Lecture 7: Neural Networks

Matt Ellis and Mike Smith

Session outline

Automatic differentiation

- Reverse mode AD

Neural networks

- Biological neurons
- (Deep) Neural networks
- Convolutional operations and networks

Learning Objectives

By the end of this session you should be able to:

1. Apply reverse mode auto-differentiation to a function.

2. Explain the structure and training of linear and convolutional neural networks.

3. Justify the architecture of a neural network model for a given task.

4. Assess the parameters in a deep neural network model.

Automatic Differentiation

Automatic differentiation (AD)

AD is concerned about exact numeric computation rather than their actual symbolic form.

It computes the derivative by only storing the values of intermediate sub-expressions.

It uses a combination of:

- **Symbolic differentiation** at the elementary operation level
- Storing **intermediate** numerical results

Derivatives with many inputs and outputs

Say that we have several functions that depend on several input variables:

$$
\begin{aligned}
y_1 &= f_1(x_1, \ldots, x_n) \\
y_2 &= f_2(x_1, \ldots, x_n) \\
&\vdots \\
y_m &= f_m(x_1, \ldots, x_n)
\end{aligned}
$$

Then the Jacobian, **J**, is a m by n matrix with entries:

$$
J_{ij} = \frac{\partial f_i}{\partial x_j}
$$

$$
\mathbf{J} = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\
\vdots & \vdots & \vdots & \vdots \\
\frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n}
\end{bmatrix}
$$

Also, the Hessian is the derivative of the Jacobian.

Evaluation trace

Evaluation trace: composition of elementary operations that lead to a full expression.

General notation:

Let $\mathbf{y} = \mathbf{f}(x_1, \ldots, x_n)$ with n inputs and m outputs ($\mathbb{R}^n \rightarrow \mathbb{R}^m$)

- Input variables: $v_{i-n} = x_i$ with $i = 1,\ldots, n$
- Intermediate variables: v_i with $i = 1,\ldots, l$
- Output variables: $y_i = v_{l+i}$ with $i = 1,\ldots, m$

Computational graph: network showing how all these variables are connected.

Example

Let's think about logistic regression with 1 input and 1 weight:

$$y = \text{Sigmoid}(w_1 x_1).$$

Inputs

$$v_{-1} = x_1$$

$$v_0 = w_1$$

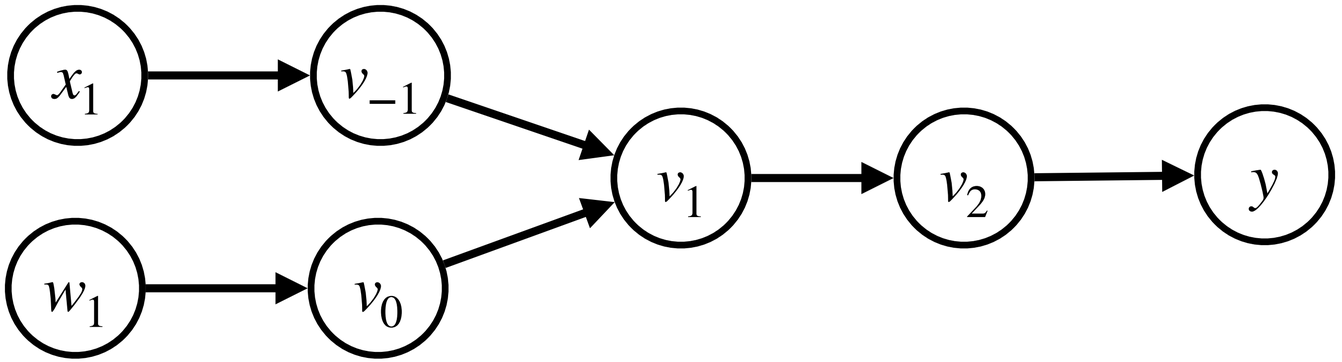
Intermediates

$$v_1 = v_{-1} v_0$$

$$v_2 = \text{Sigmoid}(v_1)$$

Outputs

$$y = v_2$$



Exercise

Consider the following example

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2).$$

The inputs are x_1 and x_2. What are the elementary operations?

$$v_{-1} = x_1$$

$$v_0 = x_2$$

$$v_1 = \ln(v_{-1})$$

$$v_2 = v_{-1} v_0$$

$$v_3 = \sin(v_0)$$

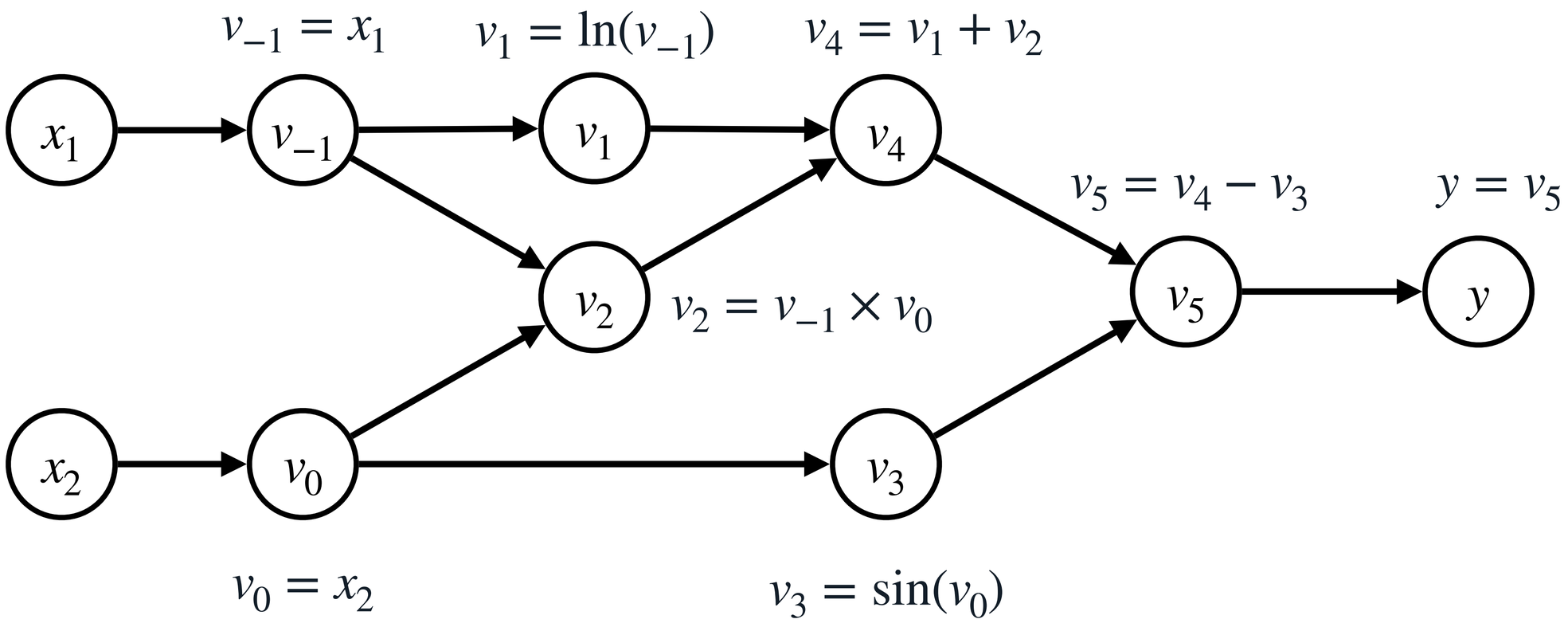
$$v_4 = v_1 + v_2$$

$$v_5 = v_4 - v_3$$

$$y = v_5$$

Let's draw a **computational graph** to show how they interact.

Computational graph



Forward accumulation mode

Also called tangent linear mode.

To compute the derivative of f w.r.t x_j, each intermediate variable v_i has a derivative

$$\dot{v}_i = \frac{\partial v_i}{\partial x_j}$$

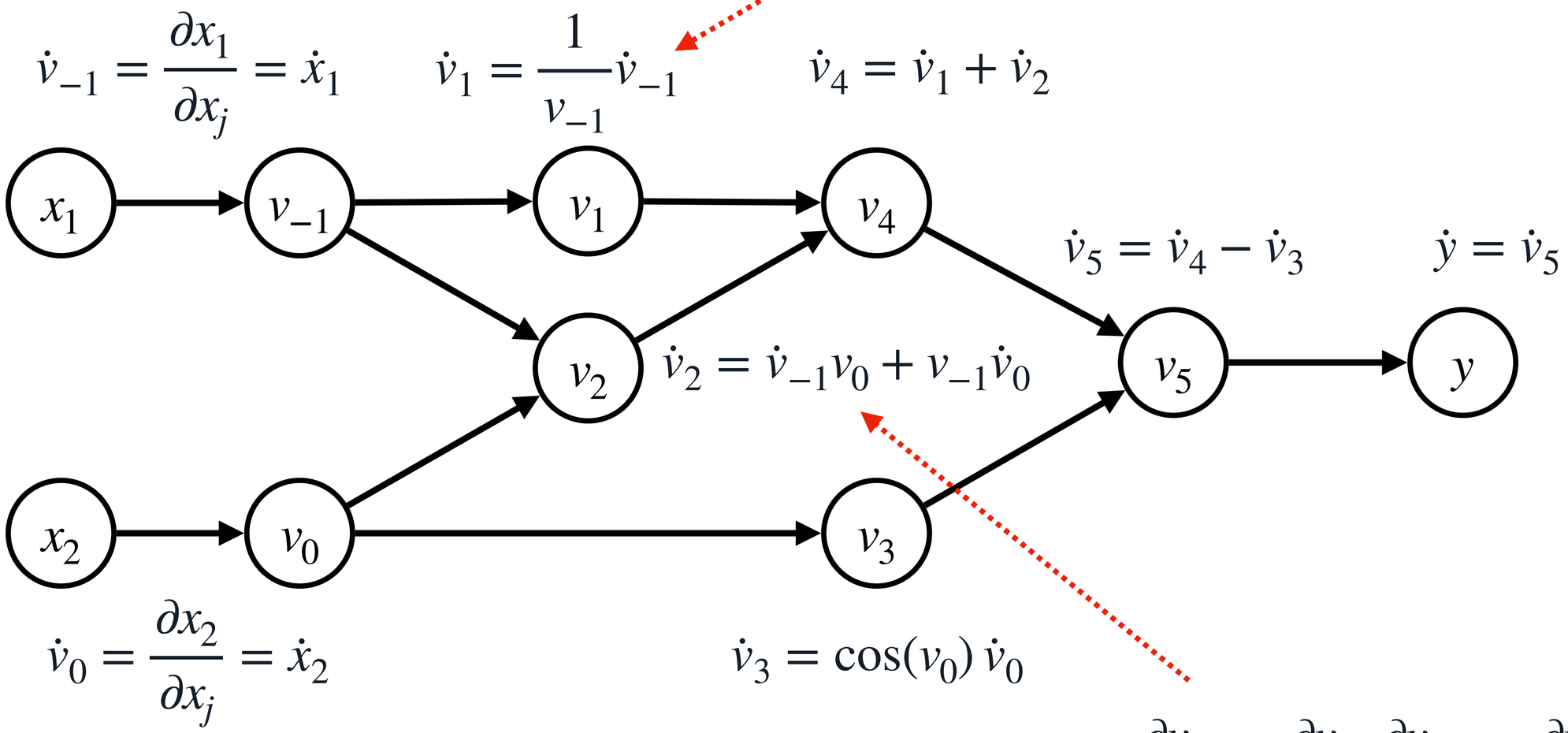
For each evaluation (or forward primal) trace, it builds a forward derivative (or tangent) trace.

Essentially, this is just implementing the chain rule:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dz}\frac{dz}{dx}.$$

Forward tangent trace

$$\dot{v}_1 = \frac{\partial v_1}{\partial x_j} = \frac{\partial v_1}{\partial v_{-1}} \frac{\partial v_{-1}}{\partial x_j}$$



$$\dot{v}_2 = \frac{\partial v_2}{\partial x_j} = \frac{\partial v_2}{\partial v_{-1}} \frac{\partial v_{-1}}{\partial x_j} + \frac{\partial v_2}{\partial v_0} \frac{\partial v_0}{\partial x_j}$$

New: Note on chain rule with multiple inputs

Previously we saw that using the chain rule for a function of a function $f(g(x))$ is

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

But if we have a function that has many inputs then it needs to be expanded. For example if $f(g_1(x), g_2(x), g_3(x))$ then the derivative is:

$$\frac{df}{dx} = \frac{\partial f}{\partial g_1}\frac{dg_1}{dx} + \frac{\partial f}{\partial g_2}\frac{dg_2}{dx} + \frac{\partial f}{\partial g_3}\frac{dg_3}{dx}$$

I.e we need to sum the independent contributions. More generally if $f(g_1(x), \ldots, g_n(x))$

$$\frac{df}{dx} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i}\frac{dg_i}{dx}.$$

For more see page 148 of Mathematics for ML

Example

Compute the derivative $\partial y/\partial x_1$ at $x_1 = 2, x_2 = 5$.

Forward primal trace	Values
$v_{-1} = x_1$	$= 2$
$v_0 = x_2$	$= 5$
$v_1 = \ln(v_{-1})$	$= \ln(2)$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$
$v_3 = \sin(v_0)$	$= \sin(5)$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
$y = v_5$	$= 11.652$

Forward tangent trace	Values
$\dot{v}_{-1} = \dot{x}_1$	$= 1$
$\dot{v}_0 = \dot{x}_2$	$= 0$
$\dot{v}_1 = \frac{1}{v_{-1}}\dot{v}_{-1}$	$= \frac{1}{2}1$
$\dot{v}_2 = \dot{v}_{-1} \times v_0 + v_{-1} \times \dot{v}_0$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3 = \cos(v_0)\,\dot{v}_0$	$= \cos(5) \times 0$
$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y} = \dot{v}_5$	$= 5.5$

If we want to compute for $\partial y/\partial x_2$ then swap initial tangent trace values.

Generalisation to the Jacobian of a function

Remember $\mathbf{f}(x_1, \ldots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a function with n independent variables and m dependent variables.

The derivatives of the Jacobian, $\partial y_i / \partial x_j$, are computed by making $\dot{x}_j = 1$ initially and all the other derivatives $\dot{x}_k = 0$ for $k \neq i$.

The values of all the derivatives $\dot{y}_i = \left. \frac{\partial y_i}{\partial x_j} \right|_{x=a}$ are obtained by a forward pass.

Note that for a specific x_j we can compute all the derivatives for all outputs I, which corresponds to the column in the Jacobian.

To compute the whole Jacobian, we need n forward passes, one per input variable.

Complexity

AD forward mode is efficient for functions like $\mathbf{f} : \mathbb{R} \rightarrow \mathbb{R}^m$.

As be saw before, this is because we can compute all the output derivatives for a single input in one pass.

In the other extreme $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}$, it would need n forward passes which becomes computationally expensive when n is large.

In general, when $n \gg m$ the reverse mode of AD is preferred.

AD reverse mode - backpropagation

AD in reverse mode propagates the derivatives backwards from a given output.

It is done by computing the **adjoints** of the intermediate variables

$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$ representing the sensitivity of the output y_j to variable v_i.

AD in reverse mode uses two-phases:

- A **forward step** to compute the variables v_i and to **store** dependencies in the computational graph.
- A backward or **reverse step**, in which the **adjoints** are used to compute the derivatives, starting from the outputs and going back to the inputs.

