DEEP LEARNING

Lecture 13

TensorFlow

TensorFlow

- TensorFlow is a popular **deep learning framework** developed and maintained by **Google**.

- TensorFlow delegates **heavy computation** to compiled C and C++ code in the background.

- The Python interface for builds a **computational graph** and passes it to the back end for execution.

- This means that it is **not straightforward** to examine the intermediate model states during training.

- We will use a front-end for TensorFlow called `keras`.

Useful Snippets

A function to list the members of a module

```
def showModuleMembers(module):
    print(*[k for k in vars(module).keys() if not k.startswith("_")], sep="\n")

[1] showModuleMembers(tf.keras.activations)

deserialize, elu, exponential, get, hard_sigmoid, linear, relu, selu,
serialize, sigmoid, softmax, softplus, softsign, tanh
```

Get help on a **specific** module member e.g. `relu`

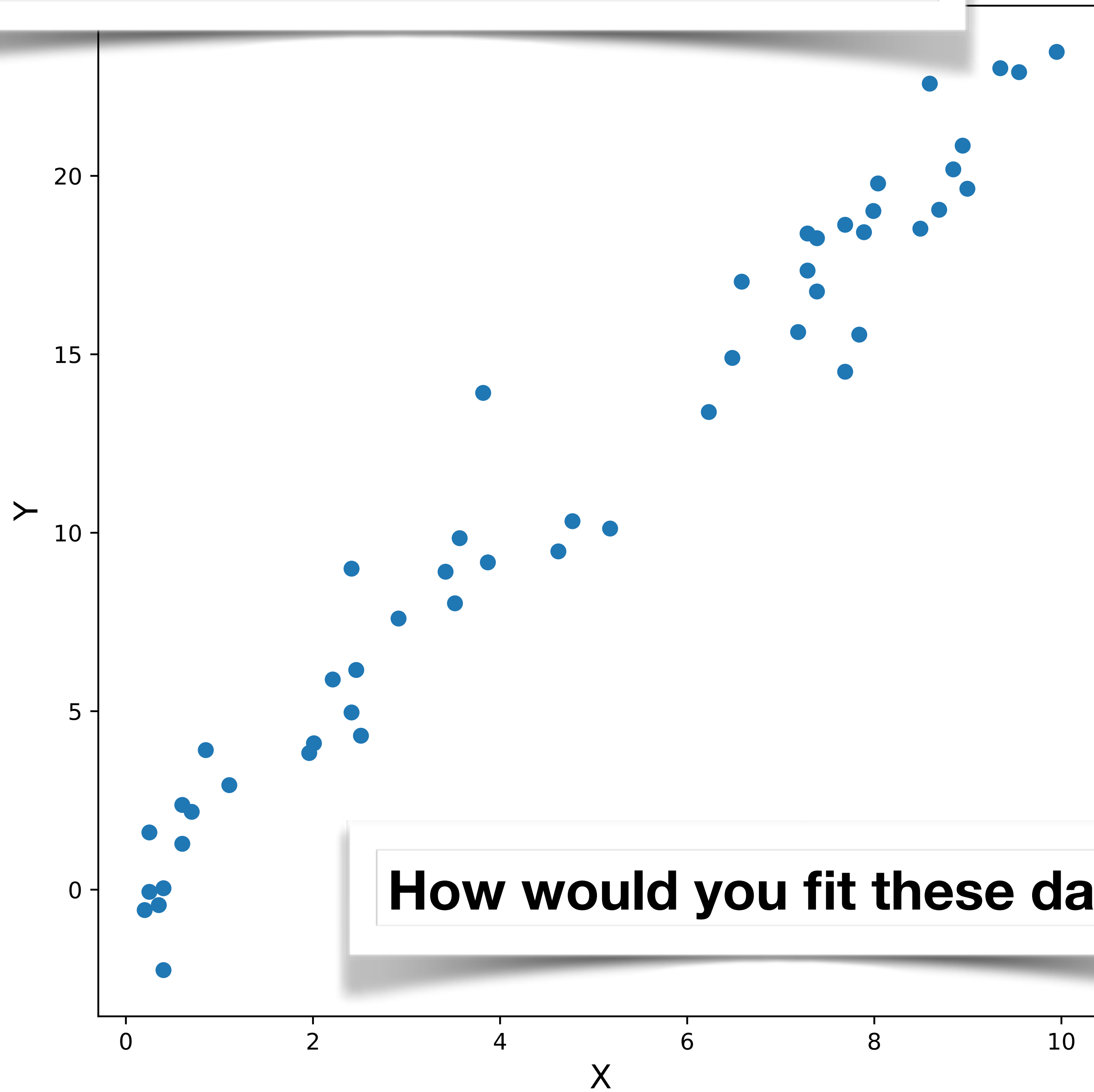
```
[1] tf.keras.activations.relu?

Signature: tf.keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0)
Docstring:
Rectified Linear Unit.

With default values, it returns element-wise `max(x, 0)`.

Otherwise, it follows:
`f(x) = max_value` for `x >= max_value`,
`f(x) = x` for `threshold <= x < max_value`,
⋮
```

Models of Data

- Scientists spend **a lot** of time **modeling data**.
- Models allow us to **abstract meaningful patterns**.
- Models allow us to **make predictions**.
- Deep learning **is not** artificial intelligence!
- Deep learning **is** a **very powerful** method for building **predictive** models of **complex data**.
- In the **ideal case**, deep learning models **generalize** well when applied to unseen data.
- Whether deep learning models are ***meaningful*** is up for debate!

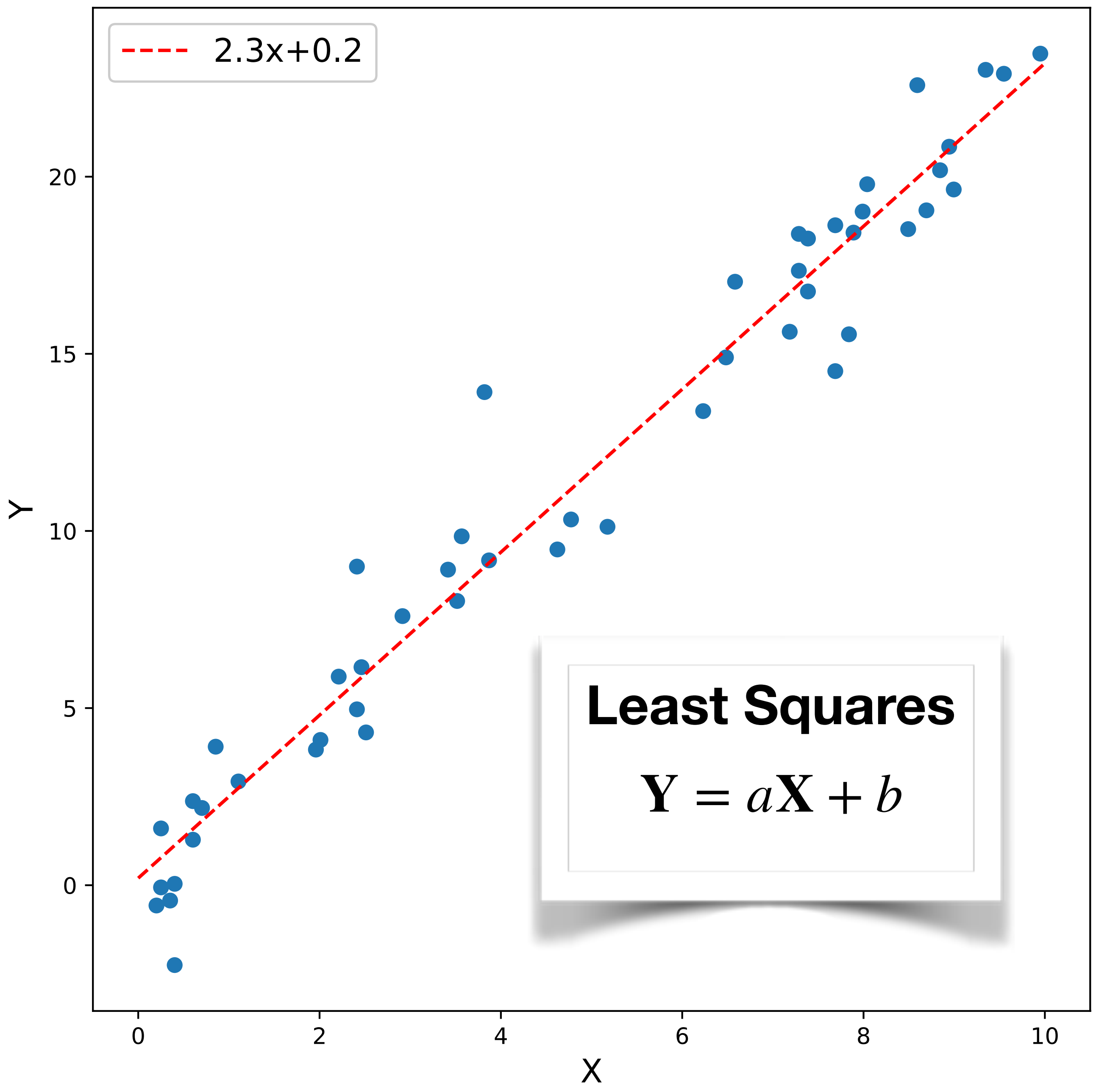
Linear Models

Fitting linear models to data is *very* fast!

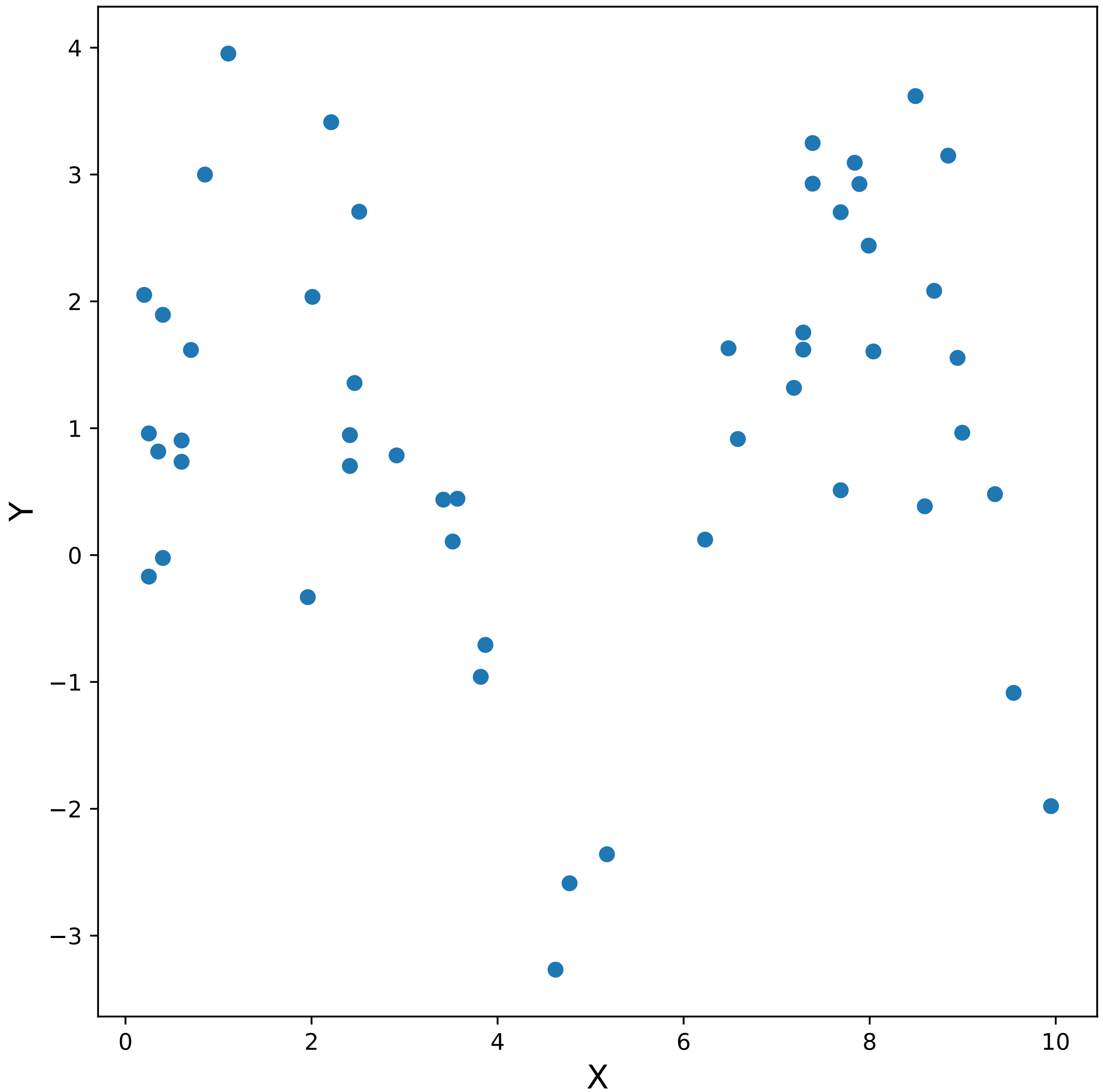


How would you fit these data?

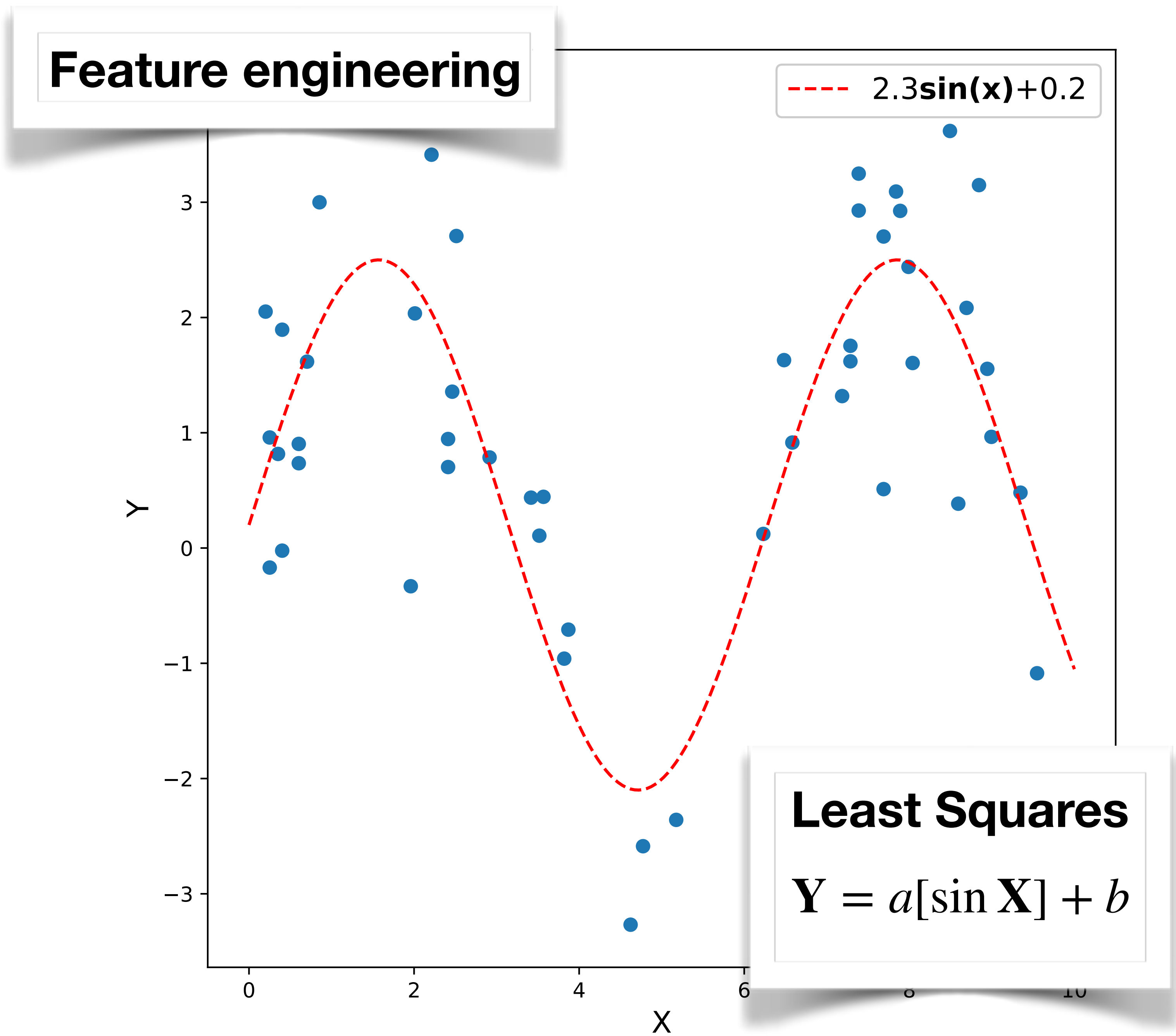
Linear Models



Nonlinear Models



~~No~~nlinear Models



Nonlinear Models

- **Linear models** are only appropriate for a **small subset** of real **physical processes**.

- When linear models are not applicable there are **several other options** we can consider.

Parametric Models

- If we have a **good intuition** for how the data should behave, we might be able to specify a **parametric model** for them.
- Parametric models assume a **finite parameter set** and are therefore bounded in their complexity.
- This can make them **easier to interpret**.
- Parametric models may arise from **theory**. For example, we expect that the **measured radioactivity** from a decaying radioisotope **declines exponentially**.
- **Empirically** derived functions can also be used. Recall that the **luminosity function** of galaxies in narrow bins of redshift can be modeled using the **Schechter function**.

