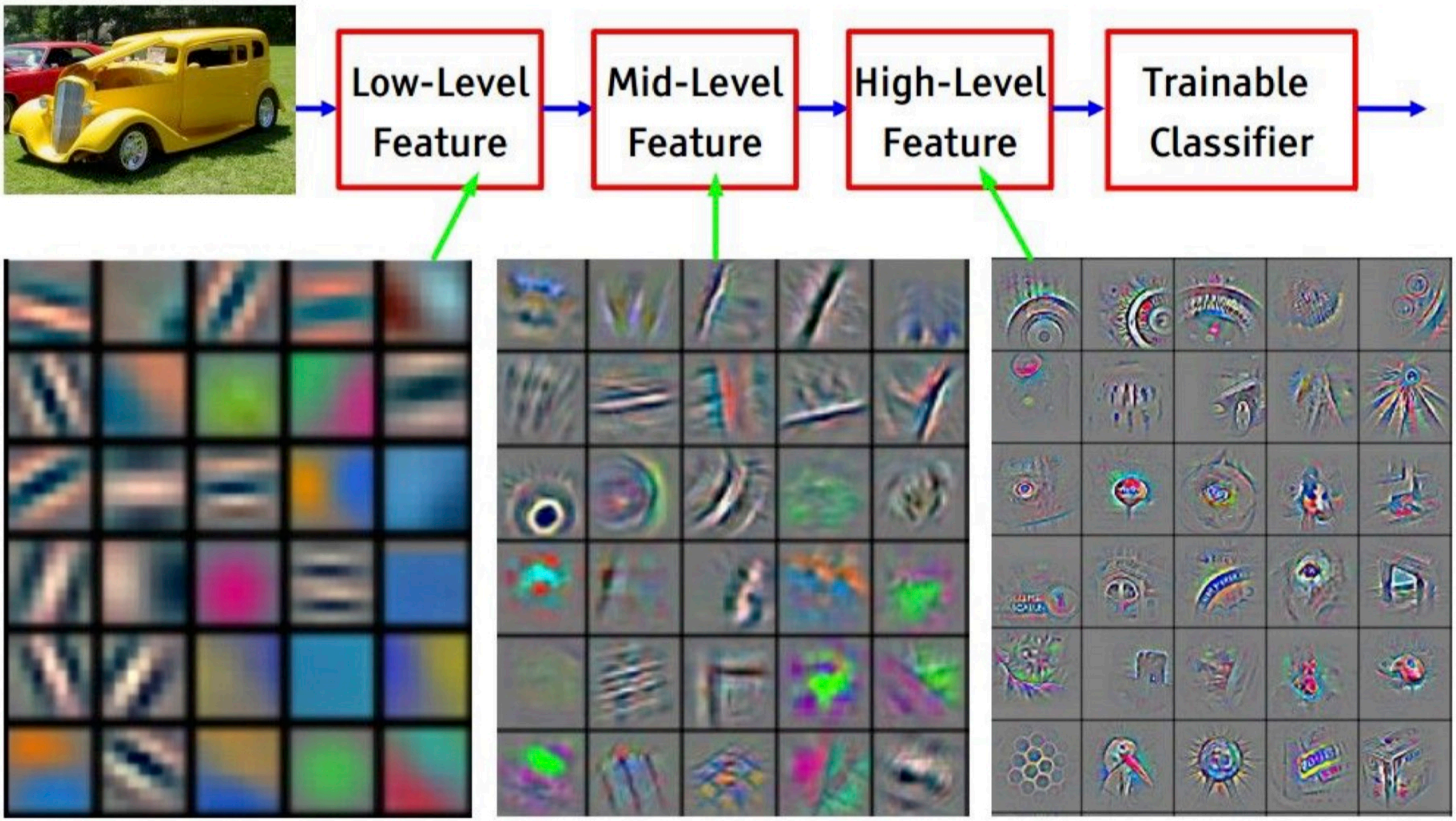
Example: GoogleNet

[Szegedy et al 2014]

Deep Neural Networks

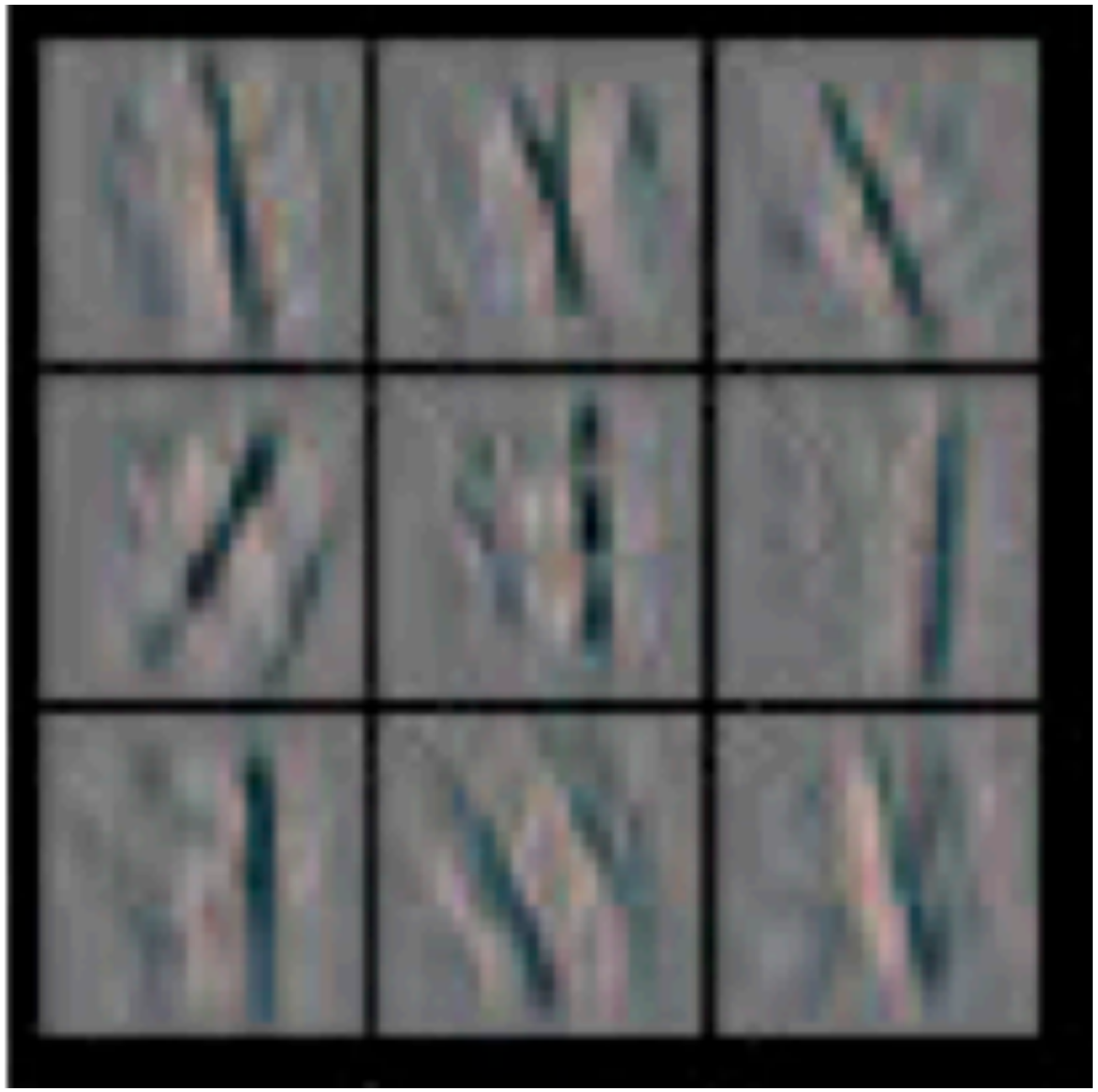
- The underlying principle behind deep learning approaches is that deeper networks allow us to learn more complex **representations**
- Effectively, we can think of deep networks as simultaneously learning complex **features** to represent inputs and how make predictions given these features
- Having multiple layers allows for more complex mappings of the inputs than shallow networks with the same number of hidden units, and thus more complex features and in turn more complex predictive models.
- Unfortunately, a lot of this motivation stems from high-level intuition and empirical evidence, rather than first principles maths
  - Theory on deep learning massively lags behind the practice

Deeper Layers can Detect More Complex Features



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Deeper Layers can Detect More Complex Features

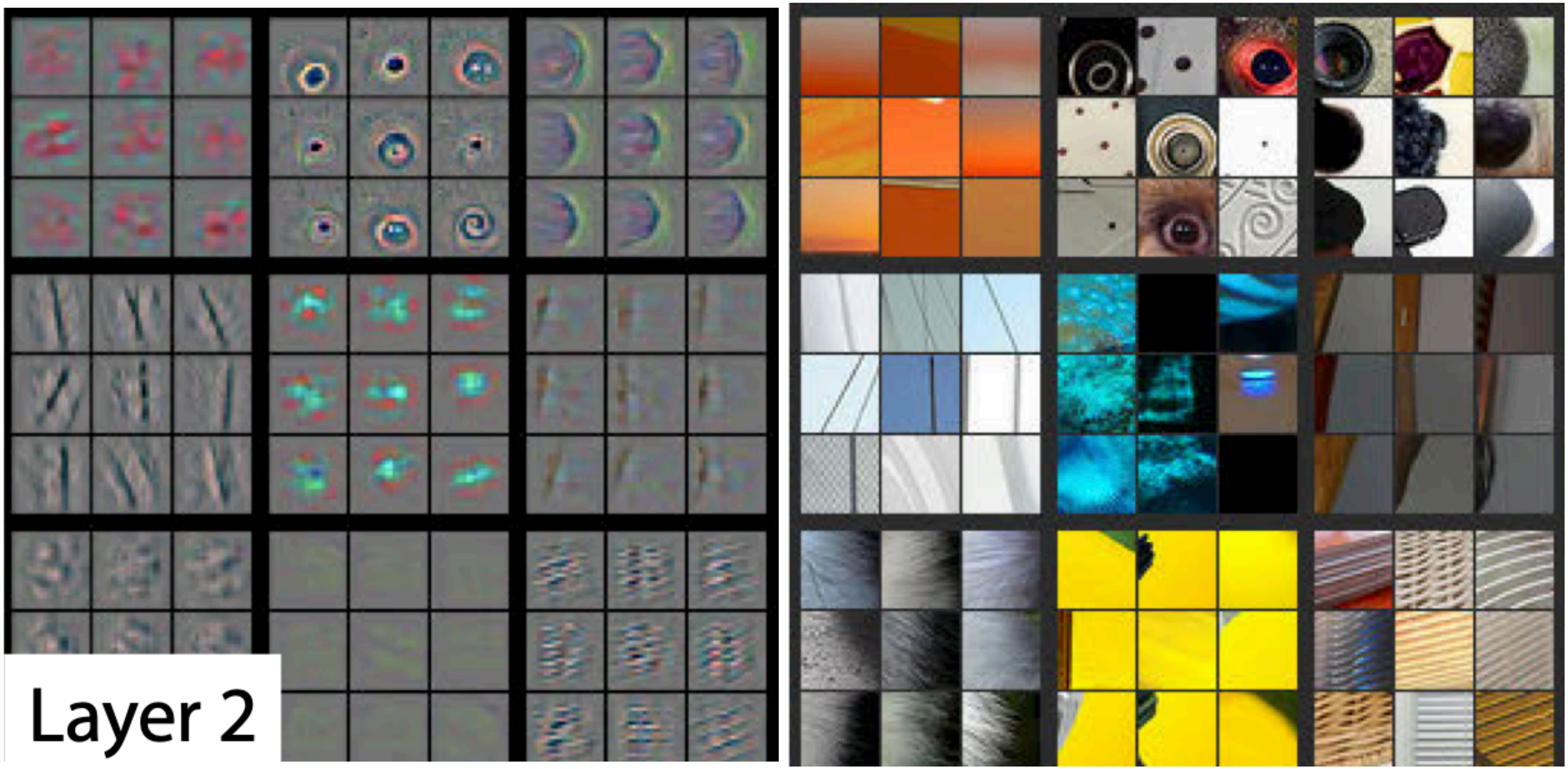
Layer 2

[Right] 9 images that produce the highest activation for a number of different neurons

[Left] Visualisations of what is triggering that activation in each case

Note these are from a CNN (more details later)