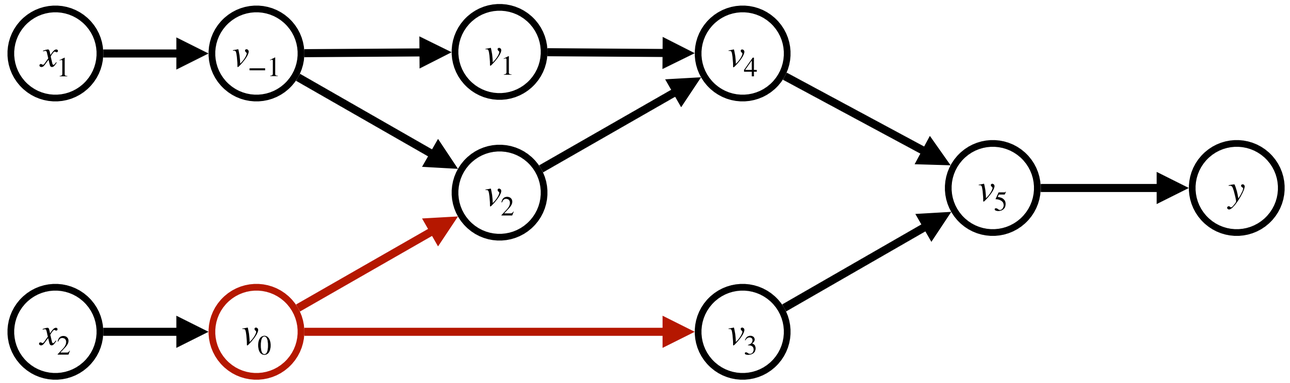
Example

Let's go back to the same example

$$f(x_1, x_2) = y = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

and focus on finding the derivative for x_2.

We need to compute the adjoint $\bar{v}_0 = \partial y / \partial v_0$, this is how a change in v_0 affects the output y. From the graph, we can see that v_0 affects y through v_2 and v_3:



Example

So the contributions of v_0 to y is given as

$$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}.$$

By definition $\frac{\partial y}{\partial v_2} = \bar{v}_2$ and $\frac{\partial y}{\partial v_3} = \bar{v}_3$ so we can simplify this to

$$\frac{\partial y}{\partial v_0} = \bar{v}_2\frac{\partial v_2}{\partial v_0} + \bar{v}_3\frac{\partial v_3}{\partial v_0}.$$

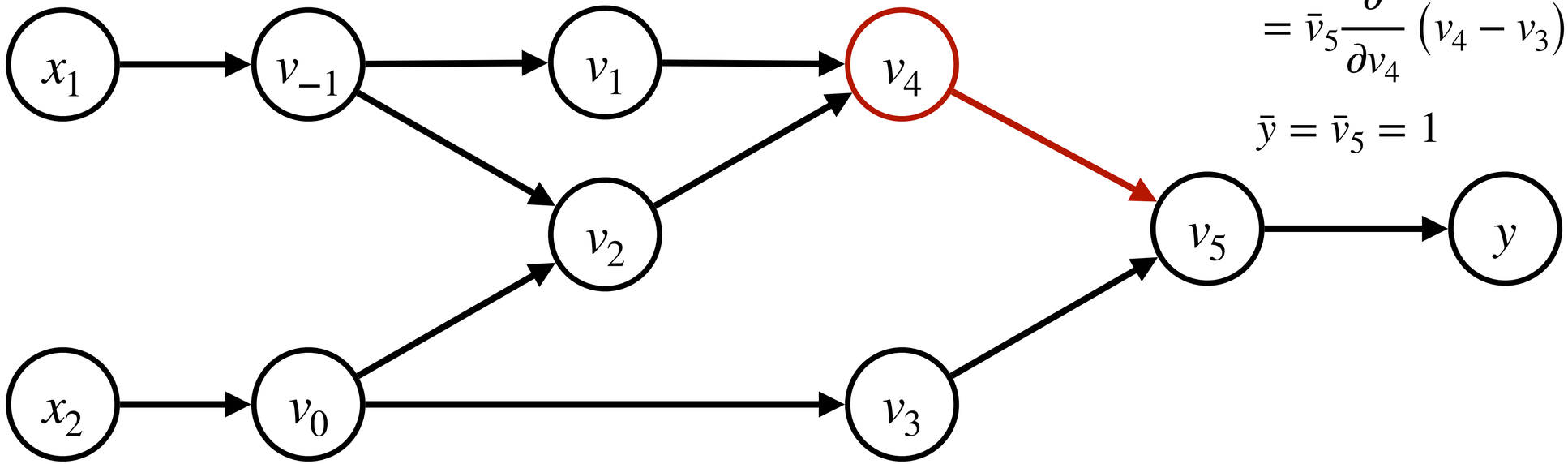
After the forward pass to compute v_i, the reverse pass computes the adjoints.
Starting with $\bar{y} = \bar{v}_5 = 1$ and computing the derivatives w.r.t the inputs at the end.

Reverse adjoint trace

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5\frac{\partial v_5}{\partial v_4}$$

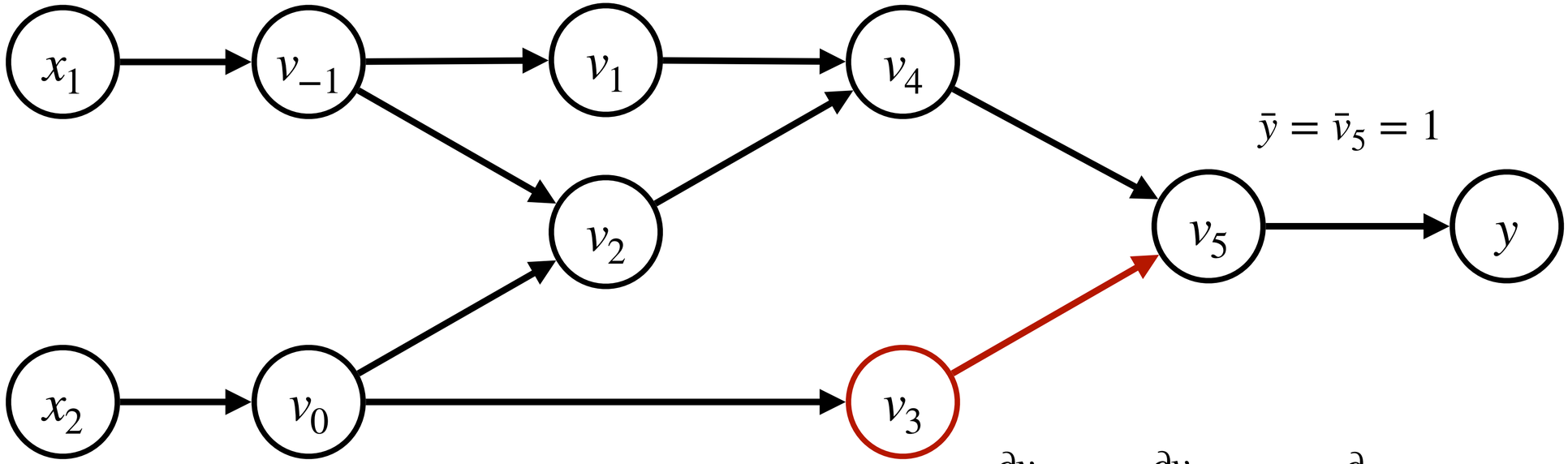
$$= \bar{v}_5\frac{\partial}{\partial v_4}\left(v_4 - v_3\right) = \bar{v}_5$$

$$\bar{y} = \bar{v}_5 = 1$$



Reverse adjoint trace

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \bar{v}_5$$

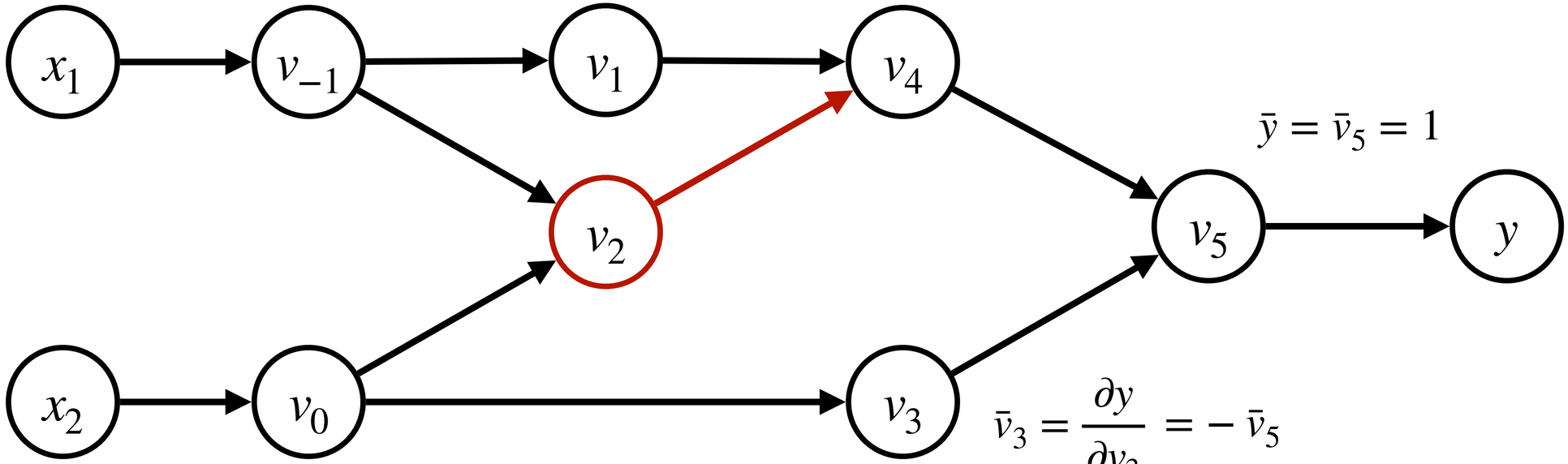


$\bar{y} = \bar{v}_5 = 1$

$$\bar{v}_3 = \frac{\partial y}{\partial v_3} = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 \frac{\partial}{\partial v_3}\left(v_4 - v_3\right) = -\bar{v}_5$$

Reverse adjoint trace

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \bar{v}_5$$

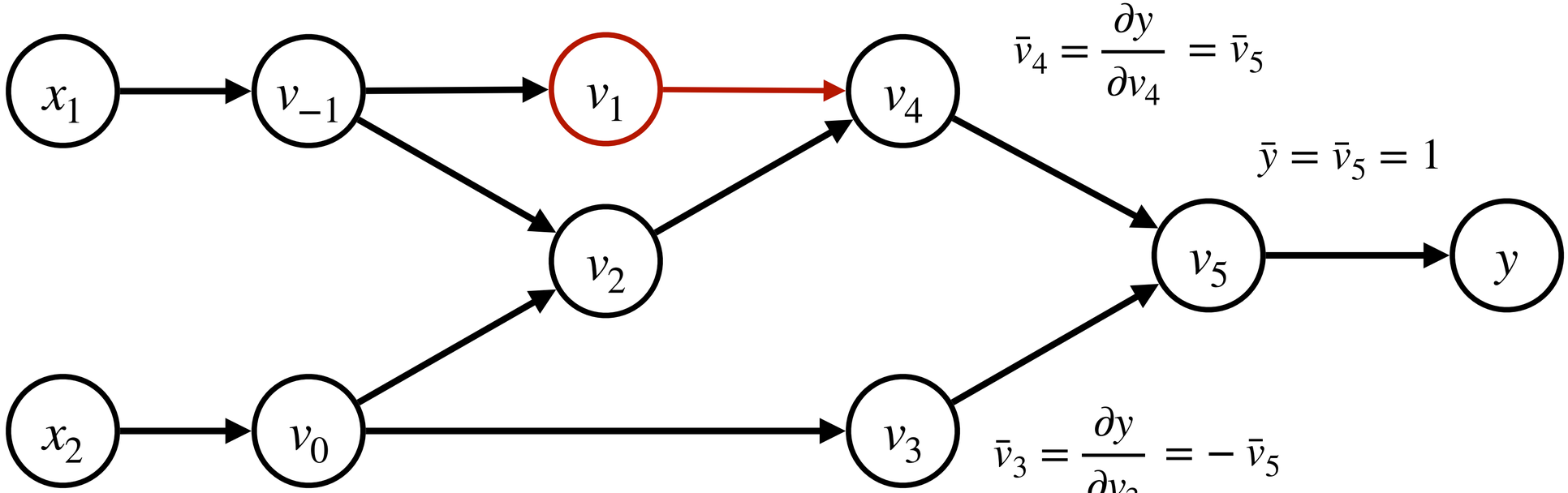


$$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$$

$$= \bar{v}_4 \frac{\partial}{\partial v_2}\left(v_1 + v_2\right) = \bar{v}_4$$

Reverse adjoint trace

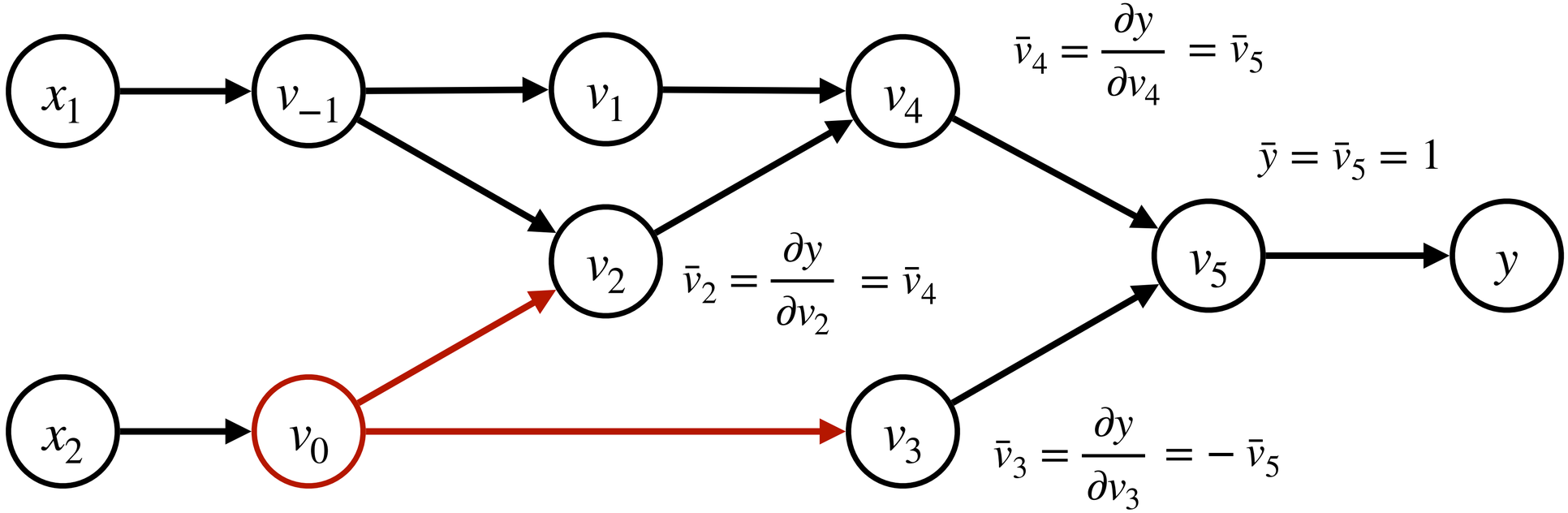
$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \frac{\partial}{\partial v_1}\left(v_1 + v_2\right) = \bar{v}_4$$



$$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_4$$

Reverse adjoint trace

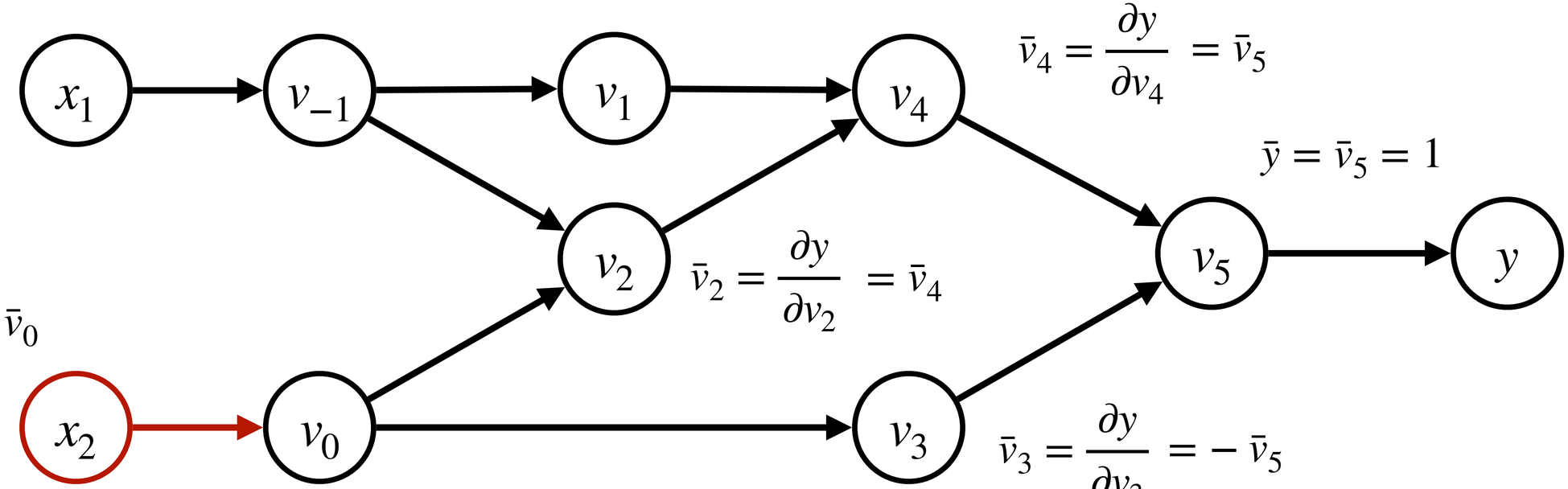
$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_4$$



$$\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0} = \bar{v}_2 \frac{\partial}{\partial v_0}\left(v_{-1} v_0\right) + \bar{v}_3 \frac{\partial}{\partial v_0}\left(\sin(v_0)\right) = \bar{v}_2 v_{-1} + \bar{v}_3 \cos(v_0)$$