Nonparametric models

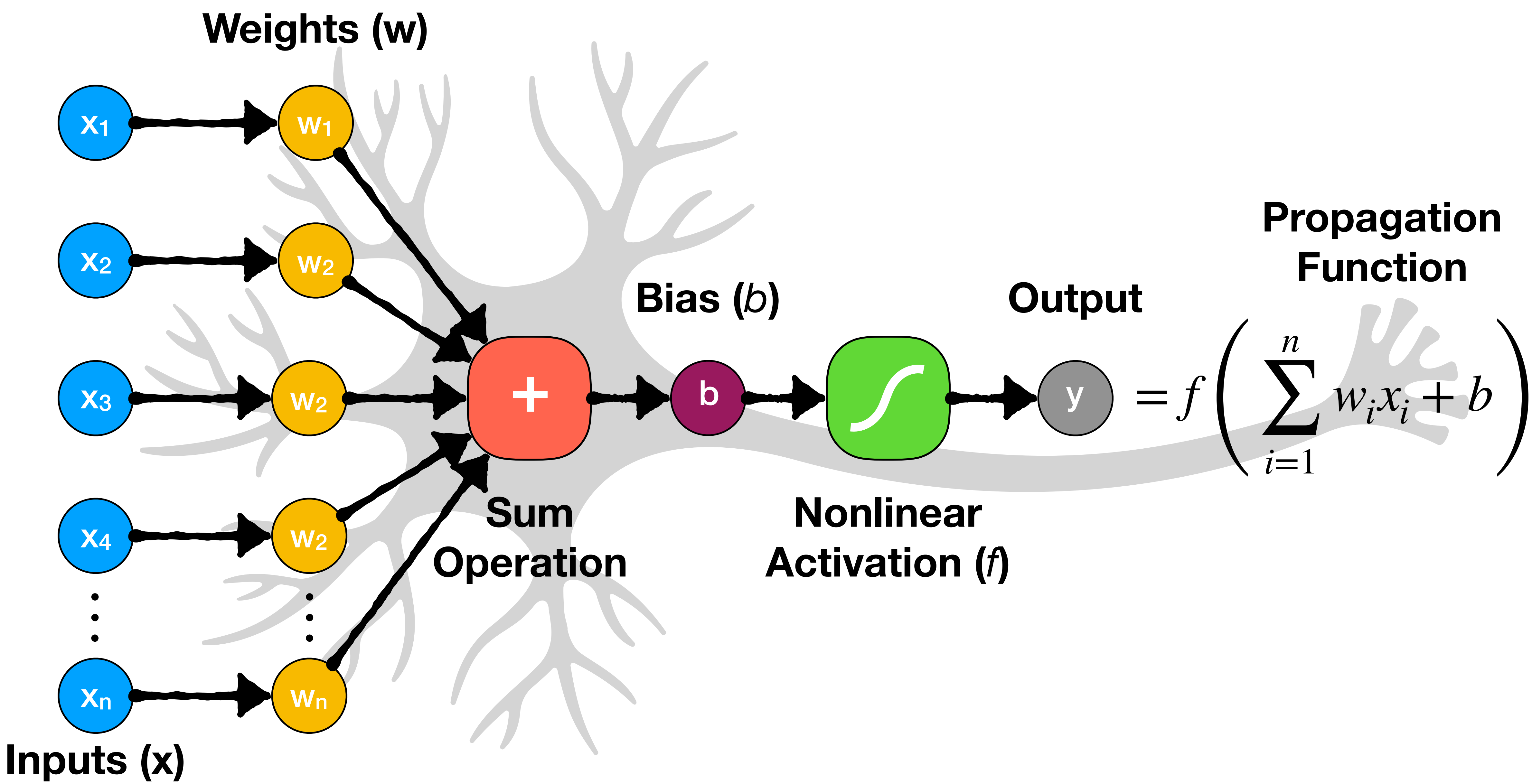
- Nonparametric models make **no assumptions** about the **number of model parameters**.
- In principle, this means that they can **represent** data of **arbitrary complexity**.
- However, **interpreting** nonparametric models can be **difficult**, particularly when the number of parameters becomes very large!
- Deep Learning is a variant of nonparametric data modeling that allows construction of complex models with **millions of parameters**!

Artificial
Neural Networks

Neural Networks

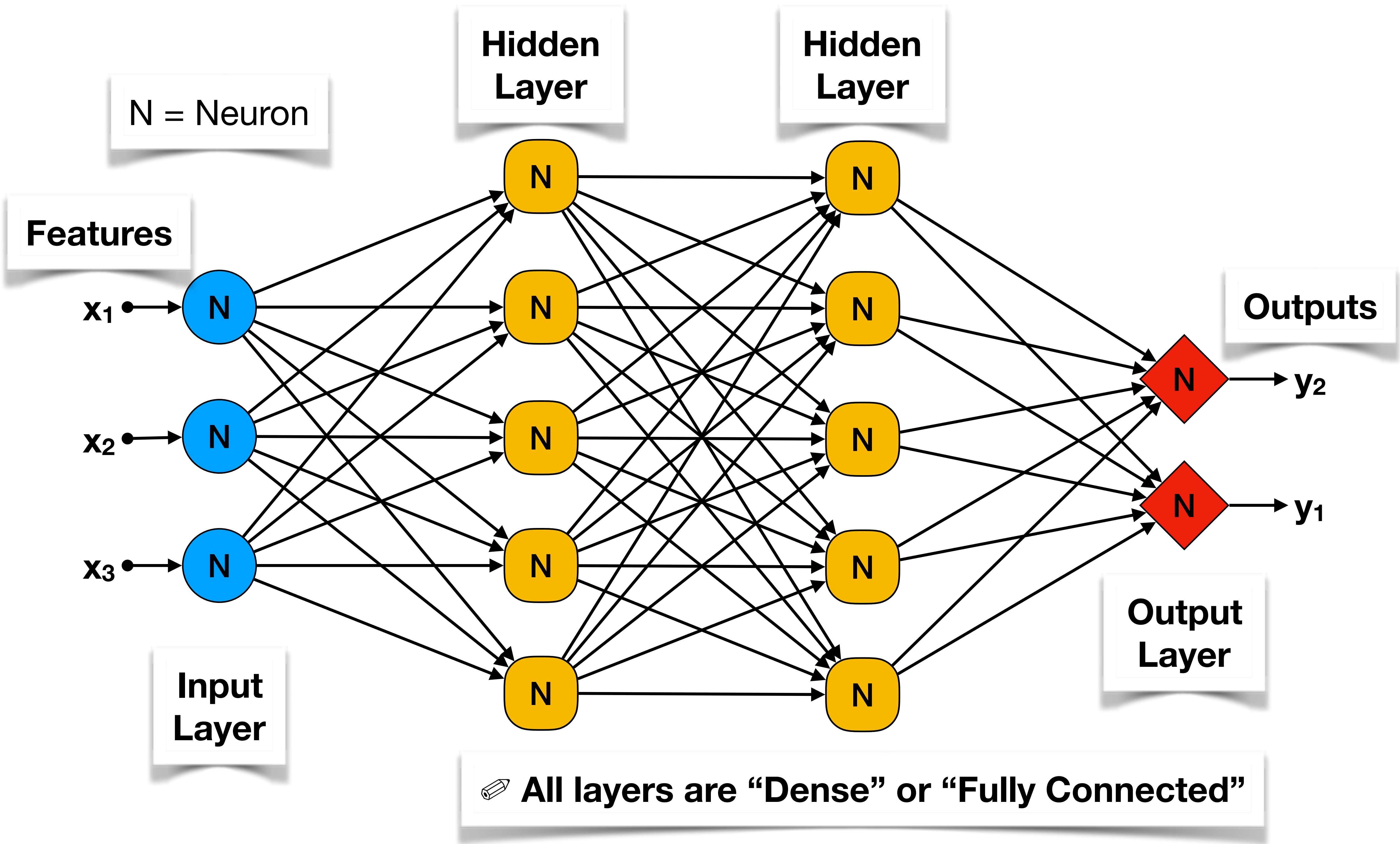
- Invented to facilitate early experiments in **artificial intelligence**.
- Designed to **mimic** the **connectivity** between neurons in the human brain.
- First implementations used dedicated **hardware**!
- Construct a complex model by combining many simple subunits - **artificial neurons**.

Artificial Neurons



✏ A *perceptron* is an artificial neuron with a *Heaviside* activation function.

Multilayer Perceptron



Activation Functions

- Activation functions inject **nonlinearity**, so we can model nonlinear processes and systems!

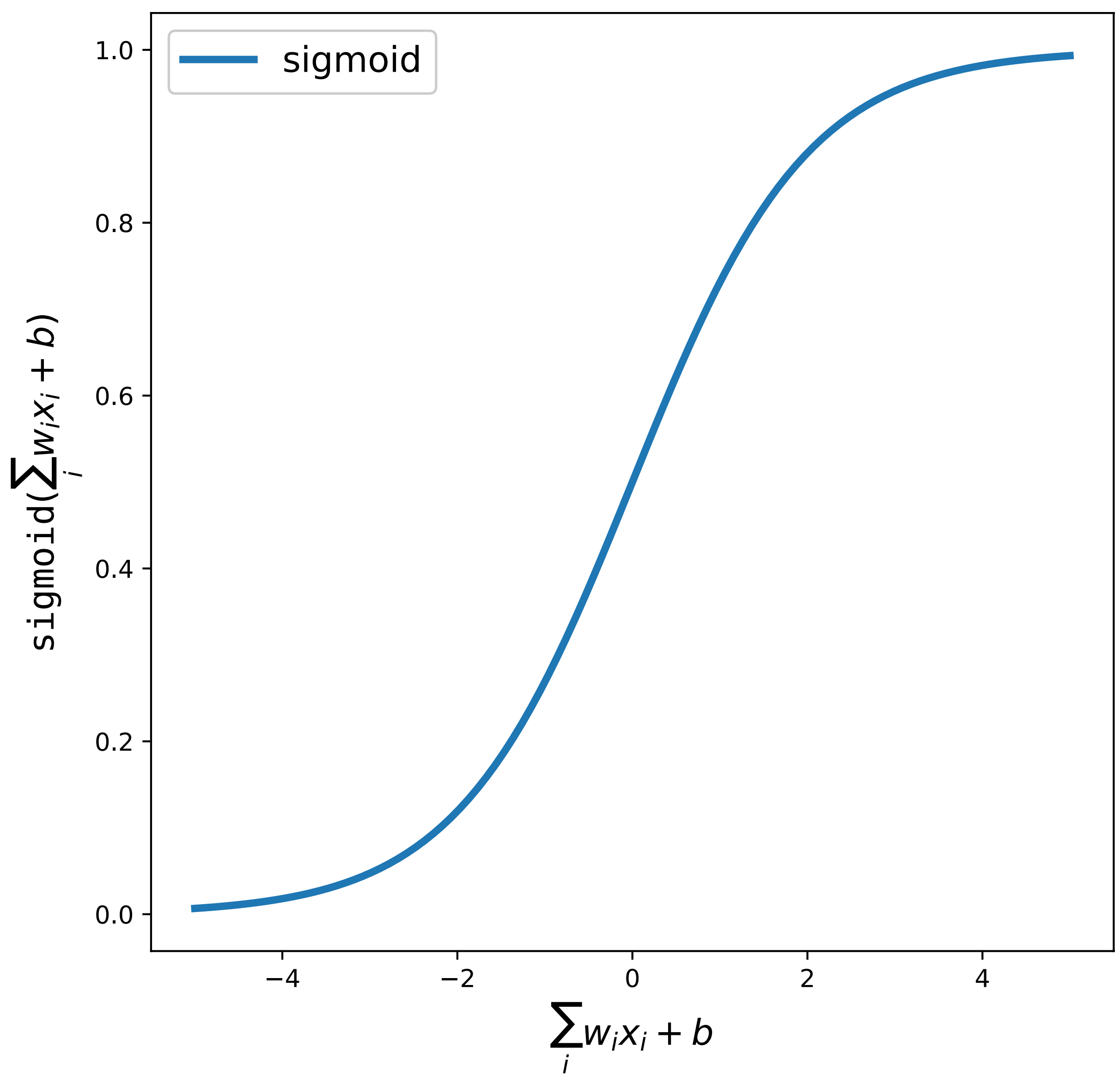
- The `tensorflow.keras` module implements several common activation functions.

```
[1] showModuleMembers(tf.keras.activations)

deserialize, elu, exponential, get, hard_sigmoid, linear, relu, selu,
serialize, sigmoid, softmax, softplus, softsign, tanh
```

- The **most commonly encountered** are `sigmoid`, `relu`, `tanh` and `softmax`.

Sigmoid

$$\mathbf{y} = \frac{e^{\mathbf{x}}}{e^{\mathbf{x}} + 1}$$



✏ **Also called the *logistic function***

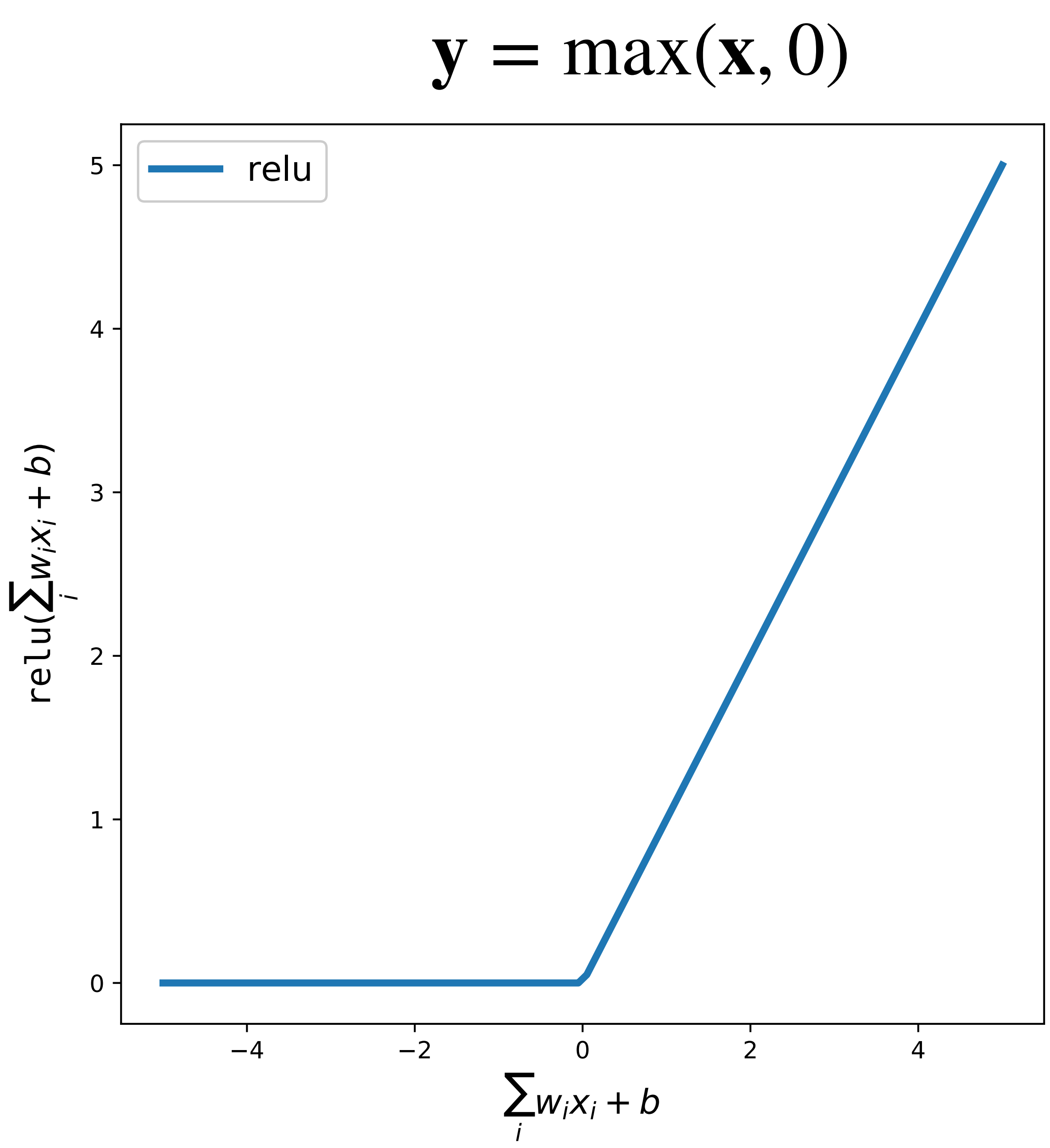
Used in logistic regression

Varies smoothly and suppresses extrema

Domain is bounded: [0, 1]

Differentiable

Relu



✏ **Short for** ***Rectified Linear Unit***

Domain is open [0, ∞]

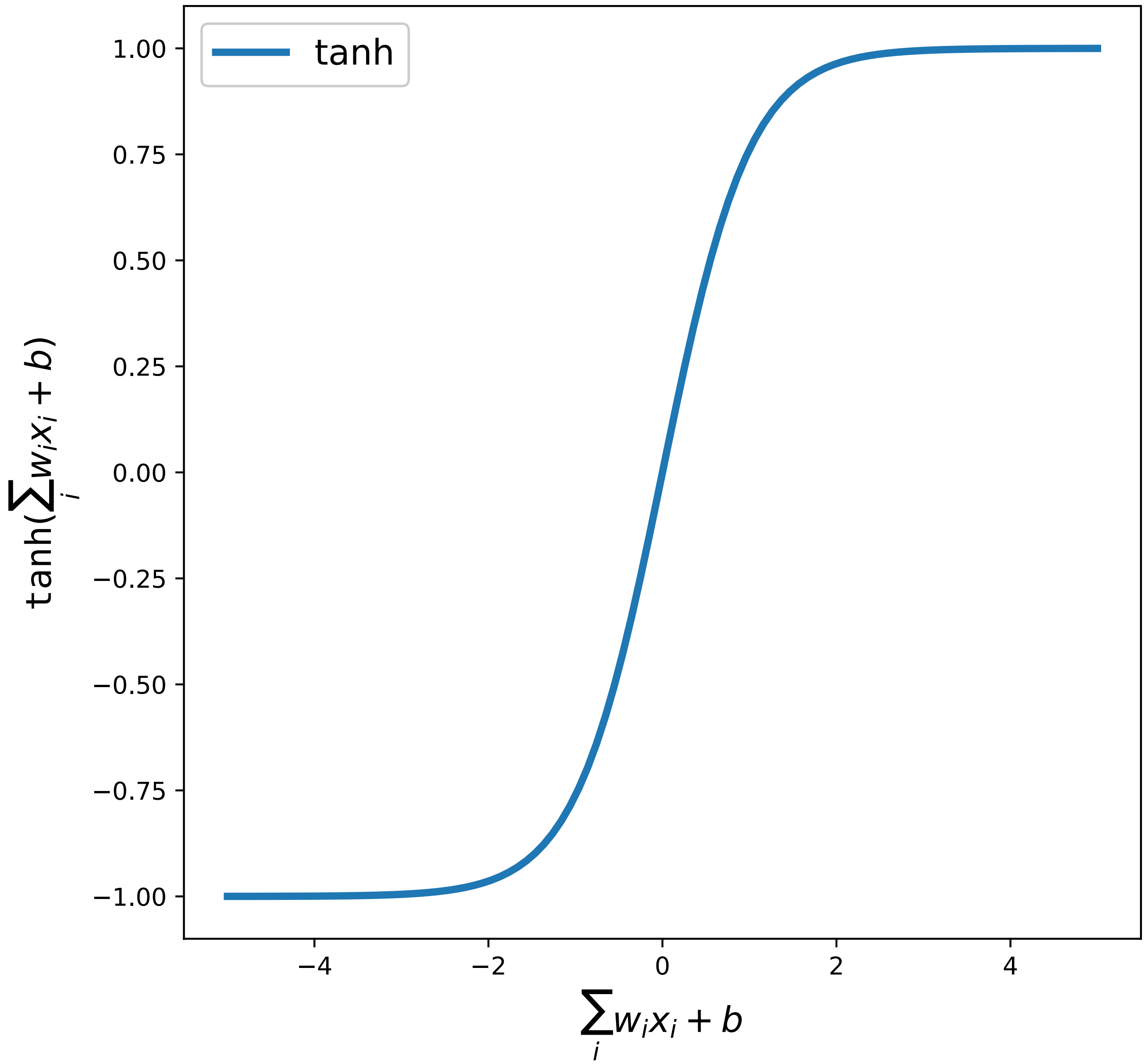
Optional parameters:
`alpha`
`max_value`
`threshold`

See the help text.