Deeper Layers can Detect More Complex Features

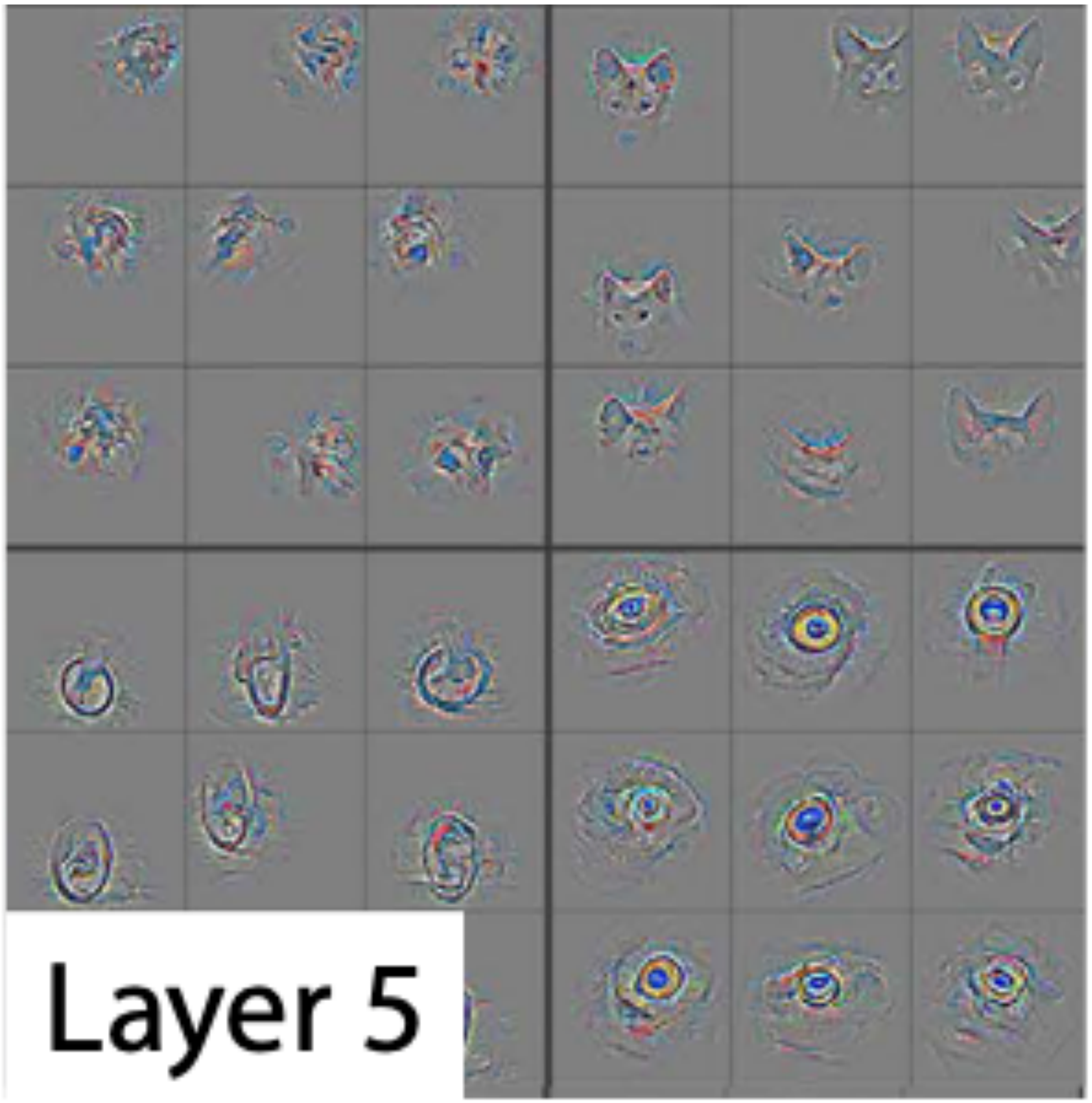


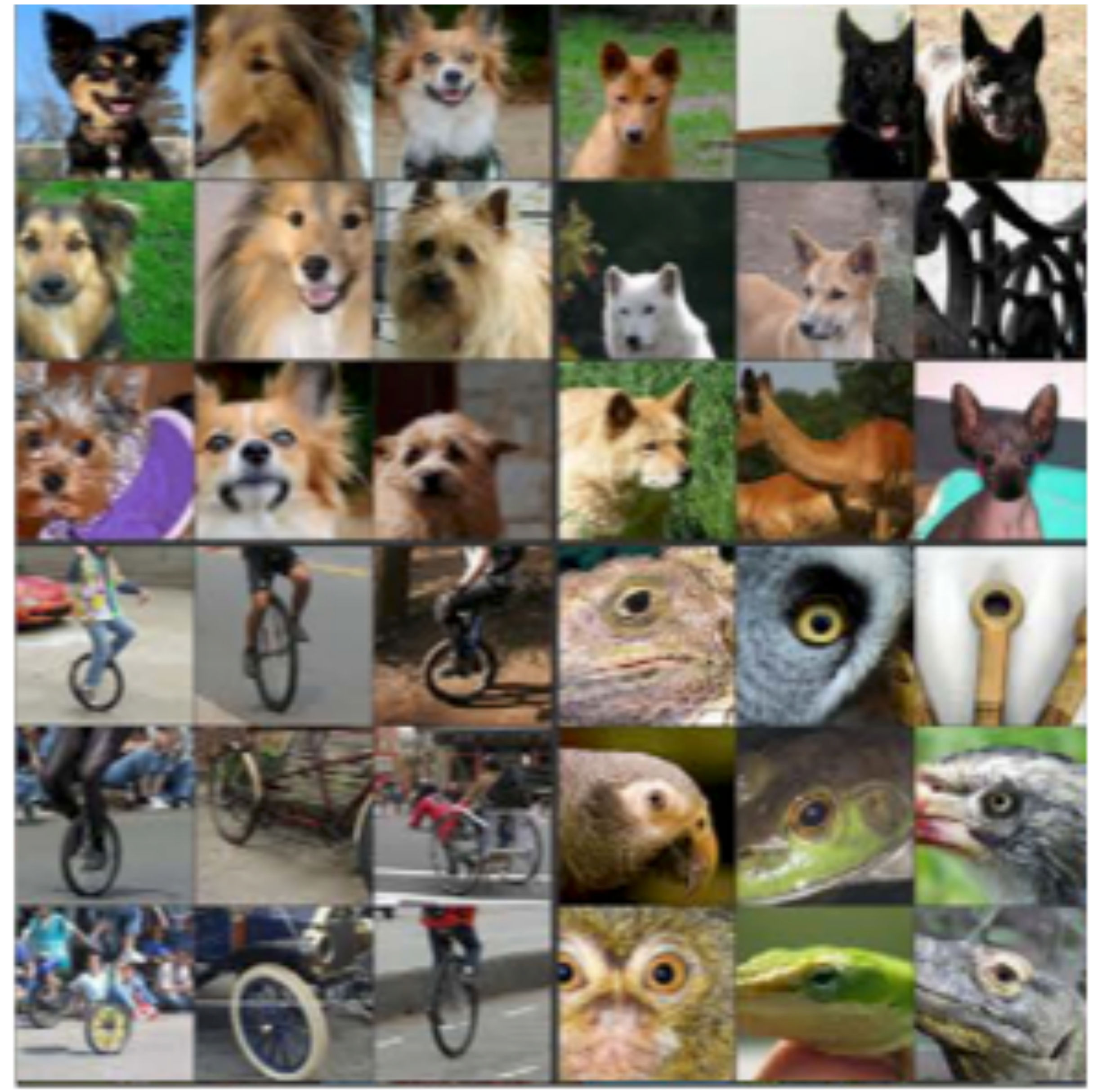
[Right] 9 images that produce the highest activation for a number of different neurons

[Left] Visualisations of what is triggering that activation in each case

Note these are from a CNN (more details later)

Deeper Layers can Detect More Complex Features





[Right] 9 images that produce the highest activation for a number of different neurons

[Left] Visualisations of what is triggering that activation in each case

Note these are from a CNN (more details later)

The Success of Deep Learning

- Success of deep learning is primarily based on:
  - Empirical prowess of large many-layered neural networks for problems where we have a **huge amount of data**, even when we only train to **local optima**
  - **Flexibility** of general framework to come up with highly customised architectures tailored to specific tasks
  - **Automatic differentiation** making models and their corresponding training schemes easy to construct
  - Effectiveness of **stochastic gradient schemes** in allowing us to successfully train huge networks
  - Suitability of these computations to running on **GPUs** allowing for big speed ups (10-100 times faster than on CPU)

Training

Training

- The empirical risk for the network is a function of the weights, the biases, and the data:

$$R_{\text{emp}} = \frac{1}{n}\sum_{i=1}^{n} L\left(W^{1:L}, b^{1:L}, x_i, y_i\right) + \lambda r\left(W^{1:L}, b^{1:L}\right)$$

  where L is our loss function, r is a regulariser, and λ is a hyper parameter controlling the level of regularisation.

- Deep learning methods are almost exclusively trained using gradient methods, for which we need to find (or at least estimate)

$$\frac{\partial R_{\text{emp}}}{\partial W^{\ell}} \quad \text{and} \quad \frac{\partial R_{\text{emp}}}{\partial b^{\ell}} \quad \forall \ell \in \{1, \ldots, L\}$$

Backpropagation

- Deep learning methods are trained using backpropagation methods just like simple neural networks
- Idea is exactly the same: just carefully apply the chain rule
- Denote $h^0 = x$ and $h^{L+1} = y$. Let h^ℓ_{ij} denote the value of the j^{th} unit of the ℓ^{th} layer when given input x_i, and let w^ℓ and b^ℓ denote an arbitrary weight and bias in the ℓ^{th} layer respectively. We can now express our backpropagation rules as follows:

$$\frac{\partial R_{\text{emp}}}{\partial w^\ell} = \lambda \frac{\partial r}{\partial w^\ell} + \frac{1}{n} \sum_{i=1}^{n} \sum_{j} \frac{\partial L(x_i, y_i)}{\partial h^\ell_{ij}} \frac{\partial h^\ell_{ij}}{\partial w^\ell}$$

$$\frac{\partial R_{\text{emp}}}{\partial b^\ell} = \lambda \frac{\partial r}{\partial b^\ell} + \frac{1}{n} \sum_{i=1}^{n} \sum_{j} \frac{\partial L(x_i, y_i)}{\partial h^\ell_{ij}} \frac{\partial h^\ell_{ij}}{\partial b^\ell}$$

$$\frac{\partial L(x_i, y_i)}{\partial h^\ell_{ij}} = \sum_{k} \frac{\partial L(x_i, y_i)}{\partial h^{\ell+1}_{ik}} \frac{\partial h^{\ell+1}_{ik}}{\partial h^\ell_{ij}}$$

Note a lot of terms are often clearly zero and can be omitted, but for complex mappings (e.g. convolutions) it can often be easiest to write down this full form and then remove any terms we don't need.

Gradient Descent

- Letting $\theta \triangleq \{W^{1:L}, b^{1:L}\}$ and $f_\theta(x)$ the application of the network to input x, our problem is of the from

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i)) + \lambda r(\theta)$$

- To train a network we need to use gradient descent methods, i.e. we apply the iterative procedure

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon_t \nabla_\theta \left(\frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i)) + \lambda r(\theta) \right)$$

where ϵ_t for which there are lots of effective clever strategies and in general we require that the Robin's Monro conditions are satisfied:

$$\sum_{t=1}^{\infty} \epsilon_t = \infty, \qquad \sum_{t=1}^{\infty} \epsilon_t^2 < \infty$$

Stochastic Gradient Descent

- The problem with this approach is that n is typically extremely large, such that carrying out these updates is typically infeasible
- Thankfully, we can fall back on a much more cost-effective strategy: **stochastic gradient descent** (SGD)
- SGD is very simple: we just make updates based on a minibatch of data $B_t \subset \{1, \dots, n\}$, rather than the full dataset:

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon_t \nabla_\theta \left(\frac{1}{|B_t|} \sum_{i \in B_t} L(y_i, f_\theta(x_i)) + \lambda r(\theta) \right)$$

- This still converges provided that the minibatches are set up in a way that the ensures all datapoints are fed to the algorithm the same amount on average (e.g. by uniformly drawing B_t from $\{1, \dots, n\}$ at each iteration or looping through the data using "**epoch**", where one epoch is one pass through the dataset).
- Updates are now $O(|B_t|)$ instead of $O(n)$
- There are lots of extensions of this approach, such as introducing momentum

Automatic Differentiation (AutoDiff)

- Many modern systems can calculate derivatives for you automatically, even for large complex programs
- In practice you never need to manually calculate network derivatives (except in exams!)
- These systems use a technique called automatic differentiation (AutoDiff)
- AutoDiff is exact (i.e. it is not a numerical approximation) but is not symbolic (it is scalable, symbolic approaches aren’t)
- Two different forms:
  - Forward mode AutoDiff
  - Reverse mode AutoDiff

Frameworks

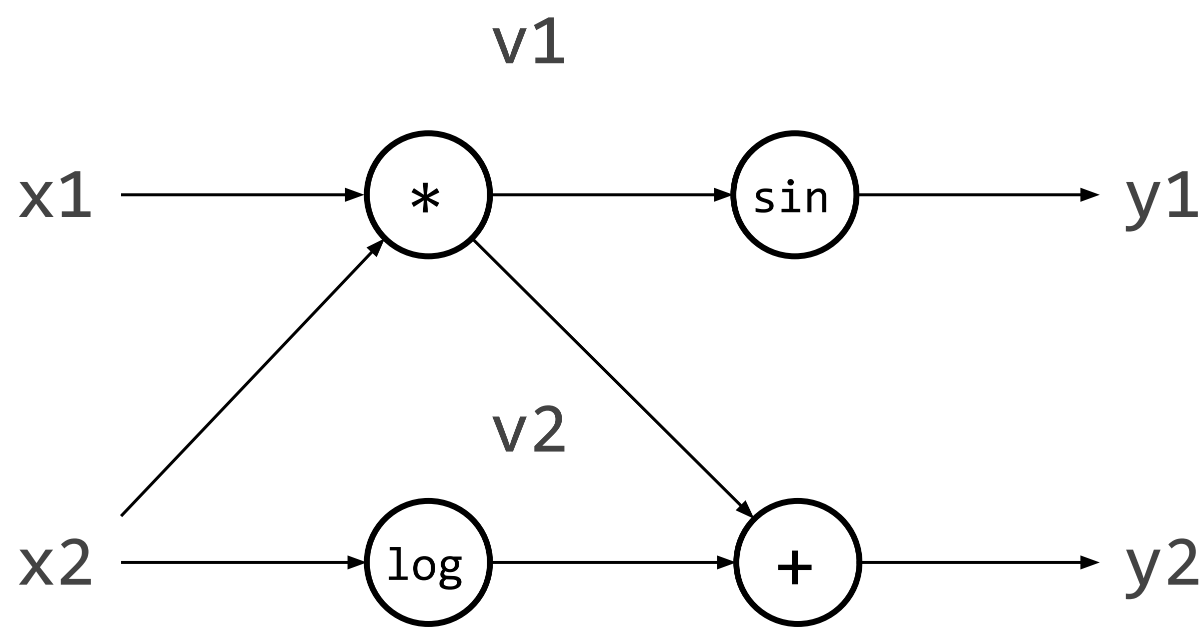
Check out Dr Baydin’s very good slides from the Computer Science Advanced ML course: https://www.cs.ox.ac.uk/teaching/courses/2019-2020/advml/

Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```



Slide Credit: Gunes Baydin

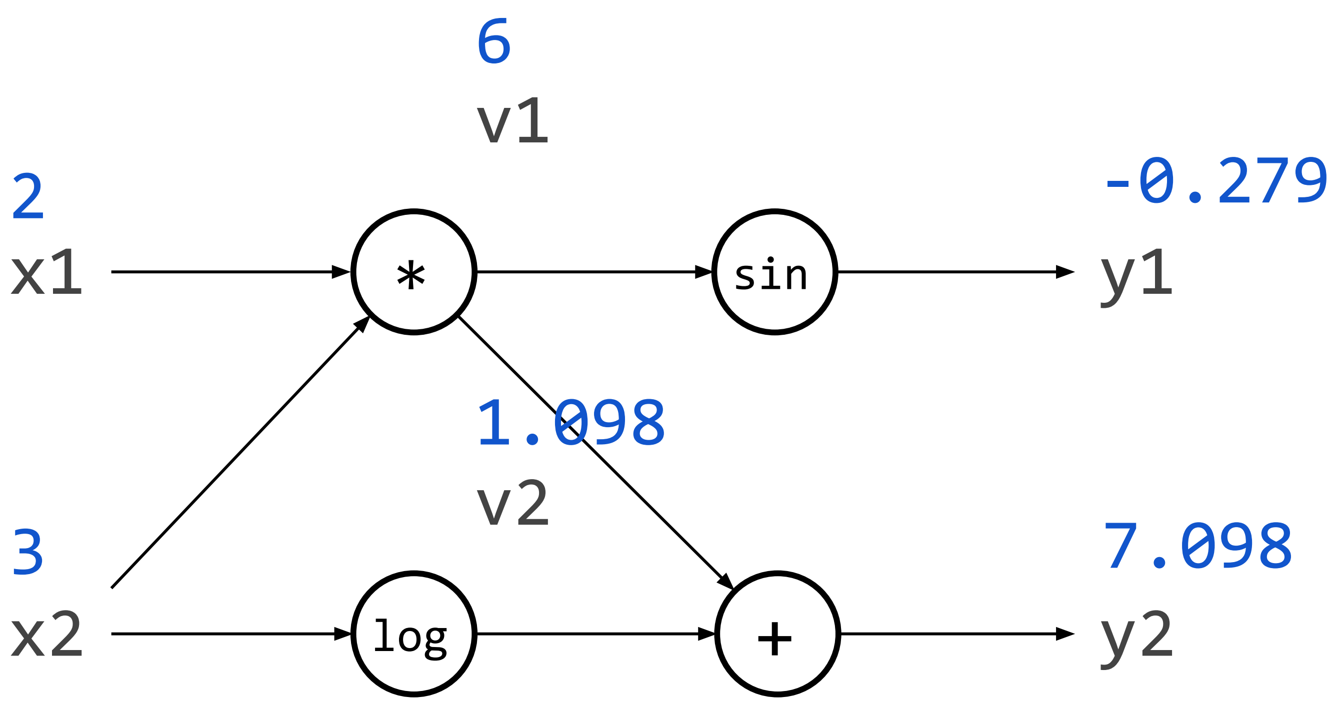
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



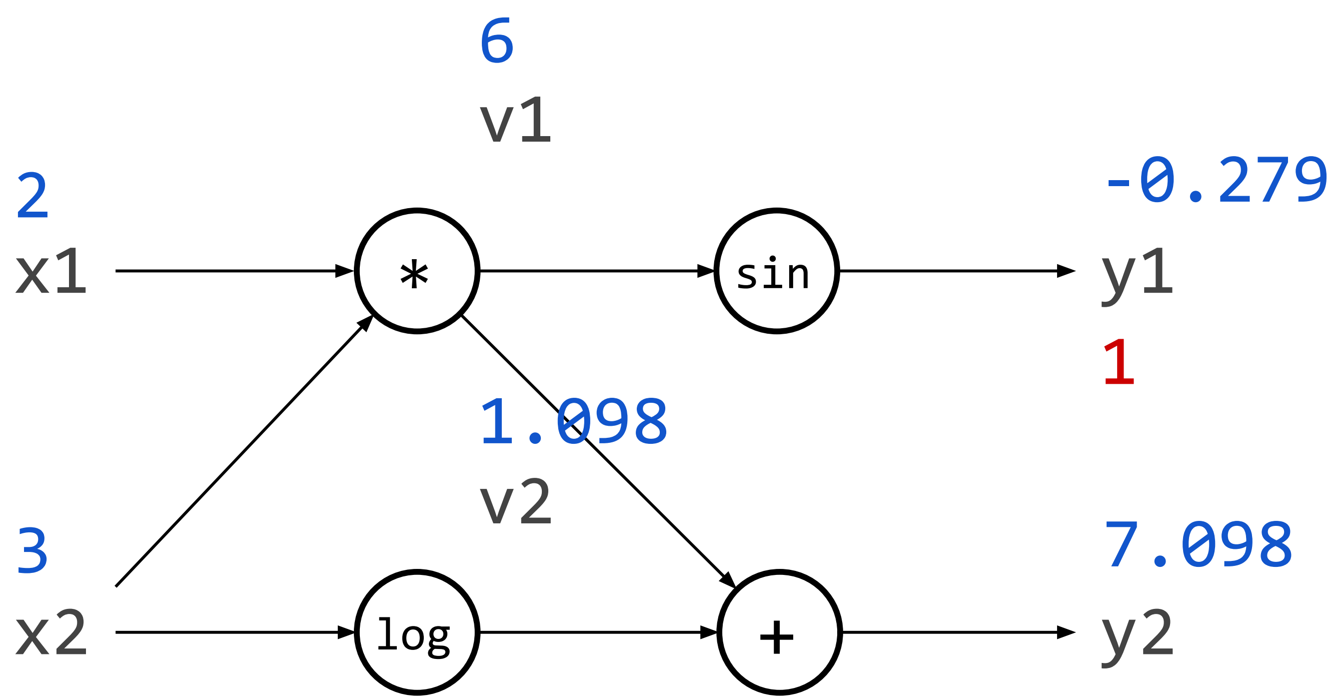
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial y_1} = 1$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

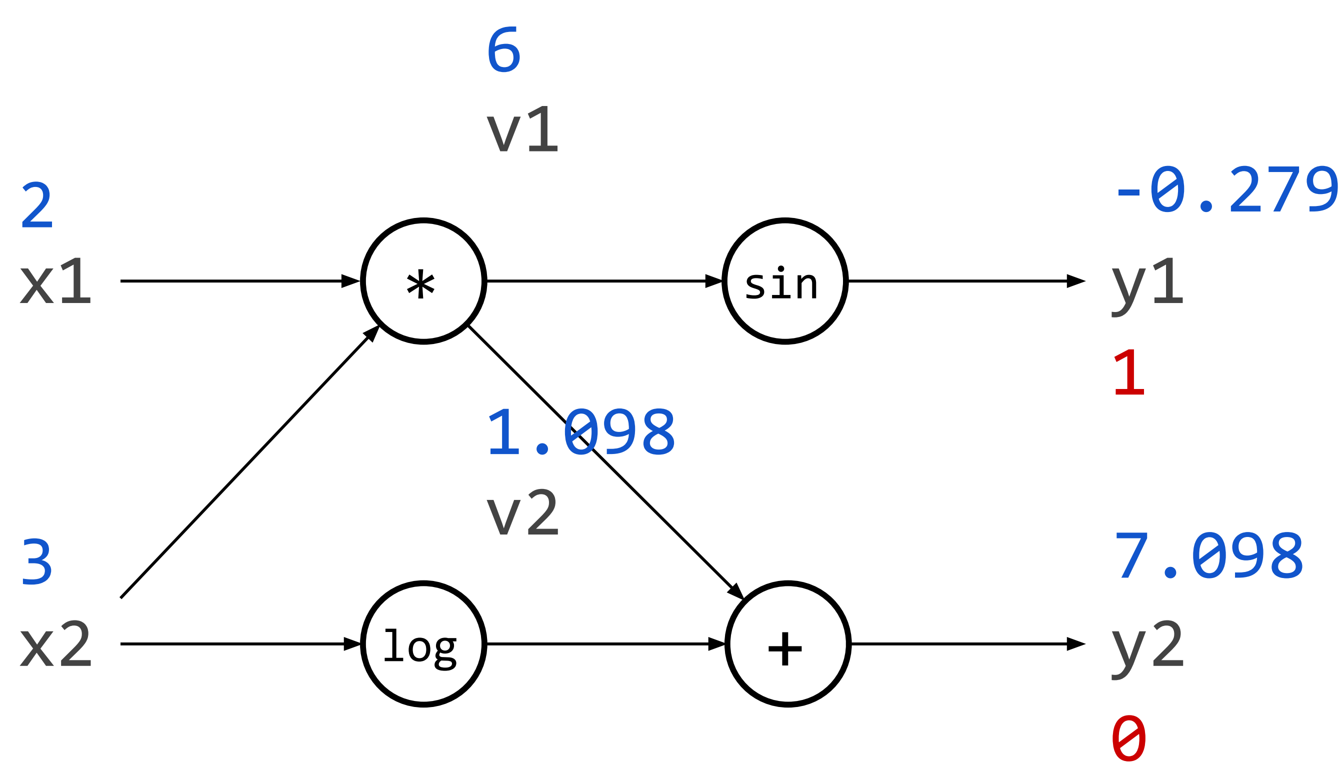
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial y_2} = 0$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

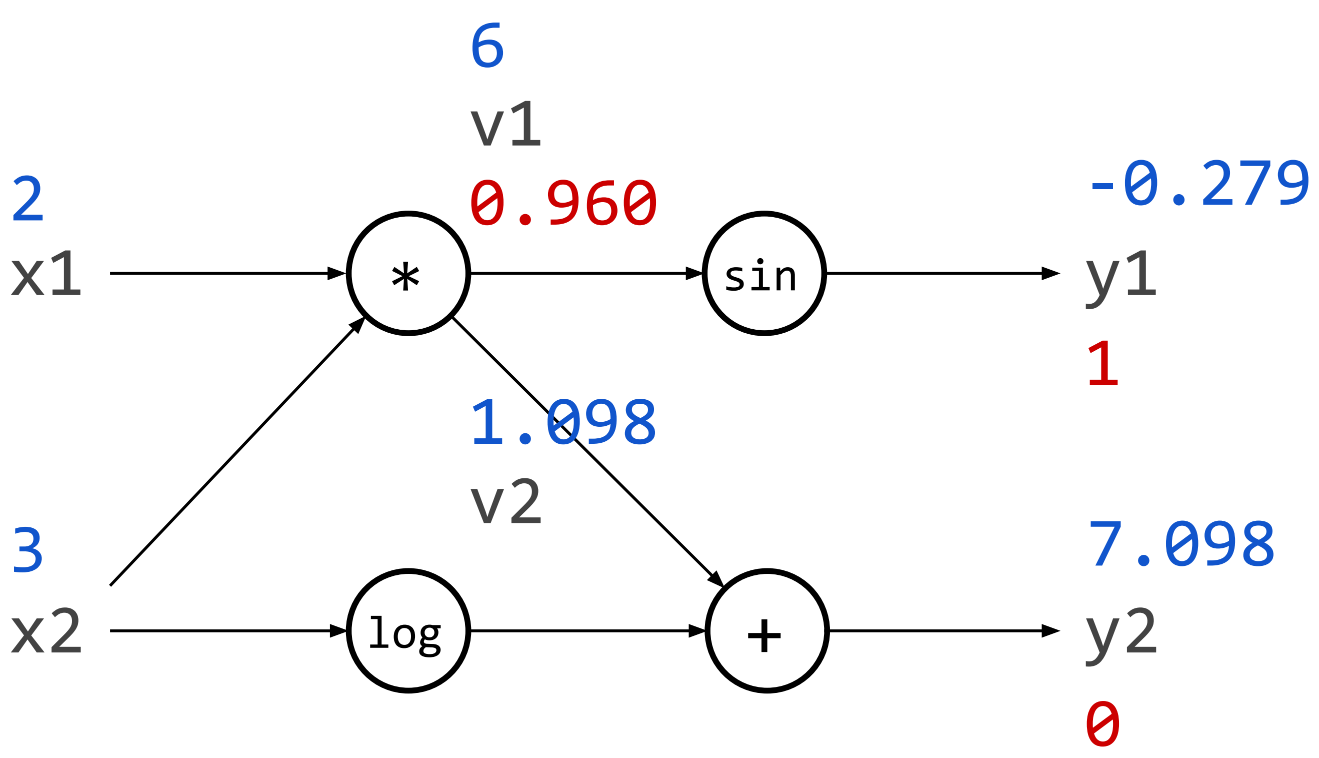
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial v_1} = \frac{\partial y_1}{\partial v_1}\frac{\partial y_1}{\partial y_1} + \frac{\partial y_2}{\partial v_1}\frac{\partial y_1}{\partial y_2} = \cos(v_1)\frac{\partial y_1}{\partial y_1}$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

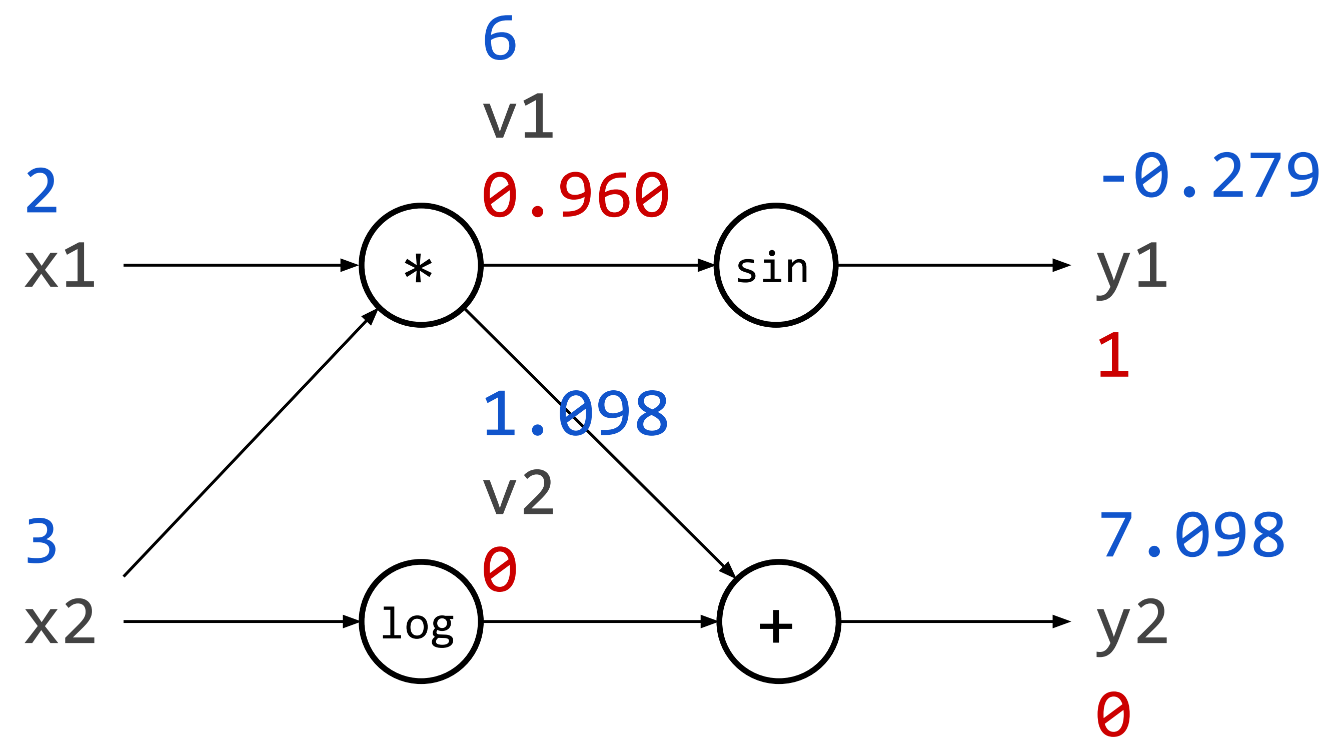
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial v_2} = \frac{\partial y_2}{\partial v_2}\frac{\partial y_1}{\partial y_2} = 0$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