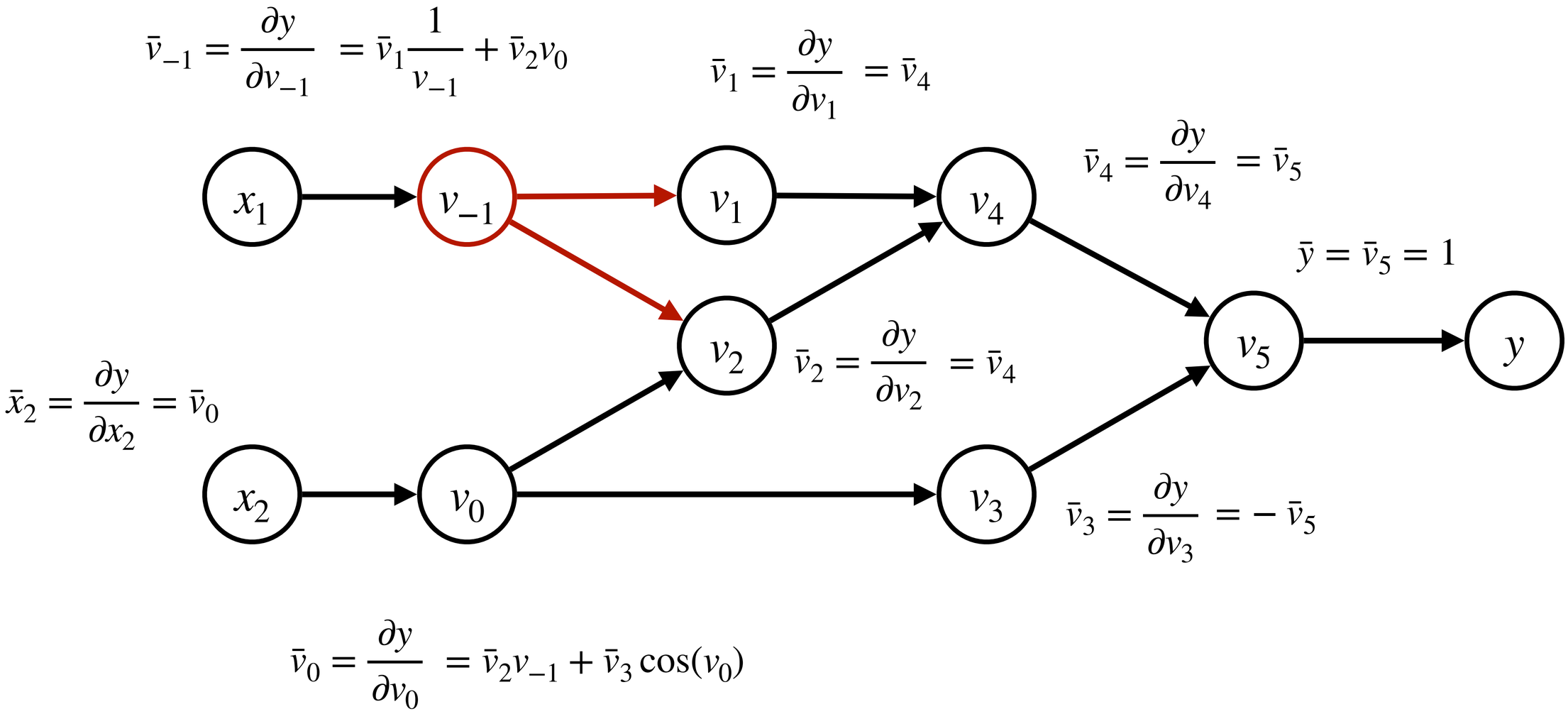
Reverse adjoint trace

$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_4$$



$$\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 v_{-1} + \bar{v}_3 \cos(v_0)$$

Reverse adjoint trace

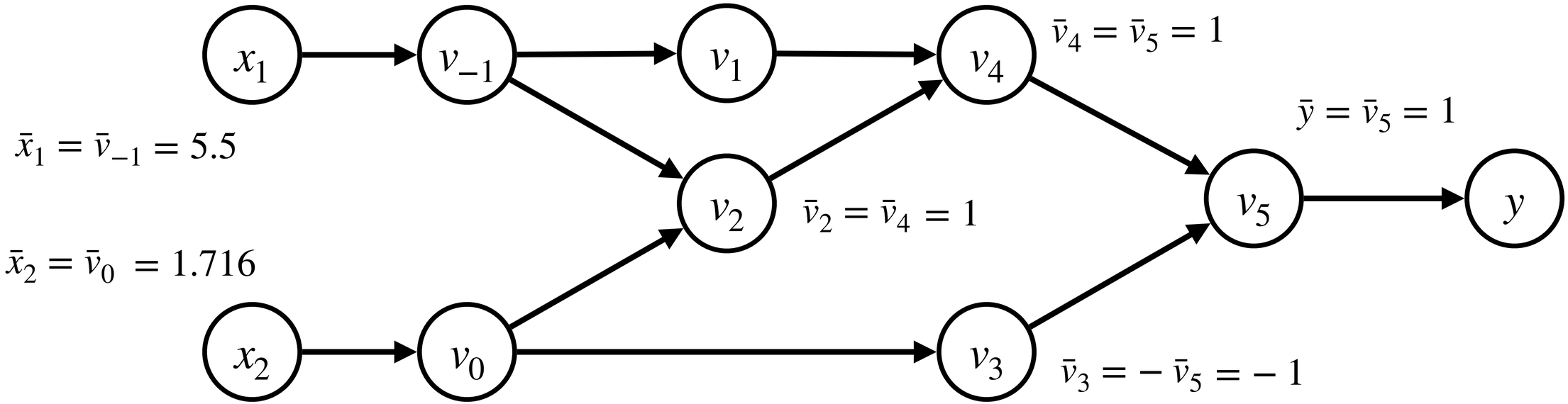


Numerical evaluation of the reverse adjoint trace

$$\bar{v}_{-1} = \bar{v}_1 \frac{1}{v_{-1}} + \bar{v}_2 v_0$$

$$= 1 \times \frac{1}{2} + 1 \times 5 = 5.5$$

$\bar{v}_1 = \bar{v}_4 = 1$



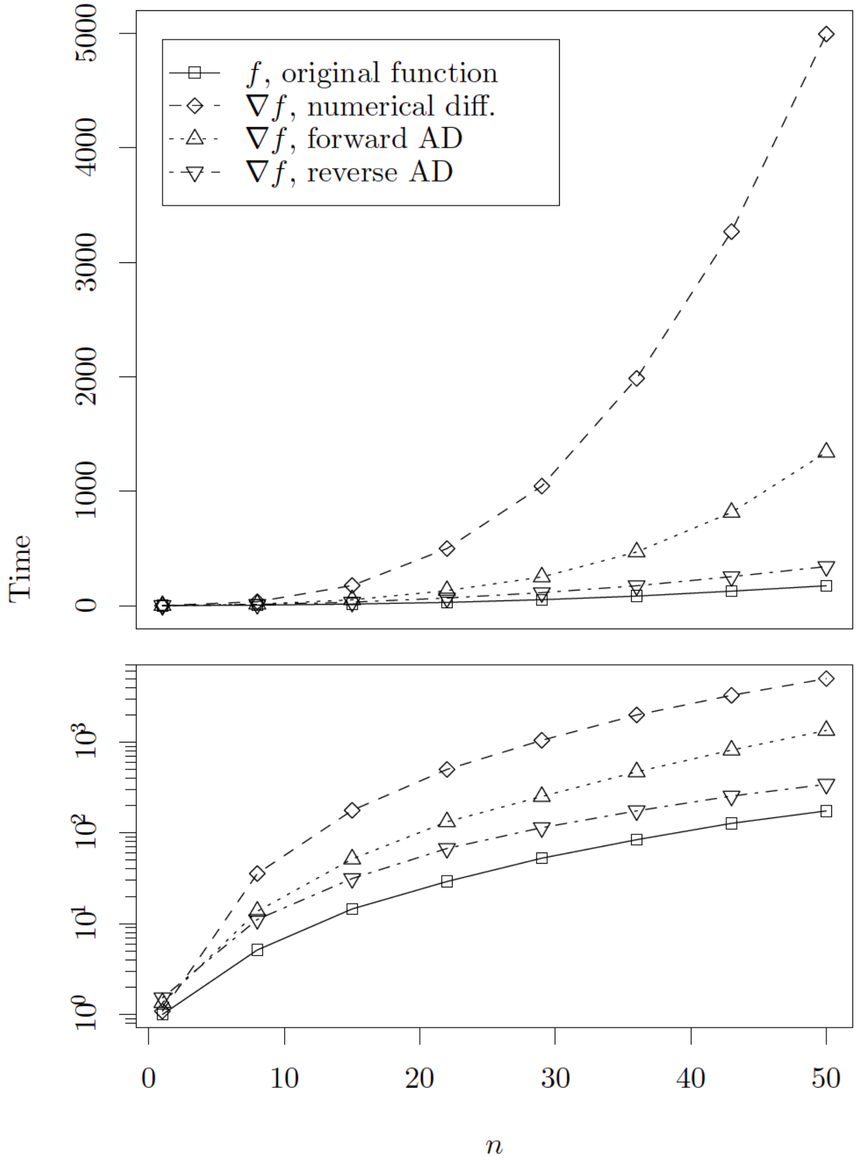
$$\bar{v}_0 = \bar{v}_2 v_{-1} + \bar{v}_3 \cos(v_0) \quad = 1 \times 2 - 1 \times \cos(5) = 1.716$$

Complexity

Reverse mode AD performs better when $n \gg m$.

As we see on the right, reverse mode is generally faster than other types.

The downside is the cost of increased storage, since we need to save the intermediate values for the adjoint evaluation trace.



From Automatic differentiation in machine learning: A survey (see readings)

Reverse mode AD and back propagation

Reverse mode AD is the algorithm used to train neural networks and deep learning models.

To train a neural network model, we optimise an objective function, $E(\mathbf{w})$ that usually depends on a high-dimensional input vector of parameters $\mathbf{w} \in \mathbb{R}^n$ (i.e many inputs but only 1 output (the error).

In the ML community, reverse mode AD goes by the name **backpropagation**, which you will see next week when we look at neural networks.

Common implementations include PyTorch, TensorFlow but many versions in different languages.

Take home messages

A complex function can be **decomposed into intermediate variables** where the derivatives are simple to perform.

Forward mode **accumulates the derivative** for a given input variable, **requires n passes**.

Reverse mode **stores the intermediate values** and **evaluates the adjoints** on a reverse trace. Better when many inputs, fewer outputs.

Forward mode calculates

$$\dot{v}_i = \frac{\partial v_i}{\partial x_j}$$

Reverse mode calculates

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

Further reading

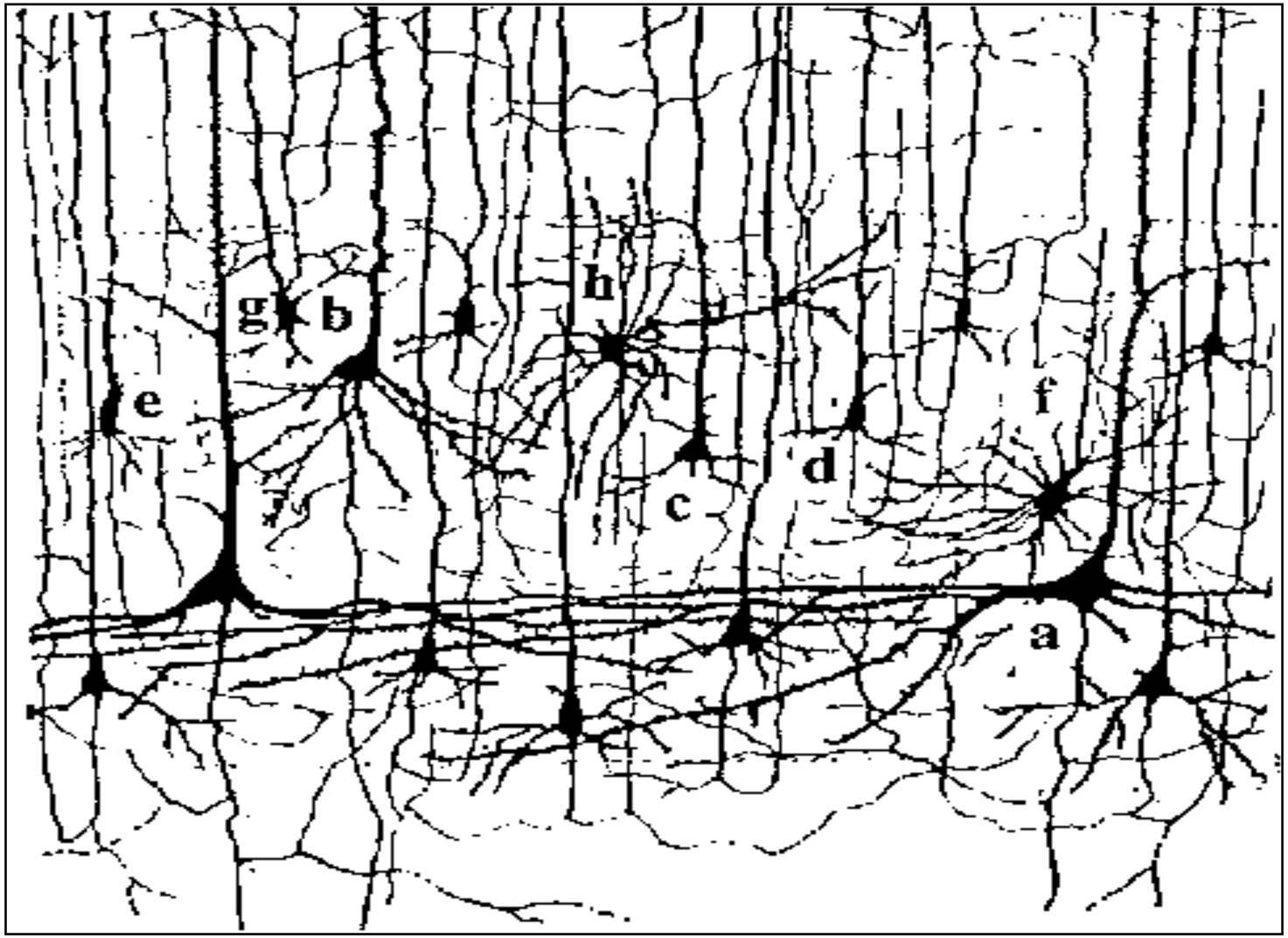
Automatic Differentiation:

Automatic differentiation in machine learning: A survey by Baydin et al.

https://arxiv.org/abs/1502.05767

Neural Networks

Neurons in the mammalian cortex



Experiment by Ramon y Cajal
Nobel prize in 1906.