Not Differentiable

Tanh

$$\mathbf{y} = \tanh(\mathbf{x})$$

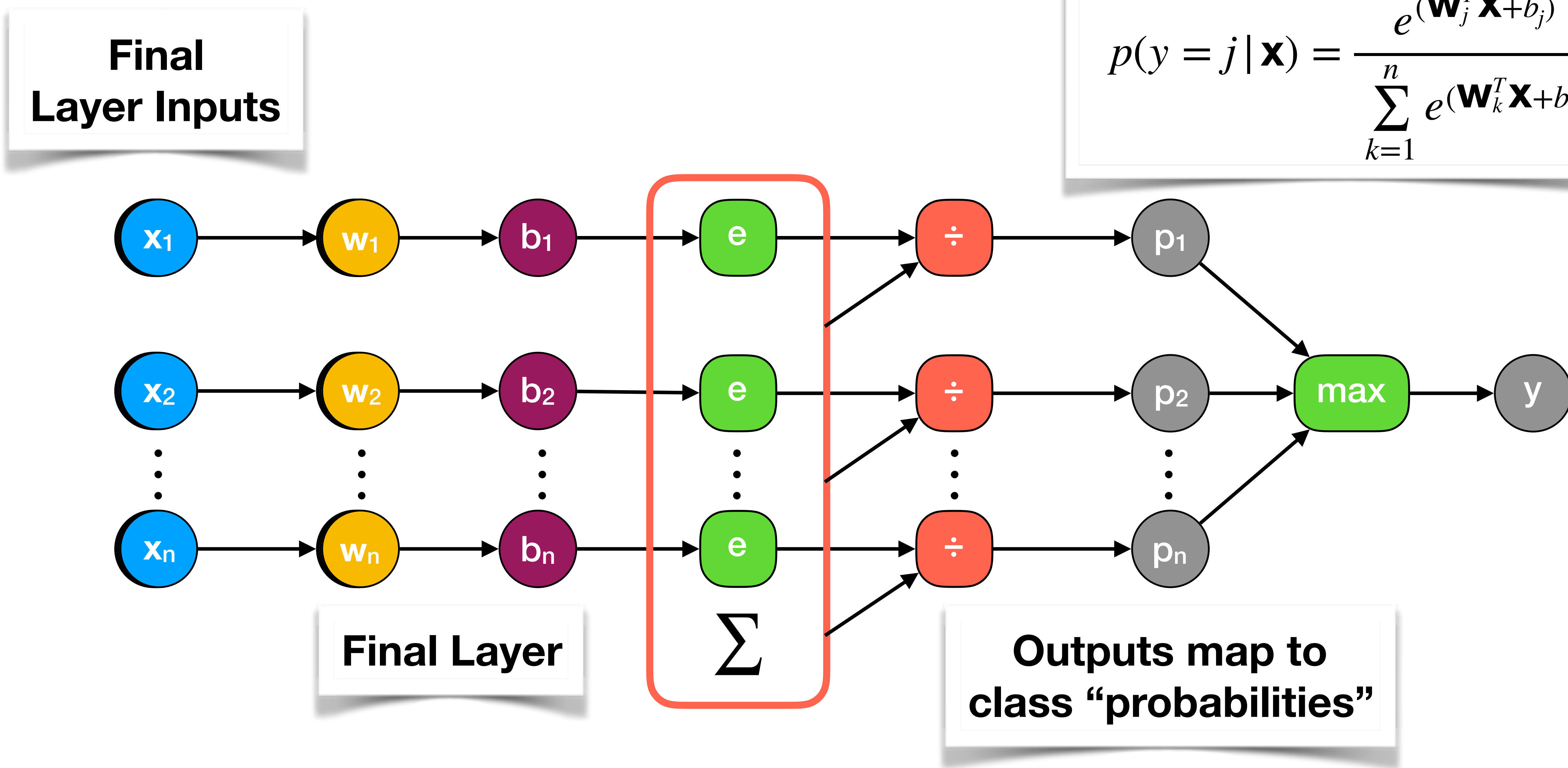


Differentiable

Domain is bounded [-1, 1] includes negative values

Second derivative only zero at origin.

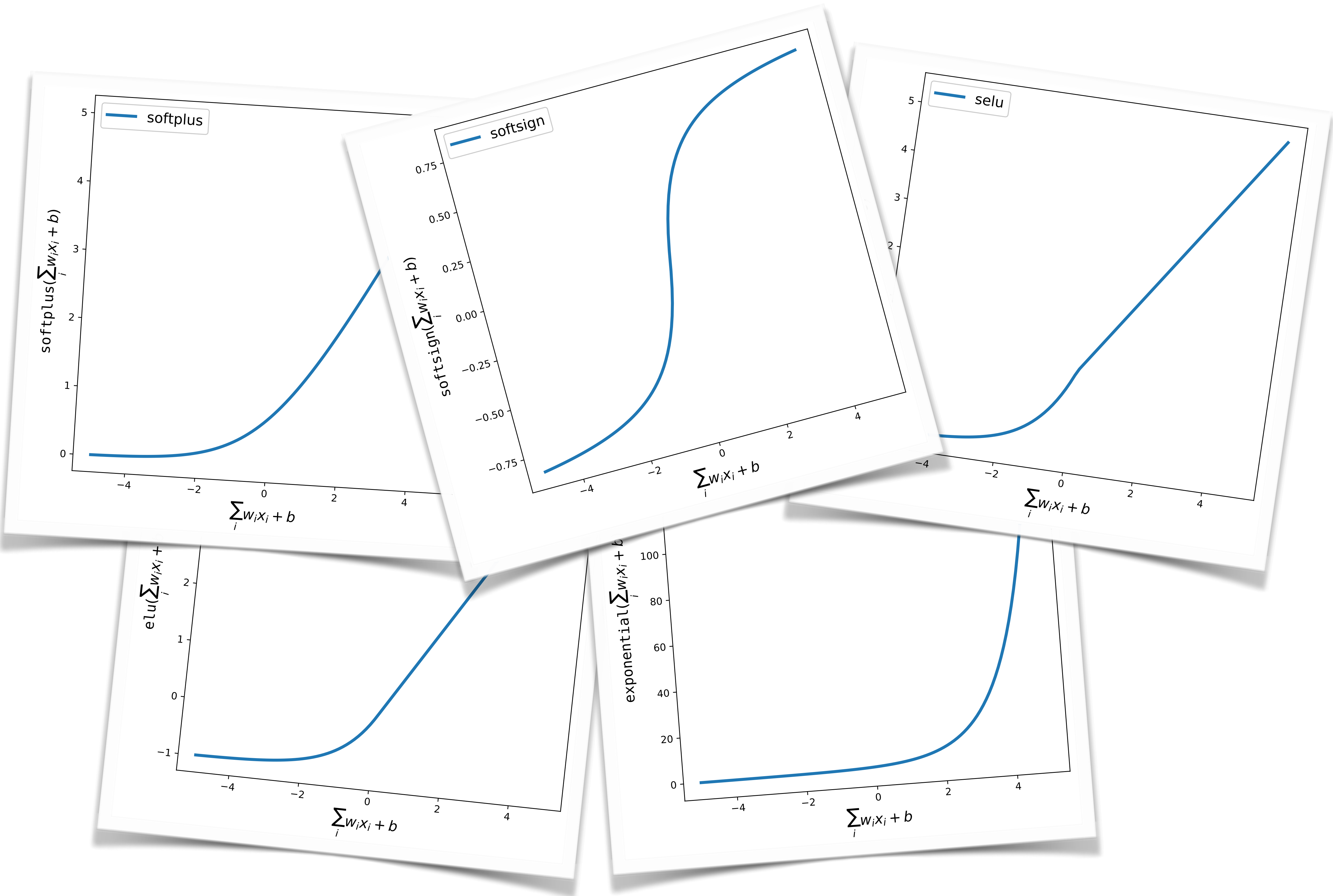
Softmax



$$p(y = j \mid \mathbf{x}) = \frac{e^{(\mathbf{w}_j^T \mathbf{x} + b_j)}}{\sum_{k=1}^{n} e^{(\mathbf{w}_k^T \mathbf{x} + b_k)}}$$

Softmax activation is used for **classification** with **more than two classes**.

Activation Zoo



Data Preparation

- Deep neural networks often perform better when the **input data** are **rescaled** to values in the range **[-1,1]**.

- Very large and/or very small input values can produce **numerical instabilities** when they are combined in an artificial neuron.

- This instability can be exacerbated when the combined inputs are fed to the **activation function**.

- This can **slow** or even prevent model **convergence** during training.

Training and Loss

- The process by which deep learning networks **improve** their **models** is called **training**.
- In training, **feature data** are presented to the network which then **predicts** a **label** based on those features.
- The prediction is used together with a **true label**, which was provided in conjunction with the feature data, to compute the **loss**.
- The loss is an **arbitrary function** that should yield **smaller** values when the prediction and the true label **agree** closely and **larger** values **otherwise**.
- Approximately, the **goal of training** is to adjust the network weights such as to **minimize the loss** function.

Training and Loss

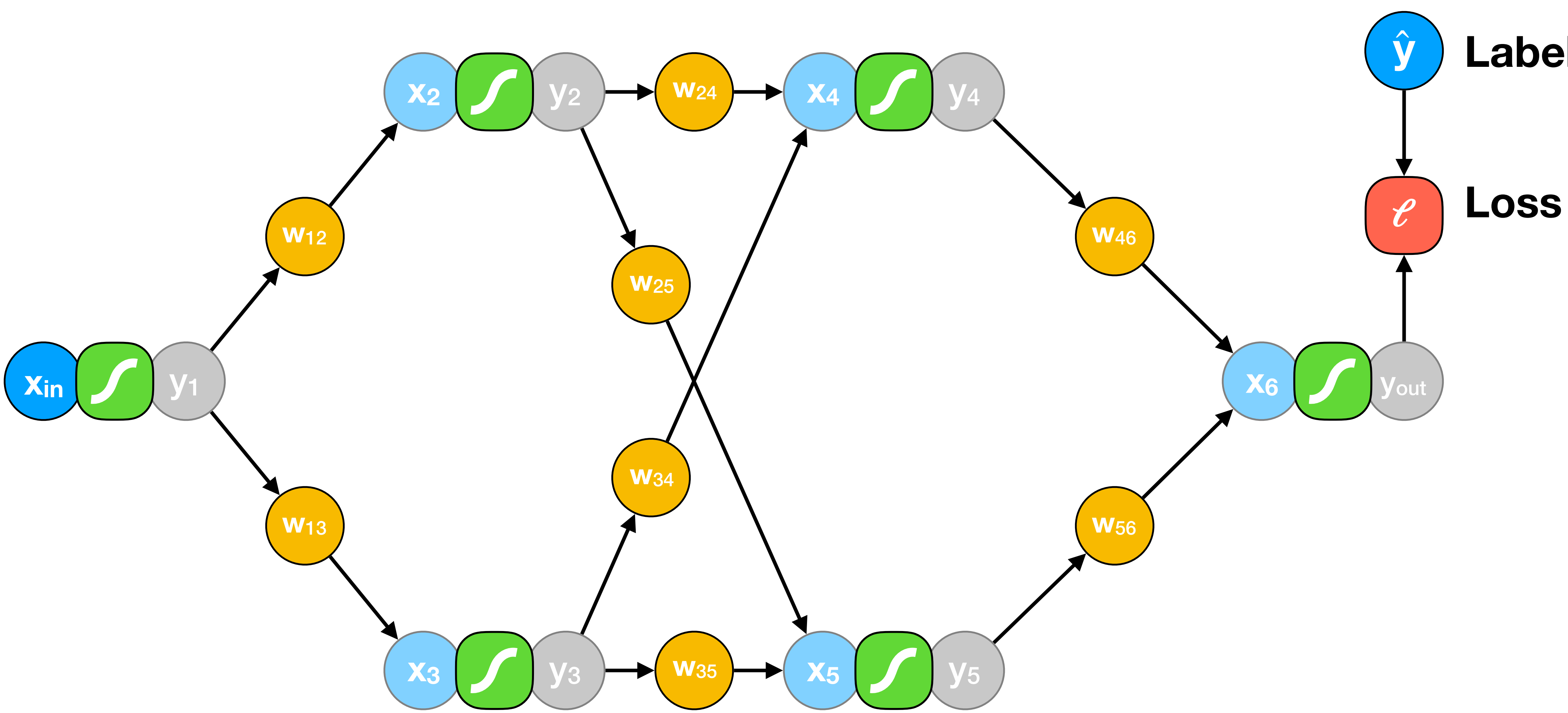
- Some useful **definitions** related to training.
  - **Step/Iteration** - The sequence of computation required to complete a single update of the network weights.
  - **Batch** - A subset of the input dataset processed by the network during a single iteration.
  - **Epoch** - One cycle through the full training set. Why there is utility in continuing to train for several epochs?
    - Training is **iterative.** The network **weights are adjusted gradually** and the loss may not minimize after just one epoch.
    - Training data are often presented in **random order**. Different data orderings may improve model **generality**.

Backpropagation

- During training the network **weights** must be **adjusted** such that the overall **loss decreases**.
- Adjust each weight using the **partial derivative** of the loss with respect to that weight ($\partial \ell / \partial w$).
- **Backpropagation** is a computationally **efficient** algorithm to compute the required partial derivatives.
- It performs part of the computation as it makes its prediction during the **forward pass**.
- The network is then **traversed in reverse** to **compute** the required **gradients** and **update** the network **weights**.