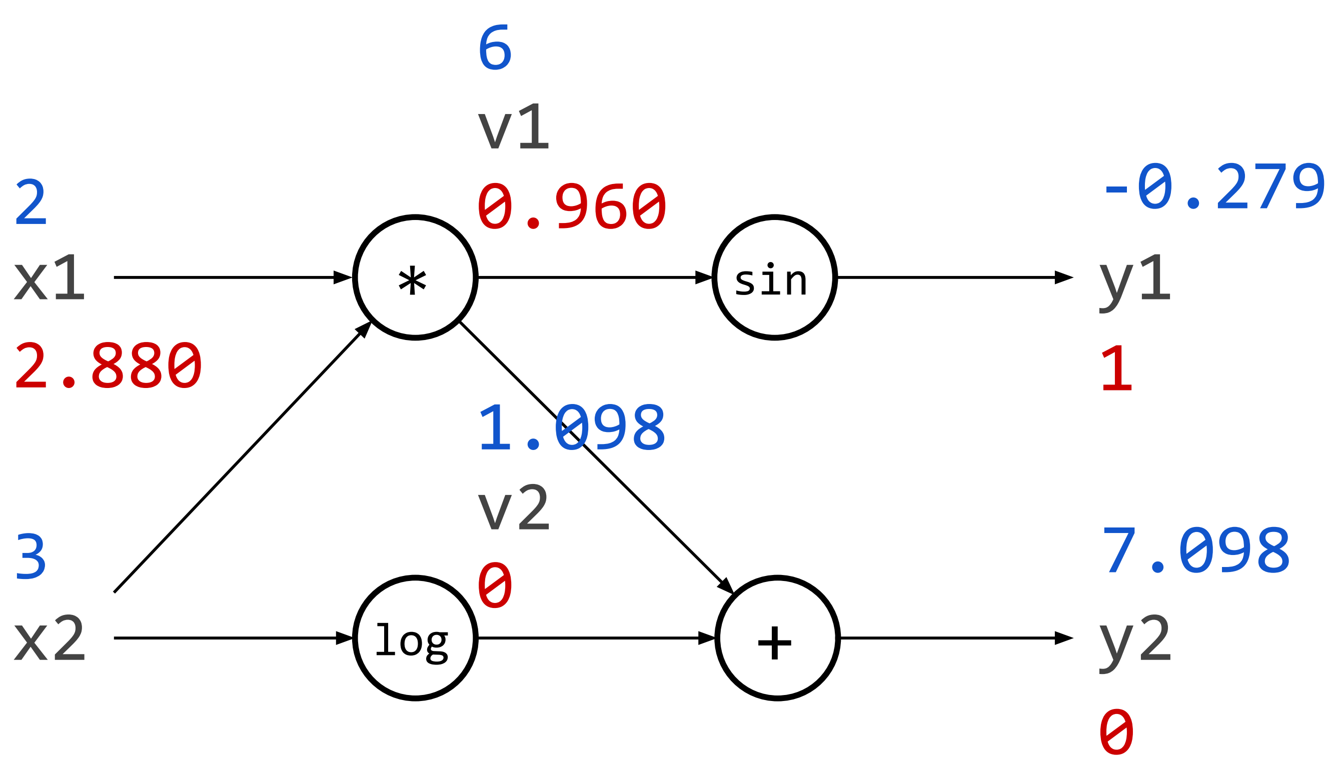
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)
```

```
f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial x_1} = \frac{\partial v_1}{\partial x_1}\frac{\partial y_1}{\partial v_1} = x_2\frac{\partial y_1}{\partial v_1}$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

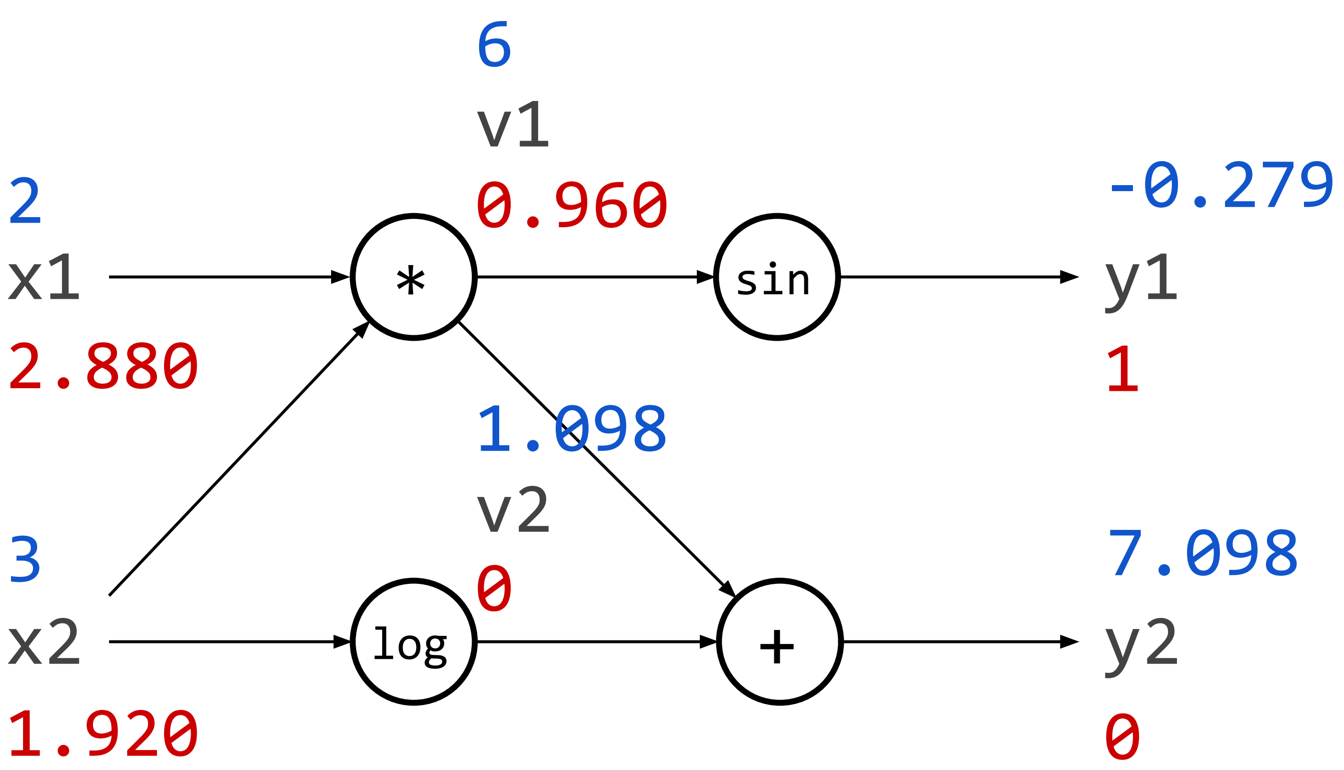
Backpropagation is Reverse Mode AutoDiff

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial x_2} = \frac{\partial v_1}{\partial x_2}\frac{\partial y_1}{\partial v_1} + \frac{\partial v_2}{\partial x_2}\frac{\partial y_2}{\partial v_2} = x_1\frac{\partial y_1}{\partial v_1}$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

The Deep Learning Pipeline

1. Get hold of some (or ideally a lot of) data and computing resources, ideally GPUs or even TPUs (Google Colabs is good if you haven't got much compute yourself)
2. Establish what you want to predict, choose a loss function and decide whether to use any regularisation (e.g. dropout, weight decay)
3. Using a deep learning system like Tensorflow or PyTorch, construct an architecture using the aforementioned building blocks (or choose a standard off-the-shelf variant), i.e. a differentiable parameterised function from inputs to predictions
4. Choose a stochastic gradient descent scheme to train with
5. Let it train until the empirical risk converges
6. Tune hyperparameters / update architecture if necessary
7. Deploy the learned network

Example Architecture: Convolutional Neural Networks (aka ConvNets, CNNs)

CNNs: Why Should I Care?

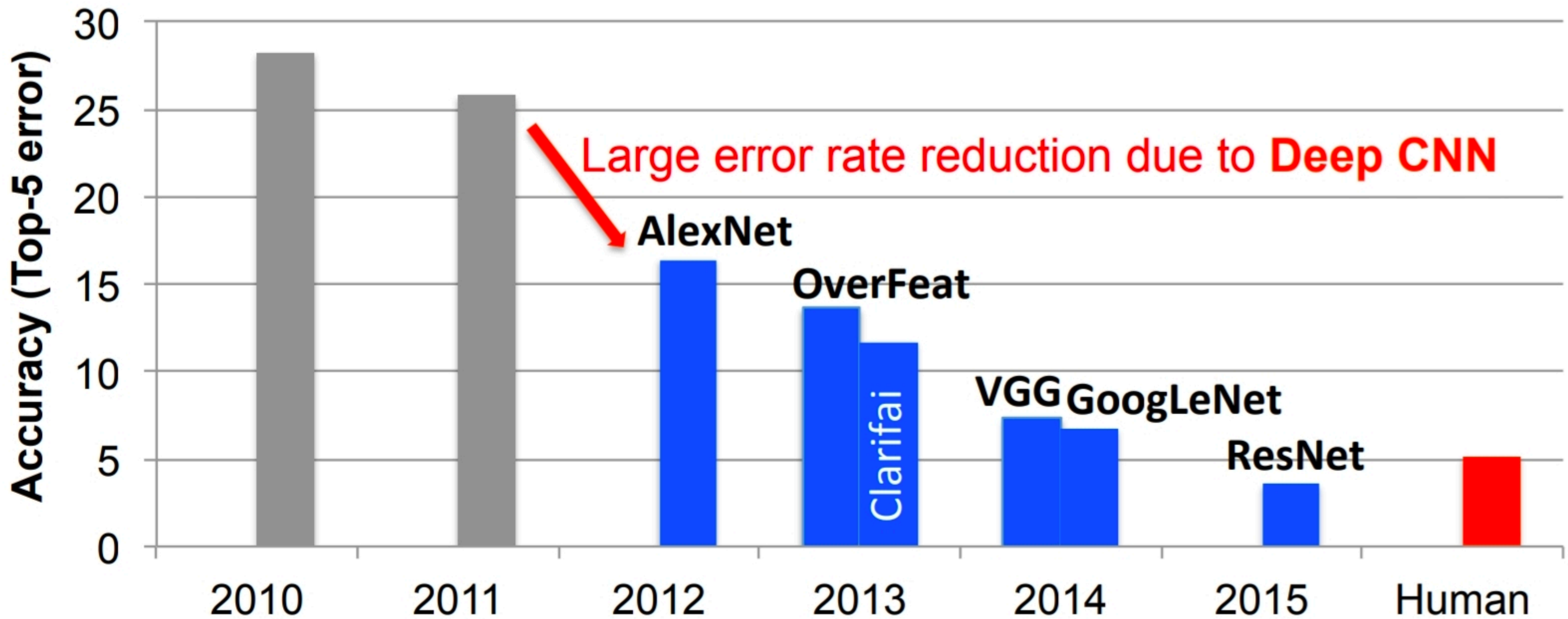
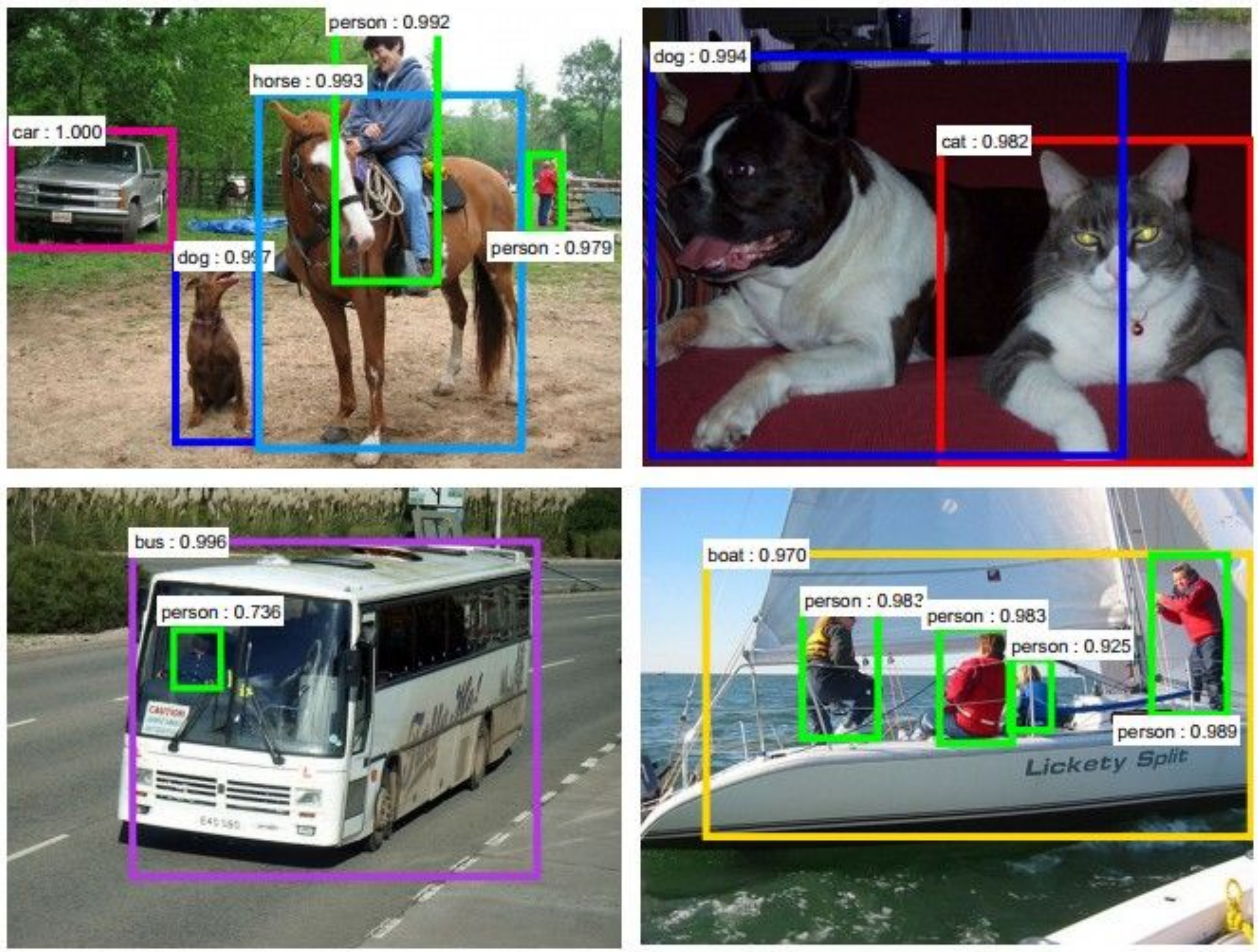


Fig. 7. Results from the ImageNet Challenge [14].

[Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. IJCV. 2015]

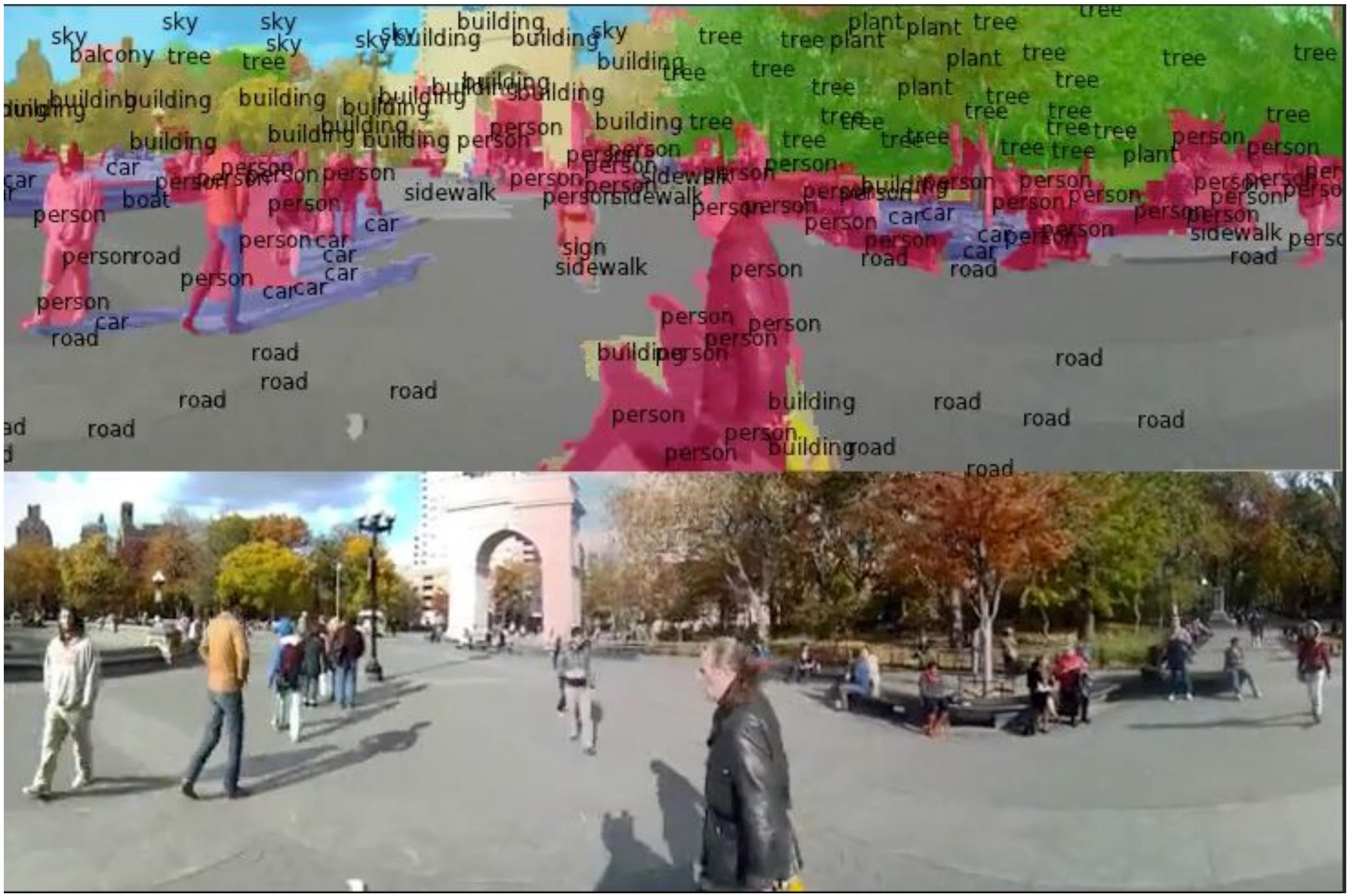
Applications — Object Detection

Detection



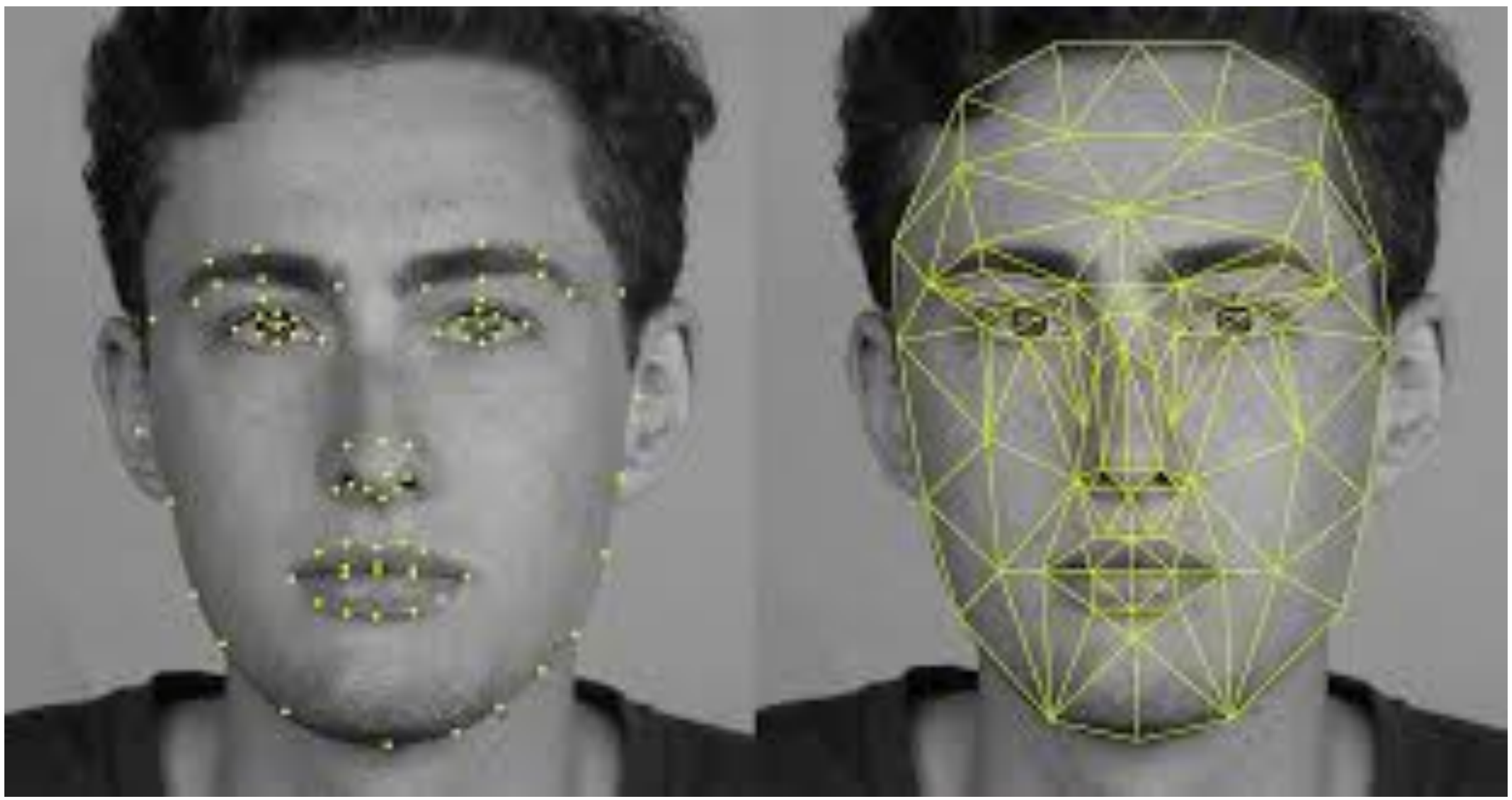
[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Applications — Segmentation



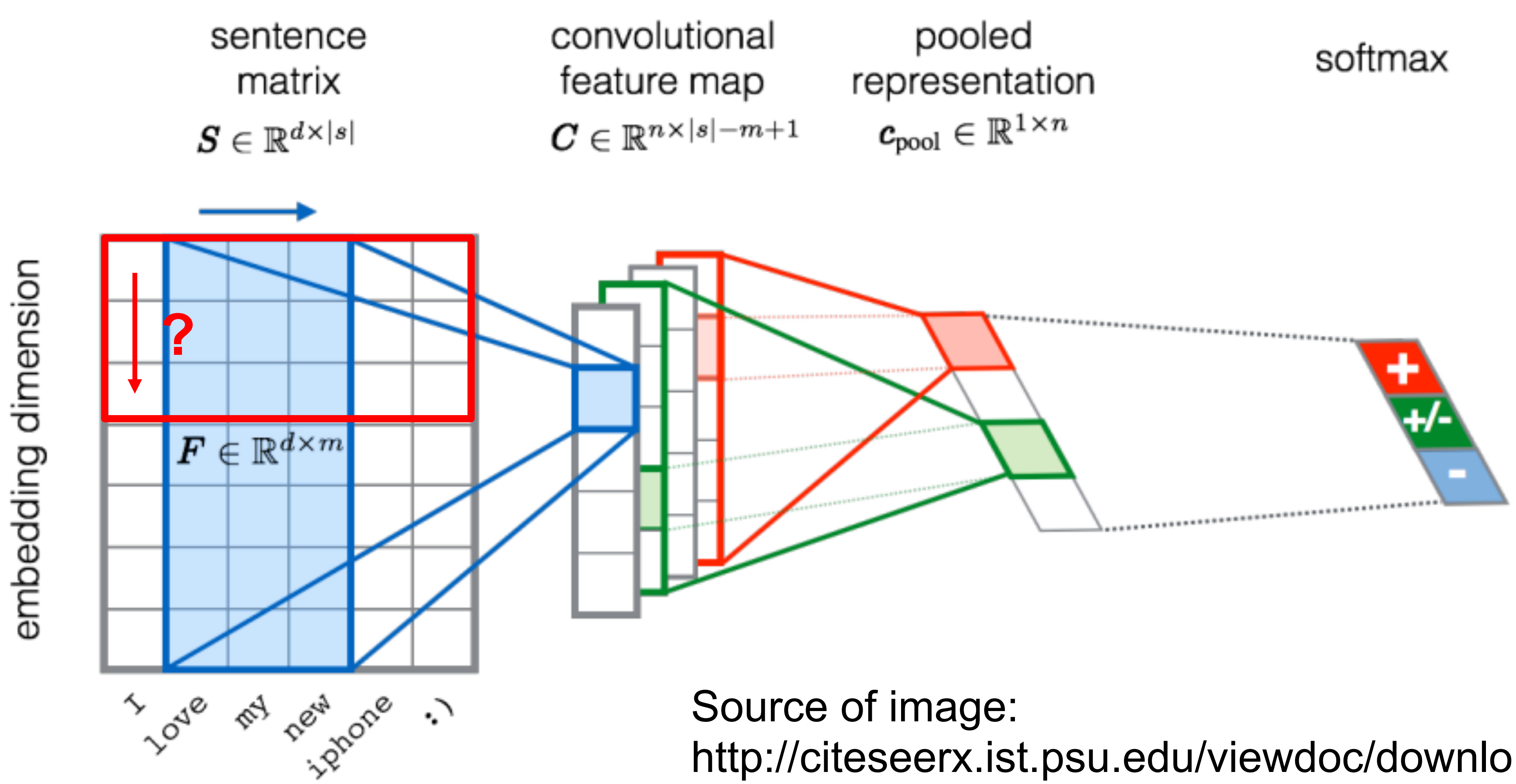
[Farabet et al., 2012]

Applications — Face Recognition



[Murray 2017]

Applications — NLP



Source of image: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.703.6858&rep=rep1&type=pdf

Convolutions

1	-1	-1
-1	1	-1
-1	-1	1

3x3 Filter

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Dot Product

3

Credit: https://cs.uwaterloo.ca/~mli/Deep-Learning-2017-Lecture5CNN.ppt

Convolutions

1	-1	-1
-1	1	-1
-1	-1	1

3x3 Filter

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Dot Product

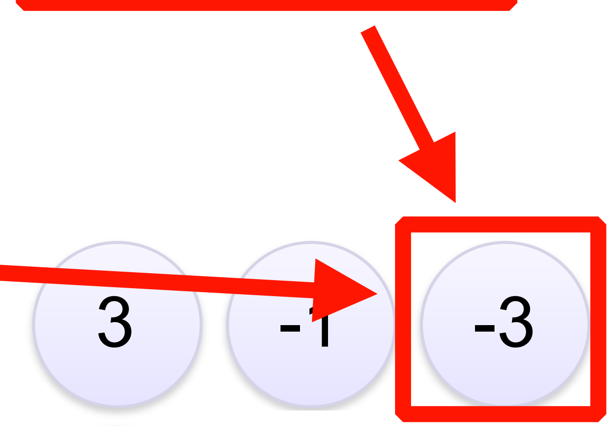
3 $\boxed{-1}$

Convolutions

1	-1	-1
-1	1	-1
-1	-1	1

3x3 Filter

Dot Product



1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Credit: https://cs.uwaterloo.ca/~mli/Deep-Learning-2017-Lecture5CNN.ppt

