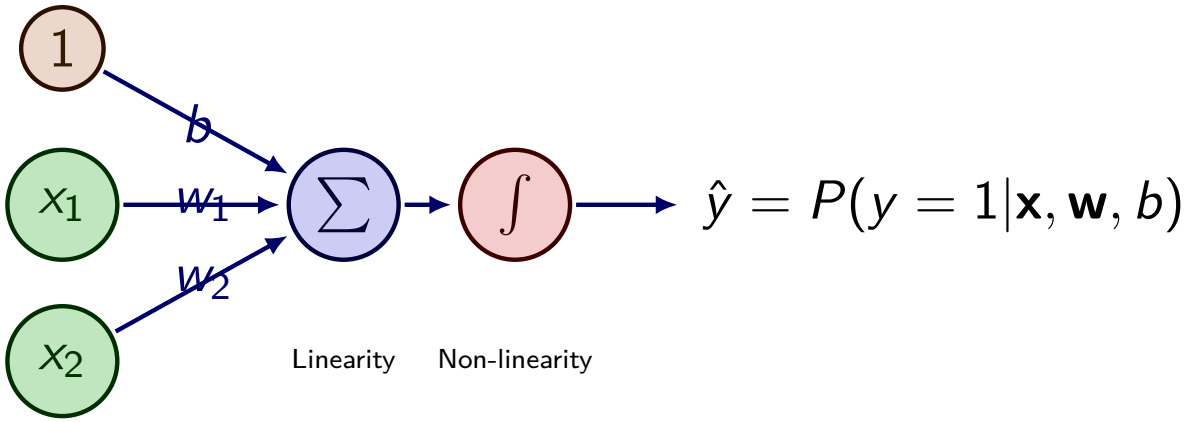
# Neural Networks

Mingfei Sun

Foundations of Machine Learning
The University of Manchester

# Outline

# Logistic Regression

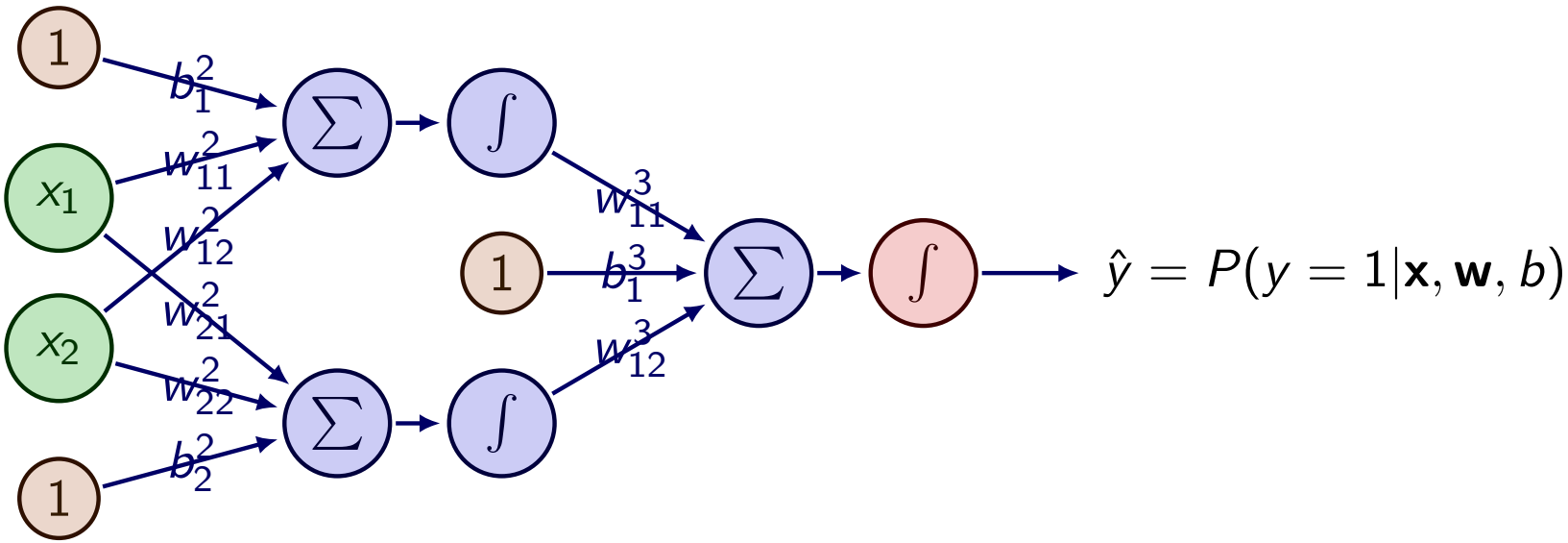



- A unit in a neural network computes a linear function of its input and is then composed with a non-linear *activation* function
- For logistic regression, the non-linear activation function is the *sigmoid*

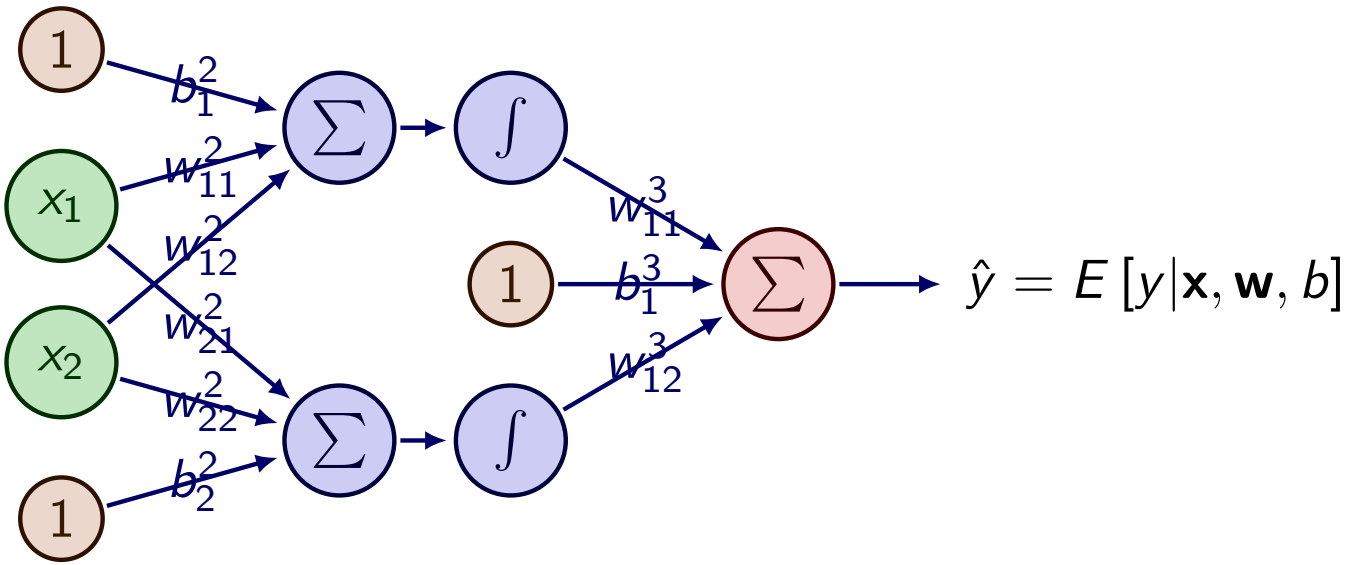
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- The separating surface is linear

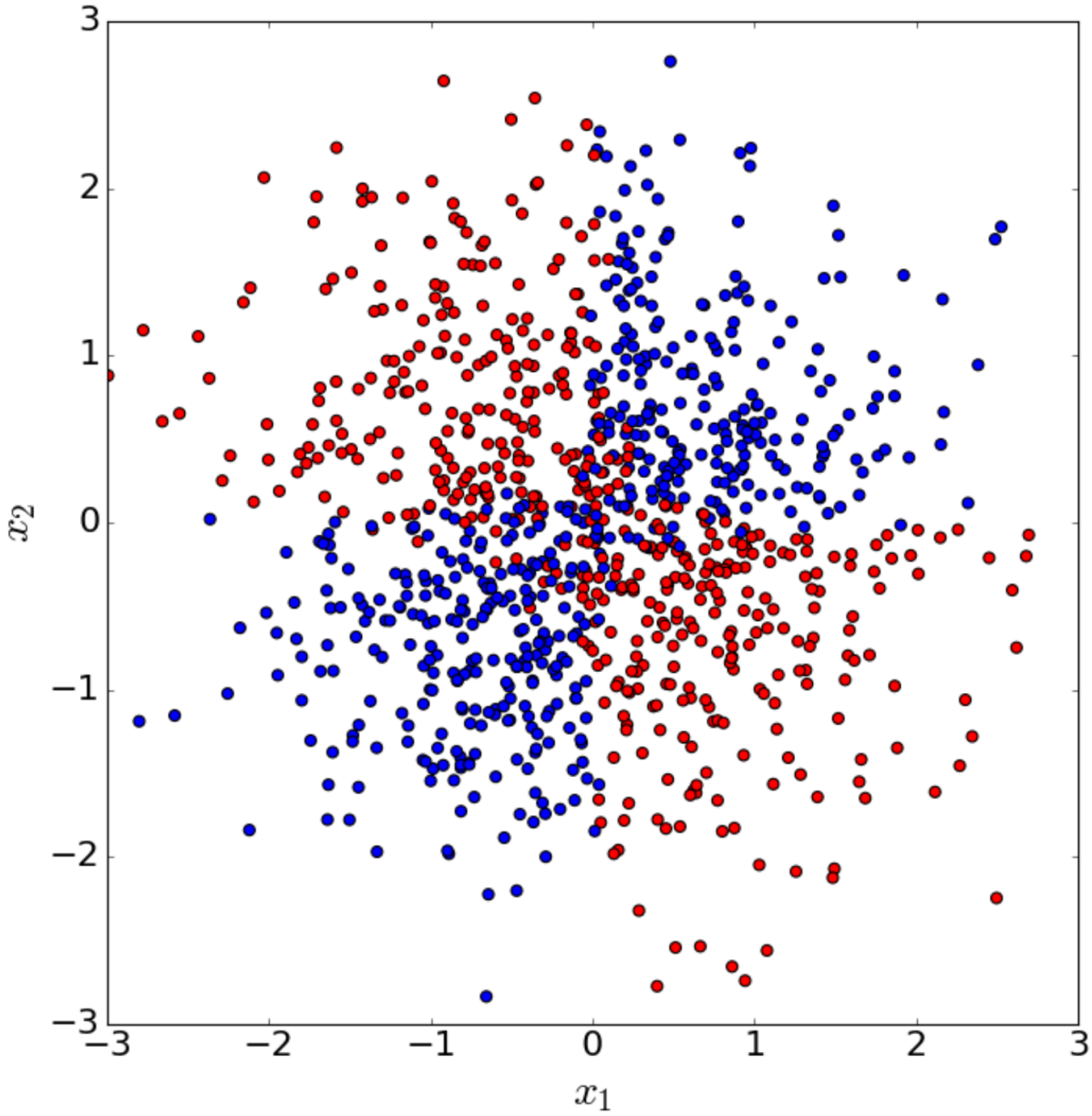
# Multi-Layer Perceptron (MLP): classification



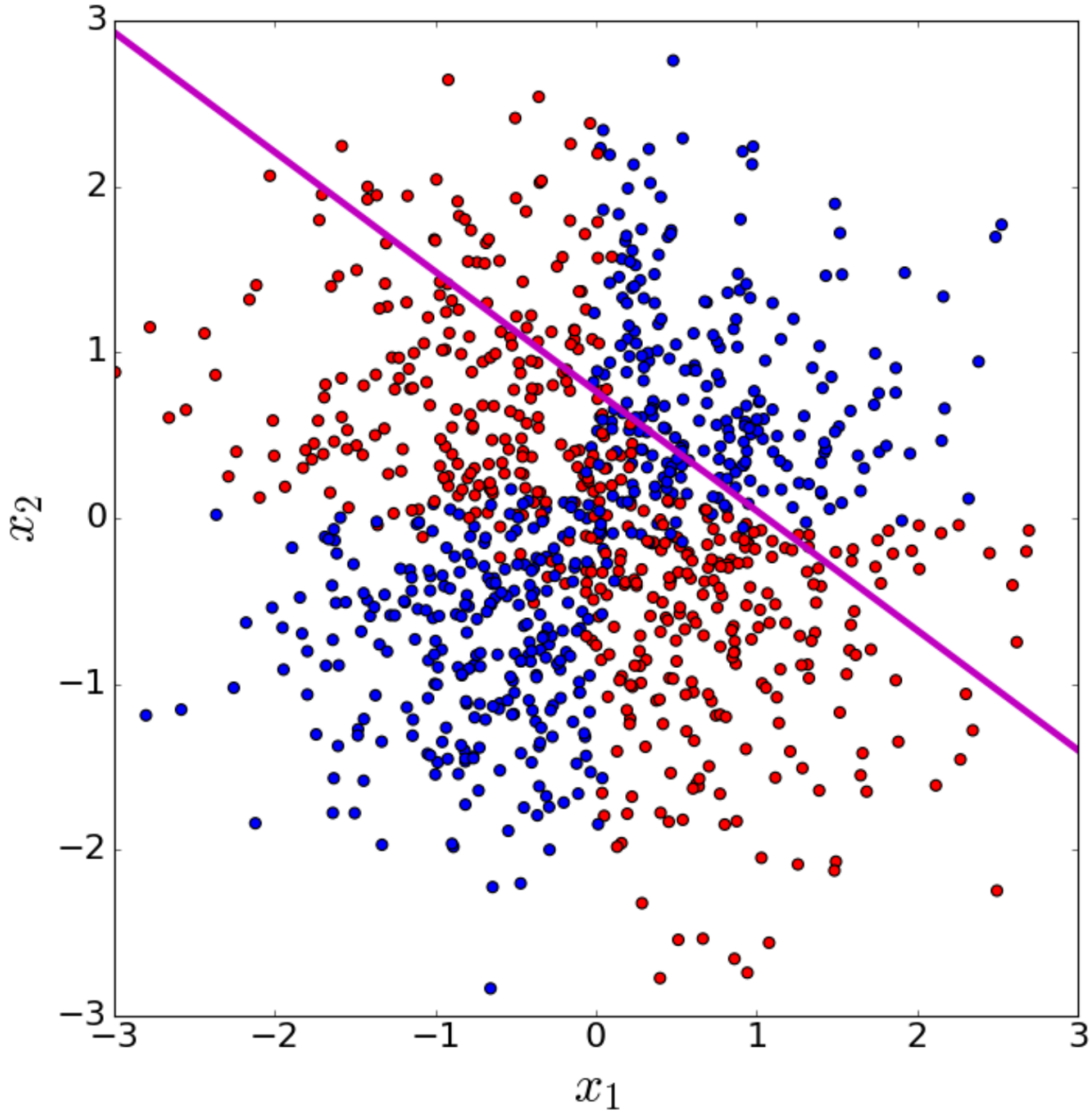
# Multi-Layer Perceptron (MLP): regression