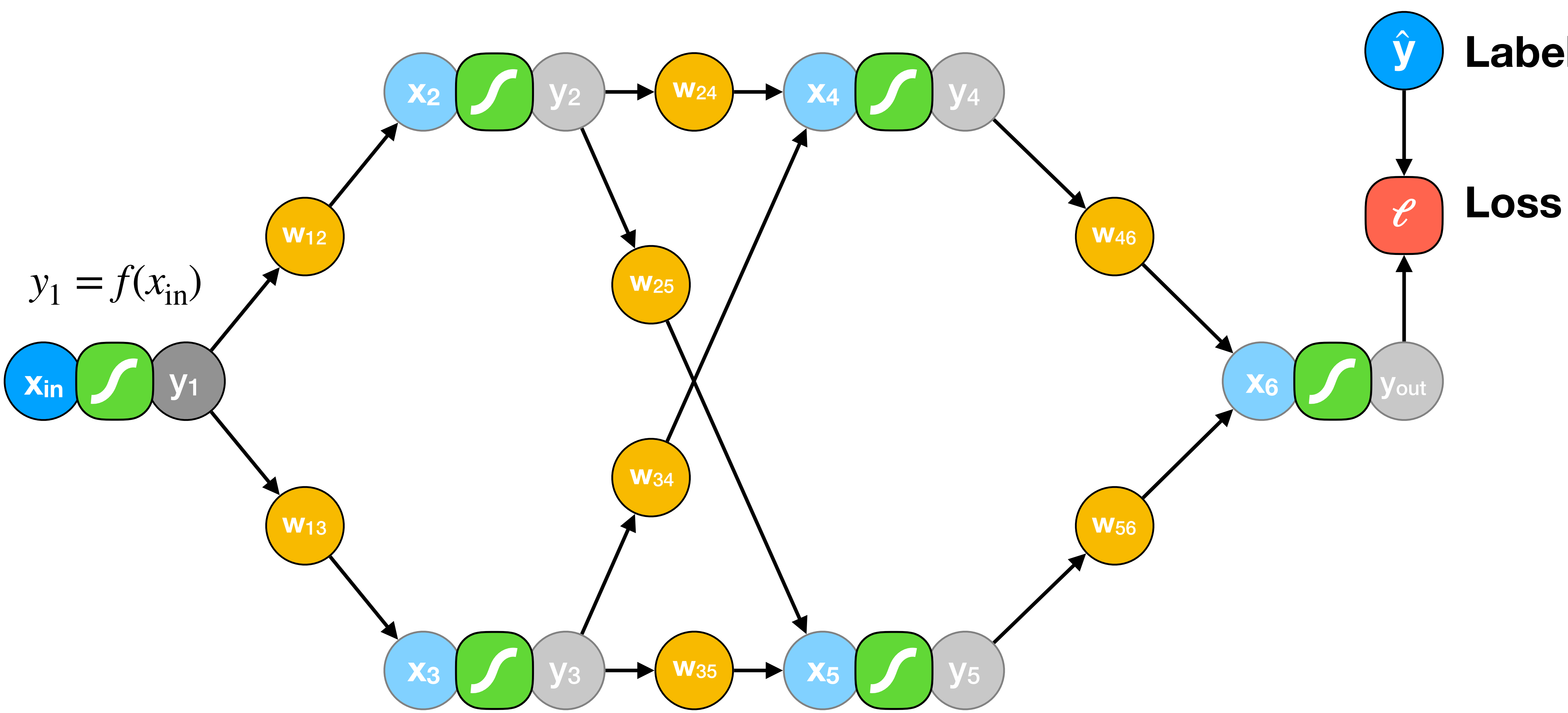
Backpropagation

The Forward Pass - Prediction



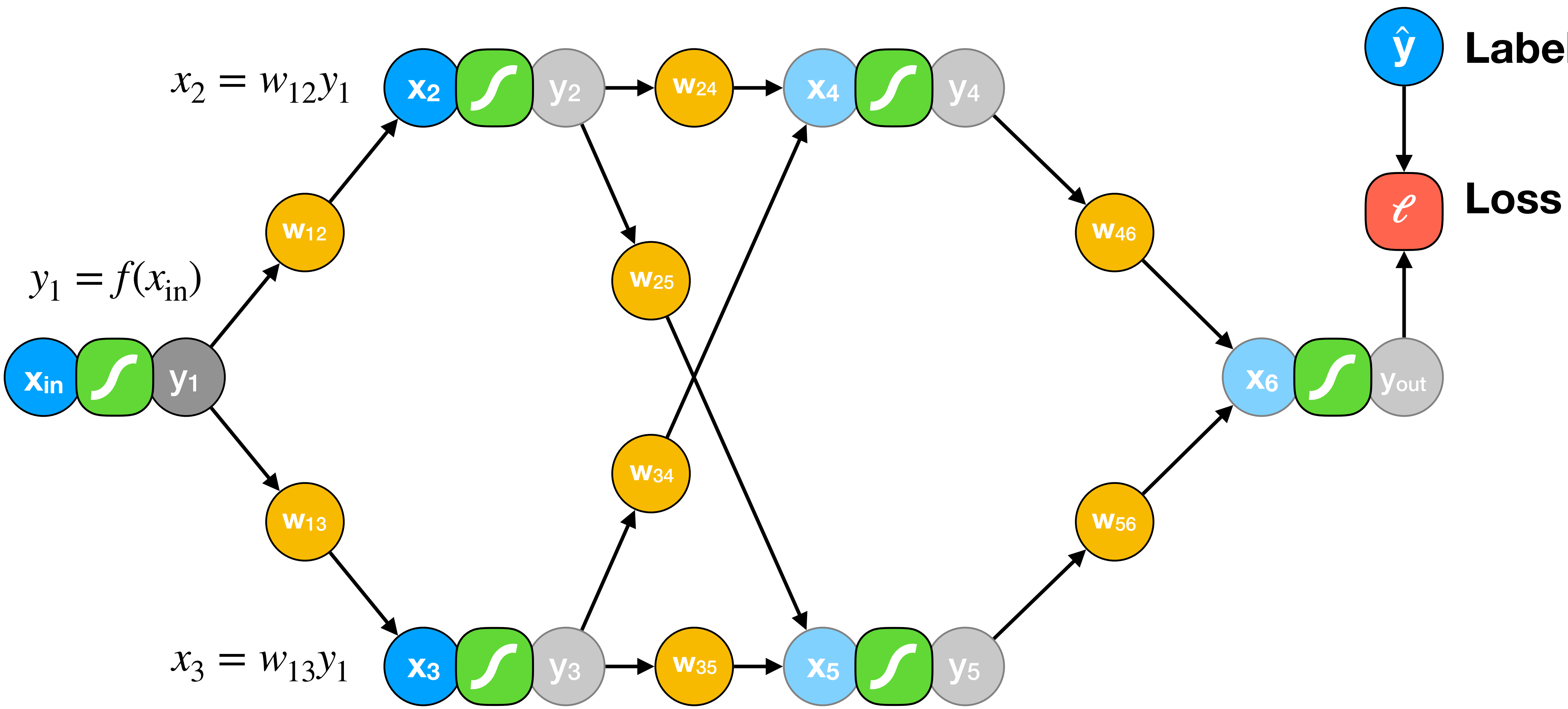
Backpropagation

The Forward Pass - Prediction



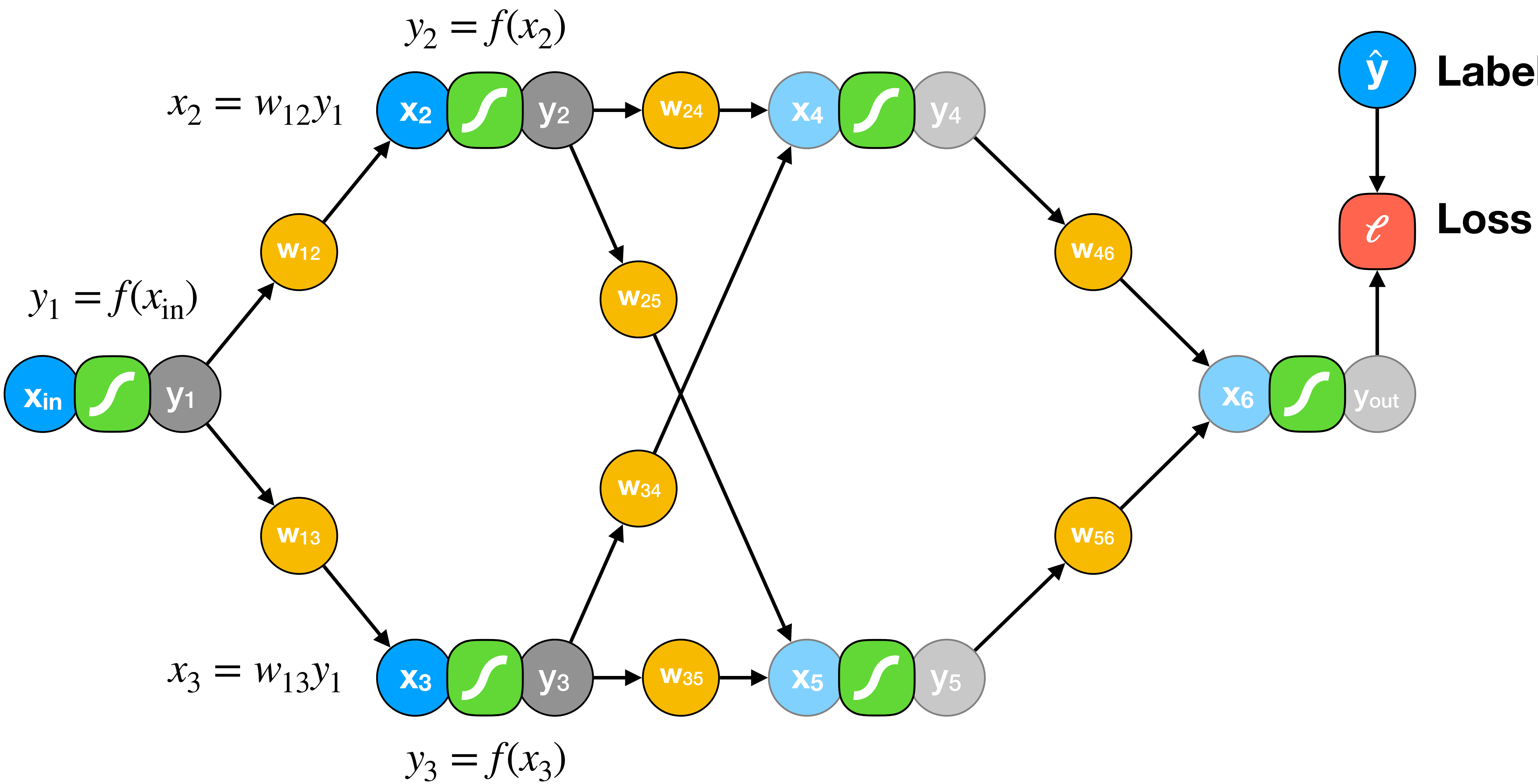
Backpropagation

The Forward Pass - Prediction

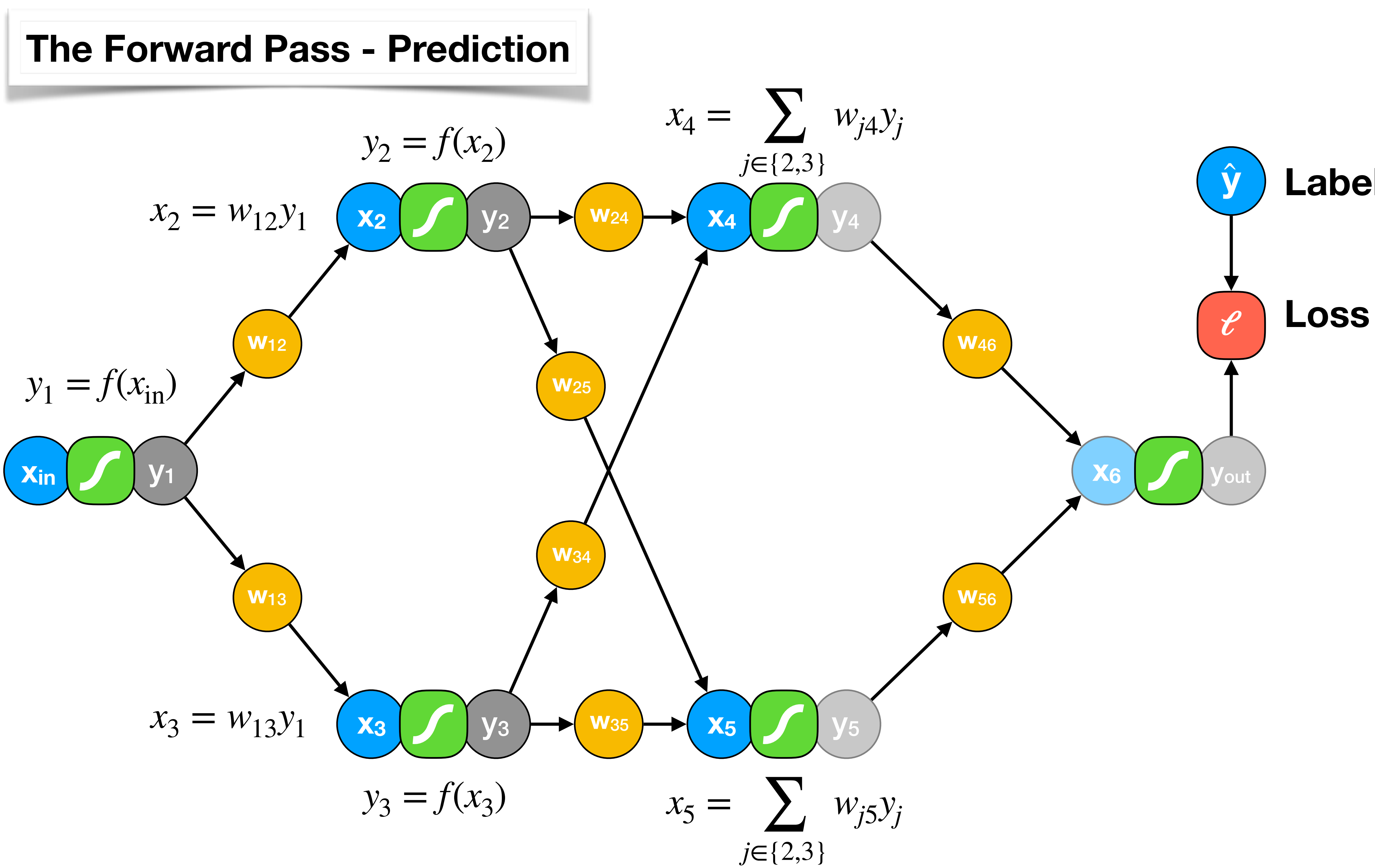


Backpropagation

The Forward Pass - Prediction

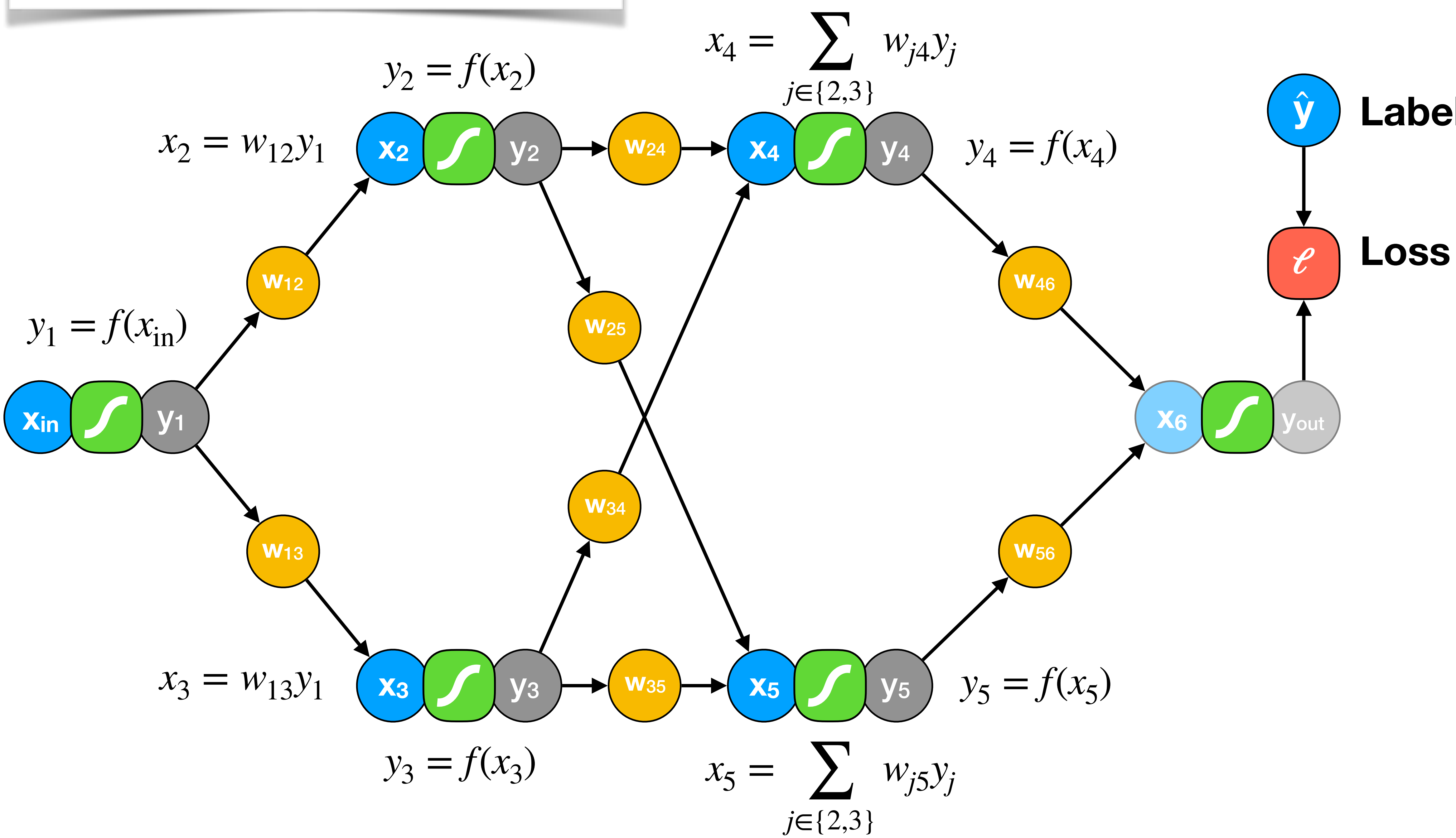


Backpropagation



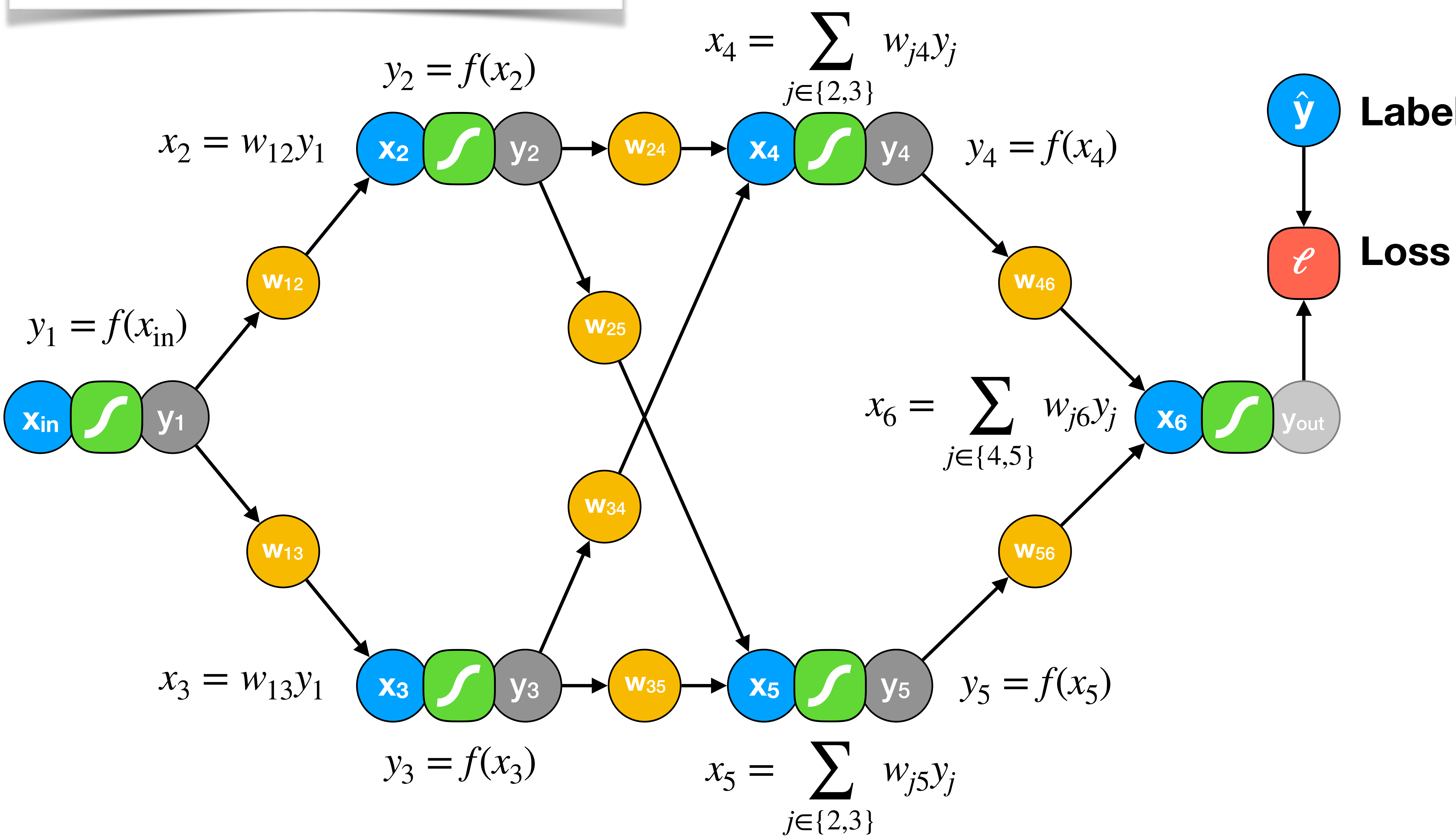
Backpropagation

The Forward Pass - Prediction



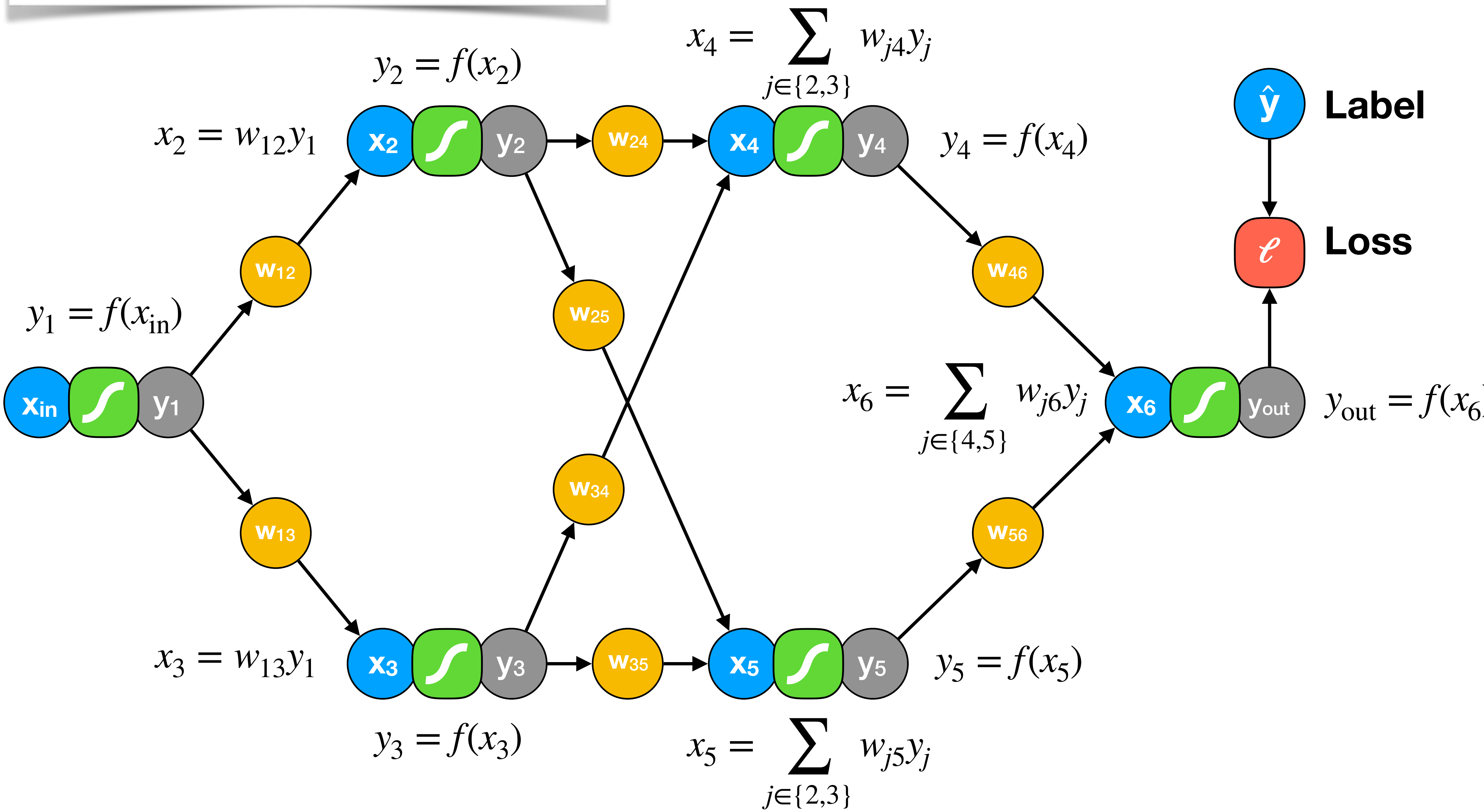
Backpropagation

The Forward Pass - Prediction



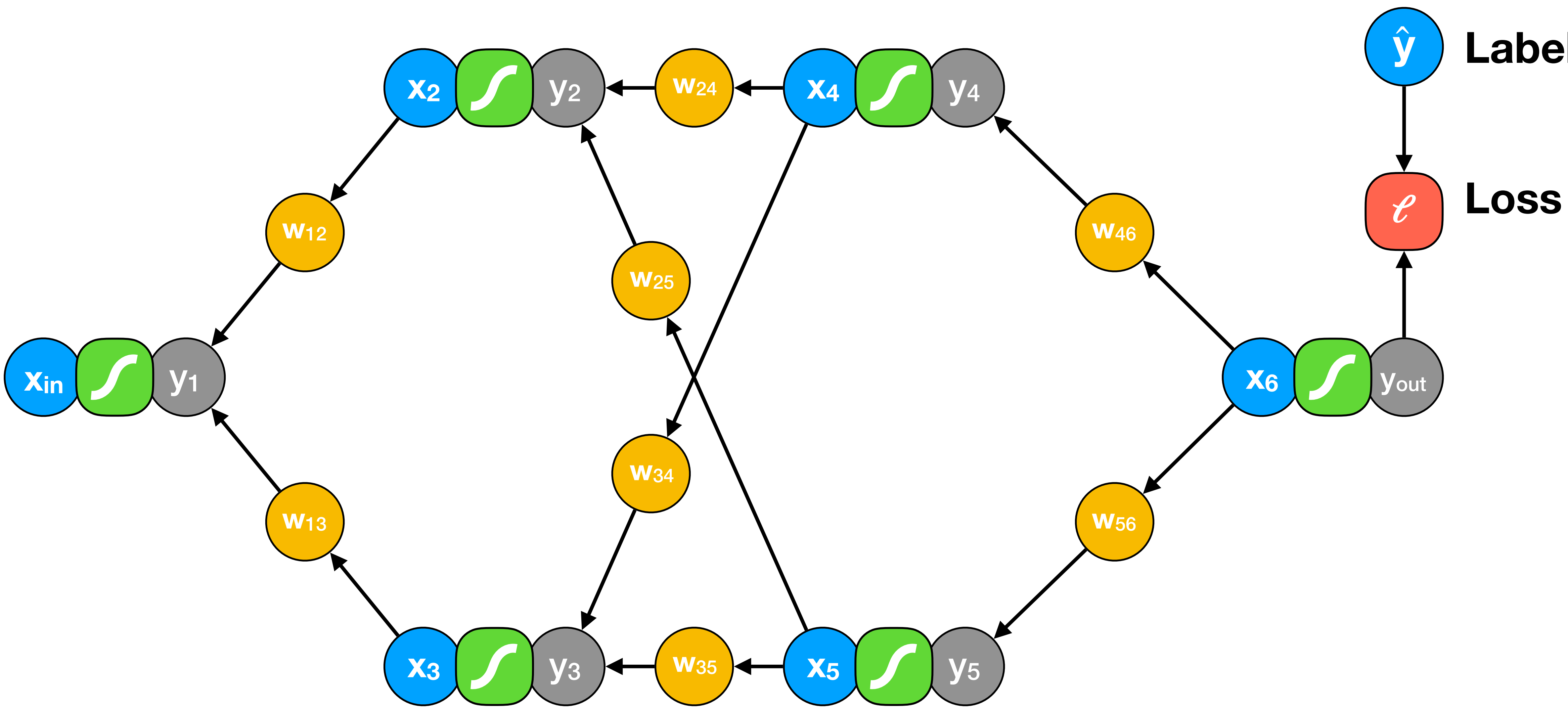
Backpropagation

The Forward Pass - Prediction

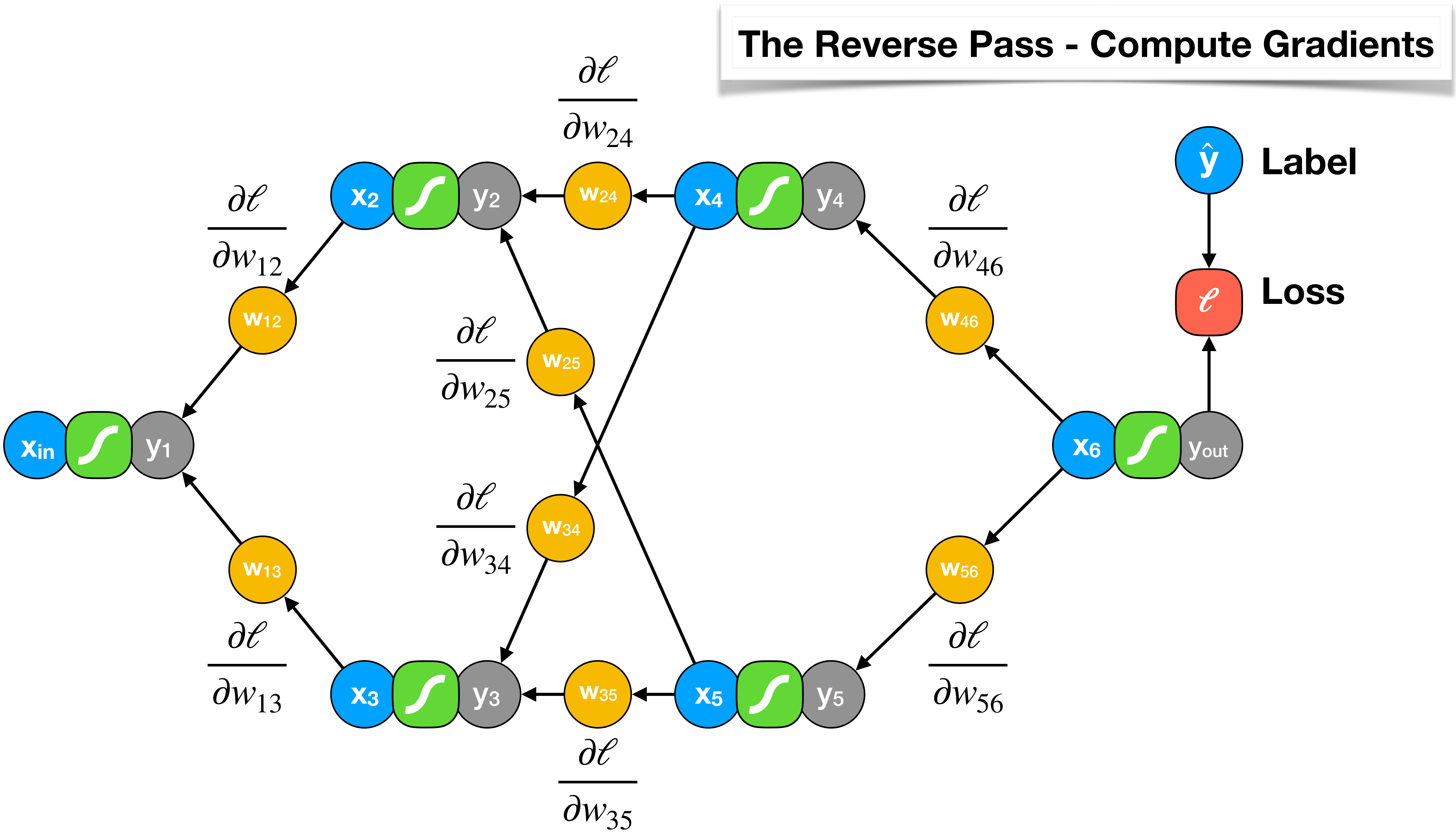


Backpropagation

The Reverse Pass - Compute Gradients

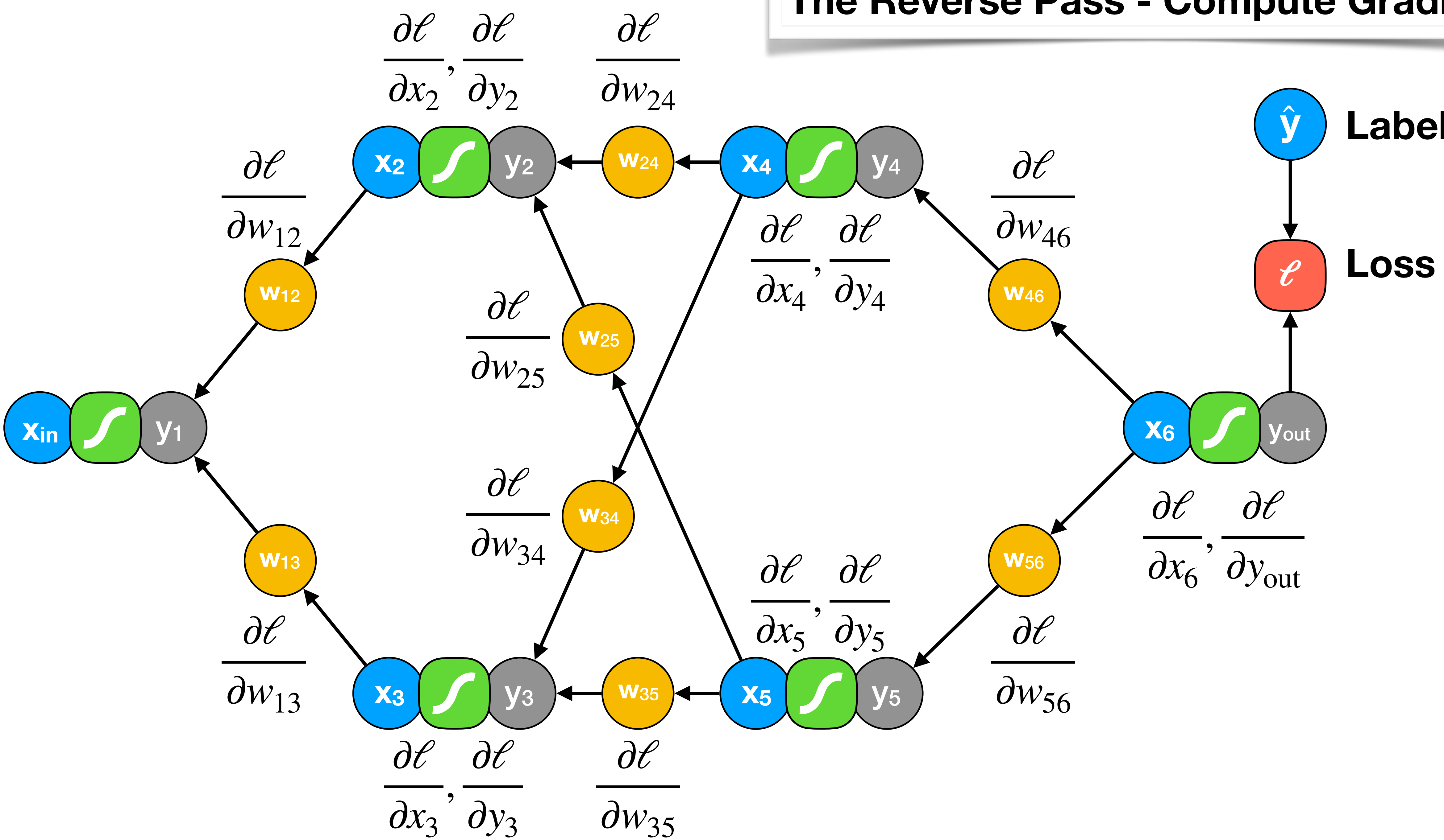


Backpropagation



Backpropagation

The Reverse Pass - Compute Gradients