Convolutions

1	-1	-1
-1	1	-1
-1	-1	1

3x3 Filter

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Dot Product

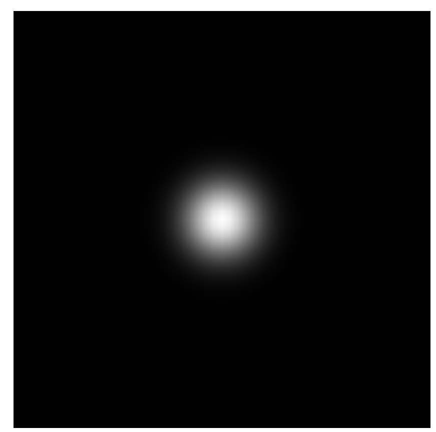
3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

4x4 Convolution

Credit: https://cs.uwaterloo.ca/~mli/Deep-Learning-2017-Lecture5CNN.ppt

Convolutions for Image Processing

Gaussian
Convolution



*

=

Credit: Frank Wood

Convolutions for Image Processing

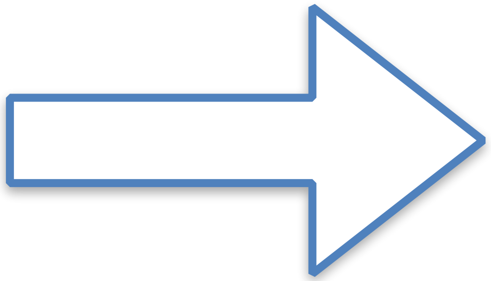
Emboss
Filter

$$* \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} =$$

Credit: Frank Wood

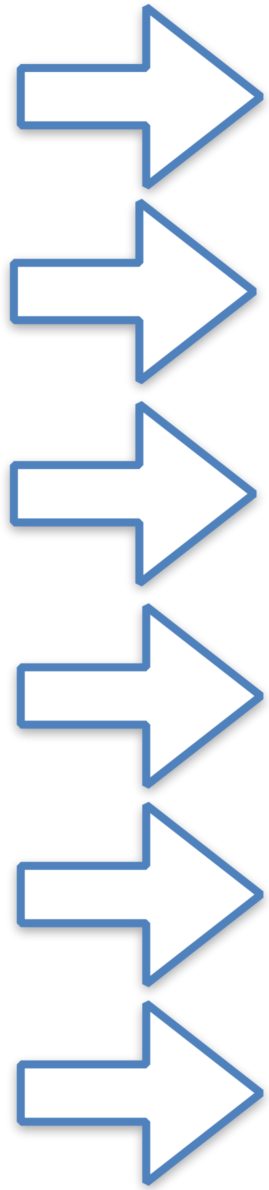
Convolutions for Image Processing

Sharpen Blue
Channel

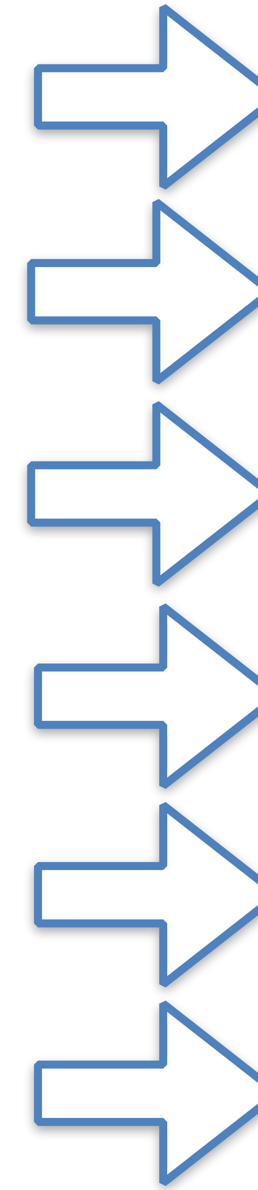


$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Credit: Frank Wood



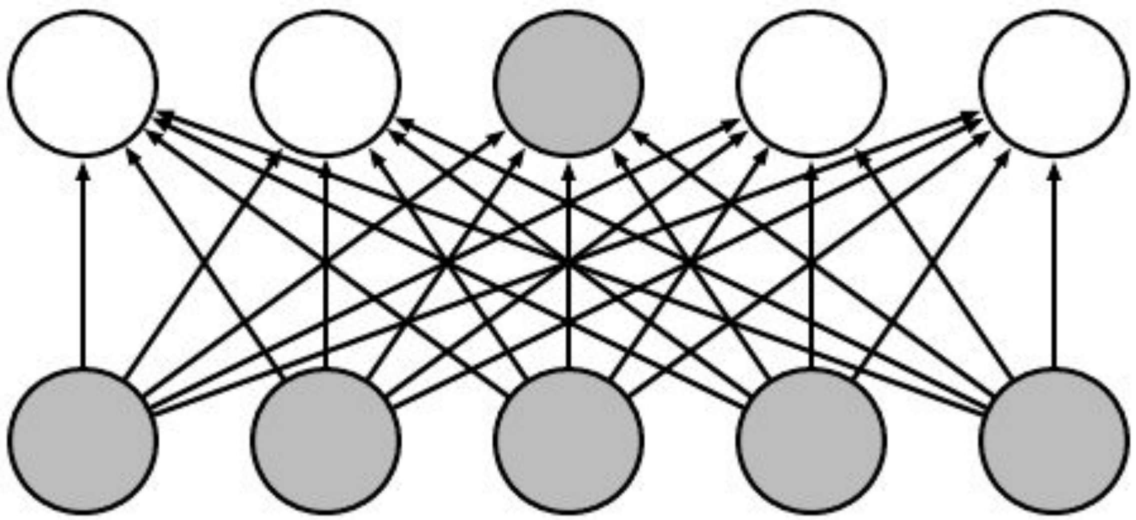
Learn filters



Input Layer

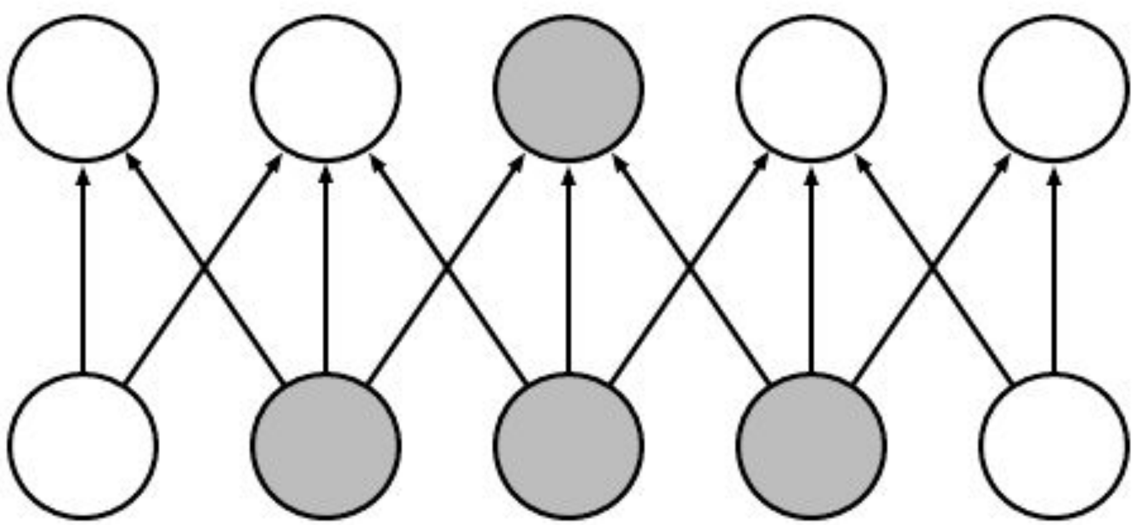
Hidden Layer 1 (before applying activations)

So how come this is a neural net?



Fully connected layer

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix}$$

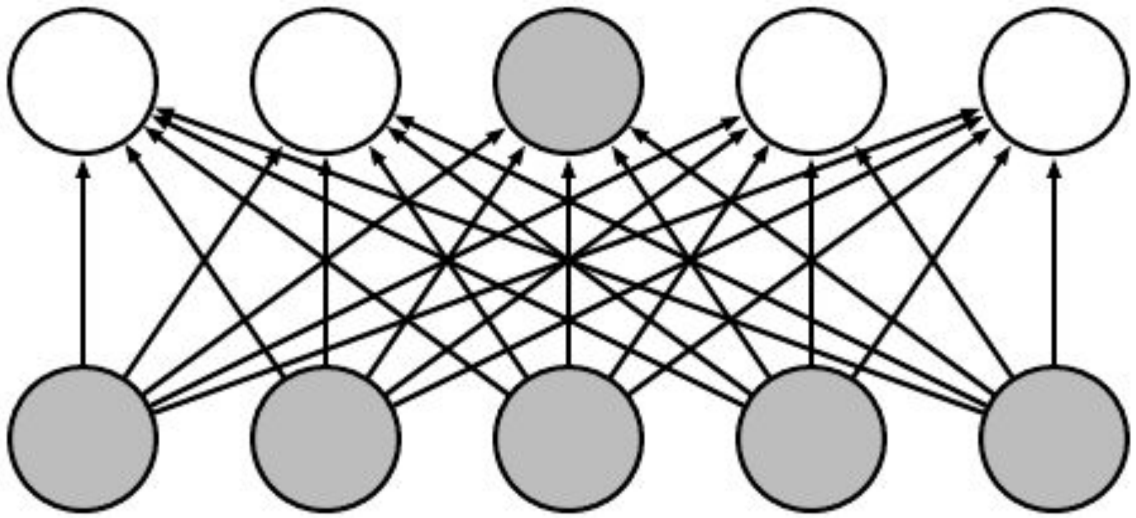


Convolutional layer

$$\begin{bmatrix} w_2 & w_3 & 0 & 0 & 0 \\ w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \\ 0 & 0 & 0 & w_1 & w_2 \end{bmatrix}$$

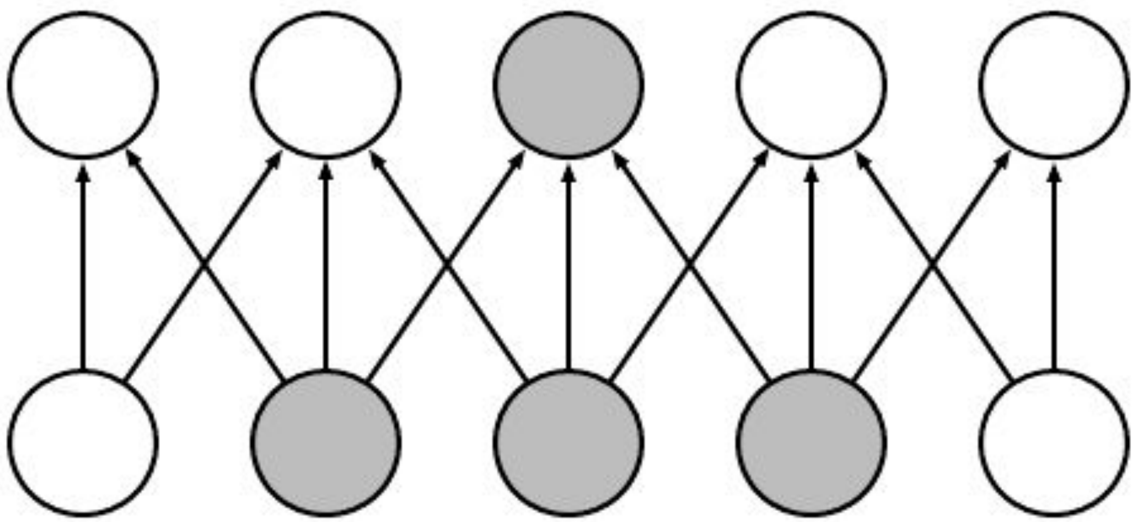
Convolution is equivalent to sparse matrix multiplication with **shared parameters**

So how come this is a neural net?



Fully connected layer

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ \boxed{w_{41} \quad w_{42} \quad w_{43} \quad w_{44} \quad w_{45}} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix}$$

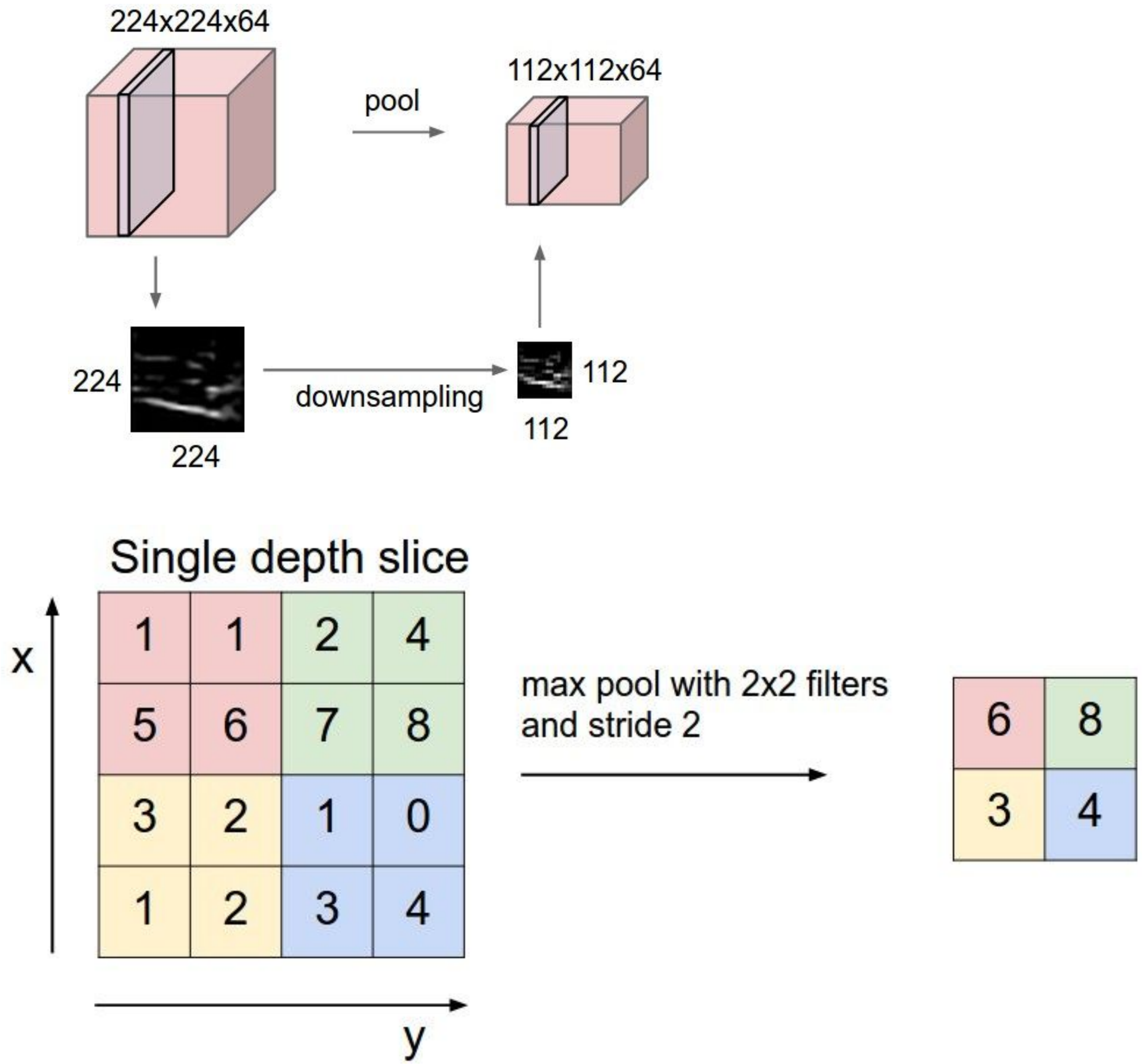


Convolutional layer

$$\begin{bmatrix} w_2 & w_3 & 0 & 0 & 0 \\ w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ \boxed{0 \quad 0 \quad w_1 \quad w_2 \quad w_3} \\ 0 & 0 & 0 & w_1 & w_2 \end{bmatrix}$$