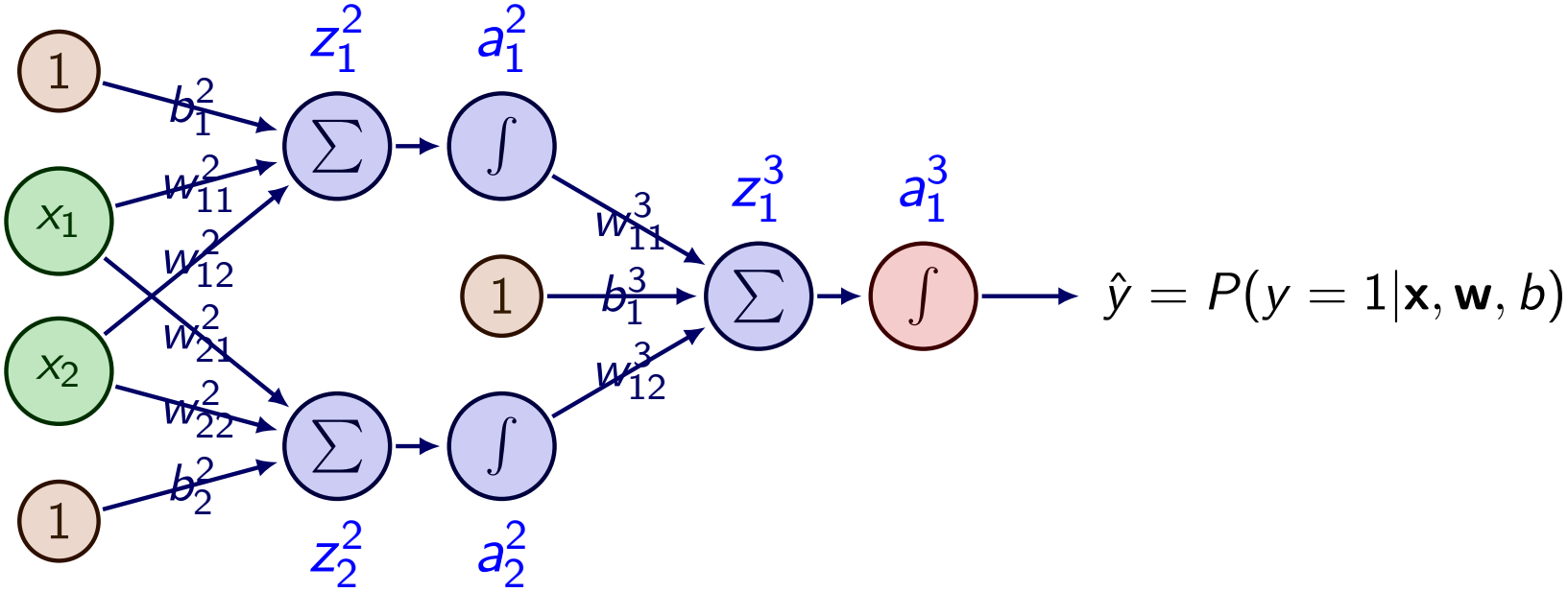
# A Toy Example



# Logistic regression fails badly



# Solve using MLP




Let us use the notation:

$$\mathbf{a}^1 = \mathbf{z}^1 = \mathbf{x}$$

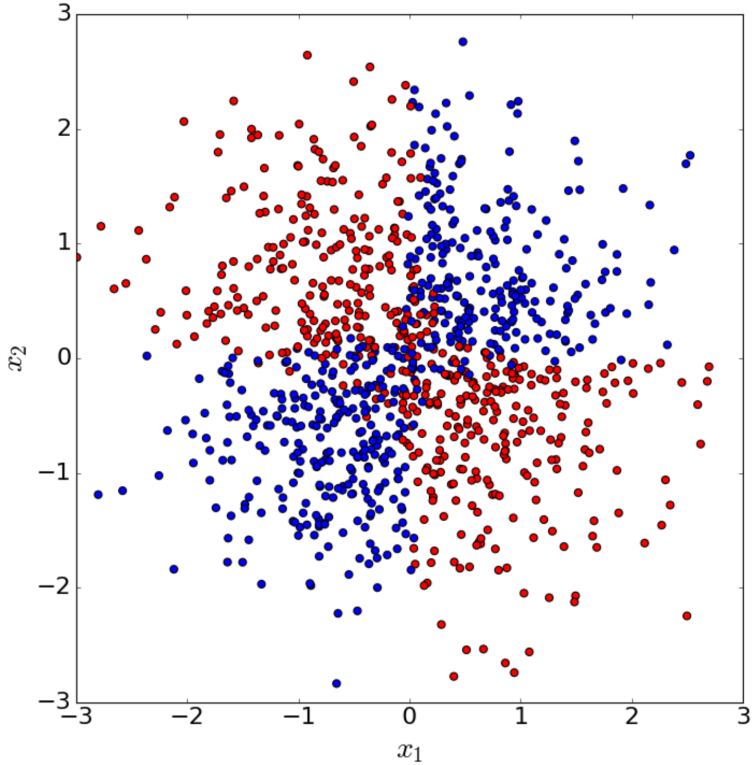
$$\mathbf{z}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2$$

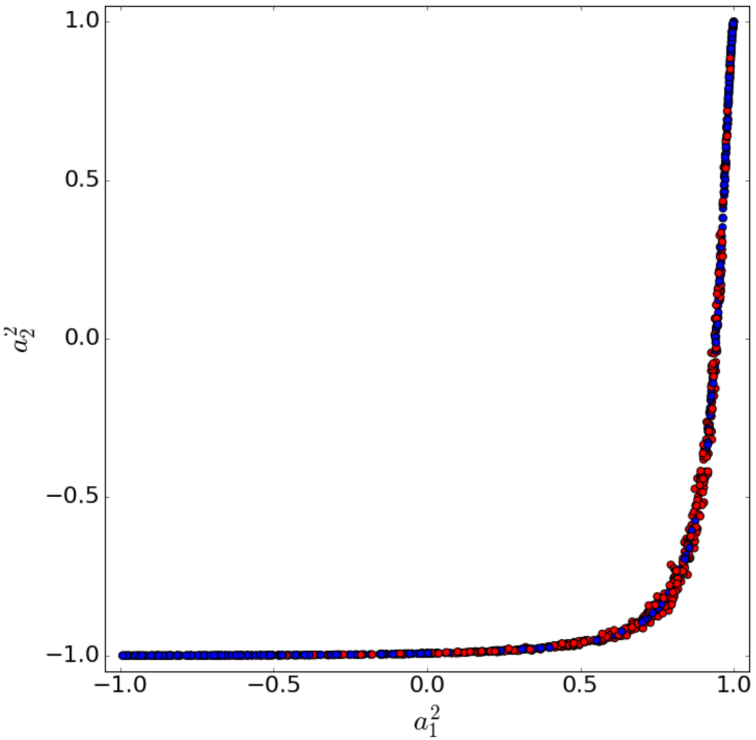
$$\mathbf{a}^2 = \tanh(\mathbf{z}^2) \qquad \text{nonlinearity: } \textbf{tanh}$$

$$z^3 = \mathbf{W}^3\mathbf{a}^2 + b^3$$

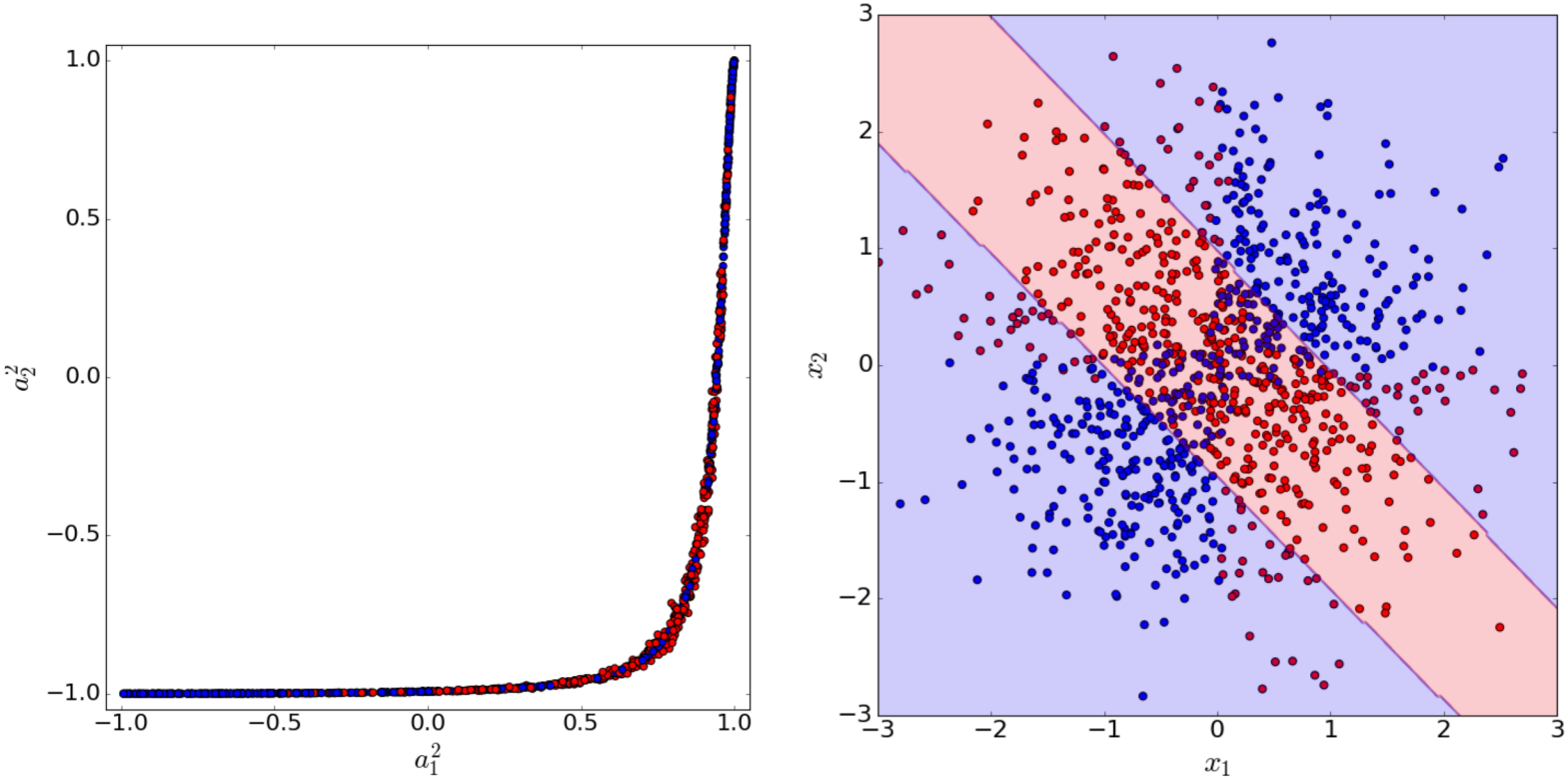
$$y = a^3 = \sigma(z^3) \qquad \text{nonlinearity: } \textbf{sigmoid}$$