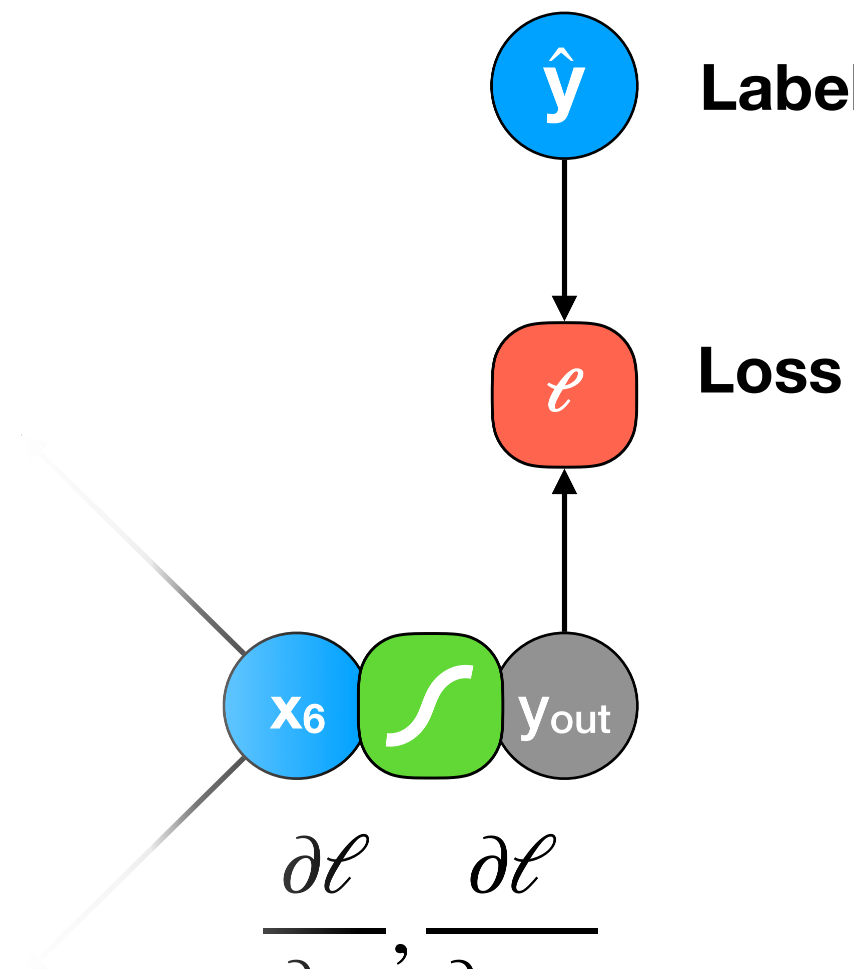
Backpropagation

- Since we **defined** the **loss function** ℓ, it is straightforward to compute $\partial \ell / \partial y_{\text{out}}$.
- For example,

$$\ell = \frac{1}{2}(\hat{y} - y_{\text{out}})^2 \implies \frac{\partial \ell}{\partial y_{\text{out}}} = \hat{y} - y_{\text{out}}$$

- Then apply the chain rule to compute

$$\frac{\partial \ell}{\partial x_6} = \frac{dy_{\text{out}}}{dx_6} \cdot \frac{\partial \ell}{\partial y_{\text{out}}} = \frac{d}{dx_6} f(x_6) \cdot \frac{\partial \ell}{\partial y_{\text{out}}}$$



$$\frac{\partial \ell}{\partial x_6}, \frac{\partial \ell}{\partial y_{\text{out}}}$$

Backpropagation

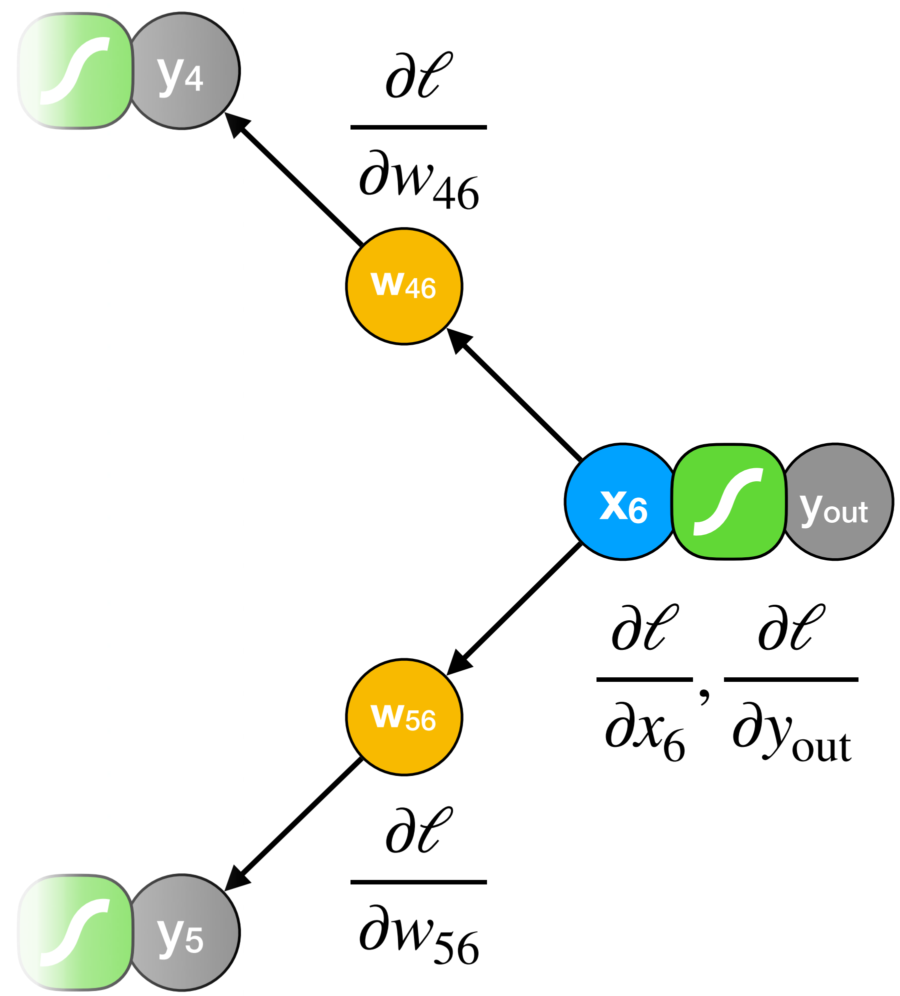
- Now that we have $\partial \ell / \partial y_{\text{out}}$ and $\partial \ell / \partial x_6$ we can compute the **derivatives with respect to the weights.**