Highly connected:
1 neuron typically connects to 10,000 post-synaptic neurons

In 1 mm^3:
~ 10,000 neurons
~ 3 km wires

Neuron Structure

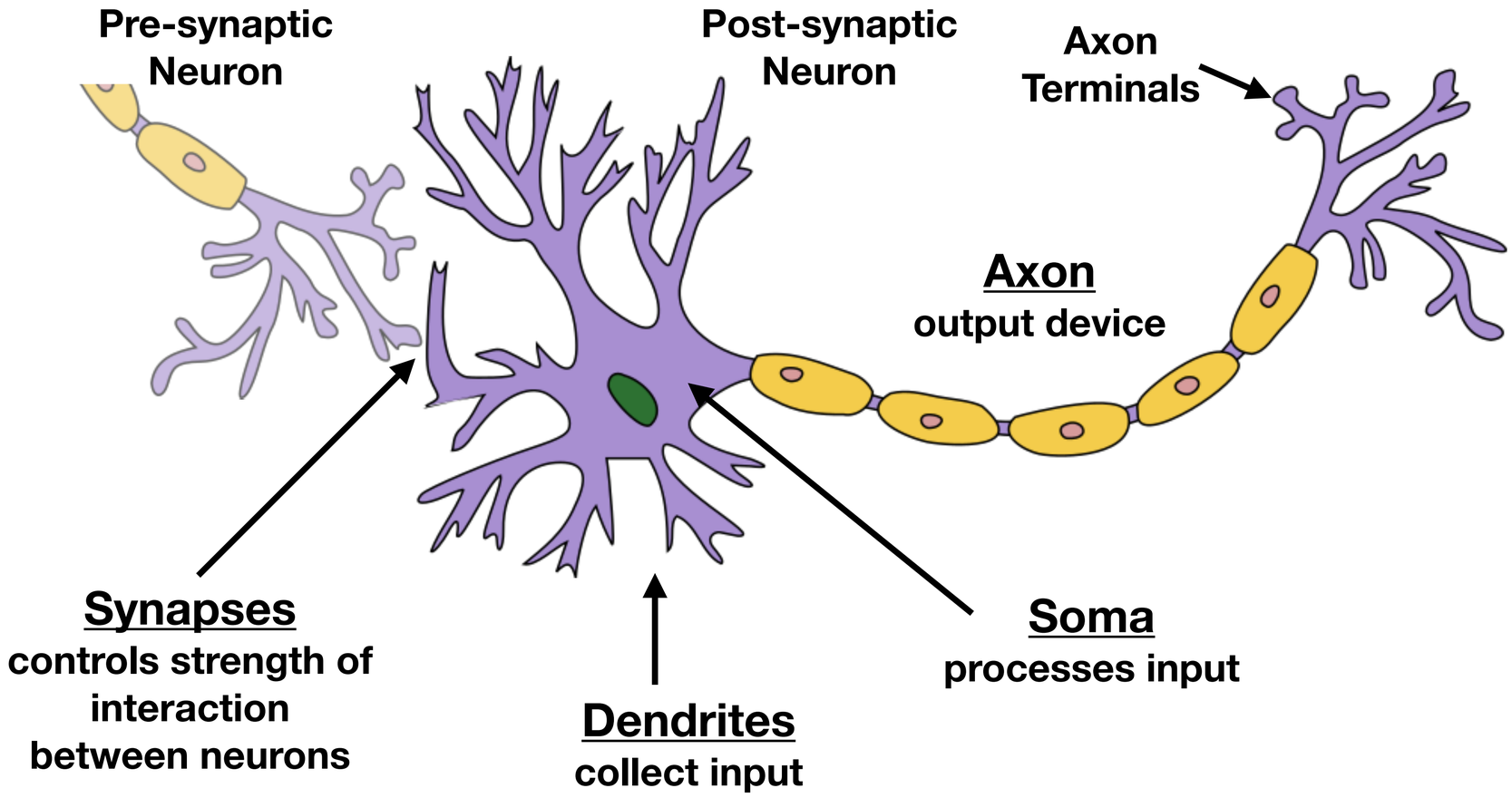
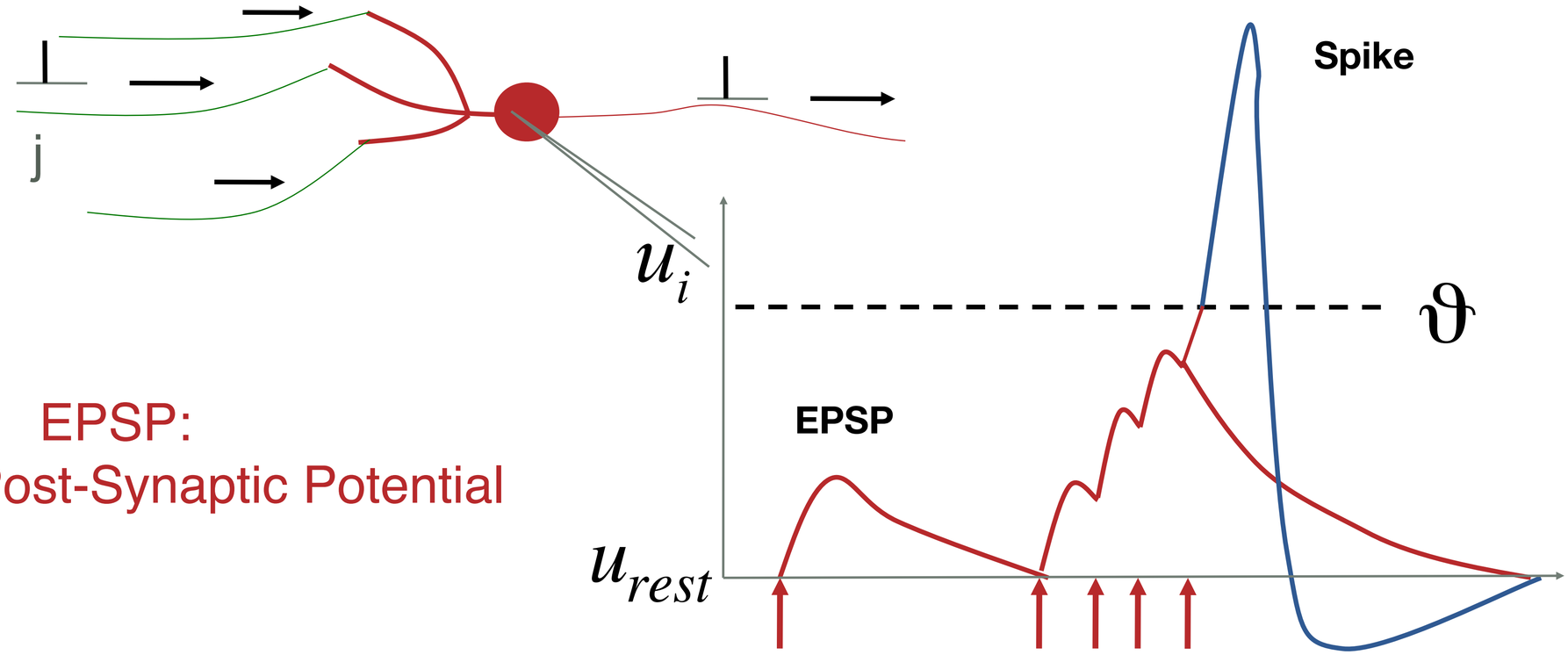


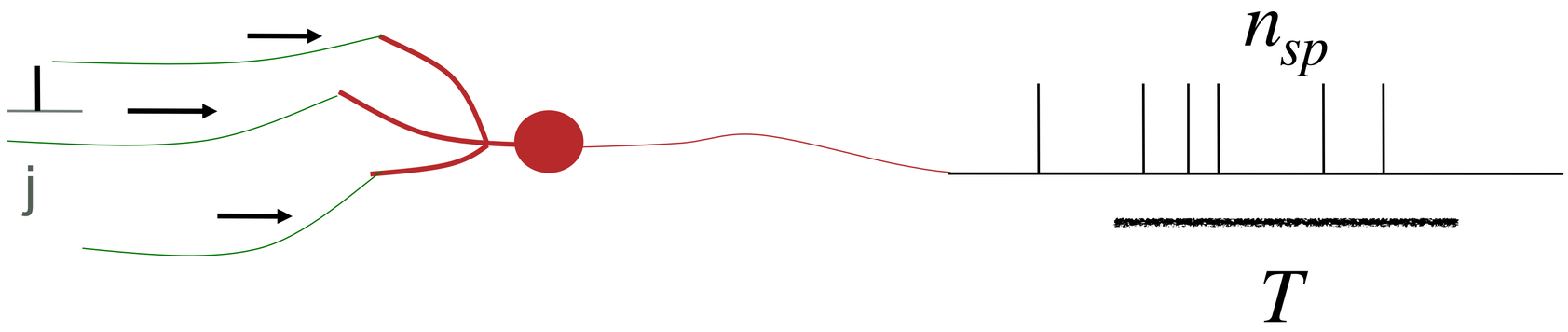
Image credit Wikimedia https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg

Real Neurons

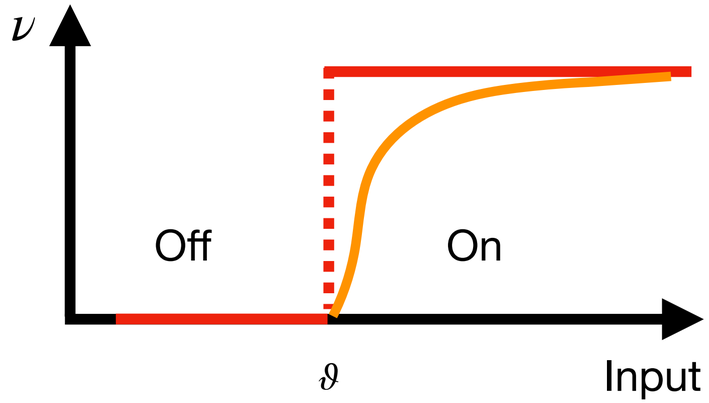


EPSP:
Excitatory Post-Synaptic Potential

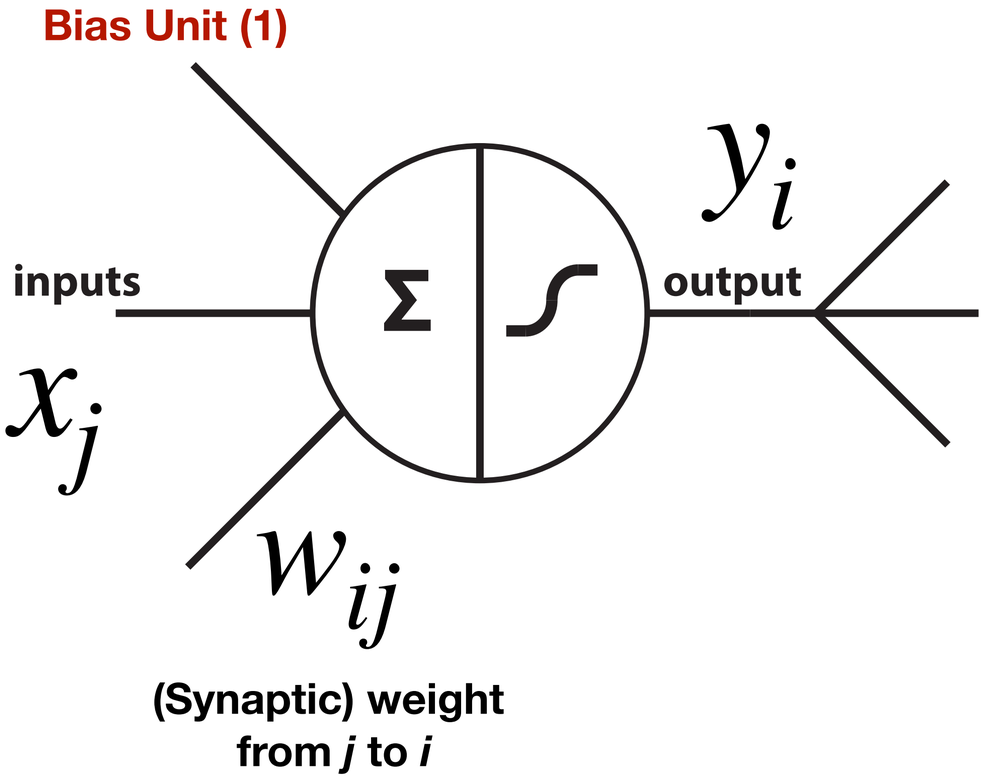
From Spikes to Rates

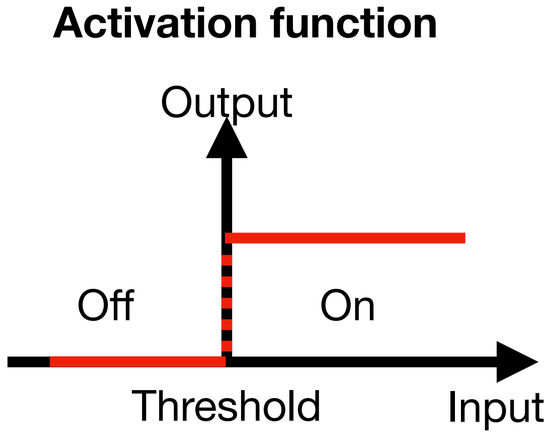


Rate $\nu = \dfrac{n_{sp}(t; t+T)}{T}$



Artificial Neurons



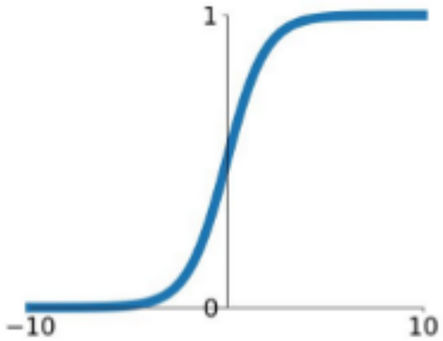


$$y_i = f\left(\sum_j w_{ij} x_j\right)$$

Common activation functions

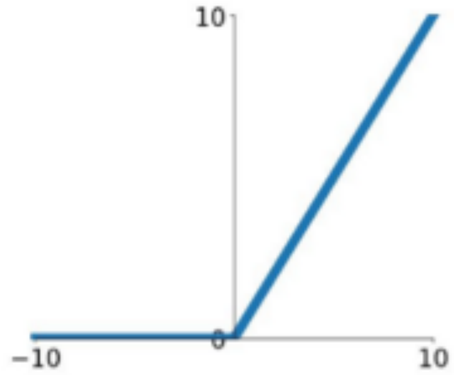
Sigmoid

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



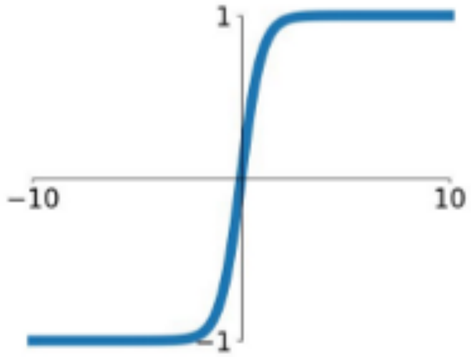
ReLU

$$\max(0, x)$$

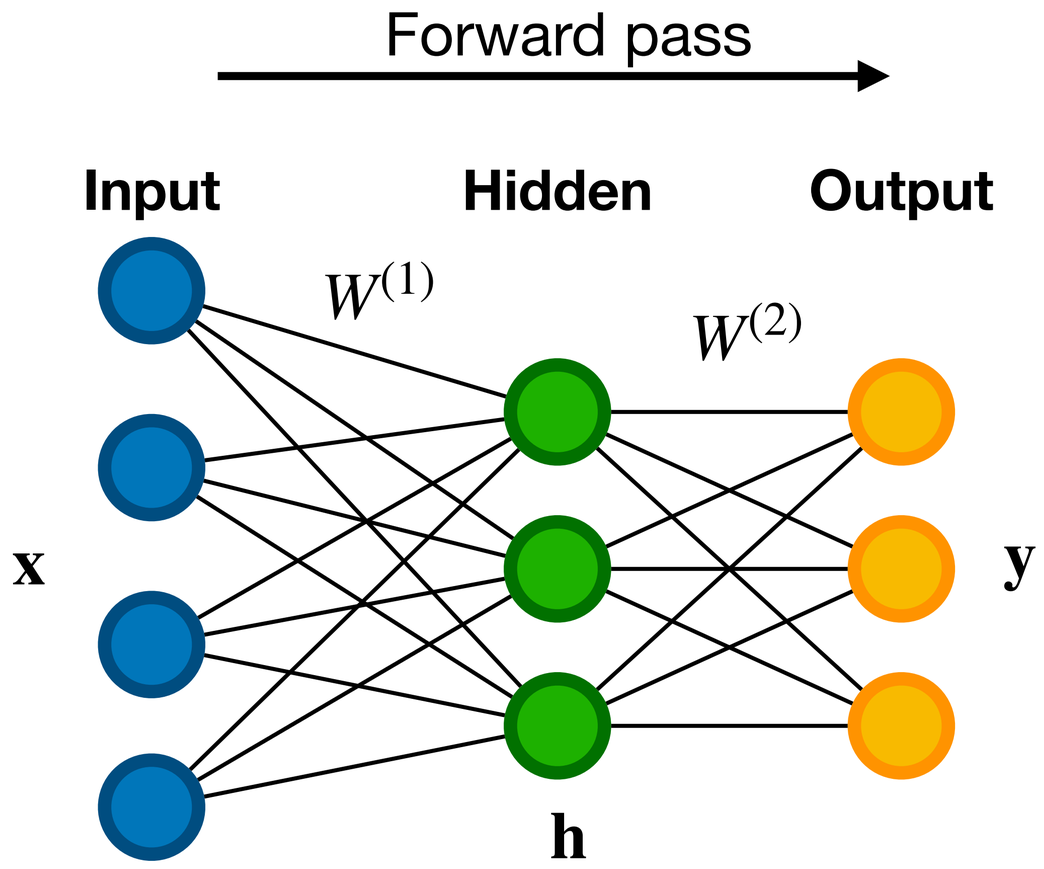


tanh

$$\tanh(x)$$



Computation in neural networks



Make predictions (decisions)
Plug in x to get y

Hidden layer neurons:

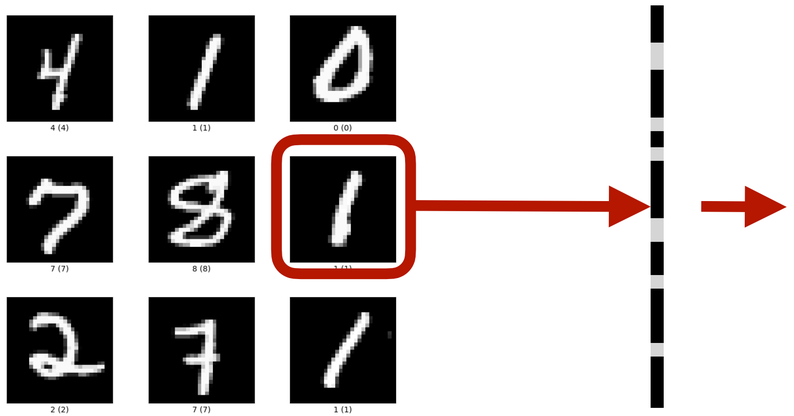
$$\mathbf{h} = f\left(W^{(1)}\mathbf{x} + b^{(1)}\right)$$

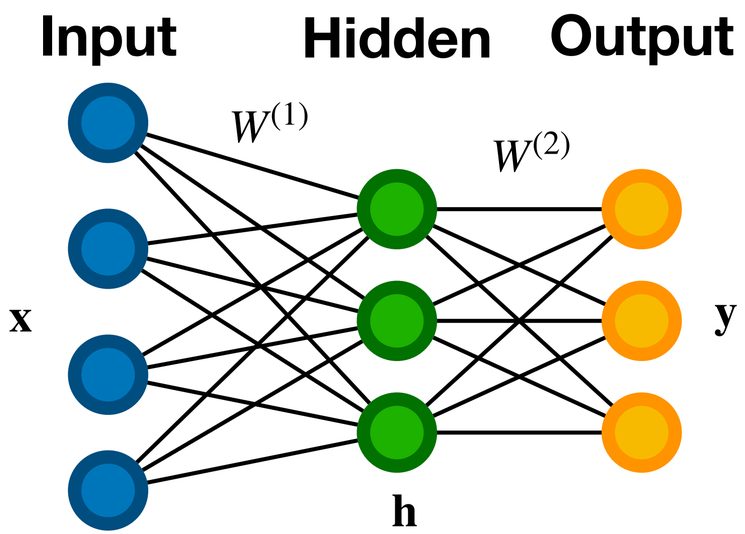
Output layer neurons:

$$\mathbf{y} = f\left(W^{(2)}\mathbf{h} + b^{(2)}\right)$$

Compute gradients of the cost (error or loss) w.r.t weights to find optimal values.

Image classification with neural networks





NN predicts class:
1 output neuron per class
(one hot encoding)

$$c = \arg\max_i \left(y_i\right)$$

Flatten to a 1D array/vector.

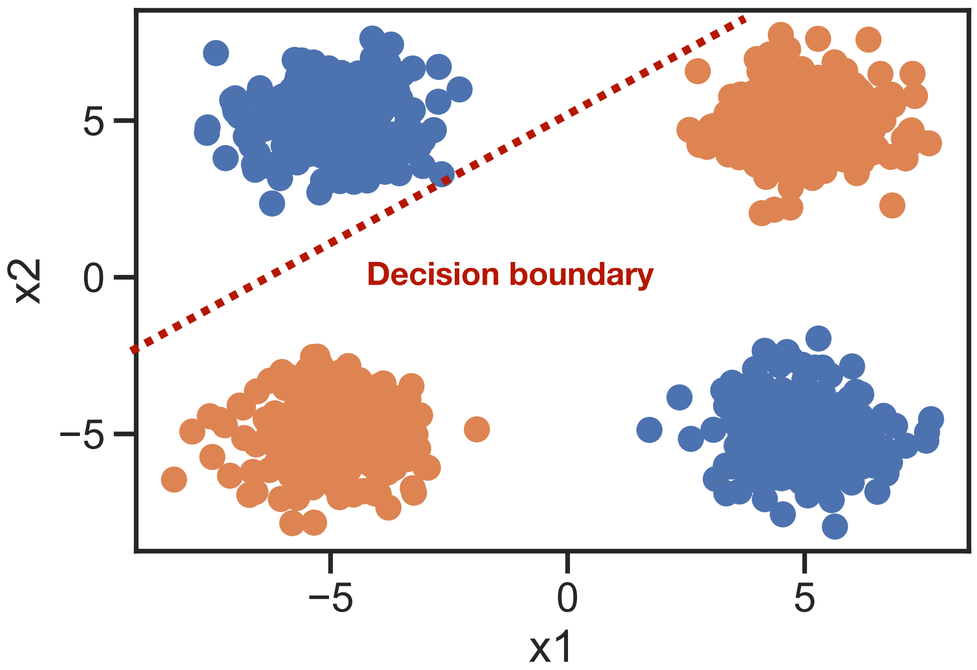
N x N image to a N^2 x 1 vector.

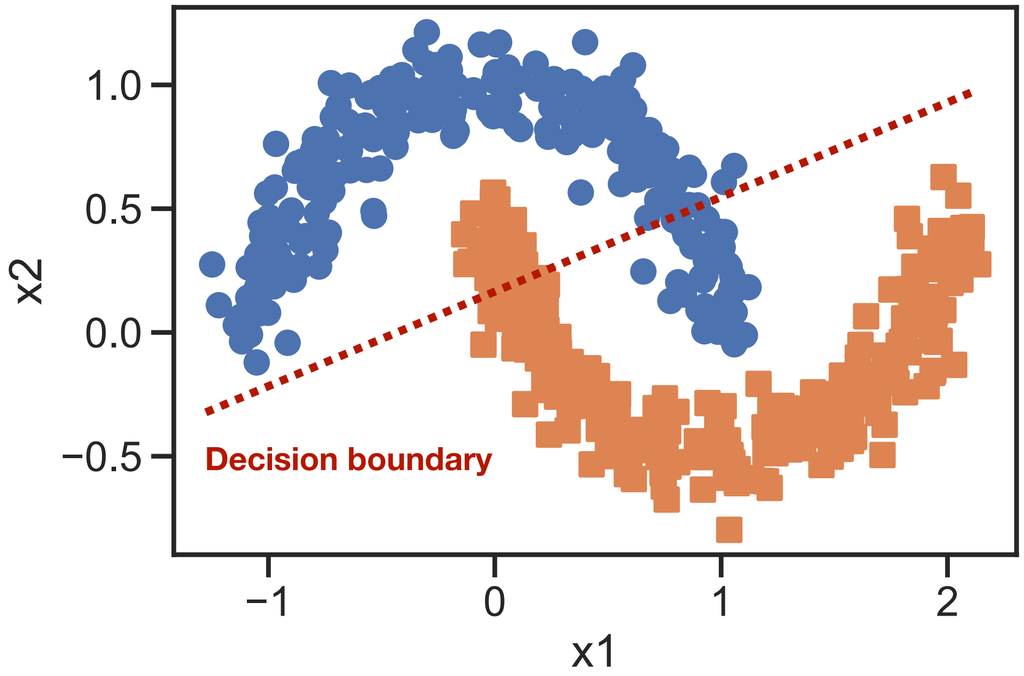
Some models convert $\mathbf{y}$ into a probability, e.g softmax

Cross entropy loss is suitable for multi class predictions.