# Scatterplot comparison $(x_1, x_2)$ vs $(a_1^2, a_2^2)$

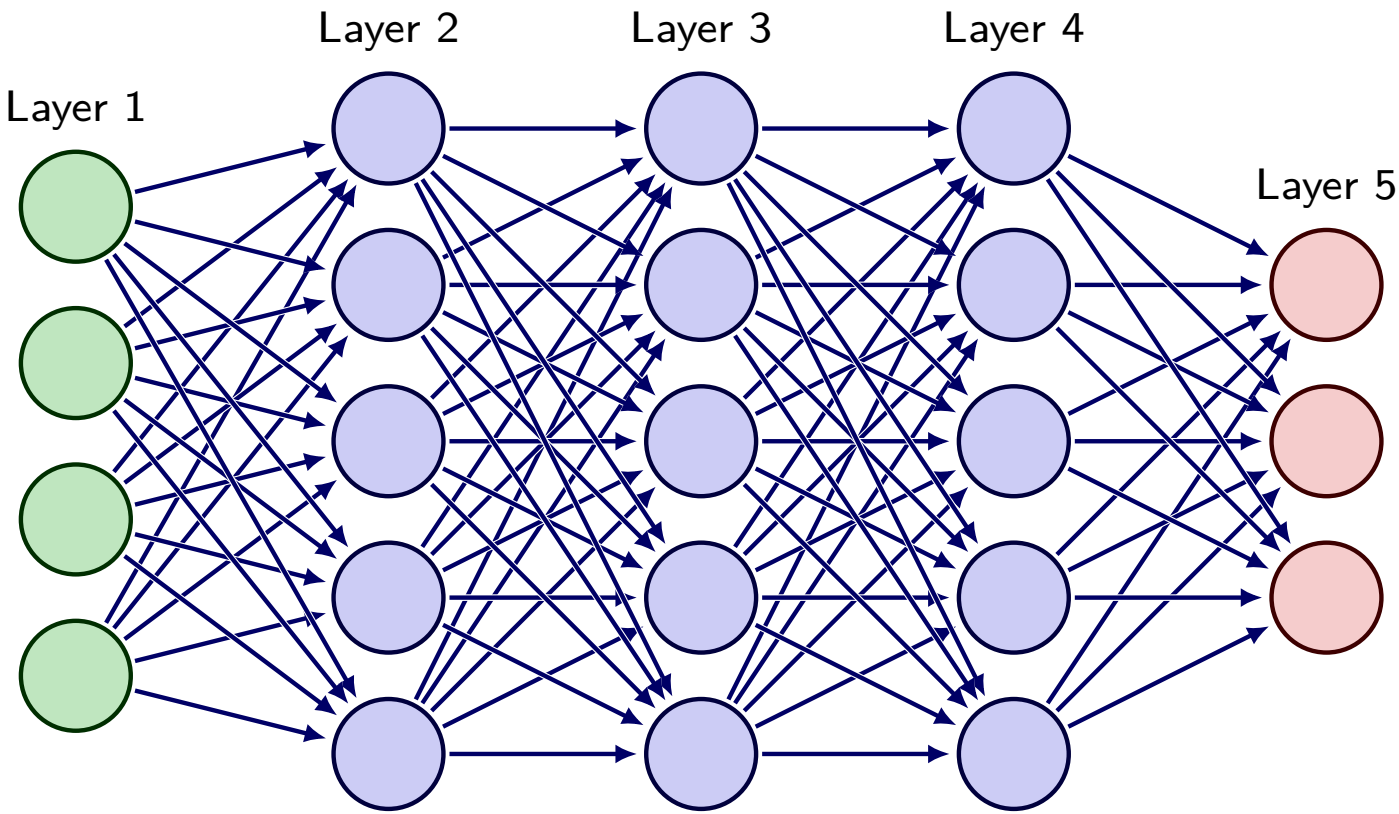


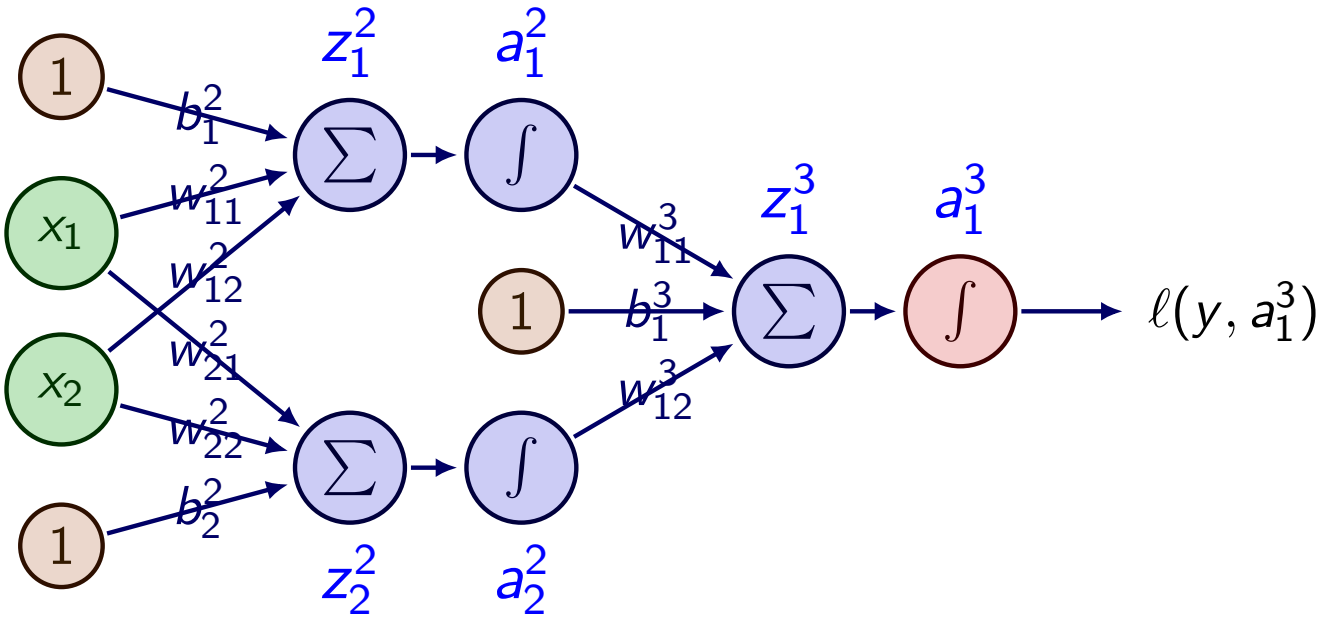
# Decision boundary of the neural net



# Feedforward neural networks



# Compute gradients on a toy example




To minimize an objective function of the form:

$$\mathcal{L}(\mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3; \mathcal{D}) = \sum_{i=1}^{N} \ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3),$$

where $\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3)$ denotes the loss on a single data point,

Want the derivatives: $\frac{\partial \ell}{\partial w_{11}^2}, \frac{\partial \ell}{\partial w_{12}^2}, \frac{\partial \ell}{\partial w_{21}^2}, \frac{\partial \ell}{\partial w_{22}^2}, \frac{\partial \ell}{\partial w_{11}^3}, \frac{\partial \ell}{\partial w_{12}^3}, \frac{\partial \ell}{\partial b_1^2}, \frac{\partial \ell}{\partial b_2^2}, \frac{\partial \ell}{\partial b_1^3}$.

Would suffice to compute $\frac{\partial \ell}{\partial z_1^3}, \frac{\partial \ell}{\partial z_1^2}, \frac{\partial \ell}{\partial z_2^2}$.

# Compute gradients on a toy example

For classification problems, such as this, we can use the cross-entropy loss function,

$$\ell(\mathbf{x}_i, y_i; \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3) = -\left(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right)$$

Let us compute the following:

1. $\frac{\partial \ell}{\partial a_1^3}$
2. $\frac{\partial a_1^3}{\partial z_1^3}$
3. $\frac{\partial z_1^3}{\partial \mathbf{a}^2}$
4. $\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}$

# Compute gradients on a toy example

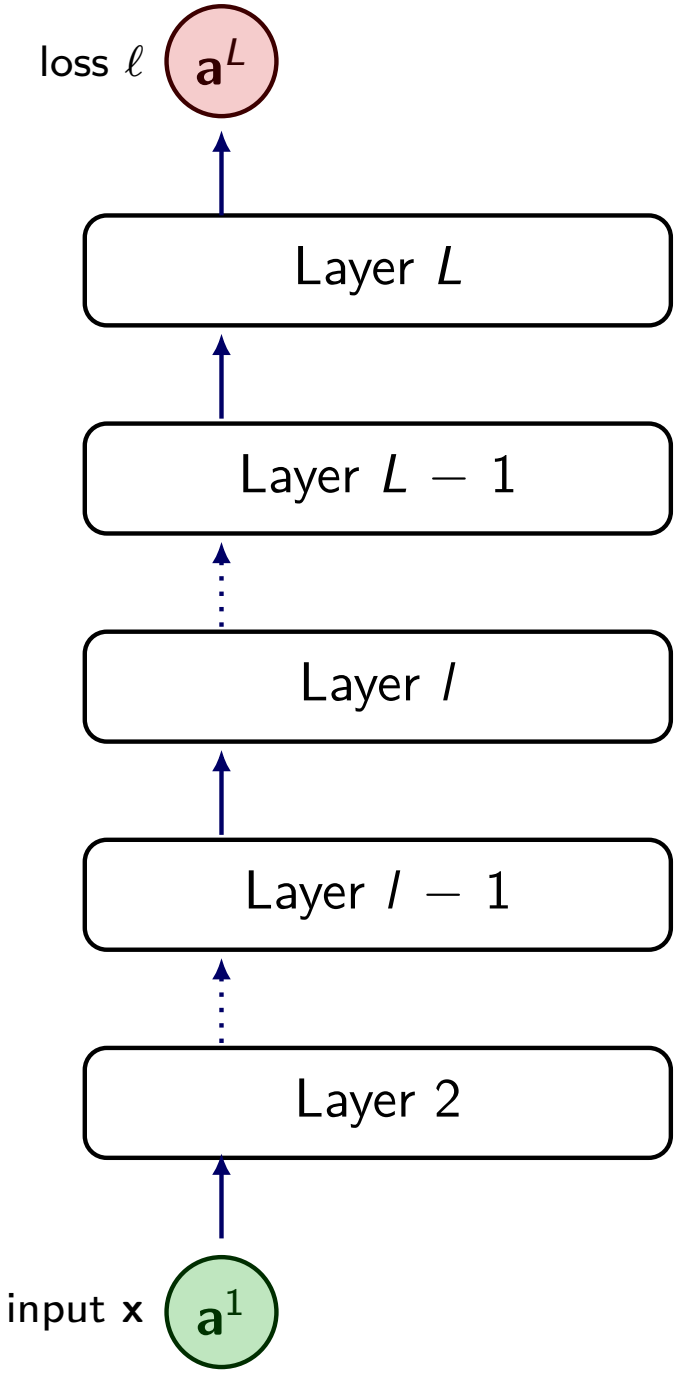
Let us compute the following:

1. $\frac{\partial \ell}{\partial a_1^3} = -\frac{y}{a_1^3} + \frac{1-y}{1-a_1^3} = \frac{a_1^3 - y}{a_1^3(1-a_1^3)}$
2. $\frac{\partial a_1^3}{\partial z_1^3} = a_1^3 \cdot (1 - a_1^3)$
3. $\frac{\partial z_1^3}{\partial \mathbf{a}^2} = [w_{11}^3, w_{12}^3]$
4. $\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} = \begin{bmatrix} 1 - \tanh^2(z_1^2) & 0 \\ 0 & 1 - \tanh^2(z_2^2) \end{bmatrix}$

Then we can calculate

$$\frac{\partial \ell}{\partial z_1^3} = \frac{\partial \ell}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial z_1^3} = a_1^3 - y$$

$$\frac{\partial \ell}{\partial \mathbf{z}^2} = \left( \frac{\partial \ell}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial z_1^3} \right) \cdot \frac{\partial z_1^3}{\partial \mathbf{a}^2} \cdot \frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} = \frac{\partial \ell}{\partial z_1^3} \cdot \frac{\partial z_1^3}{\partial \mathbf{a}^2} \cdot \frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}$$



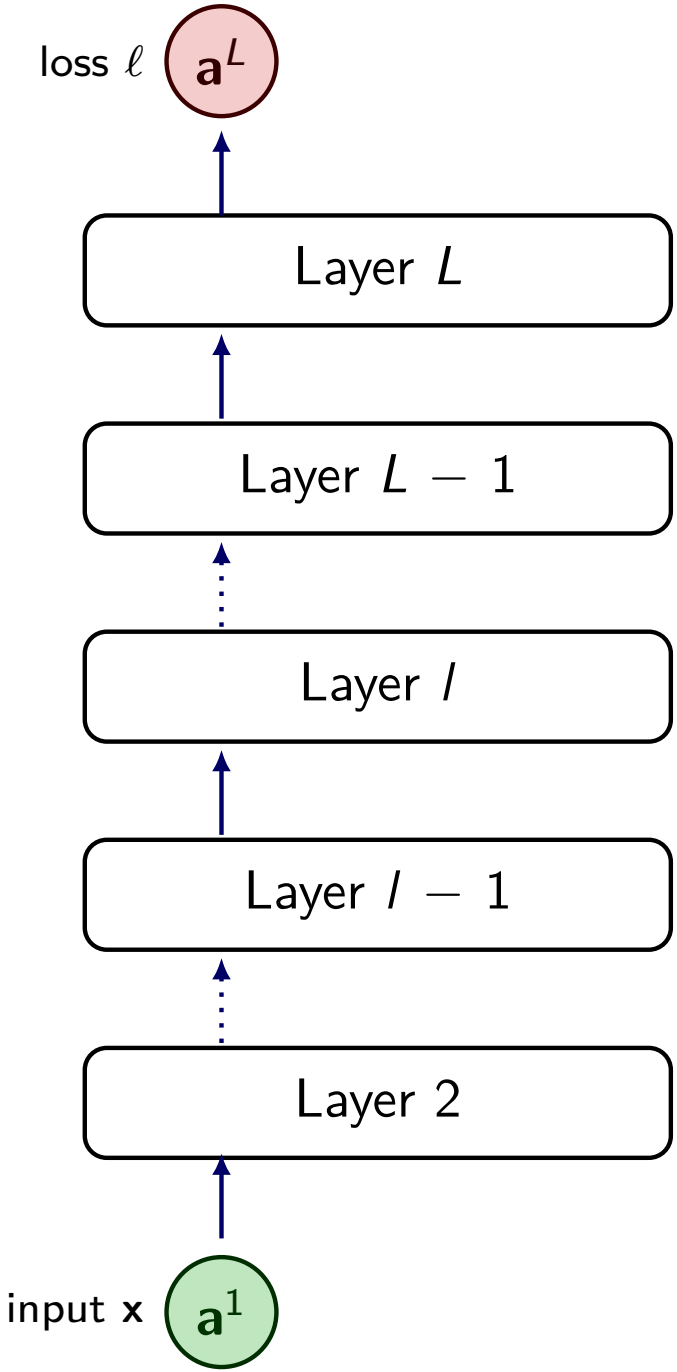

Each layer consists of a linear function and non-linear activation

Layer $l$ consists of the following:

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$
$$\mathbf{a}^l = f_l(\mathbf{z}^l)$$

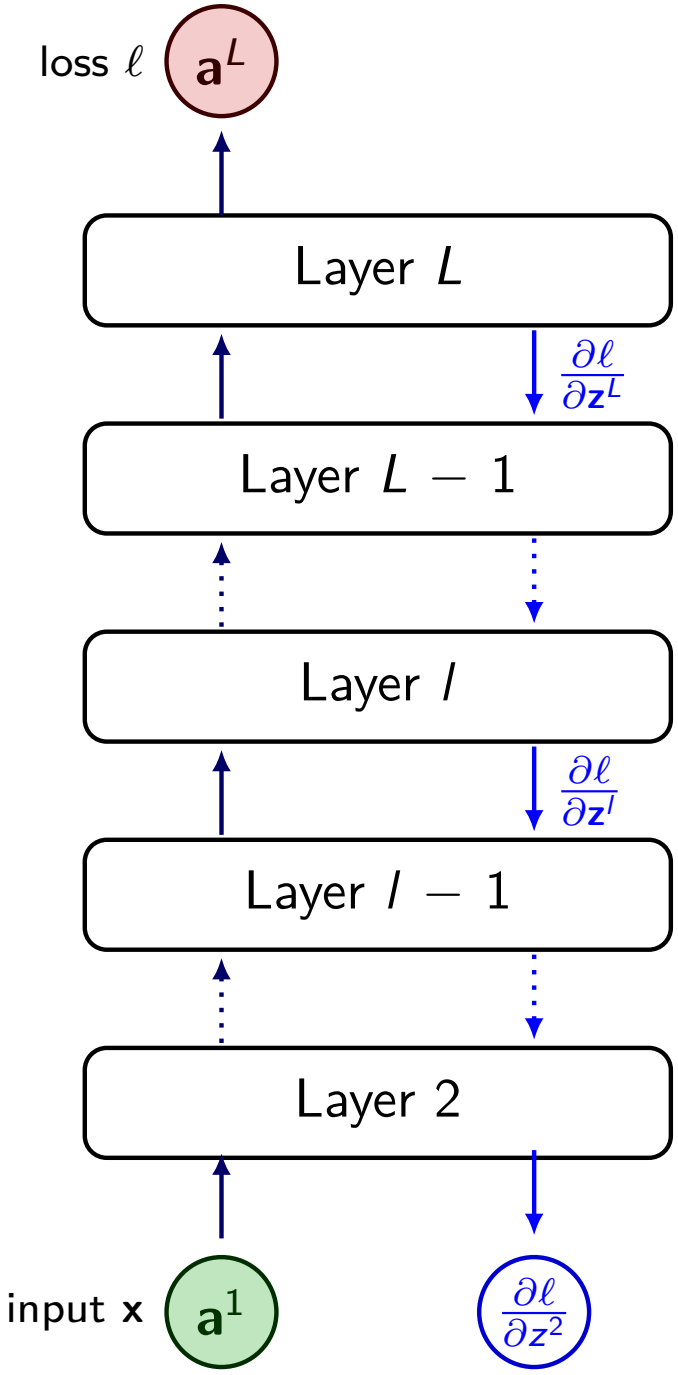
where $f_l$ is the non-linear activation in layer $l$

If there are $n_l$ units in layer $l$, then $\mathbf{W}^l$ is $n_l \times n_{l-1}$



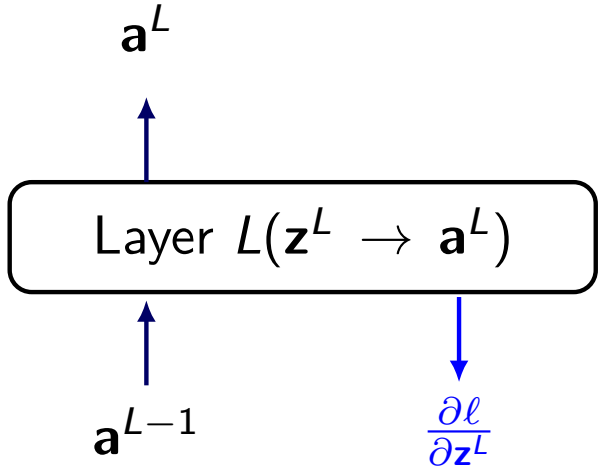

## Forward Equations

1. $\mathbf{a}^1 = \mathbf{x}(\text{input})$
2. $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
3. $\mathbf{a}^l = f_l(\mathbf{z}^l)$
4. $\ell(\mathbf{a}^L, \mathbf{y})$




Backward pass to compute derivatives

# Back propagation: output layer




$$\mathbf{z}^L = \mathbf{W}^L \mathbf{a}^{L-1} + \mathbf{b}^L$$

$$\mathbf{a}^L = f_L(\mathbf{z}^L)$$

Loss: $\ell(\mathbf{a}^L, \mathbf{y})$

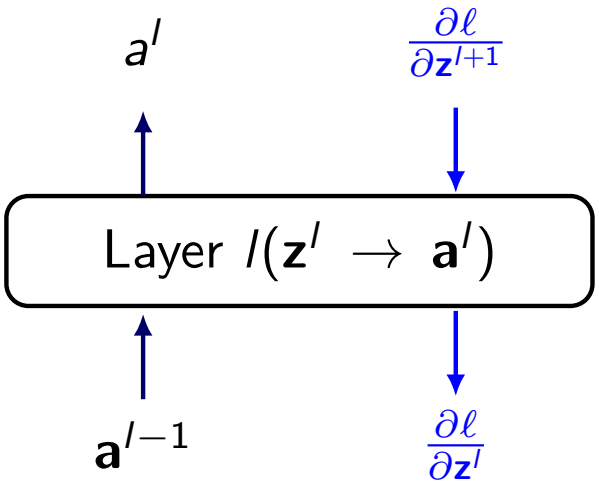
$$\frac{\partial \ell}{\partial \mathbf{z}^L} = \frac{\partial \ell}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$$

If there are $n_L$ (output) units in Layer $L$, then $\frac{\partial \ell}{\partial \mathbf{a}^L}$ and $\frac{\partial \ell}{\partial \mathbf{z}^L}$ are row vectors with $n_L$ elements and $\frac{\partial a^L}{\partial z^L}$ is the $n_L \times n_L$ Jacobian matrix:

$$\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \begin{bmatrix} \frac{\partial a_1^L}{\partial z_1^L} & \frac{\partial a_1^L}{\partial z_2^L} & \cdots & \frac{\partial a_1^L}{\partial z_{n_L}^L} \\ \frac{\partial a_2^L}{\partial z_1^L} & \frac{\partial a_2^L}{\partial z_2^L} & \cdots & \frac{\partial a_2^L}{\partial z_{n_L}^L} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_{n_L}^L}{\partial z_1^L} & \frac{\partial a_{n_L}^L}{\partial z_2^L} & \cdots & \frac{\partial a_{n_L}^L}{\partial z_{n_L}^L} \end{bmatrix}$$

If $f_L$ is applied element-wise, e.g., sigmoid, then this matrix is diagonal

# Back propagation: hidden layer




$$\mathbf{a}^l$$

the inputs into layer $l+1$

$$\mathbf{z}^{l+1} = \mathbf{W}^{l+1}\mathbf{a}^l + \mathbf{b}^{l+1}$$

$w_{j,k}^{l+1}$ weight on connection from $k$-th unit in layer $l$ to $j$-th unit in layer $l+1$

$$\mathbf{a}^l = f_l(\mathbf{z}^l)$$

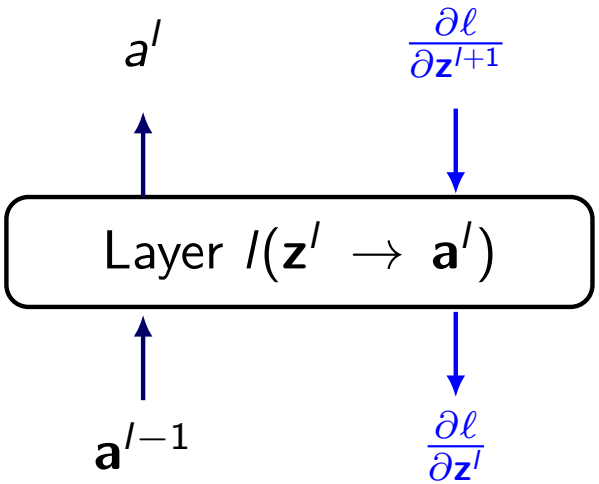
$f_l$ is a non-linearity function

$$\frac{\partial \ell}{\partial \mathbf{z}^{l+1}}$$

derivative passed from layer above

$$\begin{aligned}\frac{\partial \ell}{\partial \mathbf{z}^l} &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l} \\ &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l} \\ &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \mathbf{W}^{l+1} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\end{aligned}$$

# Gradients with respect to parameters




$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$

$w^l_{j,k}$ weight on connection from $k$-th unit in layer $l-1$ to $j$-th unit in layer $l$

$$\frac{\partial \ell}{\partial \mathbf{z}^l}$$

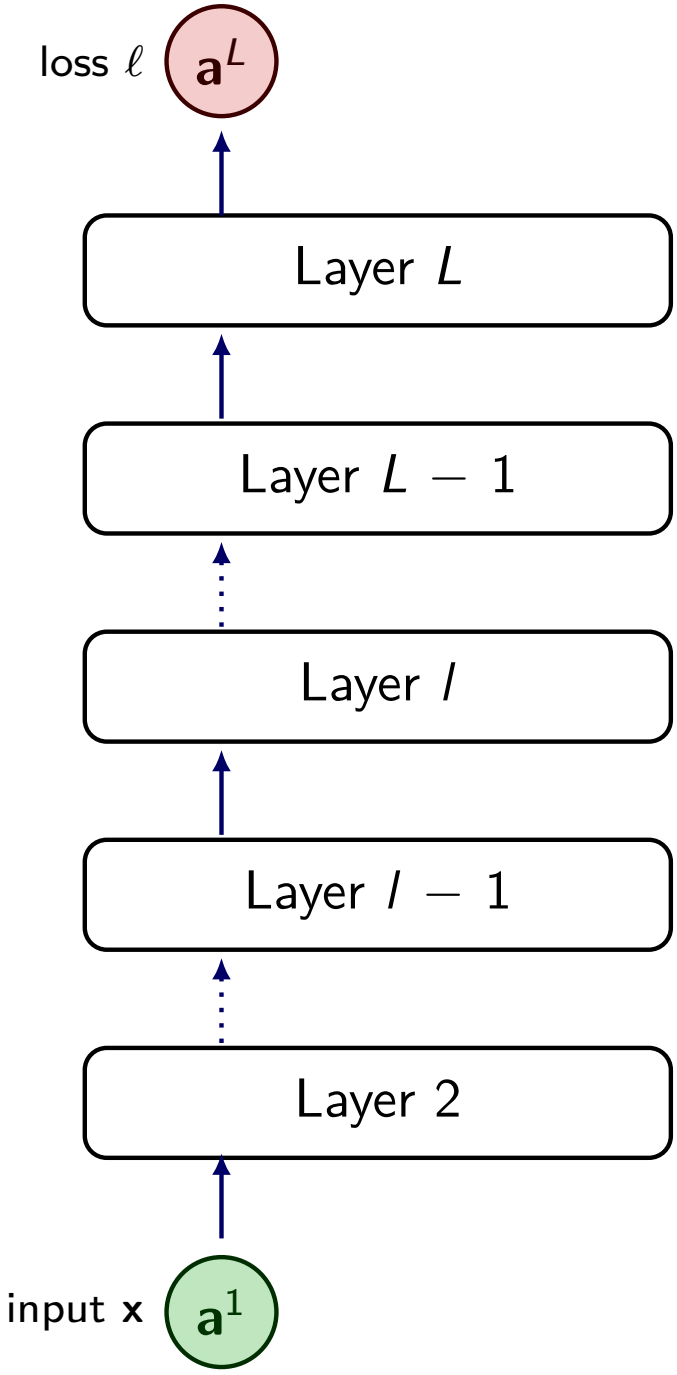
obtained using backpropagation

Consider

$$\frac{\partial \ell}{\partial w^l_{ij}} = \frac{\partial \ell}{\partial z^l_i} \cdot \frac{\partial z^l_i}{\partial w^l_{ij}} = \frac{\partial \ell}{\partial z^l_i} \cdot a^{l-1}_j \qquad \frac{\partial \ell}{\partial b^l_i} = \frac{\partial \ell}{\partial z^l_i}$$