- Again, we can **apply the chain rule**

$$\frac{\partial \ell}{\partial w_{46}} = \frac{\partial x_6}{\partial w_{46}} \frac{\partial \ell}{\partial x_6} = y_4 \frac{\partial \ell}{\partial x_6}$$
$$\frac{\partial \ell}{\partial w_{56}} = \frac{\partial x_6}{\partial w_{56}} \frac{\partial \ell}{\partial x_6} = y_5 \frac{\partial \ell}{\partial x_6}$$

- Proceeding to the hidden layers, we must now compute

$$\frac{\partial \ell}{\partial x_4}, \frac{\partial \ell}{\partial y_4}, \frac{\partial \ell}{\partial x_5}, \frac{\partial \ell}{\partial y_5}$$



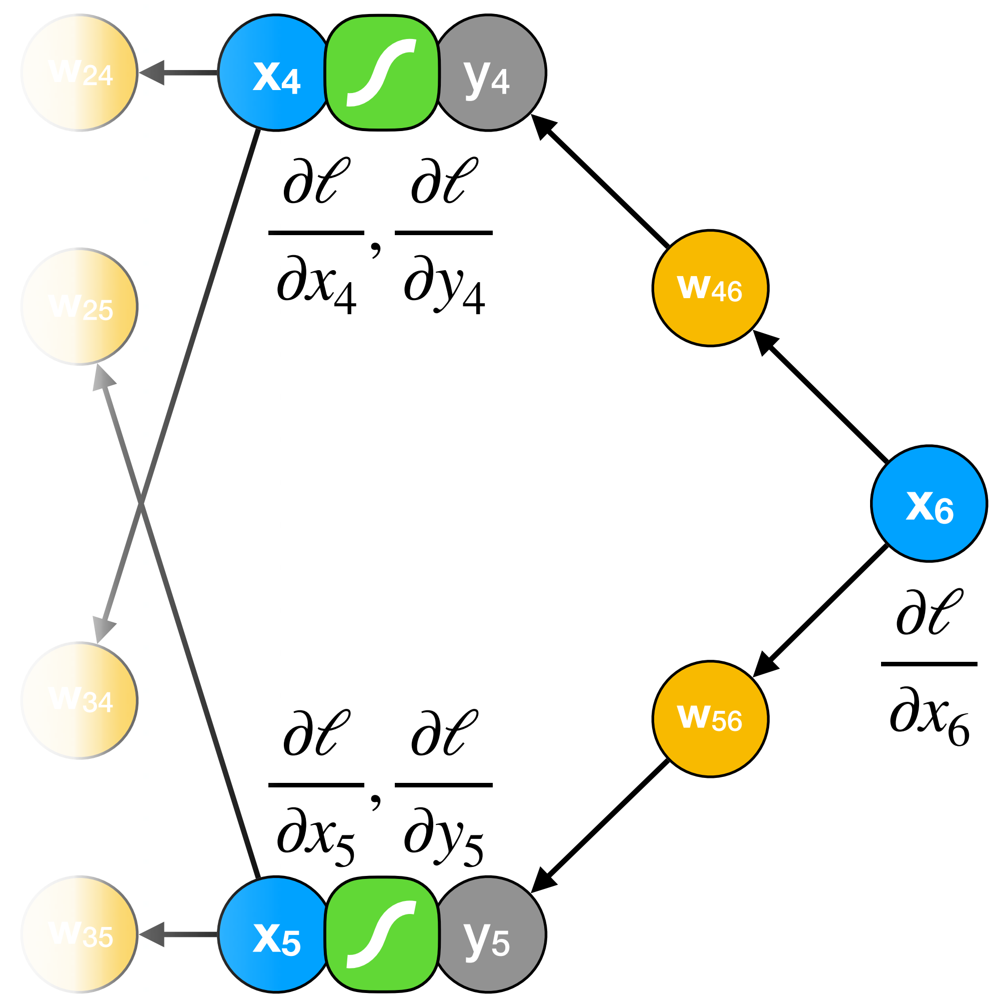
Backpropagation

- The derivatives of ℓ with respect to the **outputs** are computed using.

$$\frac{\partial \ell}{\partial y_4} = \frac{\partial x_6}{\partial y_4}\frac{\partial \ell}{\partial x_6} = \frac{\partial \ell}{\partial x_6} w_{46}$$

$$\frac{\partial \ell}{\partial y_4} = \frac{\partial x_5}{\partial y_4}\frac{\partial \ell}{\partial x_5} = \frac{\partial \ell}{\partial x_5} w_{56}$$

- When the units of a hidden layer connect to **more than one output**, then the **derivative becomes a sum** over all those outputs.



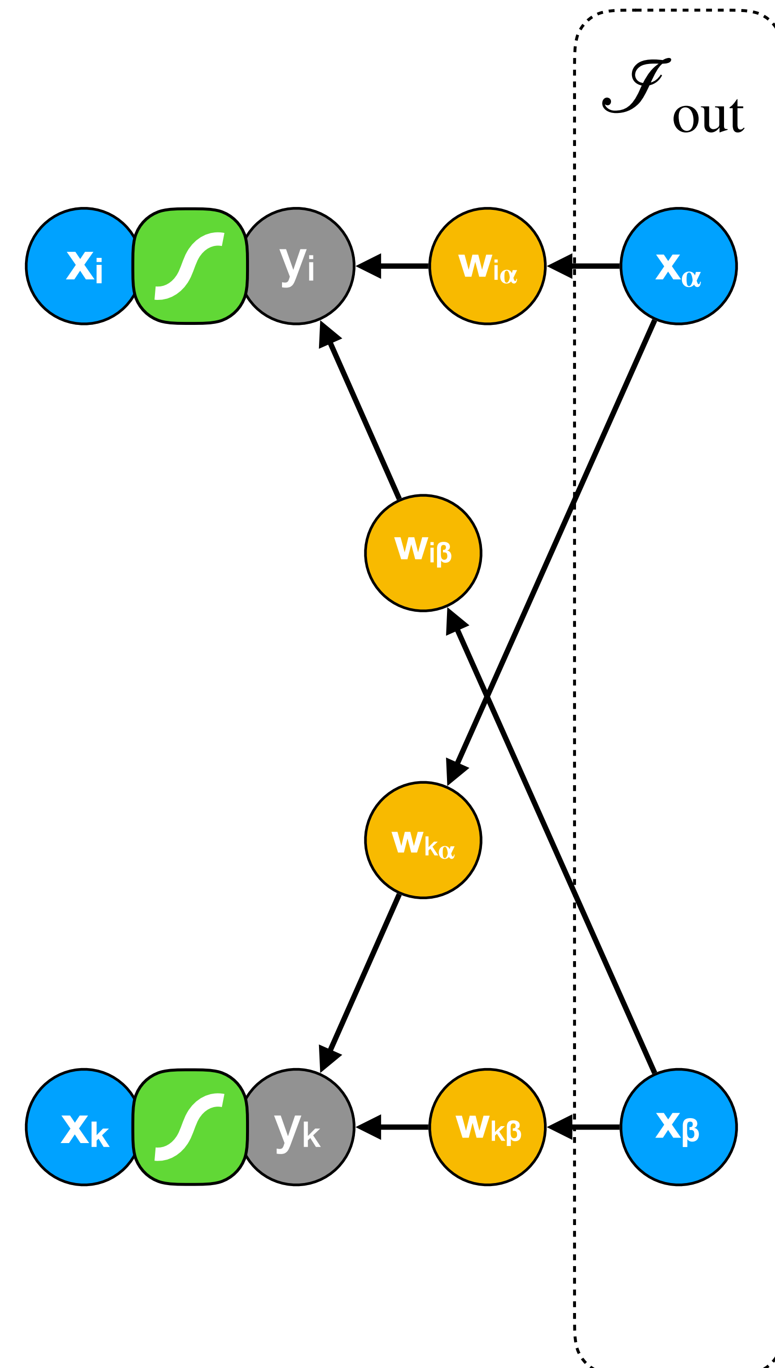
Backpropagation

- **In general**, if $\mathcal{I}_{\text{out}}$ is the set of neurons that **receive the outputs** of neuron i (in the diagram, $\mathcal{I}_{\text{out}} = \{\alpha, \beta\}$), then

$$\frac{\partial \ell}{\partial y_i} = \sum_{j \in \mathcal{I}_{out}} \frac{\partial x_j}{\partial y_i} \frac{\partial \ell}{\partial x_j} = \sum_{j \in \mathcal{I}_{out}} w_{ij} \frac{\partial \ell}{\partial x_j}$$

- The derivatives with respect to the neuron **inputs** can be computed as for the output neuron. For the general case.

$$\frac{\partial \ell}{\partial x_i} = \frac{dy_i}{dx_i} \cdot \frac{\partial \ell}{dy_i} = \frac{d}{dx_i} f(x_i) \cdot \frac{\partial \ell}{dy_i}$$



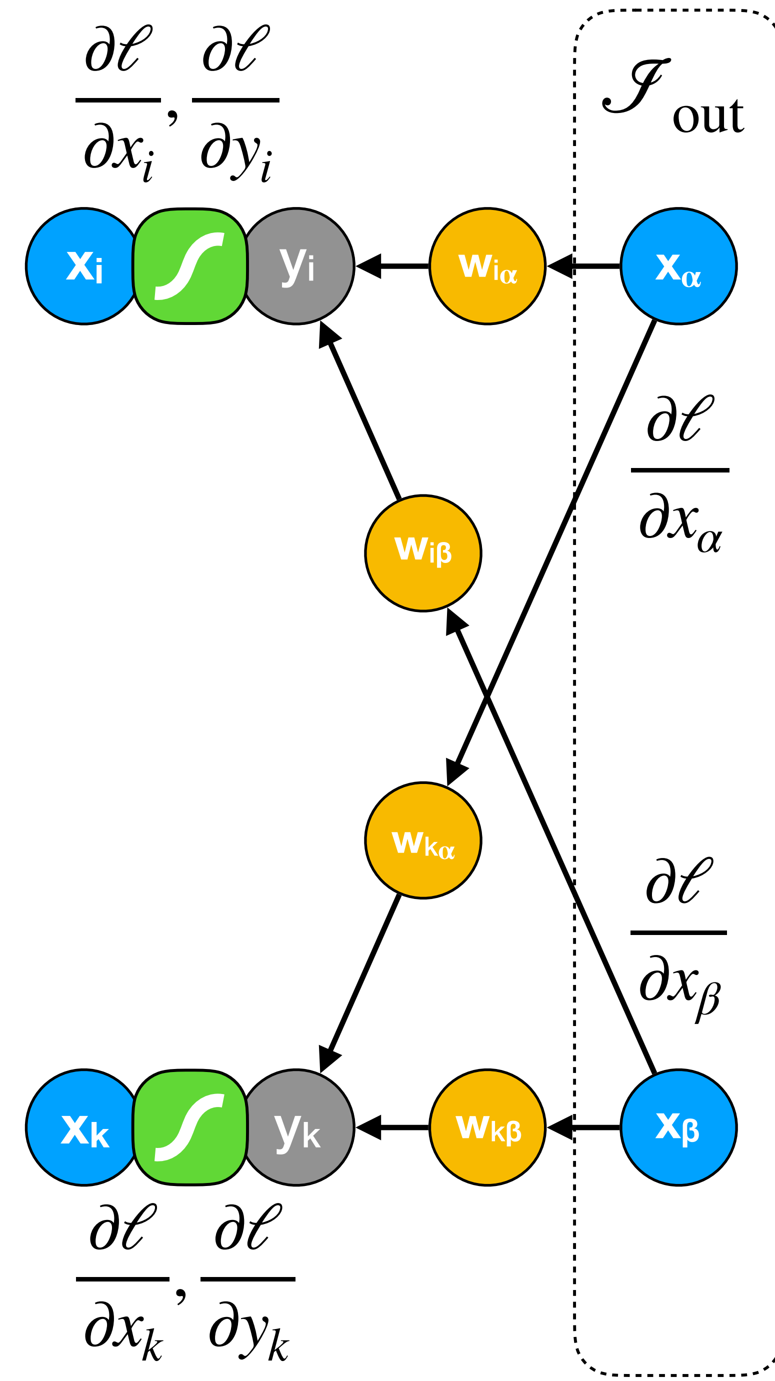
Backpropagation

- **In general**, if $\mathscr{I}_{\text{out}}$ is the set of neurons that **receive the outputs** of neuron i (in the diagram, $\mathscr{I}_{\text{out}} = \{\alpha, \beta\}$), then

$$\frac{\partial \ell}{\partial y_i} = \sum_{j \in \mathscr{I}_{out}} \frac{\partial x_j}{\partial y_i} \frac{\partial \ell}{\partial x_j} = \sum_{j \in \mathscr{I}_{out}} w_{ij} \frac{\partial \ell}{\partial x_j}$$

- The derivatives with respect to the neuron **inputs** can be computed as for the output neuron. For the general case.

$$\frac{\partial \ell}{\partial x_i} = \frac{dy_i}{dx_i} \cdot \frac{\partial \ell}{\partial y_i} = \frac{d}{dx_i} f(x_i) \cdot \frac{\partial \ell}{\partial y_i}$$

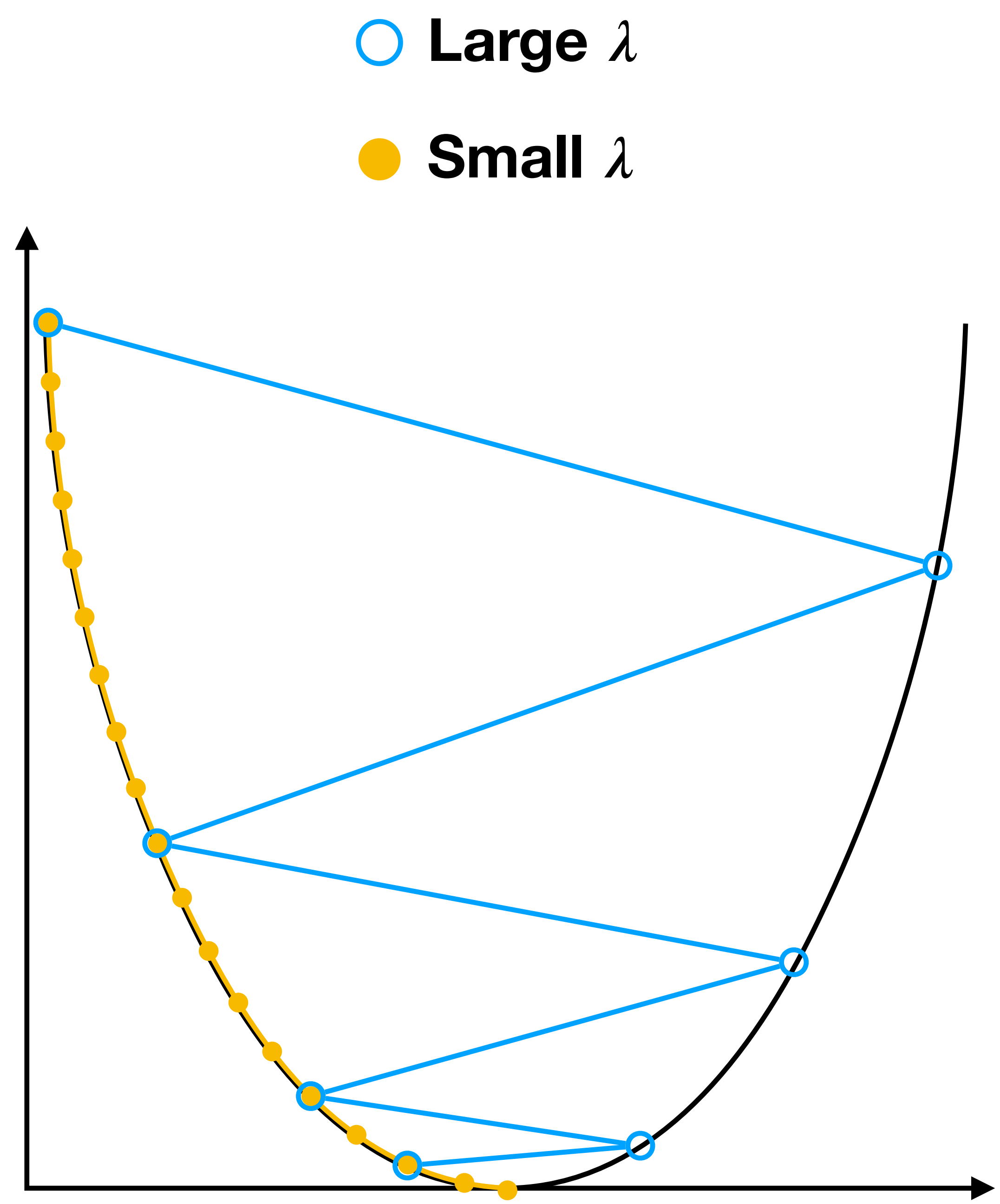


Learning rate

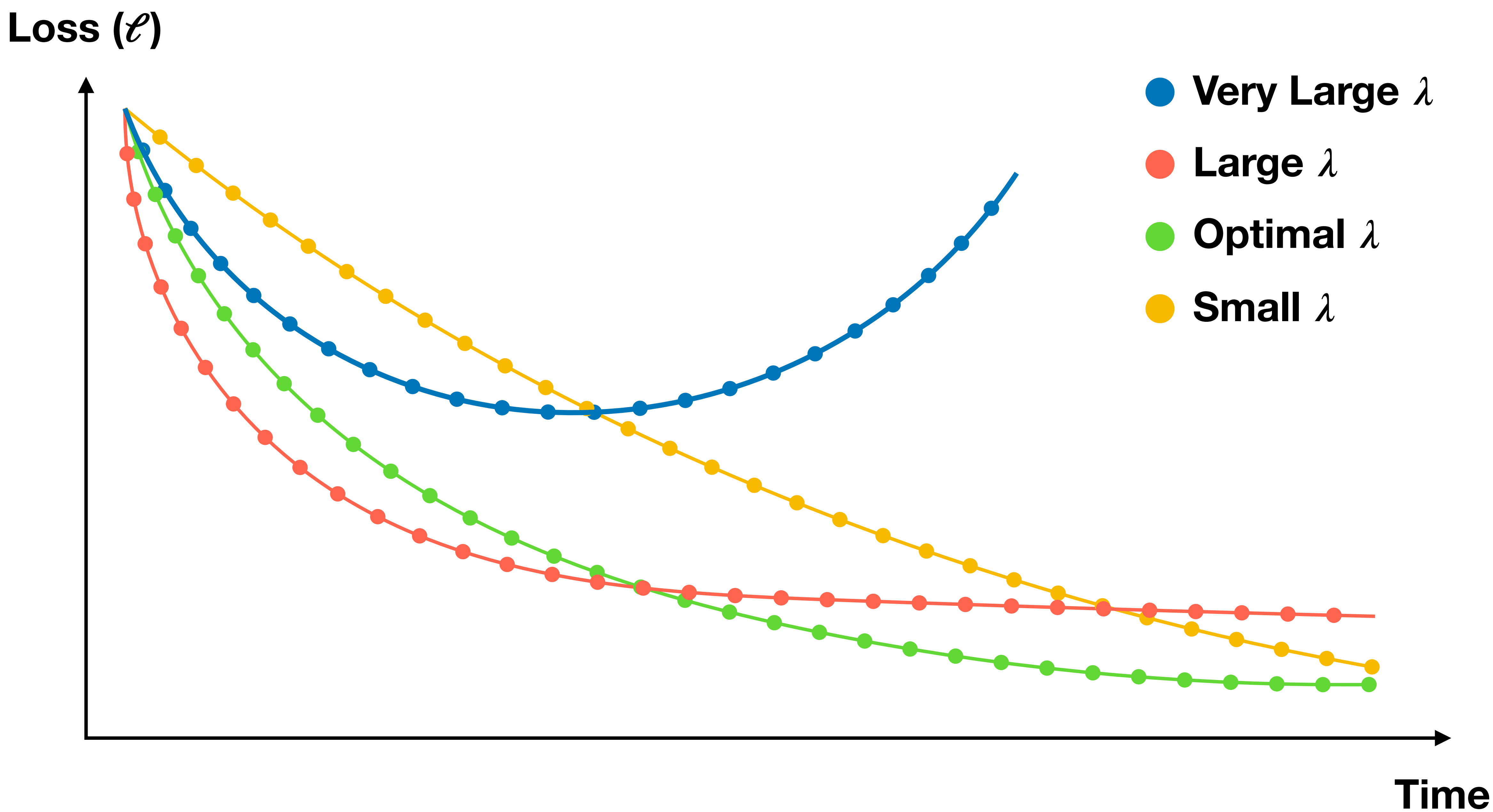
- Finally, the **network weights are updated** using

$$w_{ij} \to w_{ij} + \lambda \cdot \frac{d\ell}{dw_{ij}}$$

- λ is a tunable hyper-parameter called the **learning rate.**
- If λ is **too small** then training will be **slow.**
- If λ is **too large**, then gradient descent may **overshoot** the optimal solution.