Decision boundaries

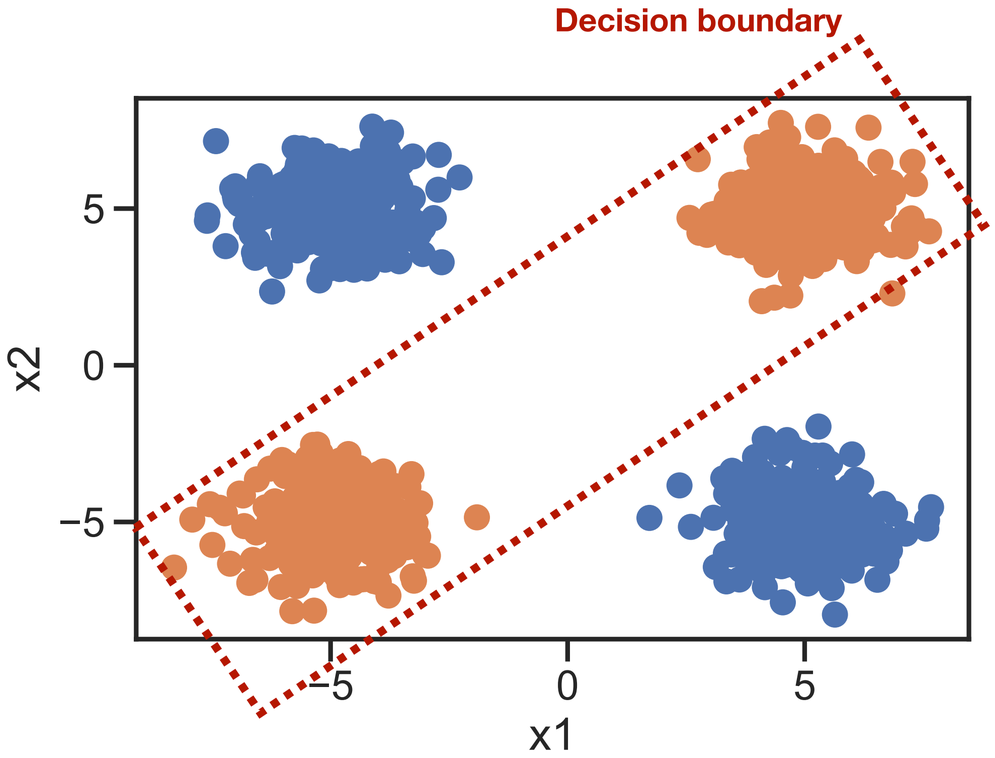
0 hidden layers - linear classifier

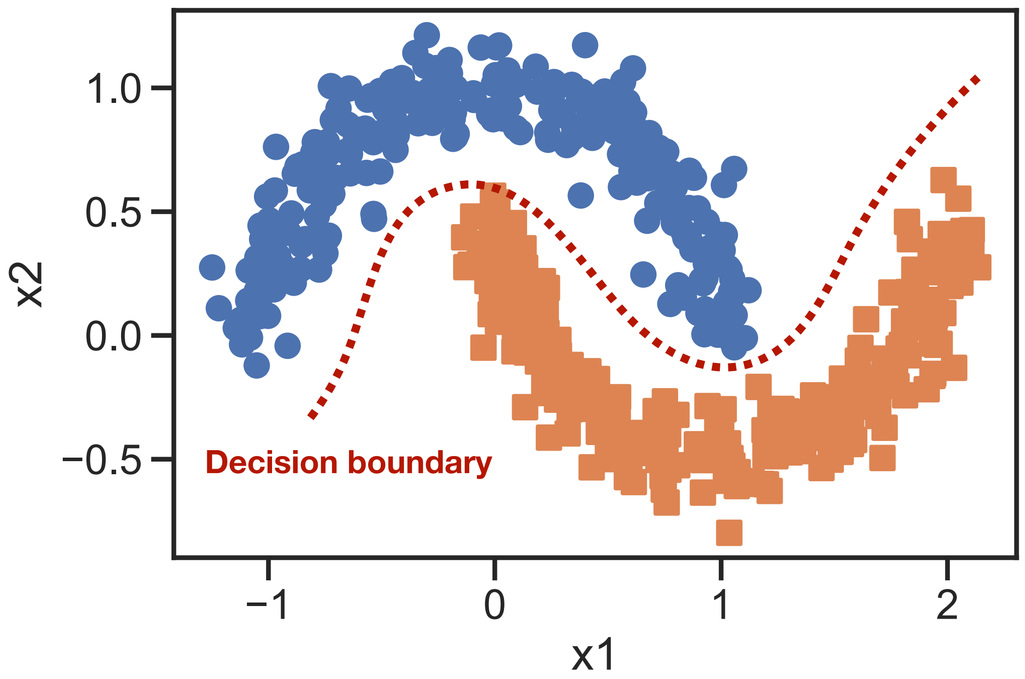




Decision boundaries

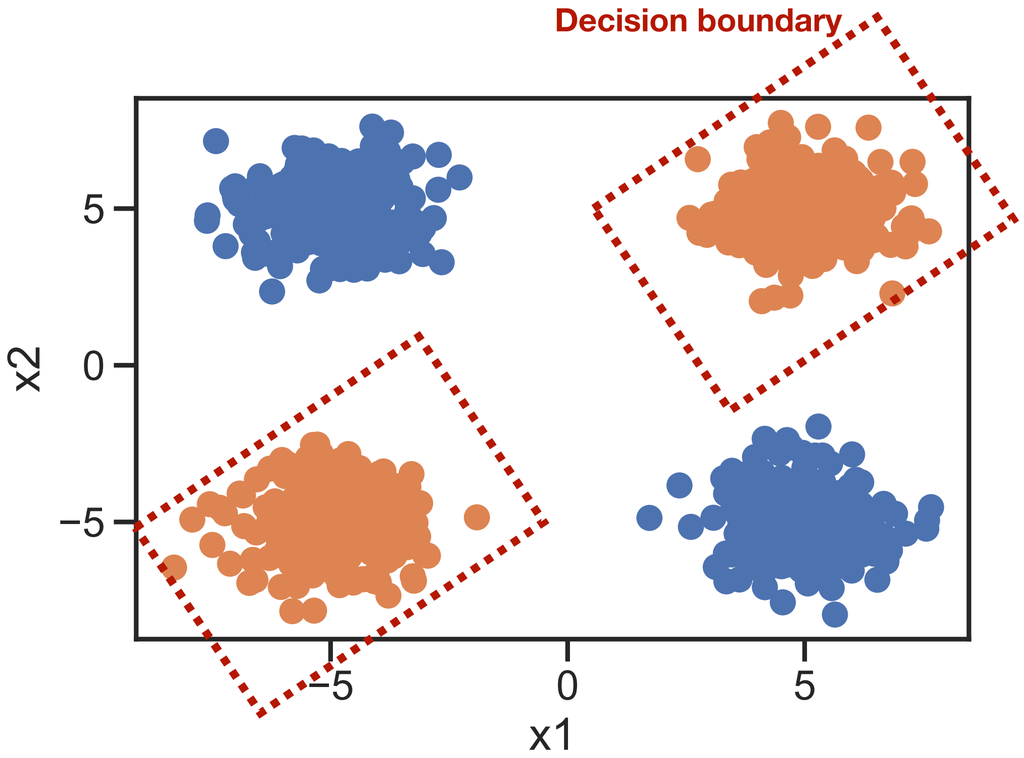
1 hidden layer - boundary of a **convex** region (open or closed)

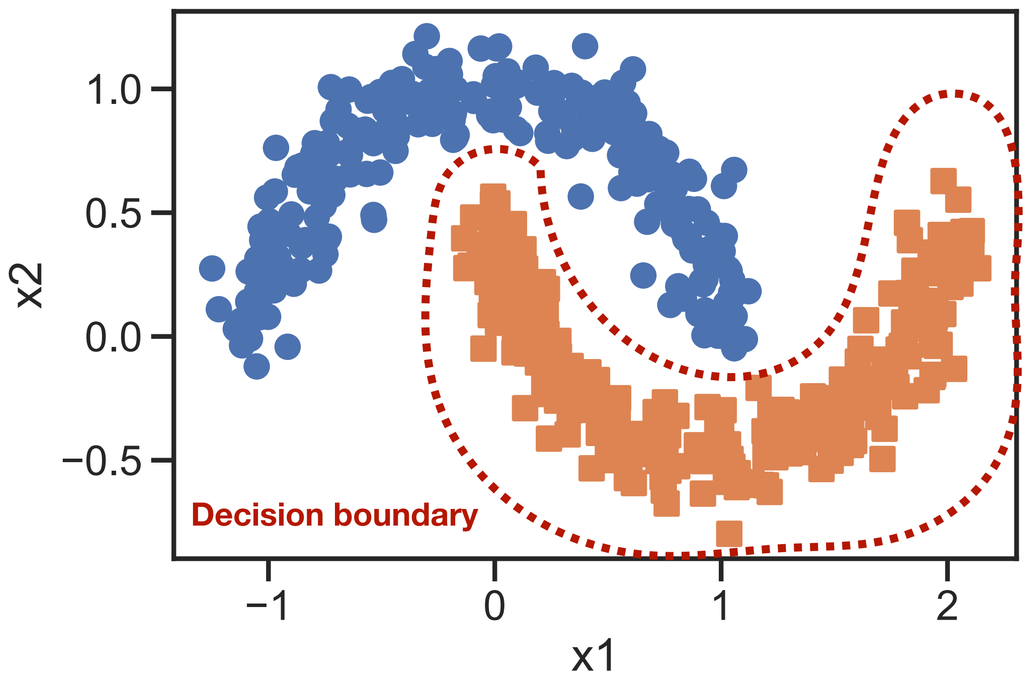




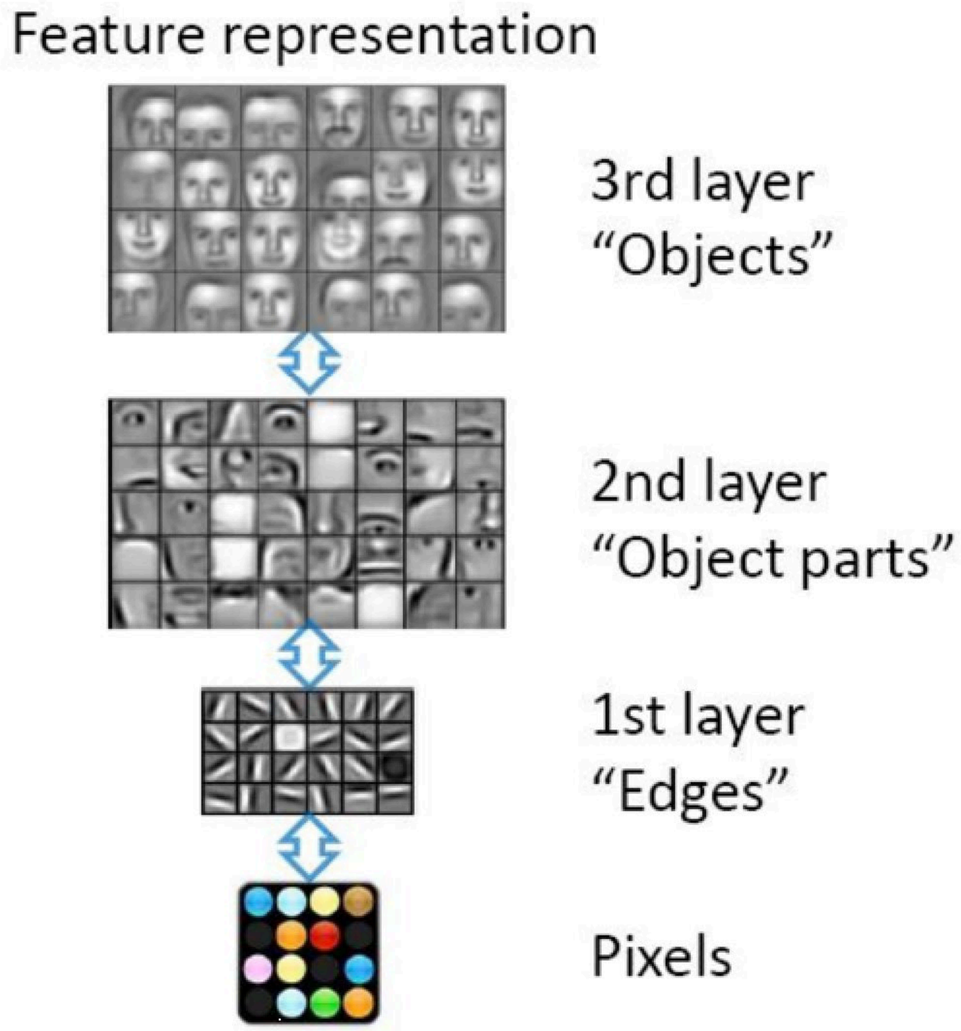
Decision boundaries

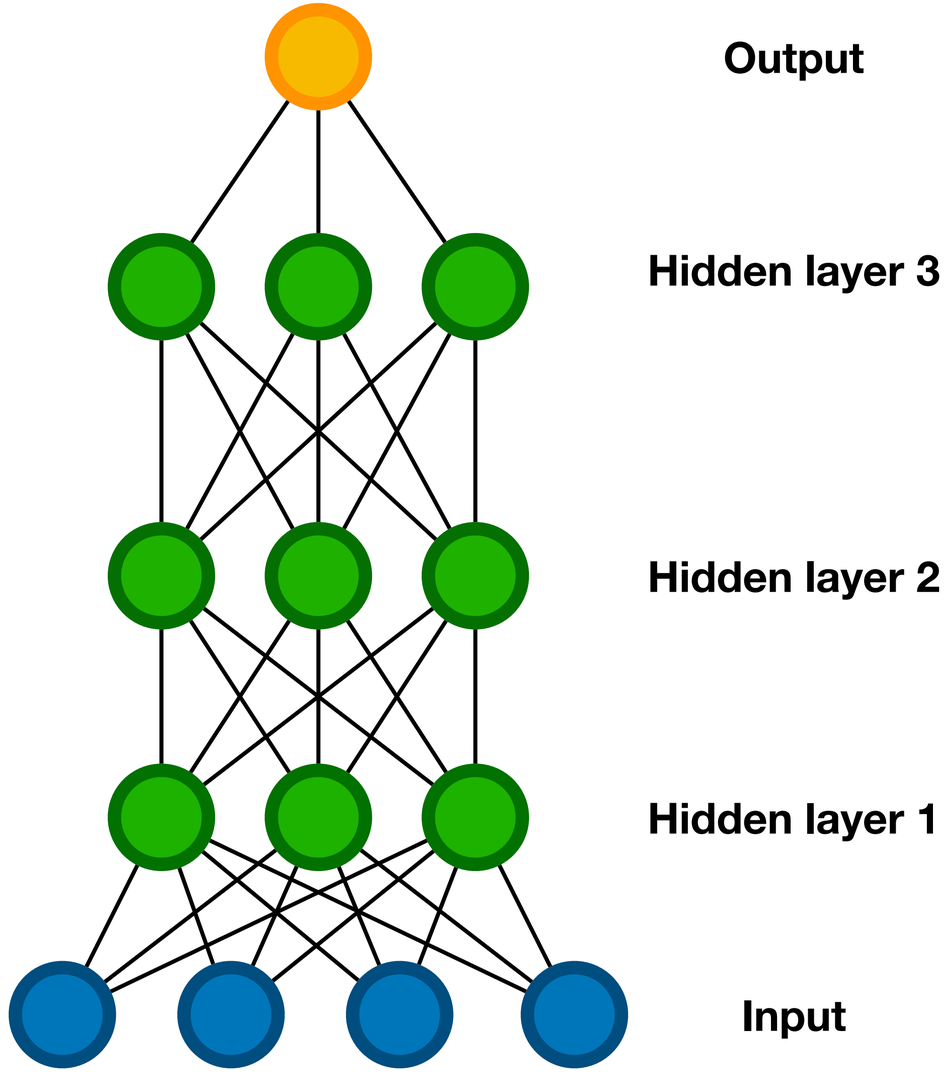
2 hidden layers - combination of **convex** regions





Different levels of abstraction





Example from Honglak Lee (NeurIPS 2010)

Data, model, metric for learning

1. Given training data

$$\{\mathbf{x}_n, \mathbf{y}_n\}_{n=1}^{N}$$

2. Choose each of these:

- Model / decision function

$$\hat{\mathbf{y}}_n = f_{\mathbf{w}}\left(\mathbf{x}_n\right)$$

- Loss function or metric

$$l(\mathbf{y}_n, \hat{\mathbf{y}}_n)$$

3. Define a goal:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{n=1}^{N} l(\mathbf{y}_n, \hat{\mathbf{y}}_n)$$

4. Optimise with gradient descent

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla l(\mathbf{y}_n, \hat{\mathbf{y}}_n)$$

Compute gradients using back propagation (using auto-diff)

Batches!

Predict for multiple inputs at once, update weights after each batch.

1 sample	N_B samples	All samples
Stochastic	Mini-Batch	Batch
More ‘noise’ Quick updates	Compromise	Smooth Slow updates

The convention used for PyTorch is number of samples as first dimension.

X.shape = (N_B = batch_size, N_{in} = input_features)

This means we have to modify our linear operation to make sure the matrix dimensions match!

$$\mathbf{y} = \mathbf{x}W^T + \mathbf{b}$$

(N_B x N_{out}) = (N_B x N_{in}) @ (N_{in} x N_{out})

Using torch.nn to create models

If we want to build our own models in PyTorch we can create classes inheriting from the **torch.nn.Module** class.
Internally this class can then hold various layers and operations.

```
class logistic_regression(nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        super().__init__()
        self.lin = nn.Linear( in_features, out_features, bias)
        self.act_func = nn.Sigmoid()

    def forward(self, x):
        return self.act_func(self.lin(x))
```

Example for a multi-layer network

```
class neural_network(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, bias=True):
        super().__init__()
        self.lin1 = nn.Linear( in_features, hidden_features, bias)
        self.act_func1 = nn.ReLU()
        self.lin2 = nn.Linear( hidden_features, out_features, bias)
        self.act_func2 = nn.Sigmoid()

    def forward(self, x):
        h = self.act_func1(self.lin1(x))
        return self.act_func2(self.lin2(h))
```

Simplify using nn.Sequential

If we are chaining together layers, we can use the built in Sequential class:

```
model = nn.Sequential(
    nn.Linear(in_features, hidden_features),
    nn.ReLU(),
    nn.Linear(hidden_features, out_features),
    nn.Sigmoid()
)
```

In each case we can use the model to predict using:

```
y_approx = model(x)
```

Fully connected layers

What if our network was bigger?

- Input image: 200 x 200 pixels, first hidden layer: 500 neurons

Q: How many weights from input to first hidden layer?

Q: Why might a FC layer be problematic for images?