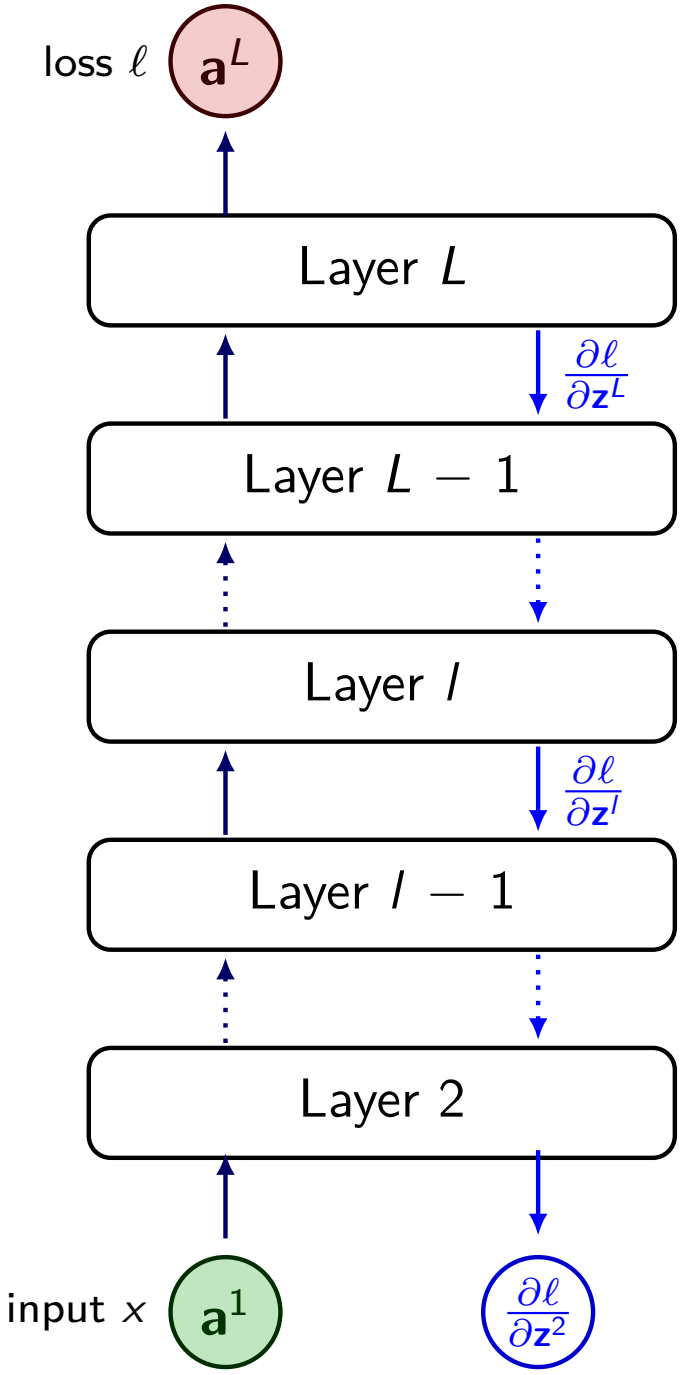
More succinctly, we may write

$$\frac{\partial \ell}{\partial \mathbf{W}^l} = \left( \mathbf{a}^{l-1} \frac{\partial \ell}{\partial \mathbf{z}^l} \right) \qquad \frac{\partial l}{\partial \mathbf{b}^l} = \frac{\partial l}{\partial \mathbf{z}^l}$$




**Forward equations**

1. $\mathbf{a}^1 = \mathbf{x}(\text{input})$
2. $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
3. $\mathbf{a}^l = f_l(\mathbf{z}^l)$
4. $\ell(\mathbf{a}^L, \mathbf{y})$




## Forward equations

1. $\mathbf{a}^1 = \mathbf{x}(\text{input})$
2. $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
3. $\mathbf{a}^l = f_l(\mathbf{z}^l)$
4. $\ell(\mathbf{a}^L, \mathbf{y})$

## Back-propagation equations

1. Compute $\frac{\partial \ell}{\partial \mathbf{z}^L} = \frac{\partial \ell}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$
2. $\frac{\partial \ell}{\partial \mathbf{z}^l} = \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \cdot \mathbf{W}^{l+1} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}$
3. $\frac{\partial \ell}{\partial \mathbf{W}^l} = \left(\mathbf{a}^{l-1} \frac{\partial \ell}{\partial \mathbf{z}^l}\right)$
4. $\frac{\partial l}{\partial \mathbf{b}^l} = \frac{\partial l}{\partial \mathbf{z}^l}$

# Computational questions

What is the running time to compute the gradient for a single data point?

- As many matrix multiplications as there are fully connected layers
- Performed twice during forward and backward pass

What is the space requirement?

- Need to store vectors $\mathbf{a}^l$, $\mathbf{z}^l$, and $\frac{\partial \ell}{\partial \mathbf{z}^l}$ for each layer

Can we process multiple examples together?

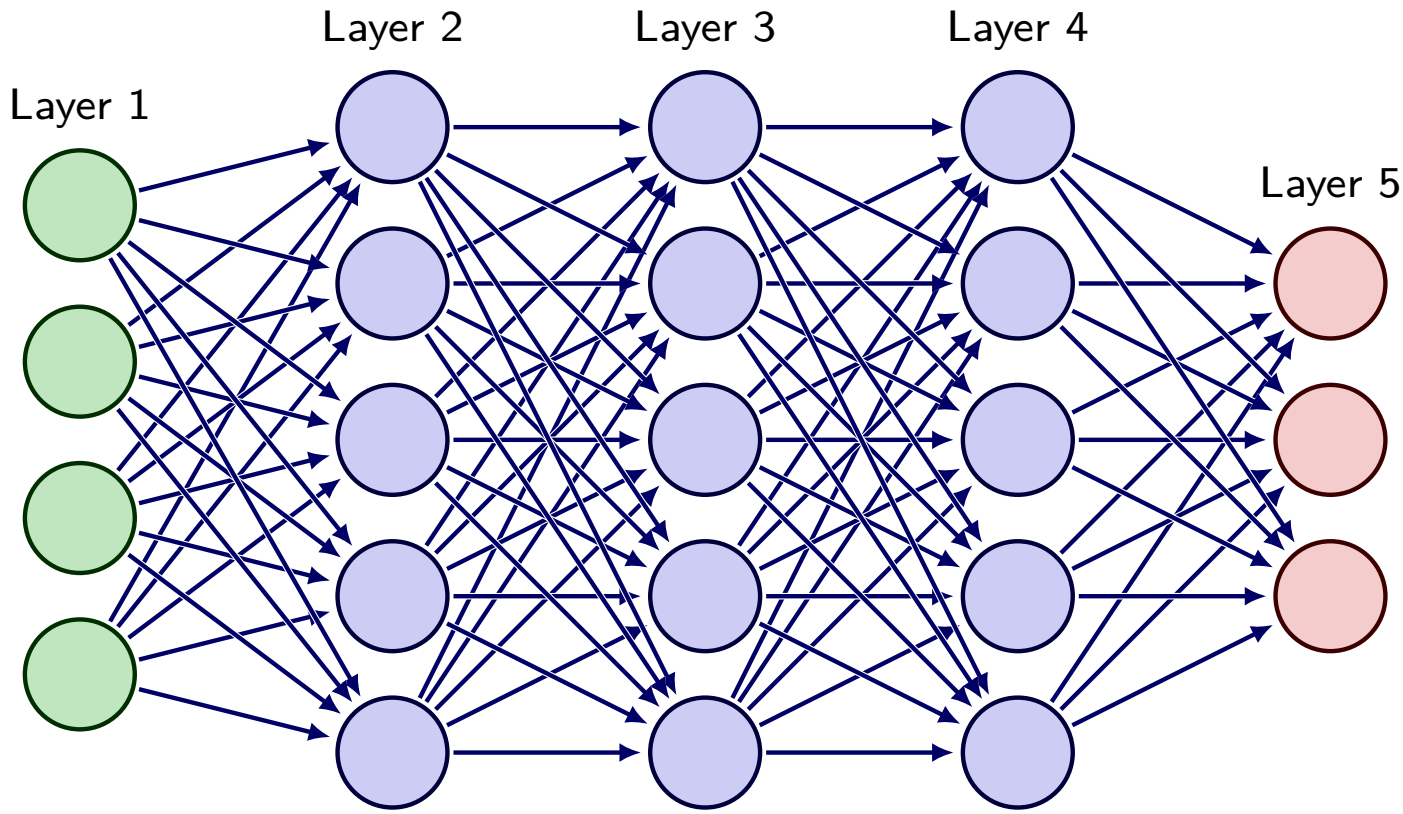
- Yes, if we minibatch, we perform tensor operations
- Make sure that all parameters fit in GPU memory

# Outline

# Training Deep Neural Networks

- Back-propagation gives gradient
- Stochastic gradient descent is the method of choice
- Regularization
  - How do we add $\ell_1$ or $\ell_2$ regularization?
  - Don't regularize bias terms
- How about convergence?
- What did we learn in the last 10 years, that we didn't know in the 80s?

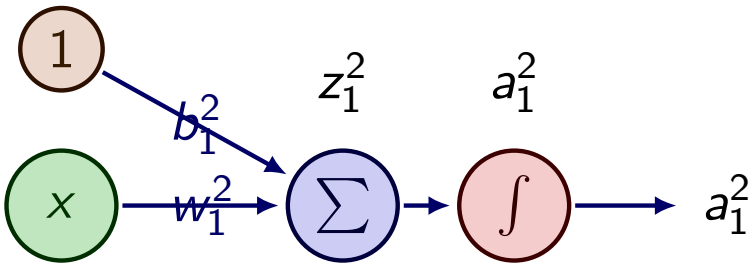
# Training Feedforward Deep Networks




Why do we get non-convex optimization problem?

All units in a layer are symmetric, hence invariant to permutations

# A toy example to illustrate saturation



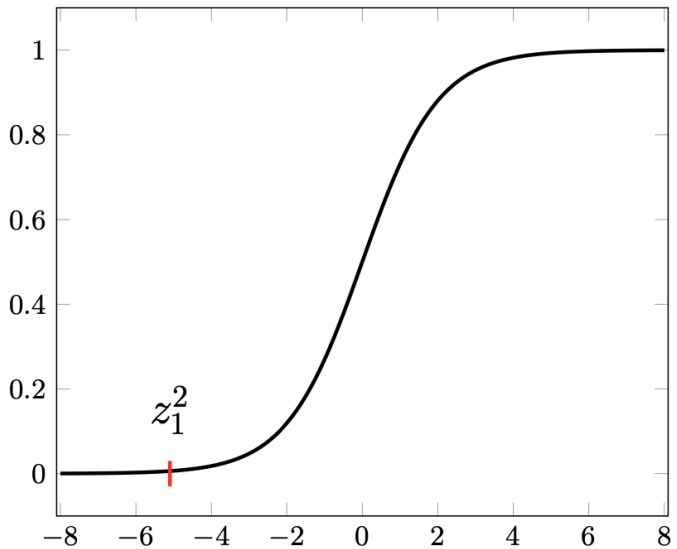

Target is $y = \frac{1-x}{2}$ and $x \in \{1, -1\}$

## Squared Loss Function

$$\ell(a_1^2, y) = (a_1^2 - y)^2$$

$$\frac{\partial \ell}{\partial z_1^2} = 2(a_1^2 - y) \cdot \frac{\partial a_1^2}{\partial z_1^2} = 2(a_1^2 - y)\sigma'(z_1^2)$$

If $x = -1$, $w_1^2 \approx 5$, $b_1^2 \approx 0$, then $\sigma'(z_1^2) \approx 0$

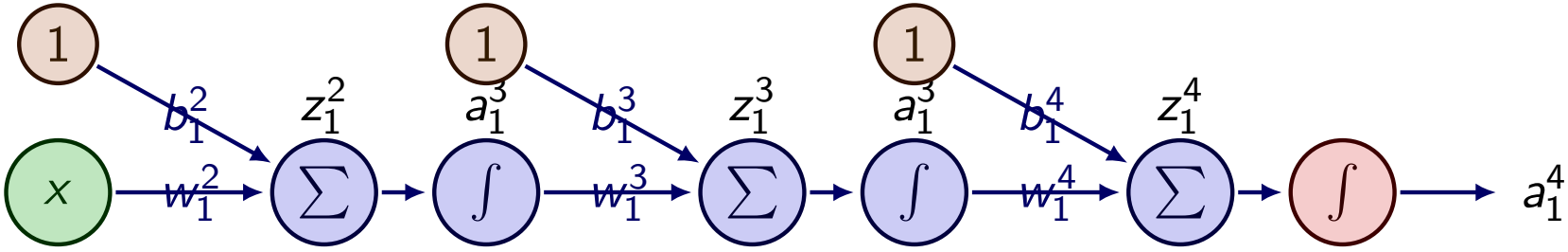



## Cross-Entropy Loss Function

$$\ell(a_1^2, y) = -\left(y \log a_1^2 + (1 - y) \log(1 - a_1^2)\right)$$

$$\frac{\partial \ell}{\partial z_1^2} = \frac{a_1^2 - y}{a_1^2(1 - a_1^2)} \cdot \frac{\partial a_1^2}{\partial z_1^2} = (a_1^2 - y)$$