Learning rate



Live Demo

Convolutional Neural Networks

ANN Downsides

- Deep, fully connected ANNs rapidly become **cumbersome** for **large input feature spaces**.
- The **number of trainable weights explodes**. The networks can be very **slow to train** and become **vulnerable to overfitting**.
- **ANN input features** must often be **hand-engineered** for best results.

CNNs

- **Convolutional Neural Networks** (CNNs) are designed to **mitigate these shortcomings**.
- They are able to **learn** a set of **pertinent features** from the data.
- They **number of trainable weights decreases** rapidly for deeper layers.
- CNNs also provide **translational invariance** in their response to the features they learn.

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Kernels replace weights

Input (X)
(m x n)

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

Output (Y)

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Kernels replace weights

Input (X)
(m x n)

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

$$
\begin{aligned}
Y(0,0) &= (0 \times 0) + (0 \times 1) + (0 \times 0) \\
&\quad + (0 \times 1) + (0 \times 0) + (1 \times 1) \\
&\quad + (0 \times 0) + (1 \times 1) + (2 \times 0) \\
&= 2
\end{aligned}
$$

Output (Y)

2					

Activation function operates on convolution result

Identity assumed for this example

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Kernels replace weights

Input (X)
(m x n)

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

$$
\begin{aligned}
Y(0,1) = {} & (0 \times 0) + (0 \times 1) + (1 \times 0) \\
& + (0 \times 1) + (0 \times 0) + (2 \times 1) \\
& + (0 \times 0) + (2 \times 1) + (3 \times 0) \\
& = \mathbf{4}
\end{aligned}
$$

Output (Y)

2	**4**				

Activation function operates on convolution result

Identity assumed for this example

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Kernels replace weights

Input (X)
(m x n)

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

$$Y(i,j) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} X(i-k, j-l) \cdot K(k,l)$$

Valid Padding

Output (Y)

2	**4**	**7**	**7**	**4**	**2**

Activation function operates on convolution result

Identity assumed for this example

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Input (X)
(m x n)

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

Output (Y)

2	4	7	7	4	2
4	5	9	9	5	4
3	7	6	6	7	3
4	4	4	4	4	4
6	1	4	4	1	6
0	8	1	1	8	0

$$Y(i,j) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} X(i-k, j-l) \cdot K(k,l)$$

Activation function operates on convolution result

Kernels replace weights

Valid Padding

Identity assumed for this example

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Kernels replace weights

Input (X)
(m x n)

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

$$Y(i,j) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} X(i-k, j-l) \cdot K(k,l)$$

Valid Padding

Output (Y)

2	5	6	6	5	2
2	6	5	5	6	2
4	6	7	7	6	4
3	4	4	4	4	3
2	6	3	3	6	2
5	0	4	4	0	5

Activation function operates on convolution result

Identity assumed for this example

Convolutions

Kernel (K)
(k x l)

0	1	0
1	0	1
0	1	0

1	0	1
0	0	0
1	0	1

1	0	1
0	1	0
1	0	1

Kernels replace weights

Input (X)
(m x n)

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

$$Y(i,j) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} X(i-k, j-l) \cdot K(k,l)$$

Valid Padding

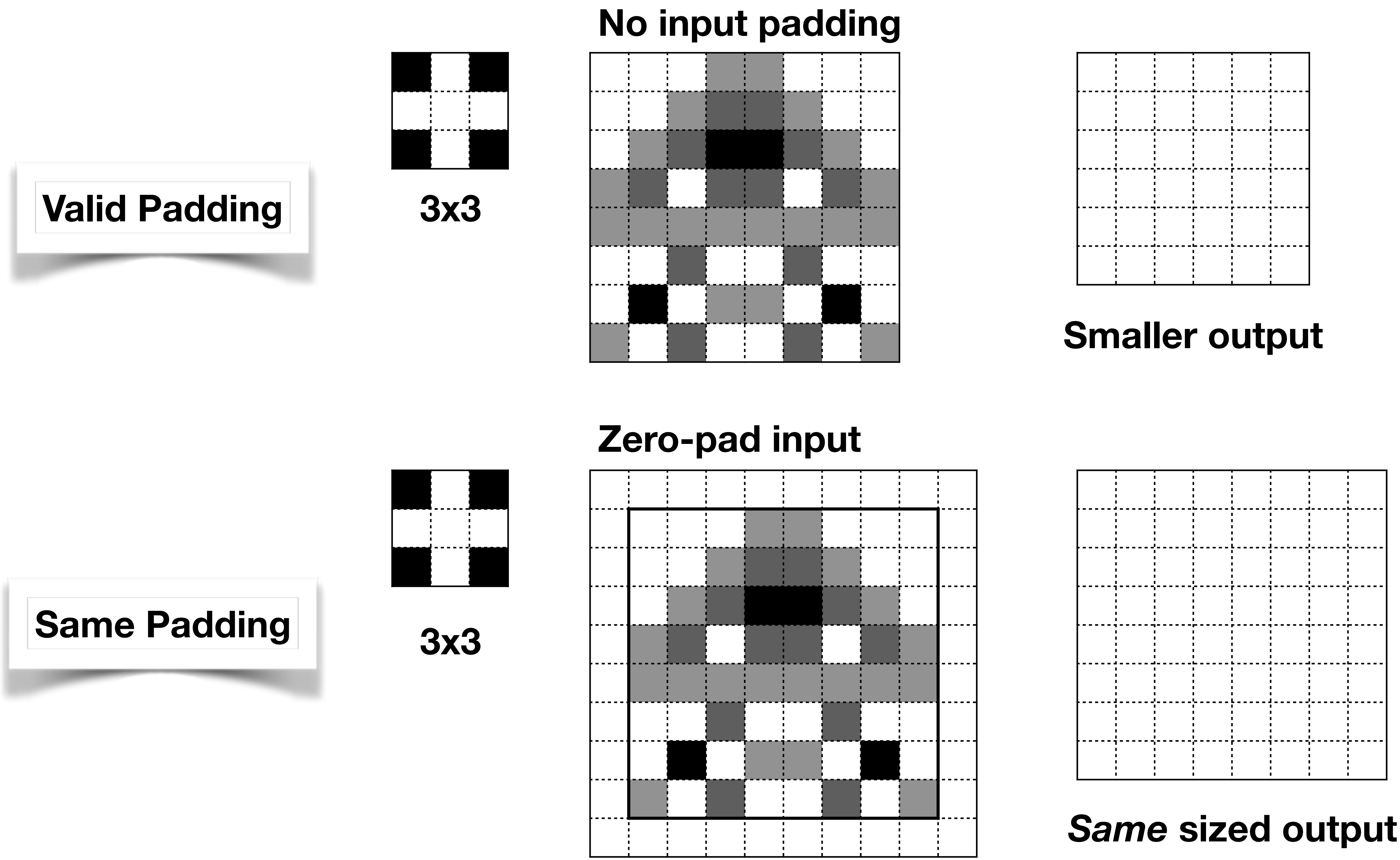
Output (Y)

2	6	8	8	6	2
3	8	8	8	8	3
6	6	9	9	6	6
4	5	5	5	5	4
2	8	3	3	8	2
8	0	5	5	0	8

3D tensor output

Output layers smaller than input.

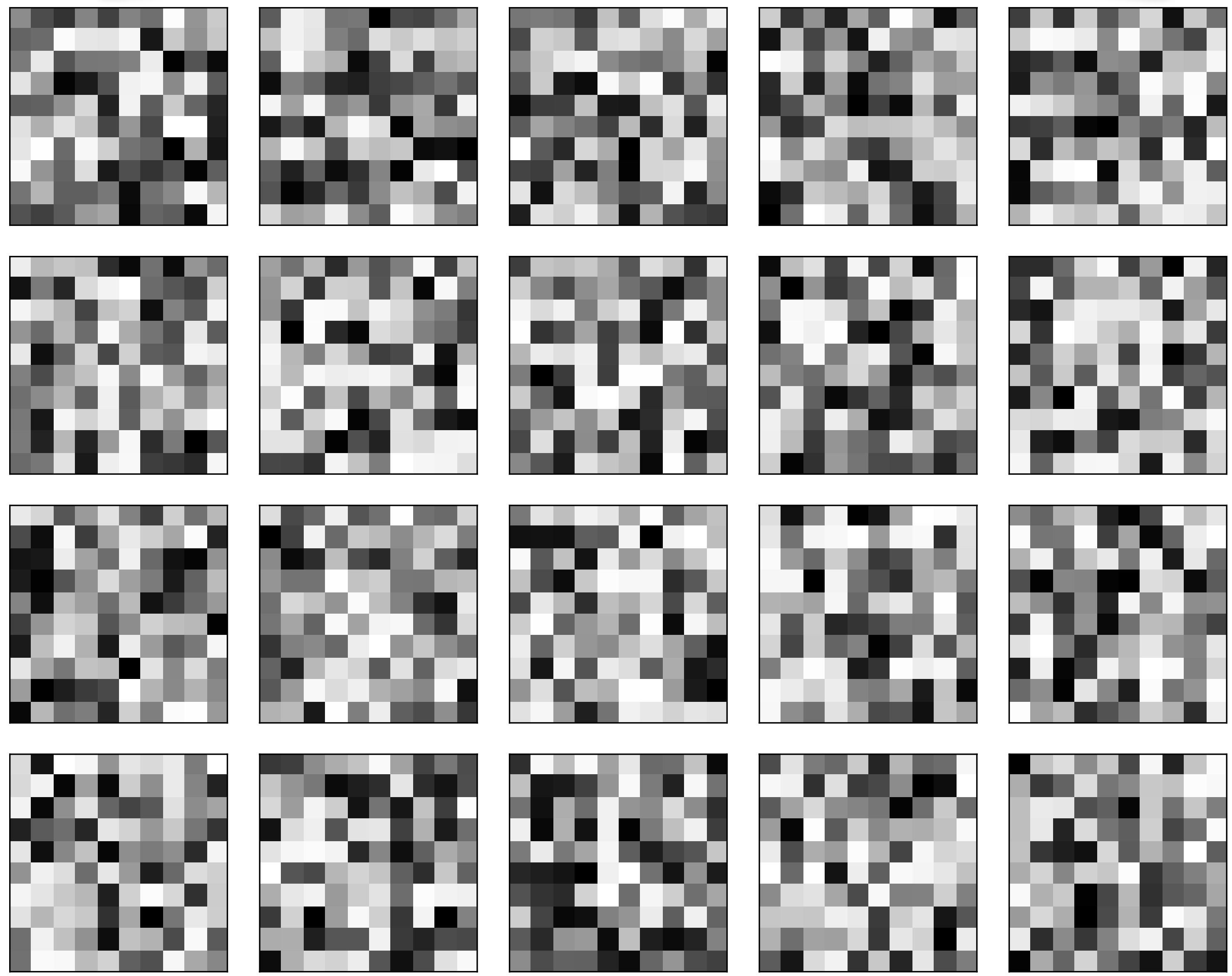
Padding



No input padding
Valid Padding
3x3
Smaller output
Zero-pad input
Same Padding
3x3
Same sized output

Kernel Initialization

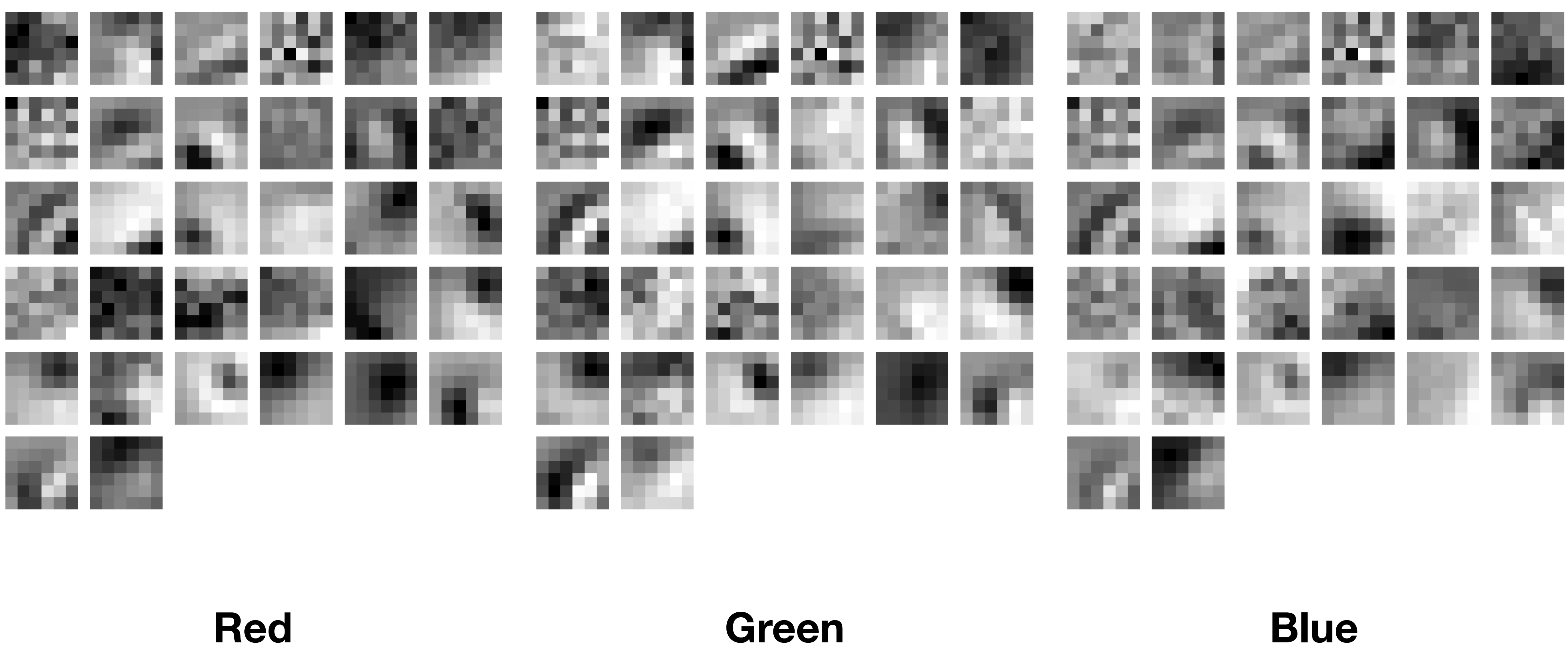
Kernels will be updated during training



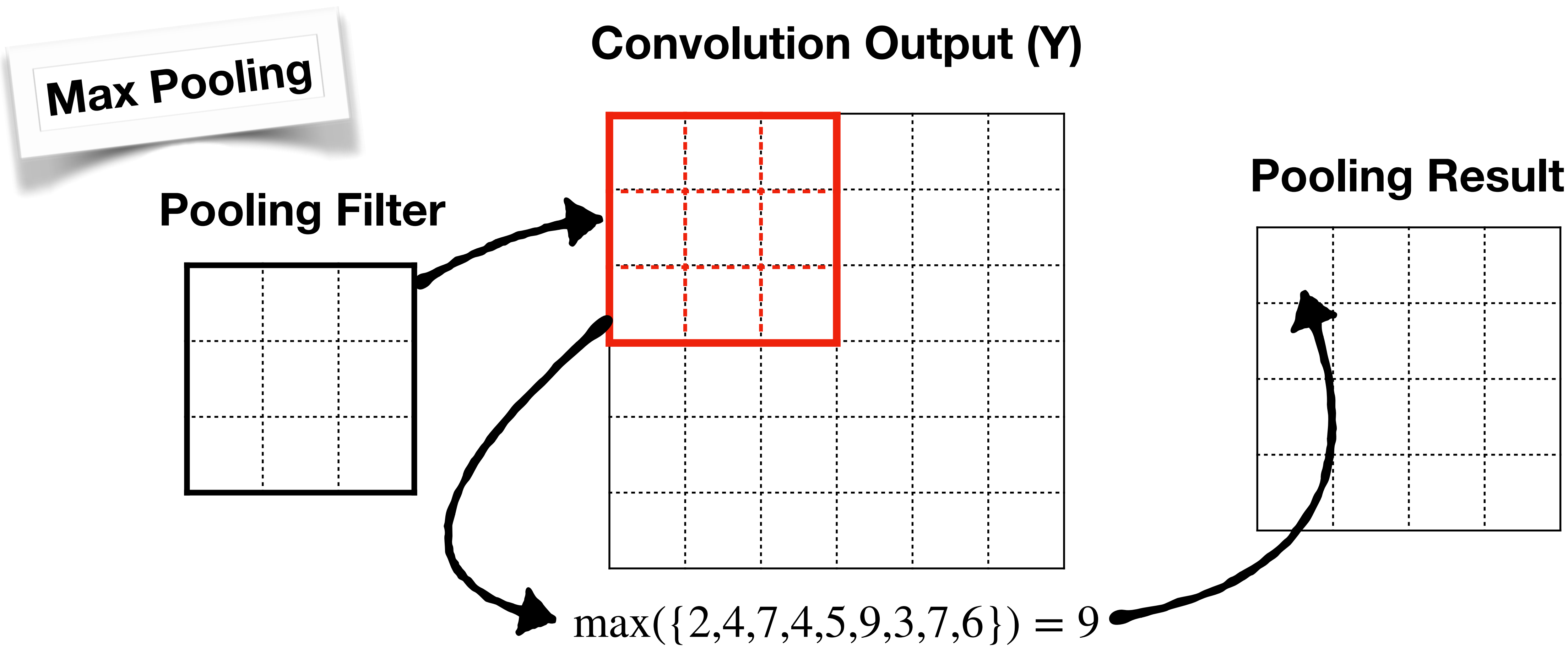
Random initialization often works well with sufficient training data