Convolutional Neural Networks

Imagenet Large Scale Visual Recognition Challenge

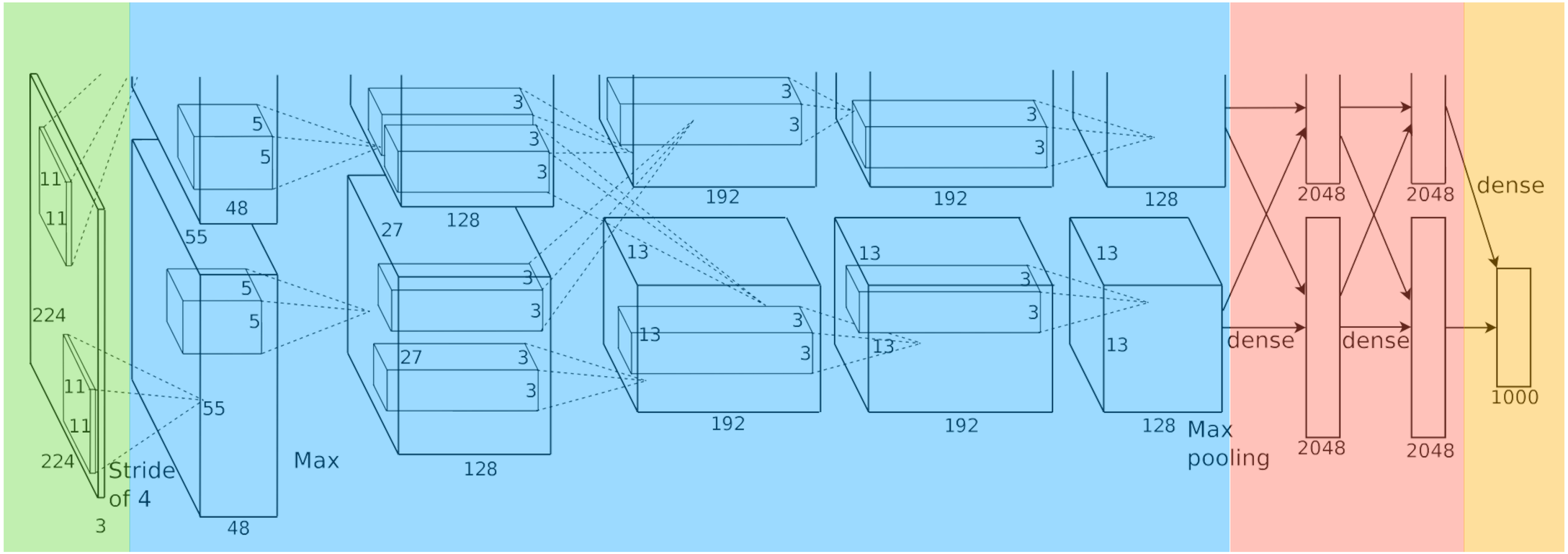
Alexnet

Input image (pixels)

5 Convolutional layers with max pooling

Fully connected layers

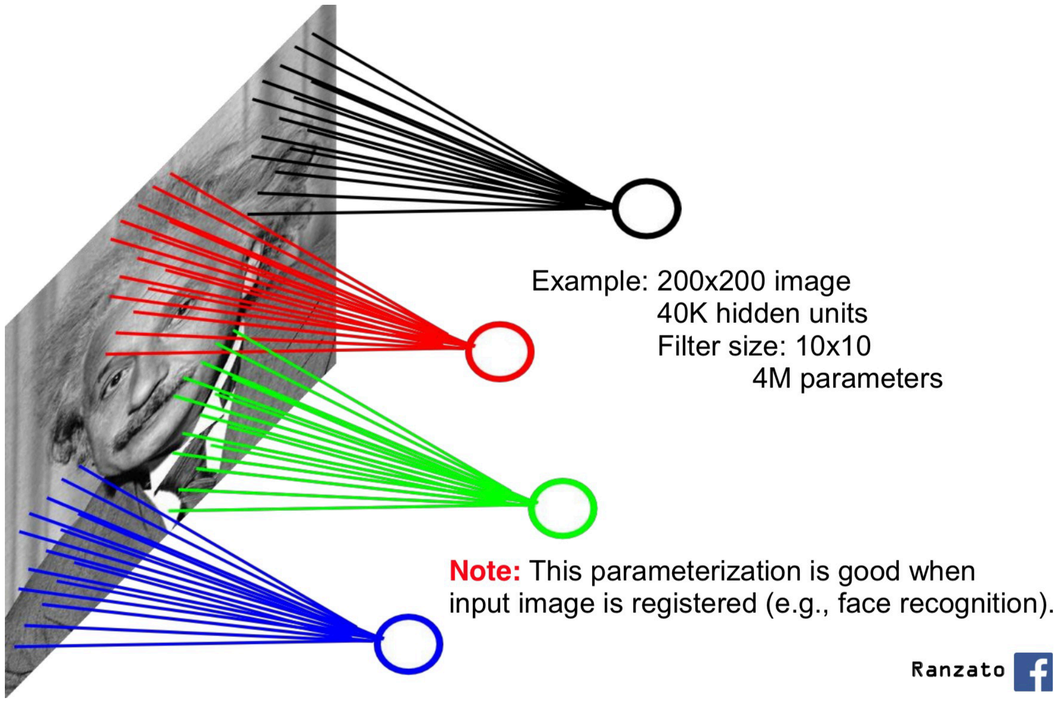
1000-way softmax

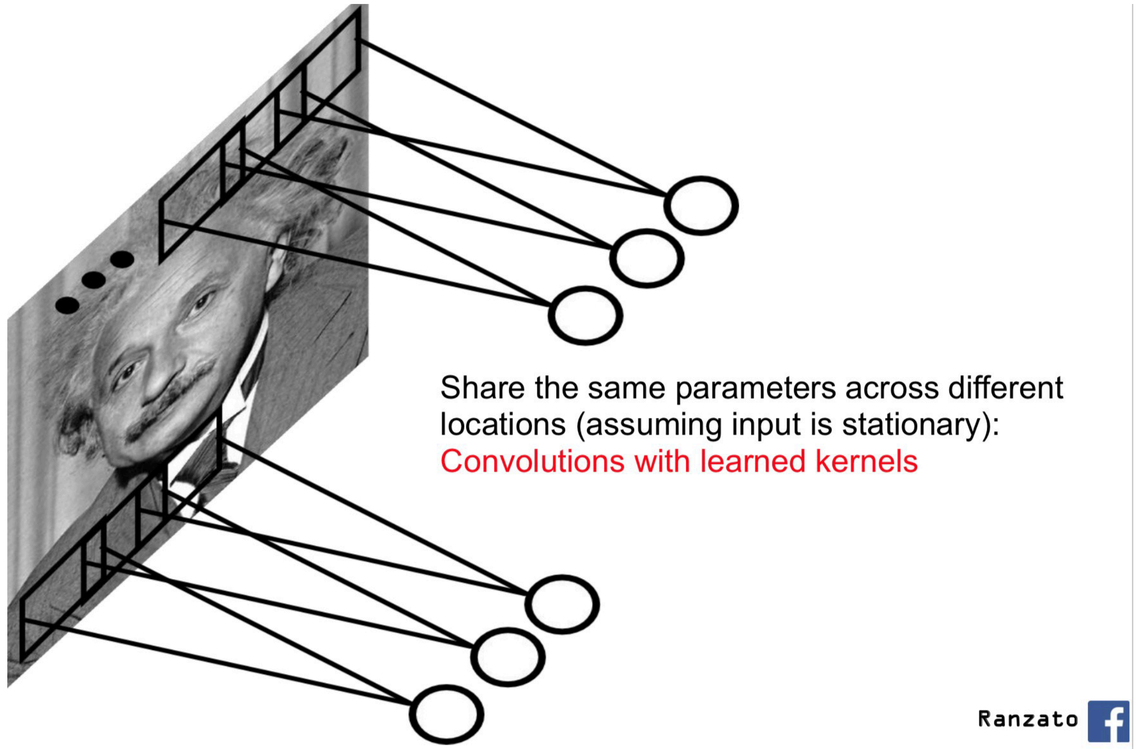


Convolutional neural network

Locally connected layers: look for local features in small regions of the image

Weight sharing: detect the same local features across the whole image





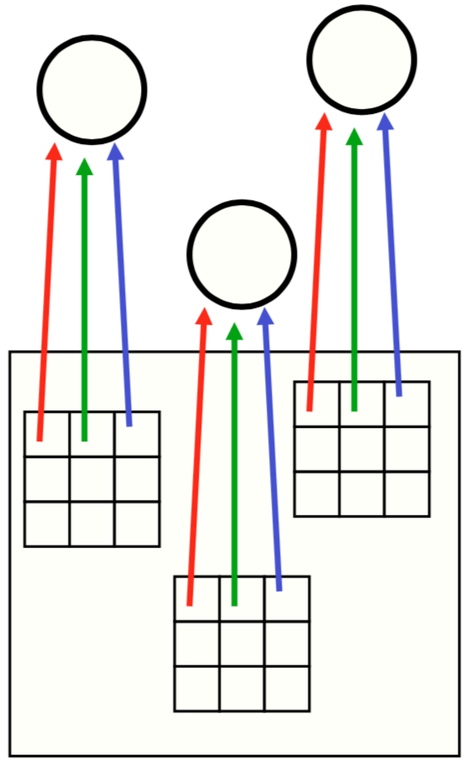
Weight sharing

The red connections all have the same weight.

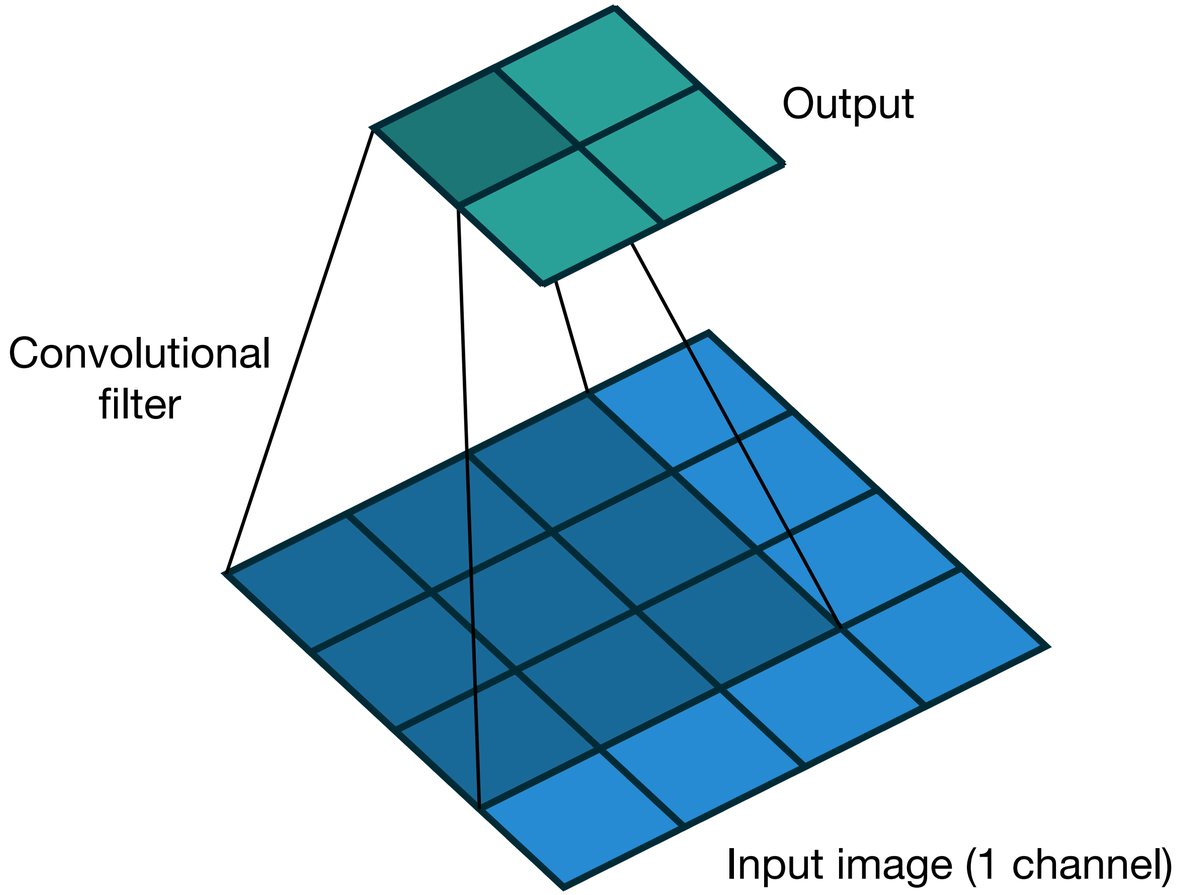
Each neuron in the higher layer detects the **same** feature, but in a **different** location in the lower layer.

Detection - the output (activation) is high if the feature is present.

Feature - something in the image (shape, blob, line) that we want to detect.



Convolutional filters



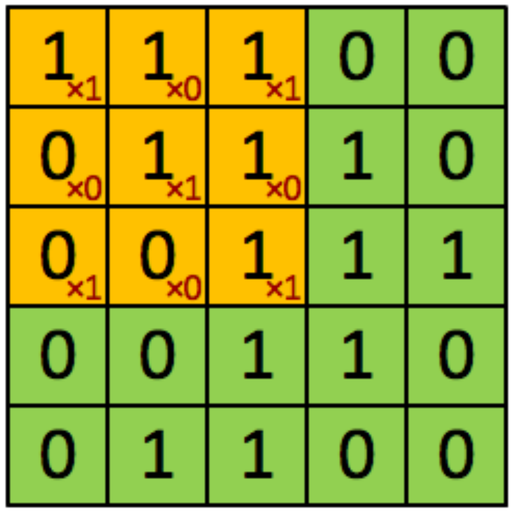
Convolution filter is applied as a moving window over the 2D input image.

$$y_{ij} = b + \sum_{k=0}^{F-1} \sum_{l=0}^{F-1} w_{kl}\, x_{i+k,j+k}$$

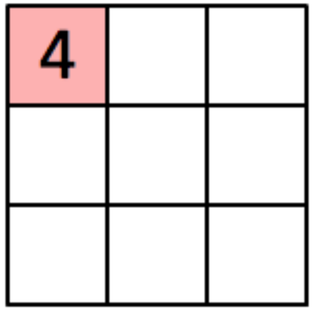
Vincent Dumoulin, Francesco Visin - A guide to convolution arithmetic for deep learning

Forward pass example

The filter/kernel (yellow) contains the trainable weights. Here filter size is 3 x 3.



Image



Convolved Feature

Exercise

Input image: 7 x 7

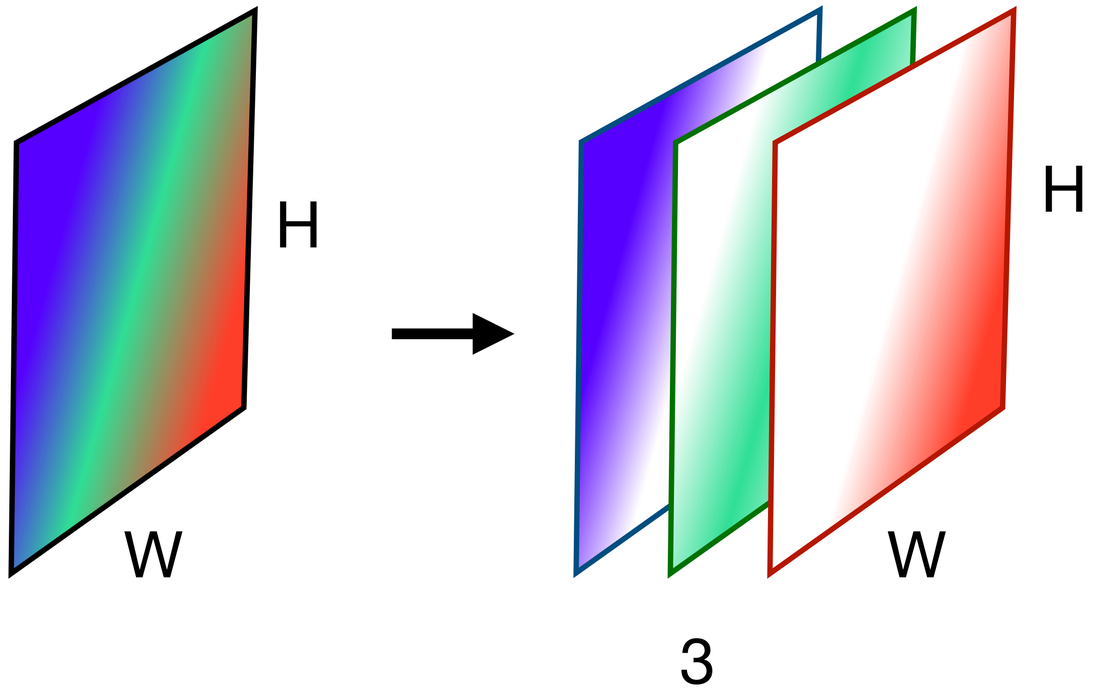
Filter size: 3 x 3

How many units in the output?