# Vanishing Gradients




- Cross entropy loss: $\ell(a_1^4, y) = -\left(y \log a_1^4 + (1-y)\log(1-a_1^4)\right)$
  - $\frac{\partial \ell}{\partial z_1^4} = a_1^4 - y$
  - $\frac{\partial \ell}{\partial z_1^3} = \frac{\partial \ell}{\partial z_1^4} \cdot \frac{\partial z_1^4}{\partial a_1^3} \cdot \frac{\partial a_1^4}{\partial z_1^3} = (a_1^4 - y) \cdot w_1^4 \cdot \sigma'(z_1^3)$
  - $\frac{\partial \ell}{\partial z_1^2} = \frac{\partial \ell}{\partial z_1^3} \cdot \frac{\partial z_1^3}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} = (a_1^4 - y) \cdot w_1^4 \cdot \sigma'(z_1^3) \cdot w_1^3 \cdot \sigma'(z_1^2)$
- Saturation: When the output of an artificial neuron is in the 'flat' part, e.g., where $\sigma'(z) \approx 0$ for sigmoid
- Vanishing Gradient Problem: Multiplying several $\sigma'(z_i^l)$ together makes the gradient $\approx 0$, when we have a large number of layers
- For example, when using sigmoid activation, $\sigma'(z) \in [0, 1/4]$

# Avoiding Saturation

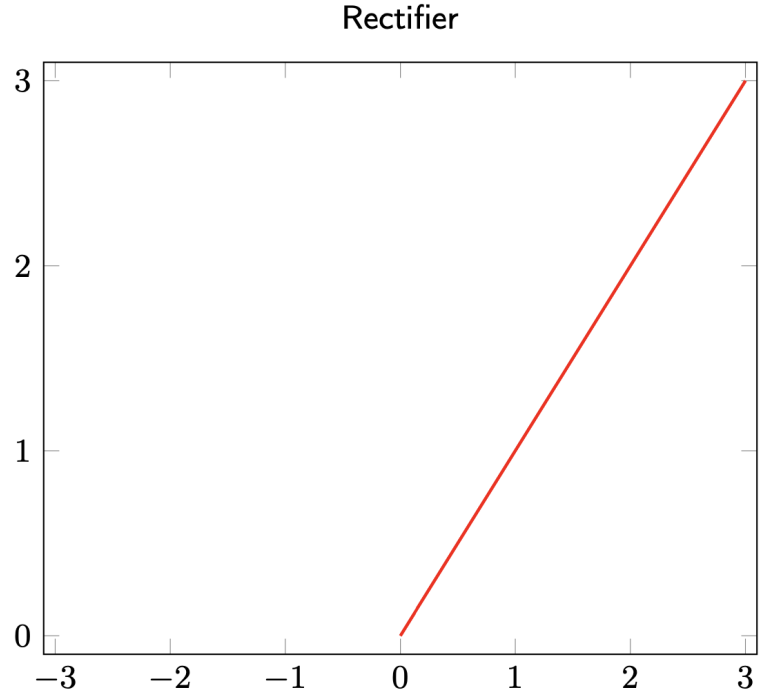
Use *rectified linear units*

Rectifier non-linearity
$f(\mathbf{z}) = \max(0, \mathbf{z})$

Rectified Linear Unit (ReLU)
$\max(0, \mathbf{Wa} + \mathbf{b})$

You can also use $f(\mathbf{z}) = |\mathbf{z}|$

**Other variants**: leaky ReLUs, parametric ReLUs



# Initializing Weights and Biases

Initializing is important when minimizing non-convex functions.
We may get very different results depending on where we start the optimization.
Suppose we were using a *sigmoid unit*, how would you initialize the weights?

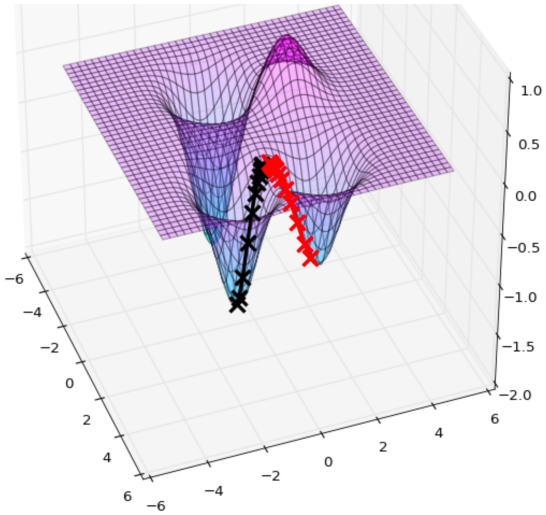
- Suppose $z = \sum_{i=1}^{D} w_i a_i$
- E.g., choose $w_i \in \left[-\frac{1}{\sqrt{D}}, \frac{1}{\sqrt{D}}\right]$ at random

What if it were a ReLU unit?

- You can initialize similarly

How about the biases?

- For sigmoid, can use 0 or a random value around 0
- For ReLU, should use a small positive constant



# Avoiding Overfitting

## Deep Neural Networks have a lot of parameters

- Fully connected layers with $n_1, n_2, ..., n_L$ units have at least $n_1 n_2 + n_2 n_3 + ... + n_{L-1} n_L$ parameters
- A typical training of an MLP for digit recognition with 2 million parameters and only 60,000 training images
- For image detection, one of the most famous models, the neural net used by Krizhevsky, Sutskever, Hinton (2012) has 60 million parameters and 1.2 million training images
- How do we prevent deep neural networks from overfitting?

# Early Stopping

Maintain validation set and stop training when error on validation set stops decreasing

What are the computational costs?

- Need to compute validation error
- Can do this every few iterations to reduce overhead

What are the advantages?

- If validation error flattens or starts increasing, can stop optimization
- Prevents overfitting

See paper by Hardt, Recht and Singer (2015)

# Add Data: Modified Data

Typically, getting additional data is either impossible or expensive

Fake the data!

Images can be translated slightly, rotated slightly, change of brightness, etc

Google Offline Translate trained on entirely fake data!

[1]

[1] Google Research Blog: https://blog.research.google/2015/07/how-google-translate-squeezes-deep.html

# Add Data: Adversarial Training

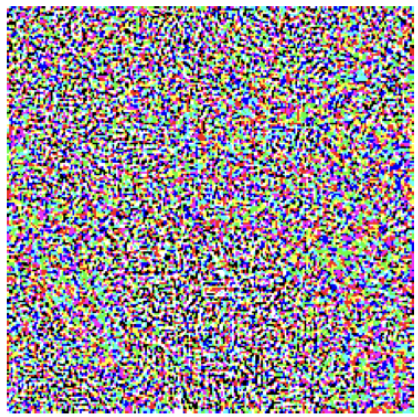
Take trained (or partially trained model)

Create examples by modifications "imperceptible to the human eye", but where the model fails

$\boldsymbol{x}$

"panda"
57.7% confidence

$+ .007 \times$



$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
8.2% confidence

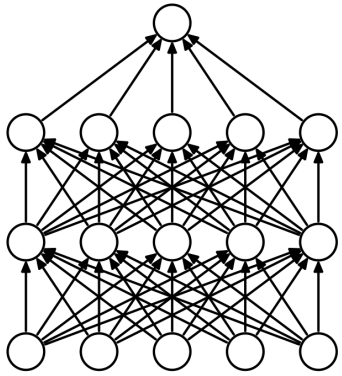
$=$

$\boldsymbol{x} + \epsilon \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

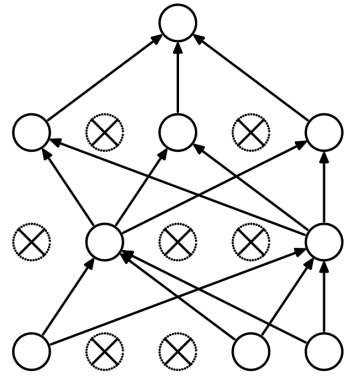
"gibbon"
99.3 % confidence

See paper by Goodfellow, Shlens and Szegedy (2014)

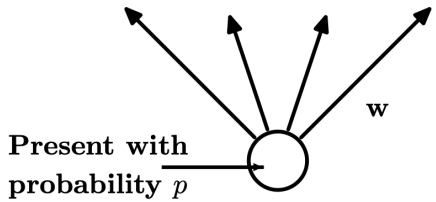
# Dropout



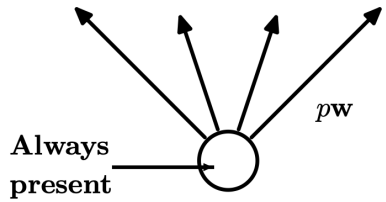
(a) Standard Neural Net



(b) After applying dropout.




(a) At training time




(b) At test time

- For input **x** drop each hidden unit with probability 1/2 independently
- Every input will have a potentially different mask
- Potentially exponentially different models, but have "same weights"
- After training whole network is used by having all the weights

See paper by Srivastava, Hinton, Krizhevsky (2014)

# Other Ideas to Reduce Overfitting

- Hard constraints on weights
- Gradient Clipping
- Inject noise into the system
- Unsupervised Pre-training
- Enforce sparsity in the neural network

# Outline

# Convolutional Neural Networks (convnets)

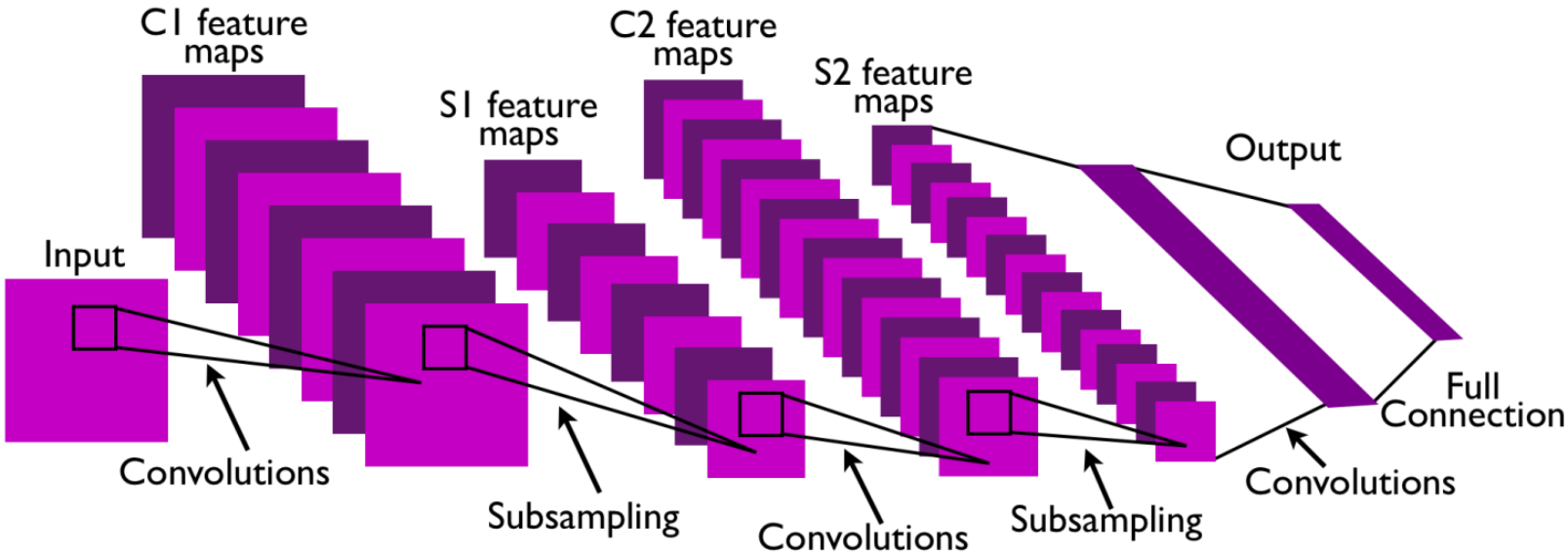
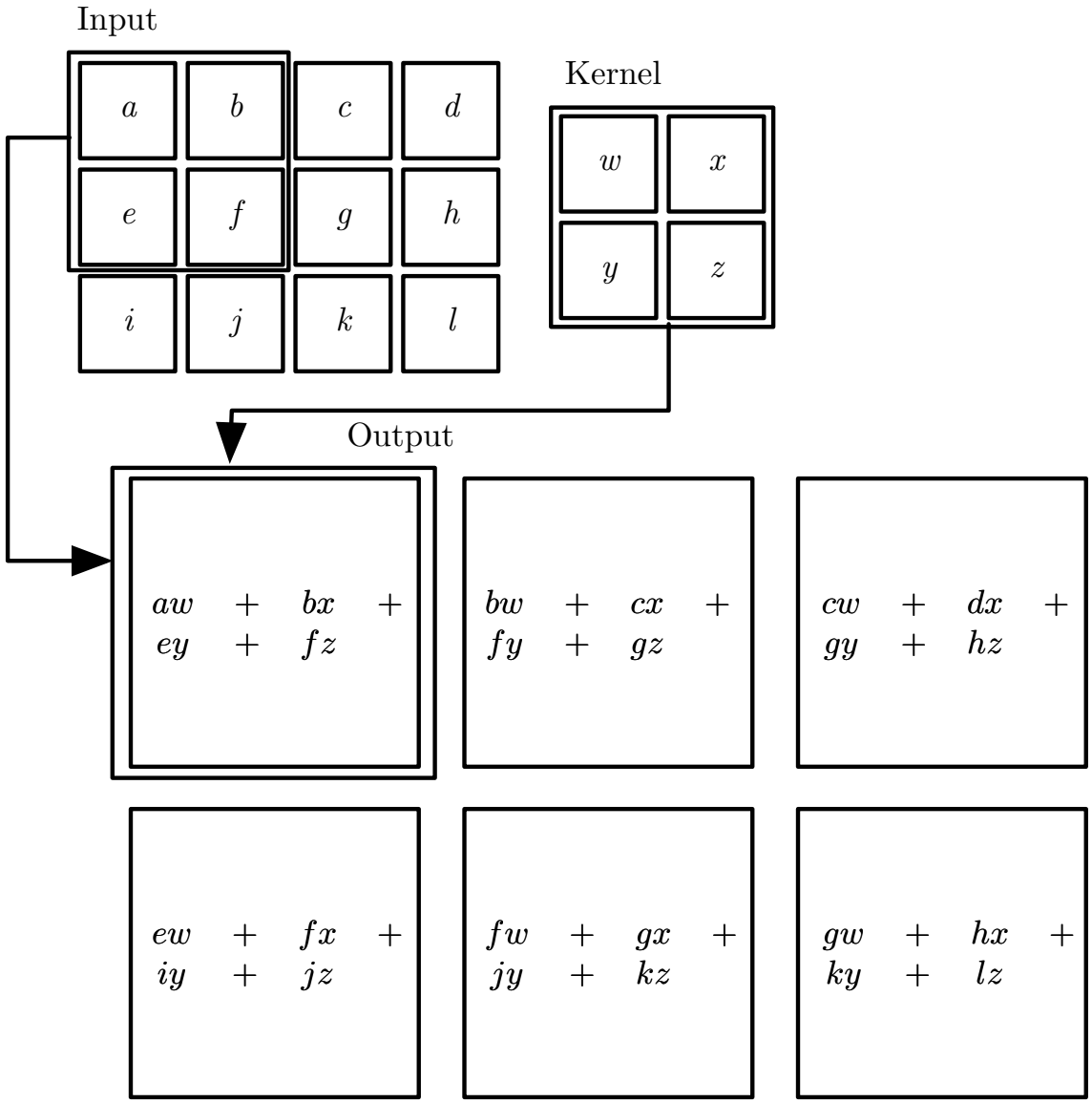