Kernel Initialization

Kernels of a trained network (Dieleman et al. 2015)

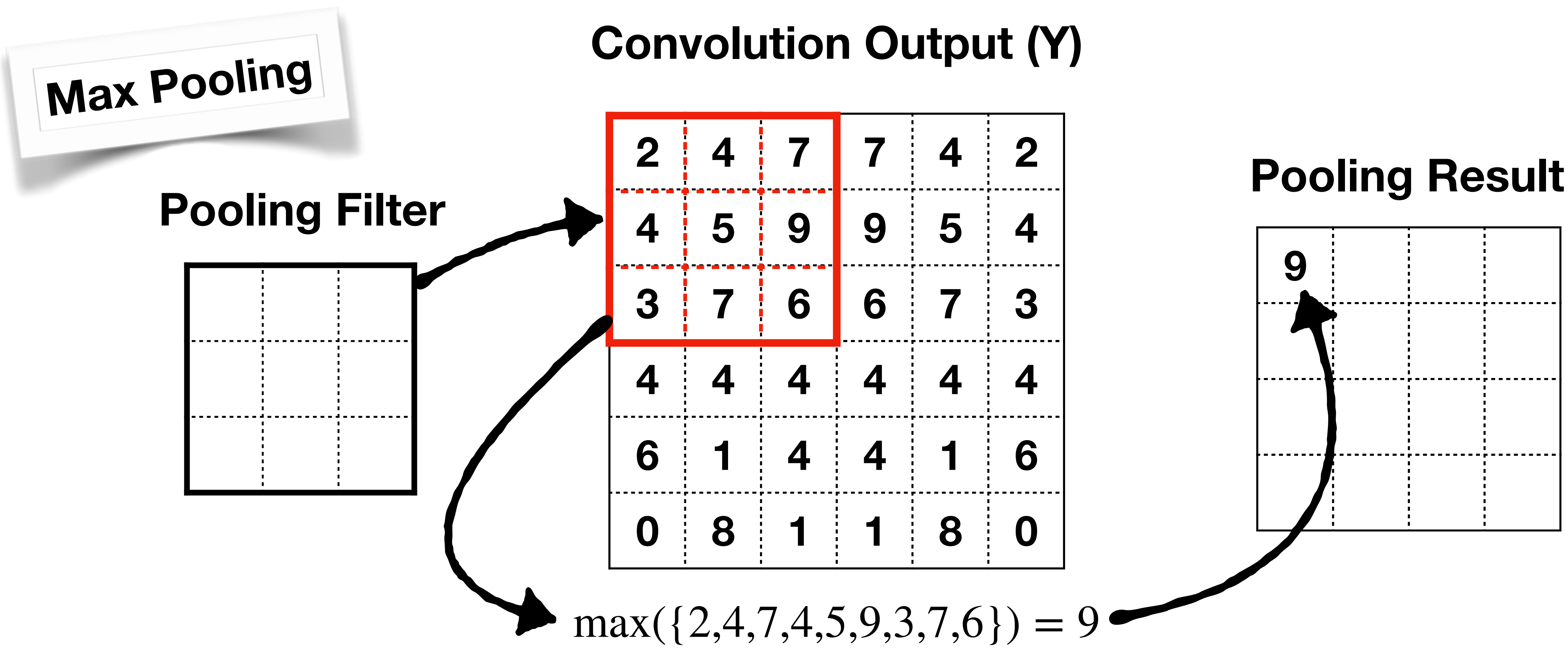


Pooling



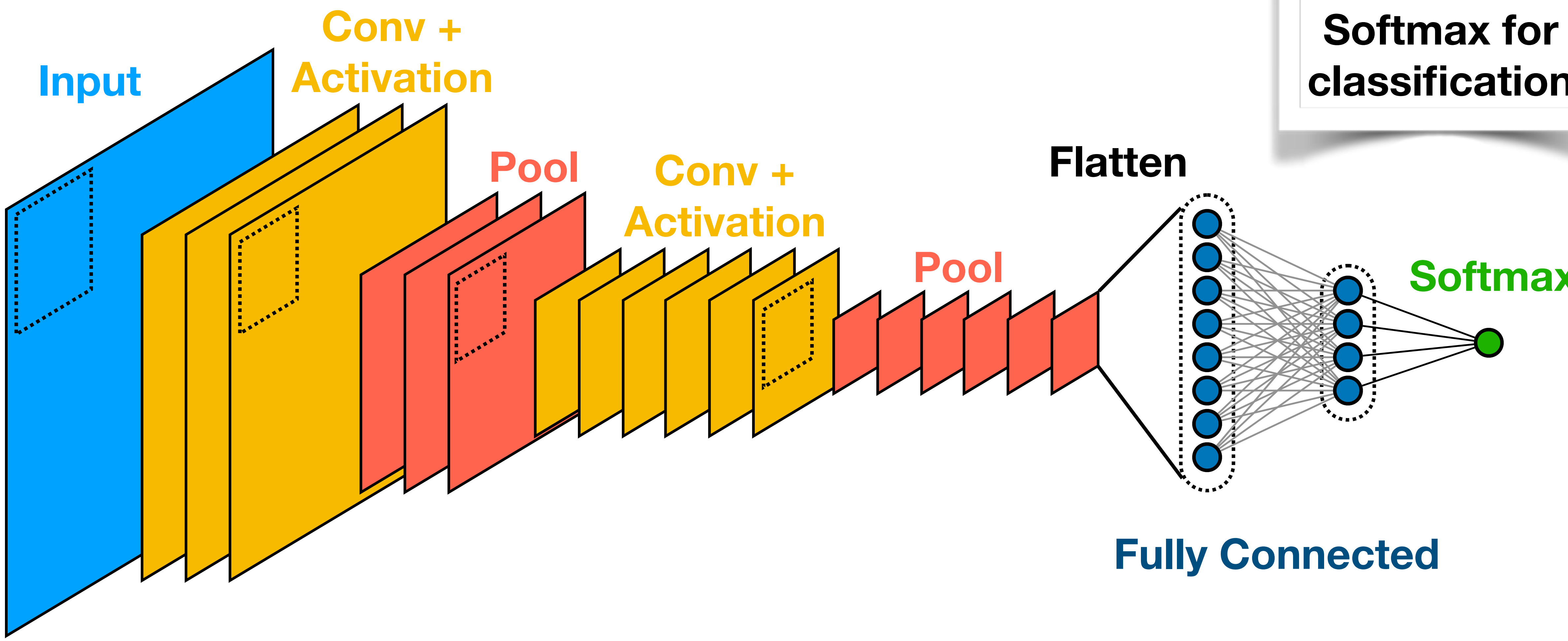
- Designed to **aggregate features** and **reduce** the total **number of trainable weights**.
- **Most common** variant is **max pooling** - take the **maximum value** within the pooling filter's **receptive field**.
- Also produces **translation invariance** in deep networks.

Pooling



- Designed to **aggregate features** and **reduce** the total **number of trainable weights**.
- **Most common** variant is **max pooling** - take the **maximum value** within the pooling filter's **receptive field**.
- Also produces **translation invariance** in deep networks.

CNN Carpentry



Softmax for classification

Convolutional layers - Mainly Feature Extraction

Dense layers - Mainly Inference

Data augmentation

- Pooling layers help to make the features learned by CNNs **translation invariant**.
- This means that **salient features will produce similar responses regardless of their location in the input data** e.g. image coordinates.
- Invariance with respect to **rotation**, **scaling** and **truncation** can be engineered by augmenting the training data.
- By adding **scaled**, **rotated**, **truncated** and even **blurred** copies of the original training data, the network can be taught to associate the **original data labels** with these **modified inputs**.

Data augmentation

Overfitting and Validation

- Deep learning models have limited utility if they do not **generalize** to unseen (albeit similar) data.
- A deep learning model can **appear** to perform very well simply by **memorizing its training data**. This is called **overfitting**.
- Without careful attention, deep networks with **millions** of trainable parameters can fit **specific** details that are **particular to the training data**.
- Such models will perform poorly and **make bad predictions** when applied to data that do not perfectly emulate the training data.

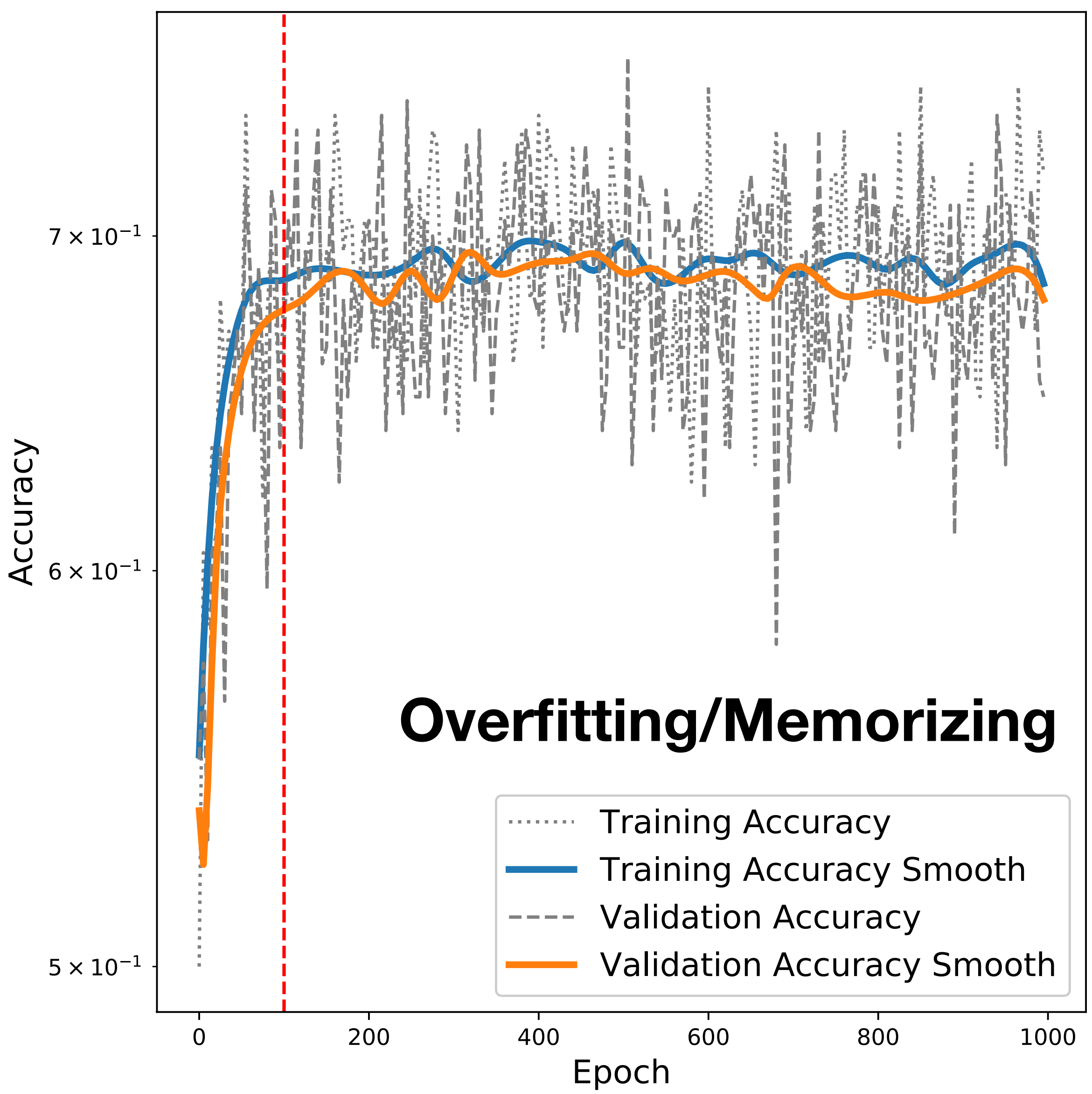
Overfitting and Validation

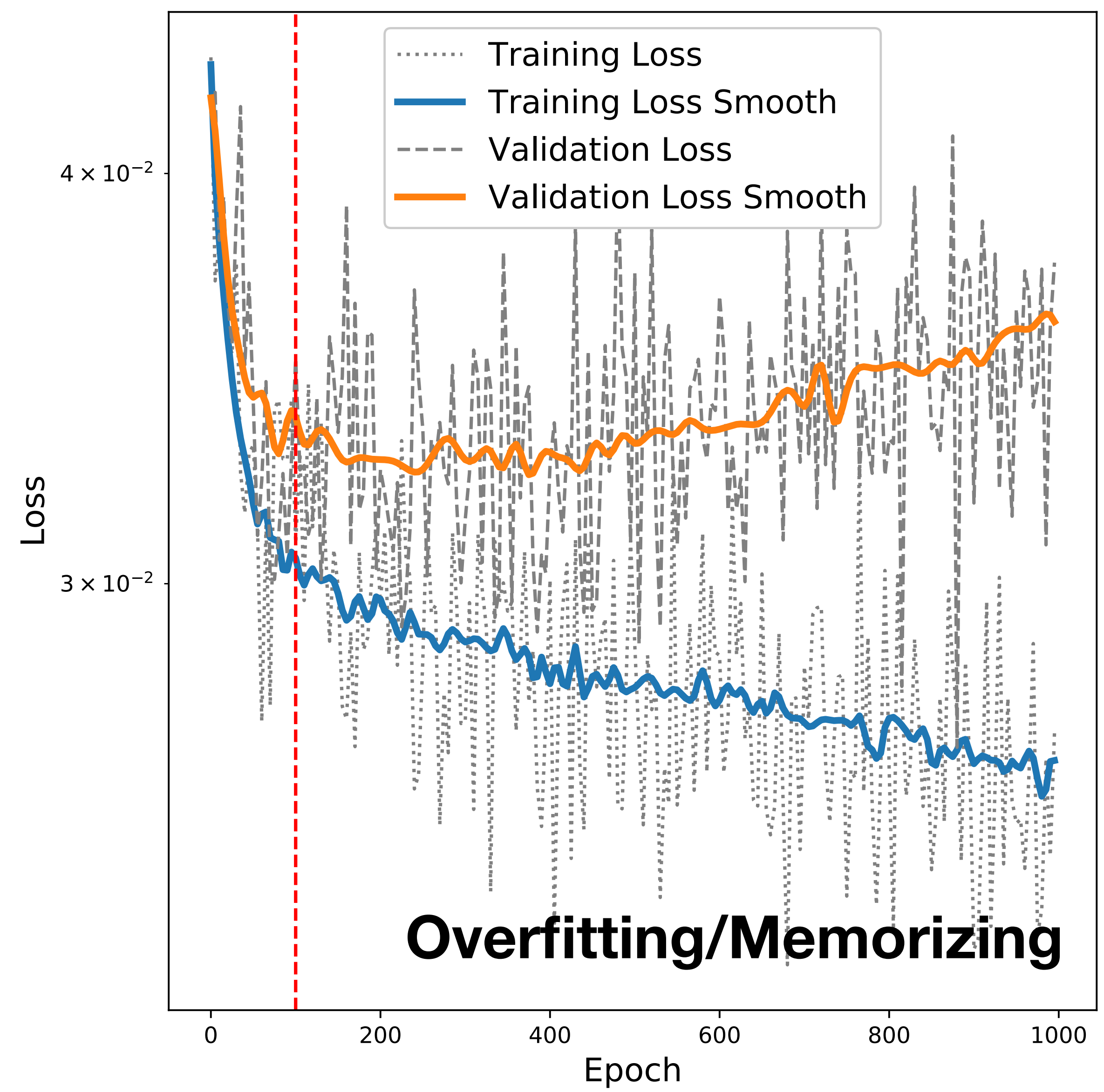
- **Validation** is a technique that is very commonly applied to **truncate** training **before overfitting** occurs.
- Before training begins, a subset of the input data - the **evaluation** set - is isolated and withheld from the training process.
- On a specified schedule - commonly after each training epoch - the **evaluation set** is used to compute the **network loss**.
- The network weights are **not** updated during validation.
- Other performance metrics like **accuracy** and **recall** can also be computed.

Overfitting and Validation

- The loss ℓ_{T} computed **on the training data** should decrease until the network has effectively **memorized the training data**.
- When the training loss plateaus, the network has already overfitted.
- The loss ℓ_{V} computed **on the validation data** will plateau **earlier** than for the training set.
- ℓ_{V} may even **begin increasing** as the network tries to match very fine details of the training data.
- Using validation allows overfitting to be **diagnosed**.

Overfitting and Validation





Mitigating Overfitting

- If your network is overfitting, there are several approaches you can try to combat this.
  - **Simplifying** your network design
    - If there are fewer trainable parameters, then it is more difficult to precisely model the training data.
    - The downside is that your network is less able to model very complicated data.
  - **Regularizing** your network
    - Fundamentally, this involves adding an extra term in the loss function that **penalizes model complexity**.
  - Using **dropout** layers
    - **Randomly discarding** ***different*** subsets of layer outputs **during each iteration**.

Dropout

- Dropout involves **randomly** discarding ***different*** subsets of layer outputs **during each iteration** of the **training** process.
- Effectively this means that a **simpler model** is trained during each iteration. Each **simpler model** is less likely to fit fine details of the training data.
- **All outputs** are **reinstated** at prediction time, so the final model is an "average" of the simpler models.
- The final model represents **robust features** that apply to a **many random samples** of the training data.
- Training using dropout generally requires **more epochs** to converge, but the **computation time for each epoch is lower**.

Transfer Learning

- Training **reliable** deep learning models can require **huge *labeled*** training datasets.
- Often, physicists and astronomers cannot assemble the required training examples.
- **Transfer learning** is an approach that uses a **pre-trained** deep learning network to extract features from **similar** data.
- The extracted features can then be provided to a **custom inference network** to analyze the new data.
- **Fine tuning** of the feature extraction network **requires far fewer training data** than training a network from scratch.

Applied Deep Learning

- Deep learning is rapidly being adopted as a tool in astronomy. Some notable applications include:
  - Automated determination of galaxy morphology (Huertas-Company et al. 2015).
  - Improving galaxy photometric redshift accuracy (Pasquet et al. 2018).
  - Detecting outflows from radio galaxies (Wu et al. 2018).
  - Detecting star-forming clumps in SDSS galaxy images (Nico Adams, Private Communication).