How many trainable weights?

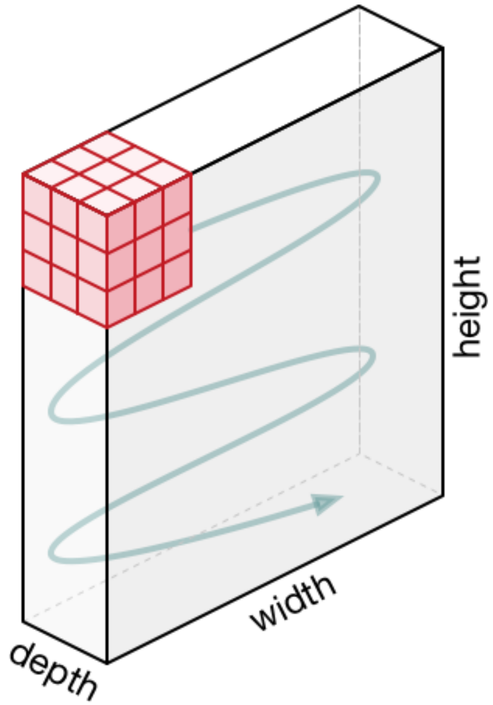
What are the values of the bottom output row?

Convolutions for colour images



2D image with 3 channels (RGB)

A tensor!: PyTorch uses (N_B,C,H,W) shape

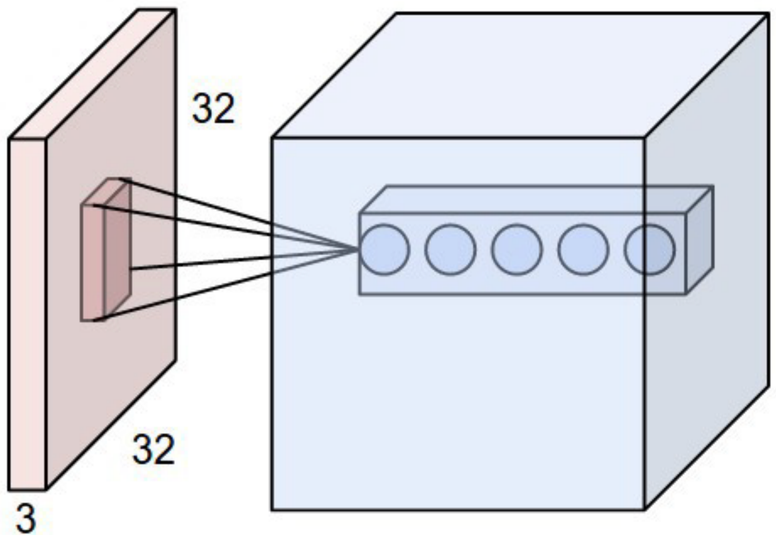


Kernel is also a 3D tensor. This example is **3** x 3 x 3

Number of **input channels** or **feature maps**

Detecting multiple features

Given a single filter map, how many features are being detected?



Have multiple filter maps to detect different features.

Example:

Input image size: 3 x 32 x 32

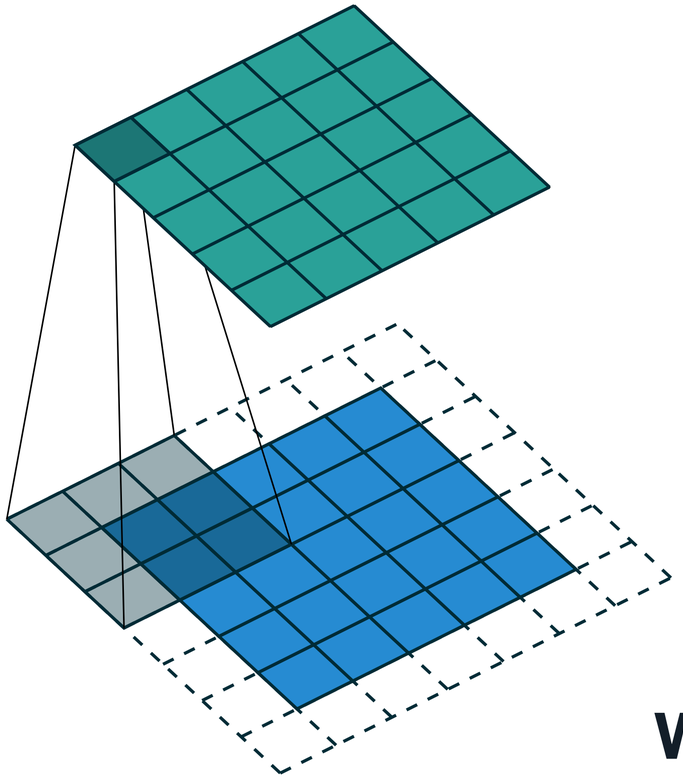
Convolutional kernel (4D): **3** x 3 x 3 x **5**

3 : number of **input channels** or **input feature maps**

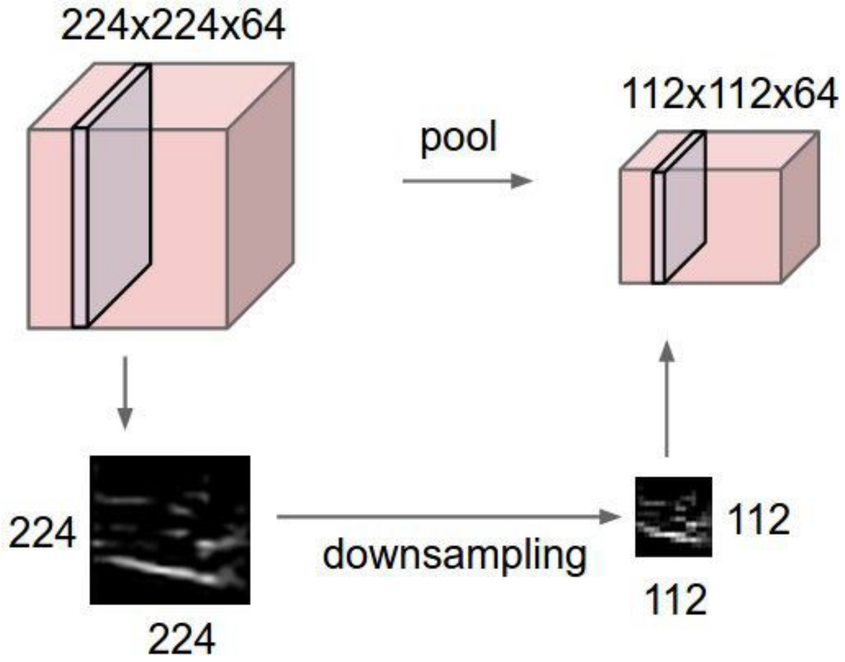
5 : number of **output channels** or **output feature maps**

Zero padding and pooling

Add zeros around the edge of the input image. Common padding size is (F-1)/2.

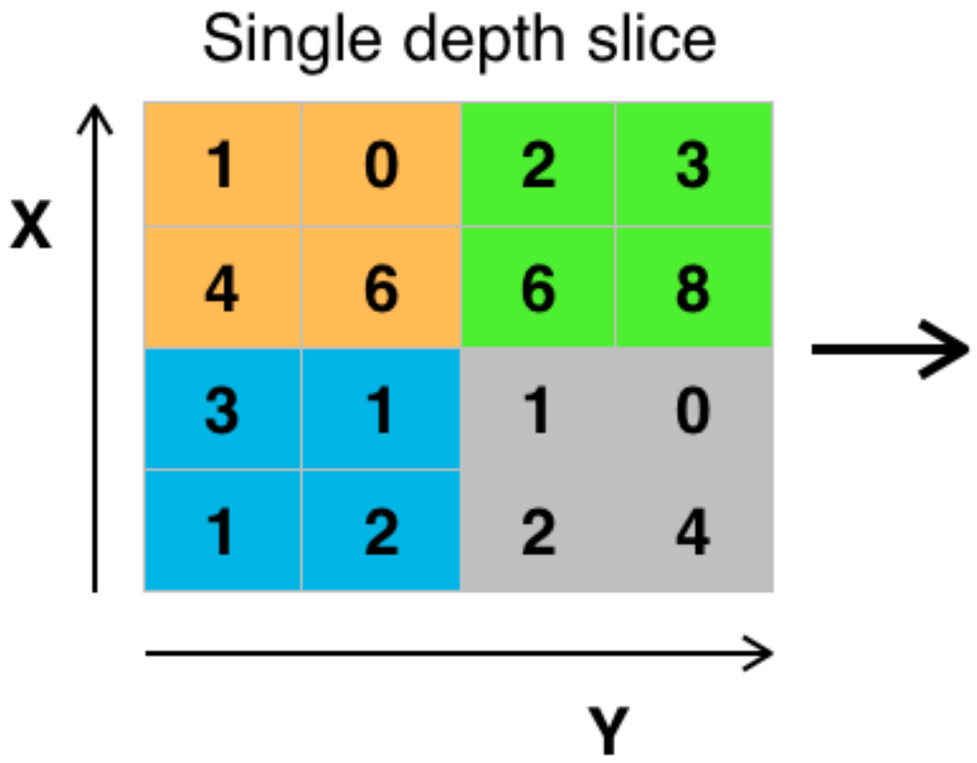


Downsample the feature maps by using either a max or average operation.



What are the benefits of these operations?

Max pool example

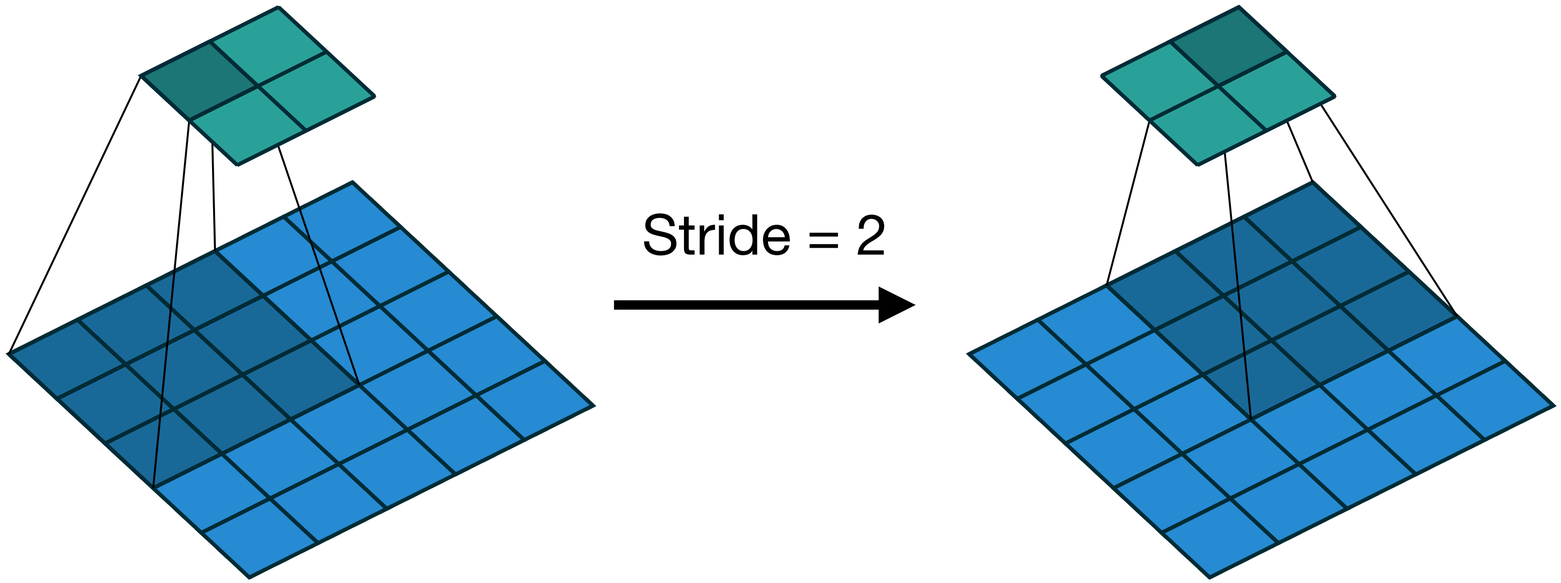


What is the output?

Image credit: Wikimedia (https://commons.wikimedia.org/wiki/File:Max_pooling.png)

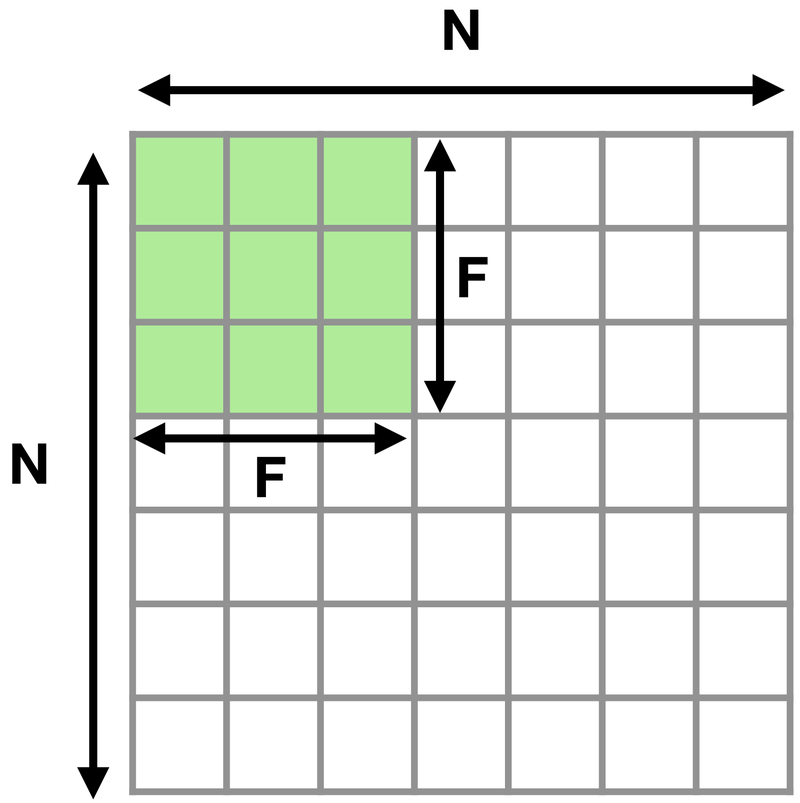
Strided convolutions

Shift the kernel by multiple pixels when computing the output feature:



Objective: to consolidate (summarise) information.

Size of output



Output size = (N-F)/stride + 1

Example

N = 7, F = 3

Stride = 1 → O = (7-3)/1 + 1 = 5

Stride = 2 or 3 ?

Exercise

Input volume = **3 x 32 x 32;** **10** filters, **5 x 5** shape with stride = **1**, pad = **2**

How many input and output channels?

What is the output volume size?

How many parameters are in this layer?

Take home messages

- Neurons perform a weighted sum of inputs followed by a non-linear function
- Neural networks connect many neurons together, the non-linearity allows for complex decision boundaries or functions to be learned.
- Deeper layers in a network detect smaller features of the input.
- Convolutional networks (generally) better for image data than fully connected layers.
- Convolutional layers share weights (filters) which detect whether features are within a local region.
- Zero padding, pooling and strides can be used to consolidate information.

Reading

Chapter 6 and 9 of Deep Learning by Goodfellow.

Available at https://www.deeplearningbook.org/