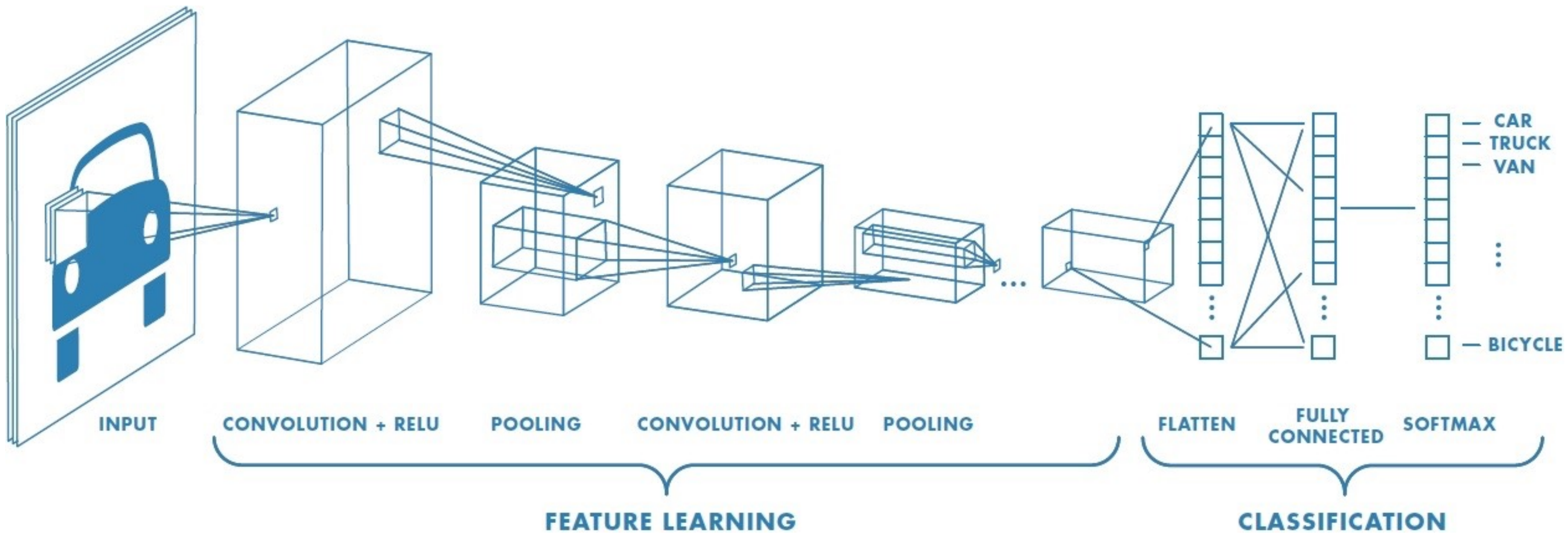
Convolution is equivalent to sparse matrix multiplication with **shared parameters**

Max Pooling



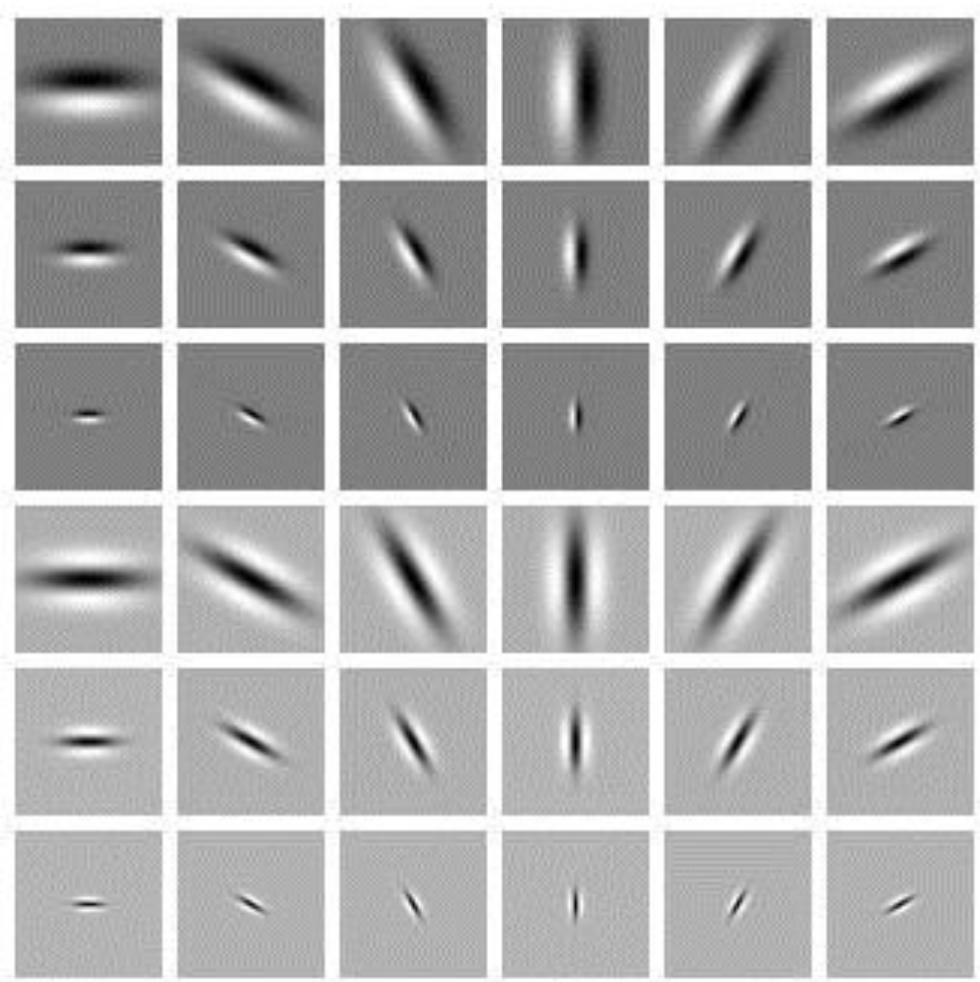
Credit http://cs231n.github.io/convolutional-networks/

Conventional CNNs are a mixture of convolutional layers, max pooling layers, and fully connected layers



Why convolutions + max pooling?

- Convolutions often provide good representations, particularly for images



Credit: Frank Wood

Why convolutions + max pooling?

- Massively reduce number of parameters stored in memory and computations need to carry out
- Restricts towards networks we expect to be effective: can also help for avoiding overfitting

Fully connected layer

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix}$$

25 Parameters

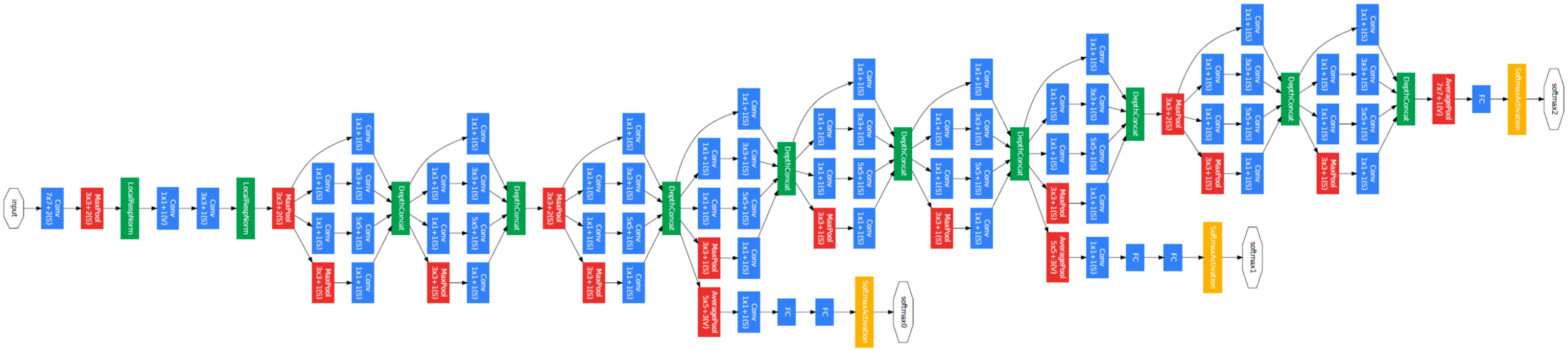
Convolutional layer

$$\begin{bmatrix} w_2 & w_3 & 0 & 0 & 0 \\ w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \\ 0 & 0 & 0 & w_1 & w_2 \end{bmatrix}$$

3 Parameters

Why convolutions + max pooling?

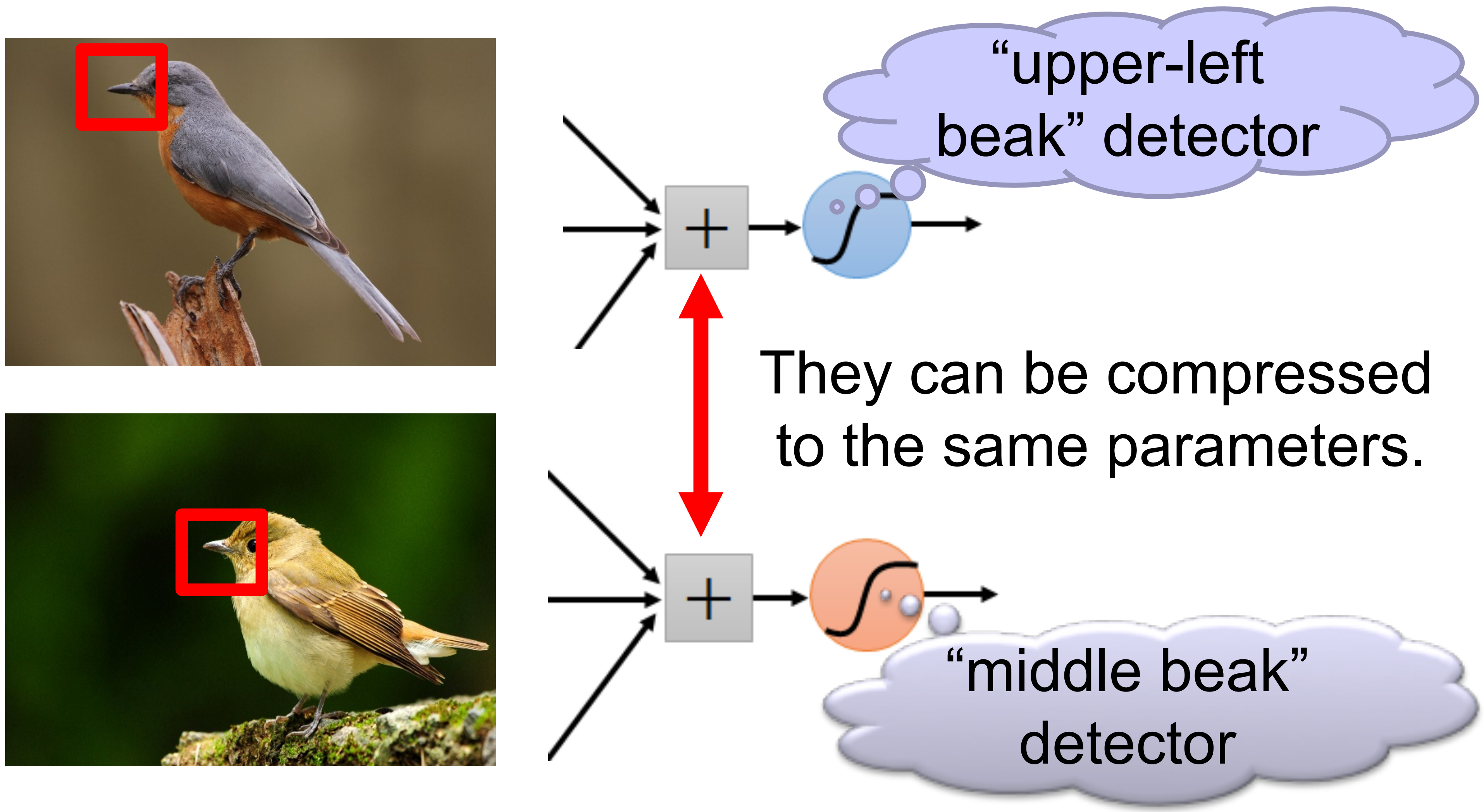
- Computational savings allow us to go deeper:



[GoogleNet. Szegedy et al 2014]

Why convolutions + max pooling?

- Naturally introduce spatial invariances



Credit: https://cs.uwaterloo.ca/~mli/Deep-Learning-2017-Lecture5CNN.ppt

CNNs Take Homes

- CNNs are very powerful machine learning tool with a lot of successful applications, particularly for image data
- They work by using a series of convolution and max pool layers to try and learn features, before having a number of fully connect layers to do the final prediction
- They are effective because they form a principled means of designing large networks without exploding the number of parameters

Other Cool Applications (not examinable)

MS COCO Image Captioning Challenge

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."