Fig. 1. A typical ConvNet architecture with two feature stages

See paper by LeCun, Kavukcuoglu, Farabet (2010)

# Convolution Operation

## 2D convolution



# Convolution

In general, a convolution filter $f$ is a tensor of dimension $W_f \times H_f \times F_l$, where $F_l$ is the number of channels in the previous layer

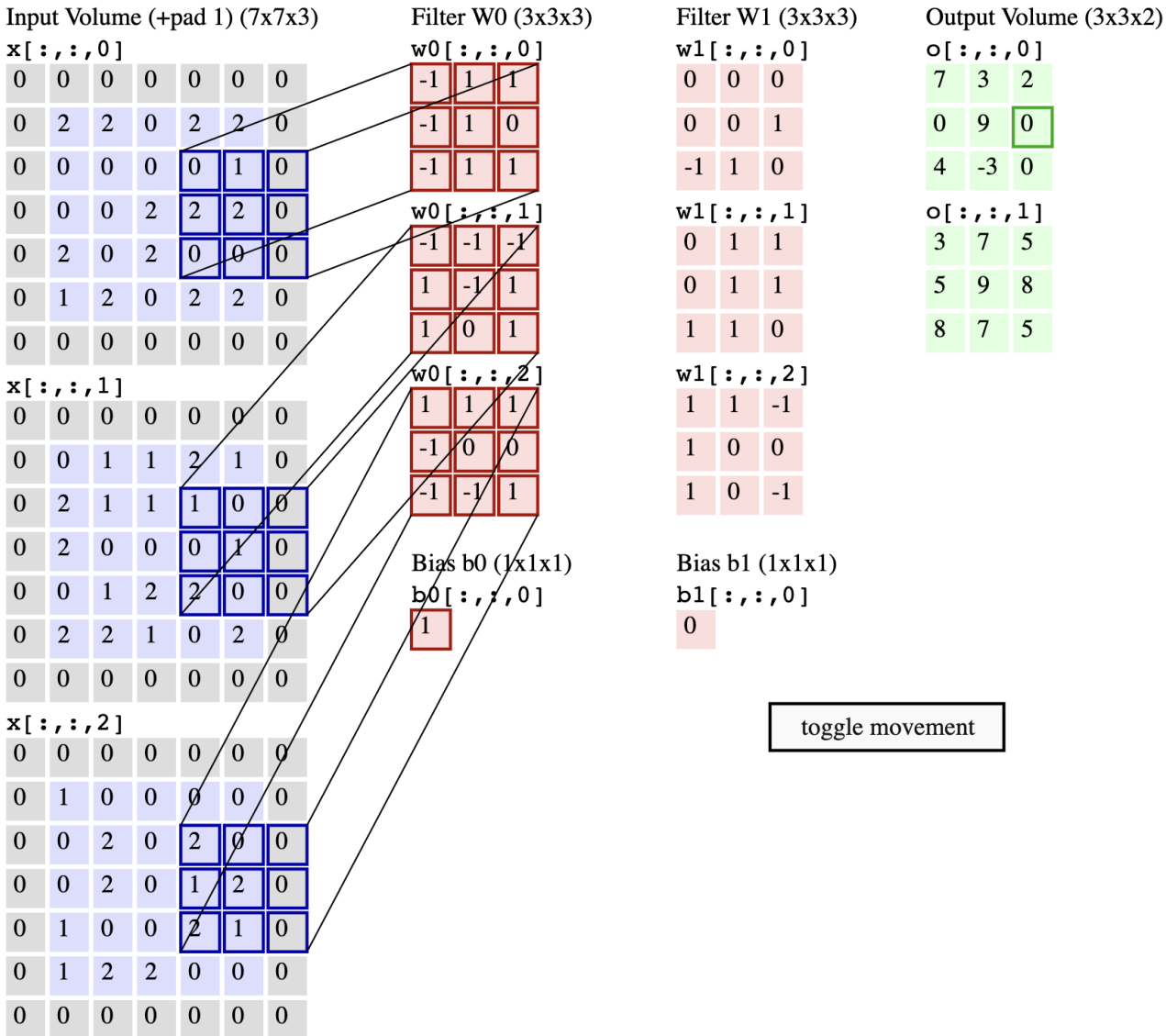
Strides in $x$ and $y$ directions dictate which convolutions are computed to obtain the next layer

Zero-padding can be used if required to adjust layer sizes and boundaries

Typically, a convolution layer will have a large number of filters, the number of channels in the next layer will be the same as the number of filters used

# Image Convolution

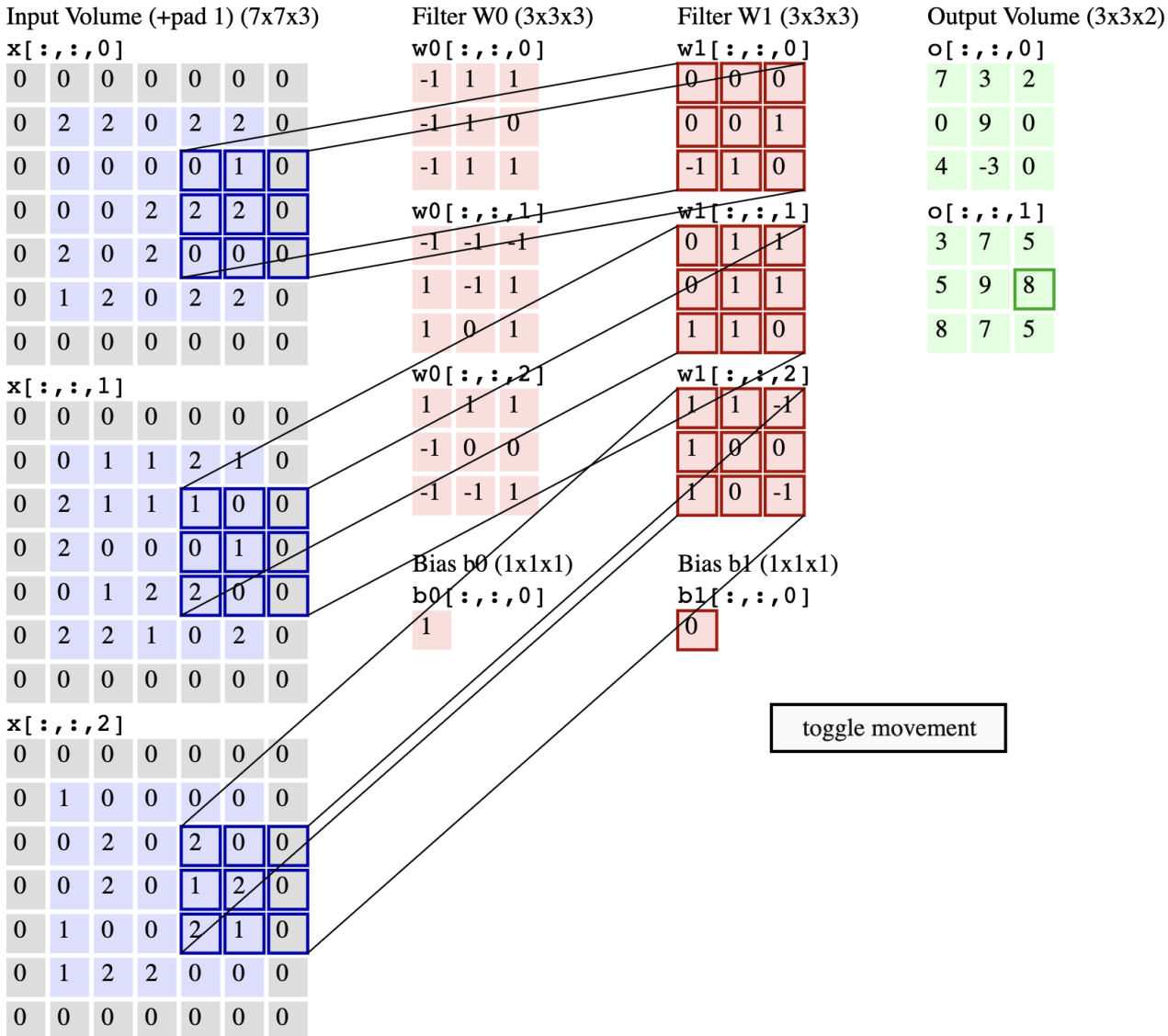
Two filters, zero padding, stride one[2]




[2] https://cs231n.github.io/convolutional-networks/

# Image Convolution

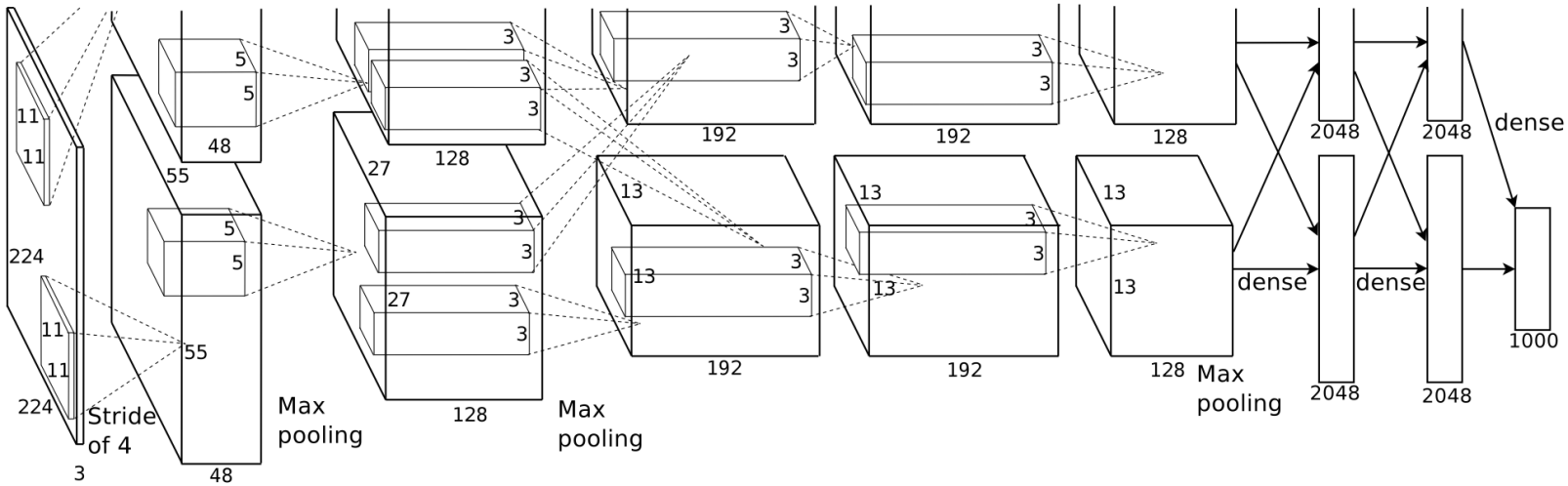
Two filters, zero padding, stride one[3]