"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

"young girl in pink shirt is swinging on swing."

"man in blue wetsuit is surfing on wave."

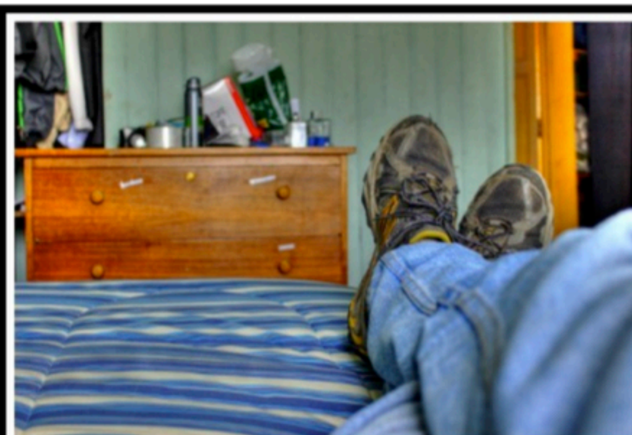
Karpathy & Fei-Fei, 2015; Donahue et al., 2015; Xu et al, 2015; many more

Slide Credit: Dan Klein and Pieter Abbeel

Visual QA Challenge

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, Devi Parikh

What vegetable is on the plate?
Neural Net: broccoli
Ground Truth: broccoli

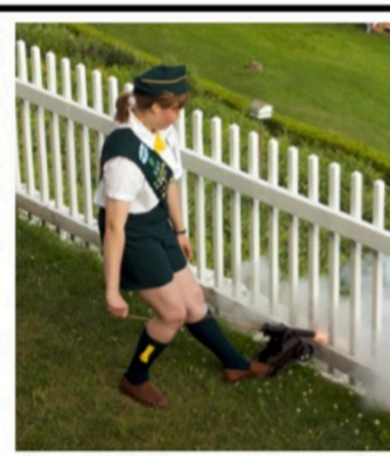


What color are the shoes on the person's feet ?
Neural Net: brown
Ground Truth: brown

How many school busses are there?
Neural Net: 2
Ground Truth: 2

What sport is this?
Neural Net: baseball
Ground Truth: baseball

What is on top of the refrigerator?
Neural Net: magnets
Ground Truth: cereal



What uniform is she wearing?
Neural Net: shorts
Ground Truth: girl scout

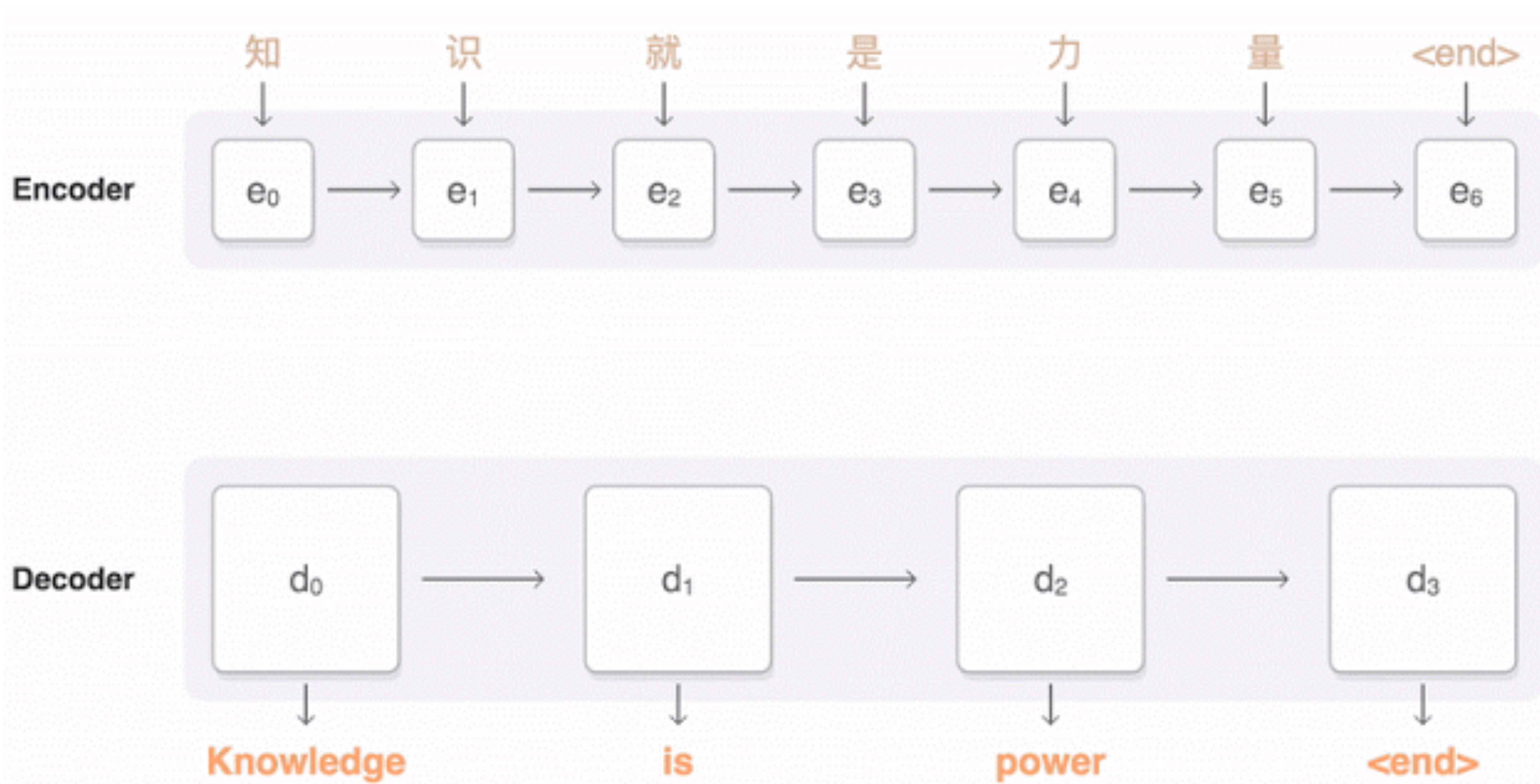
What is the table number?
Neural Net: 4
Ground Truth:40

What are people sitting under in the back?
Neural Net: bench
Ground Truth: tent

Slide Credit: Dan Klein and Pieter Abbeel

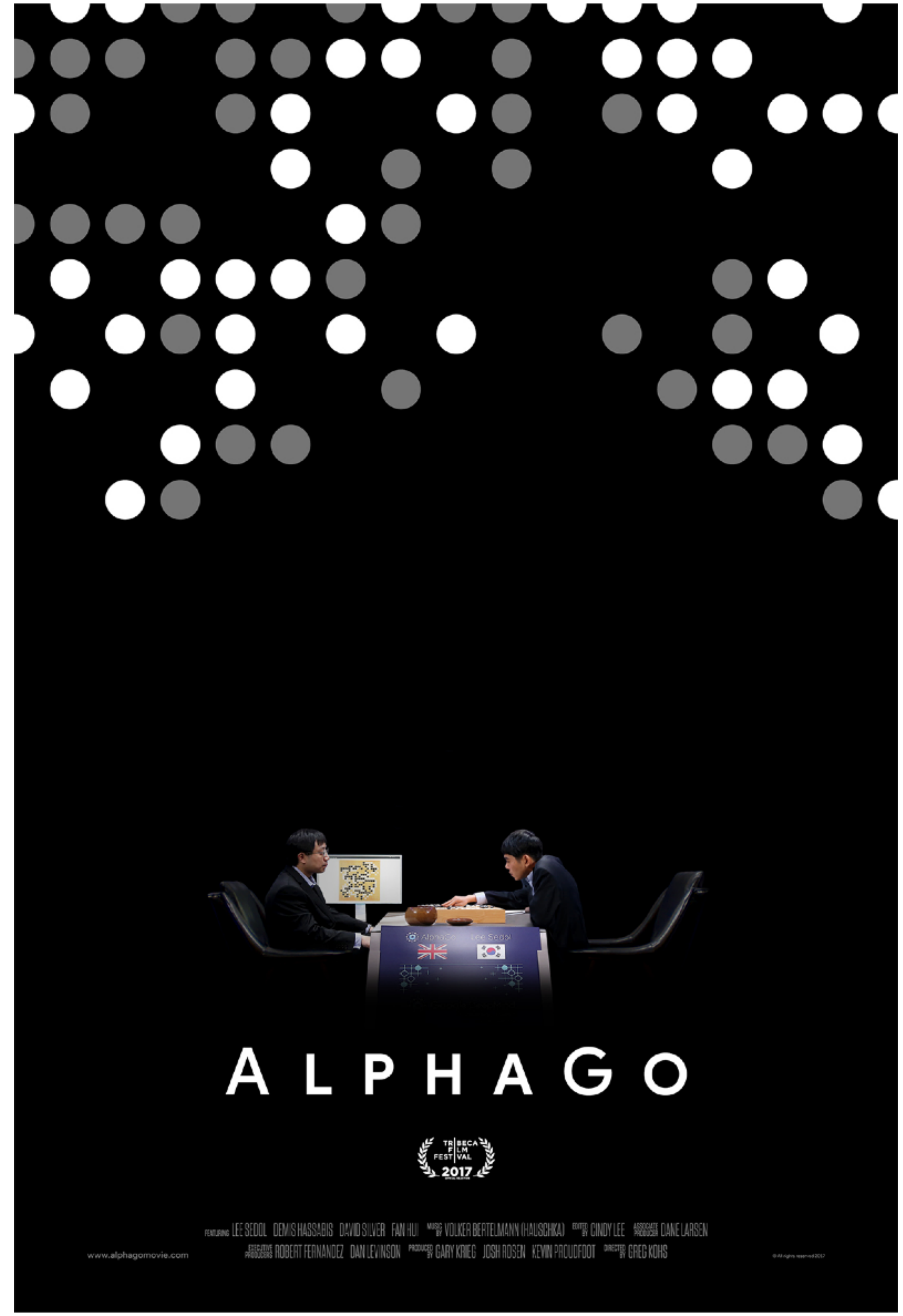
Machine Translation

Google Neural Machine Translation (in production)



Slide Credit: Google AI Research Blog

Deep Reinforcement Learning



Deep Generative Models

Kingma and Dhariwal. Glow. NeurIPS 2018

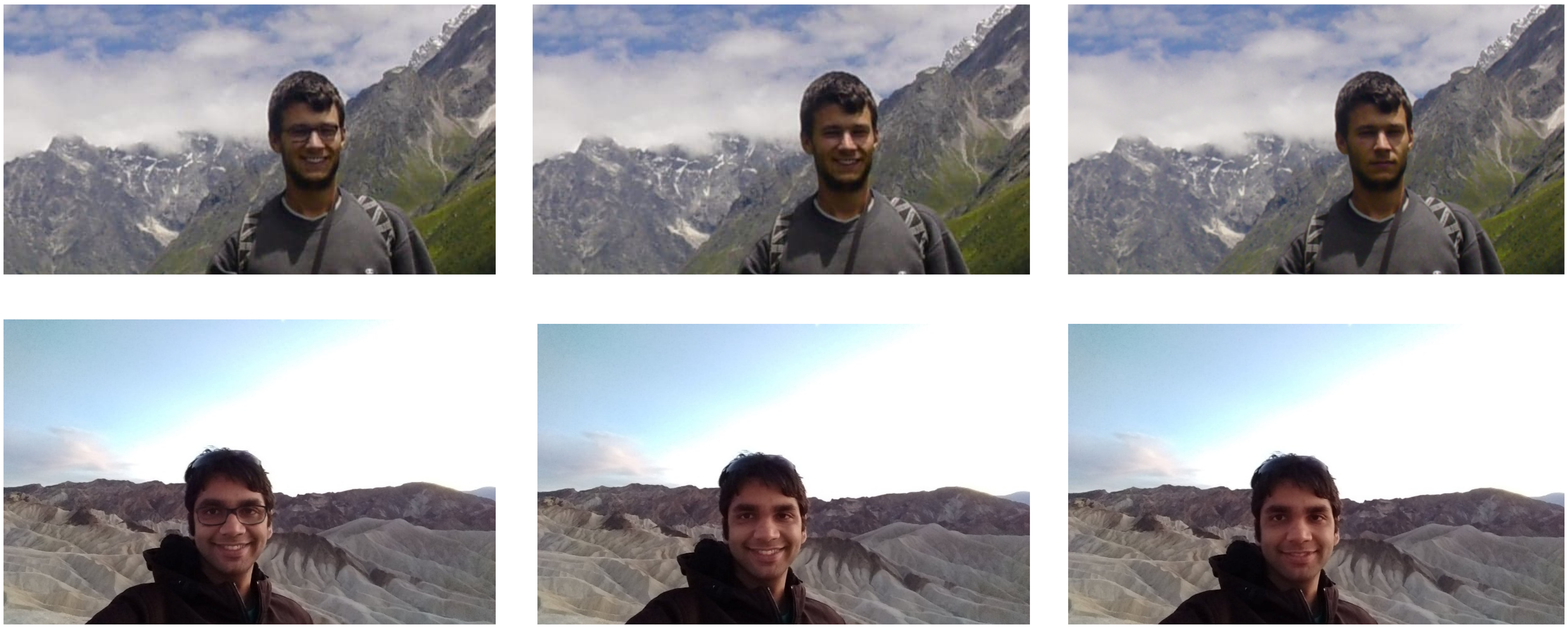
Deep Fakes

Which image is real?

Images credit: Stefano Ermon and Aditya Grover

Deep Fakes

Neither!



No glasses!

No smile!

Images credit: Stefano Ermon and Aditya Grover

Sentence Generation

https://talktotransformer.com/

Further Resources

- Deep learning book (free online version): https://www.deeplearningbook.org/
- Coursera machine learning course: https://www.coursera.org/learn/machine-learning#syllabus
- Stanford deep learning course: http://cs231n.stanford.edu/
- Advanced topics in machine learning course from Computer Science: https://www.cs.ox.ac.uk/teaching/courses/2019-2020/advml/
- Tensorflow https://www.tensorflow.org/tutorials/ and PyTorch tutorials https://pytorch.org/tutorials/
- Google Colab: colab.research.google.com

Fin!
Thanks for listening