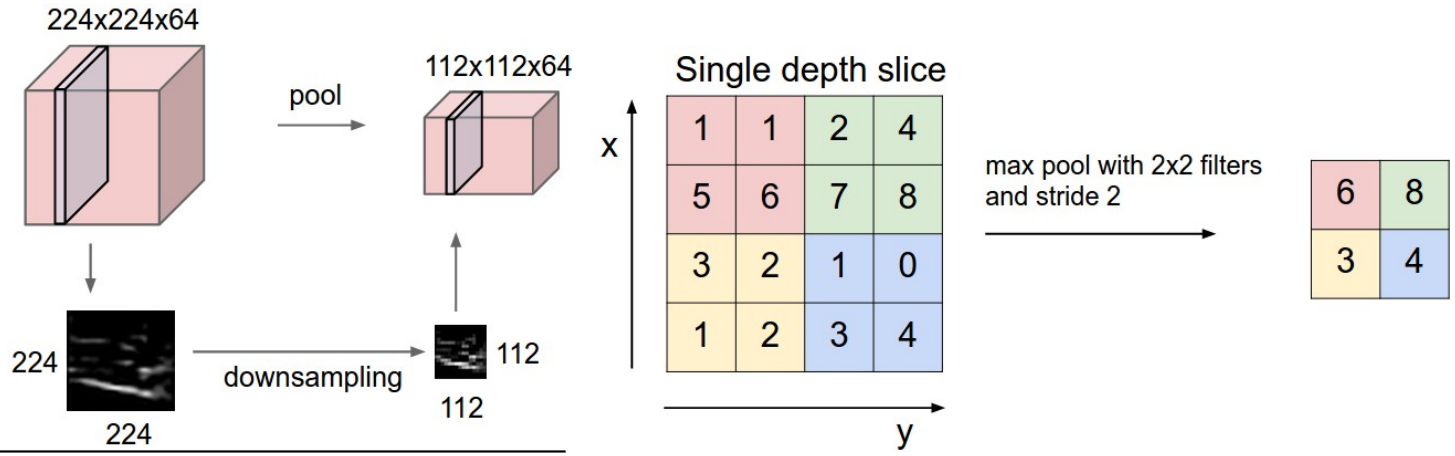

[3] https://cs231n.github.io/convolutional-networks/

# Deep Convnets

See paper by Krizhevsky, Sutskever and Hinton (2012)

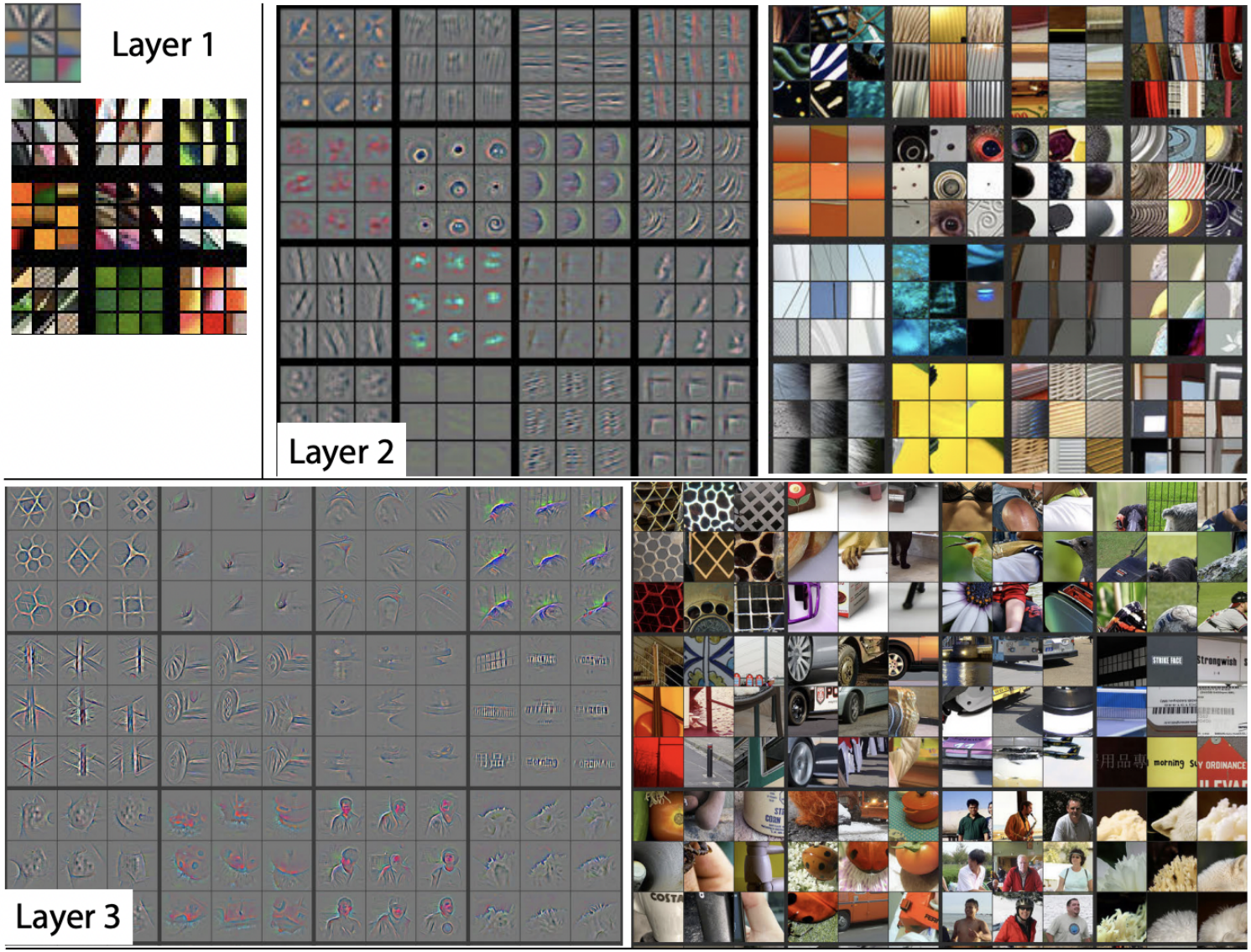



Pooling[4]



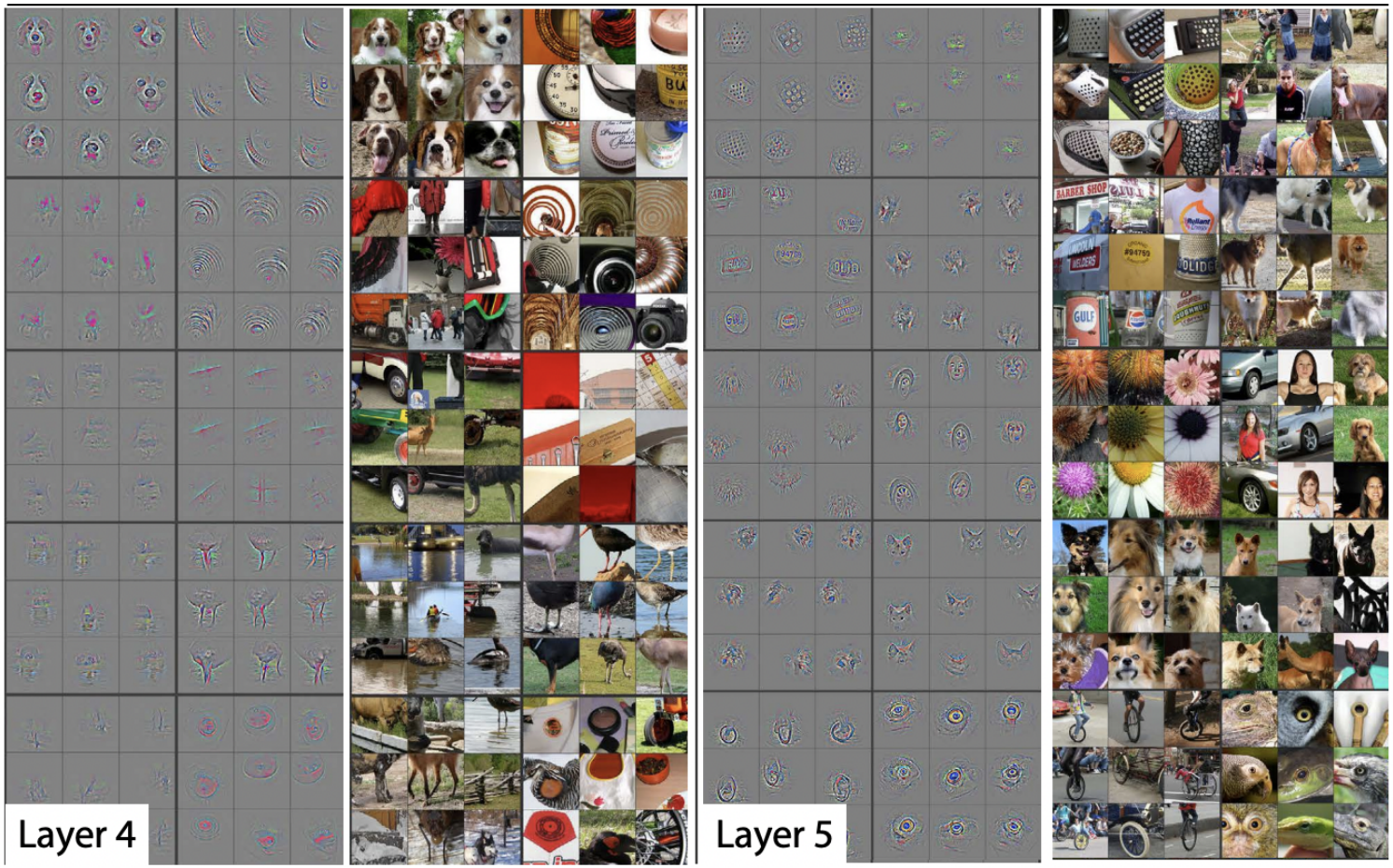

[4] https://cs231n.github.io/convolutional-networks/

# Understanding Convnets



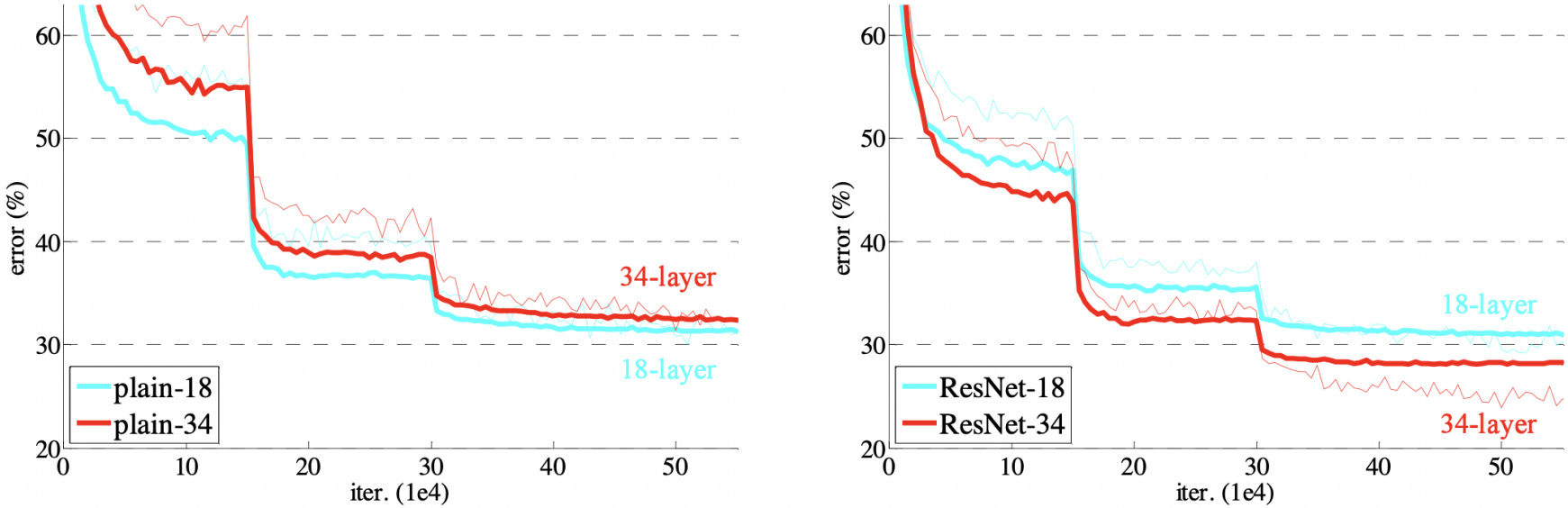

See paper by Zeiler and Fergus (2013)

# Understanding Convnets




See paper by Zeiler and Fergus (2013)

# ResNets: Why more layers can give worse models

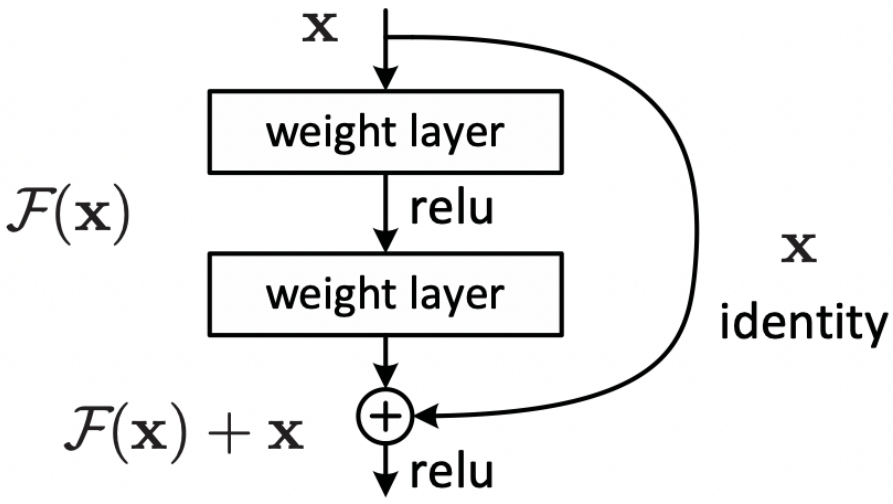



Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers

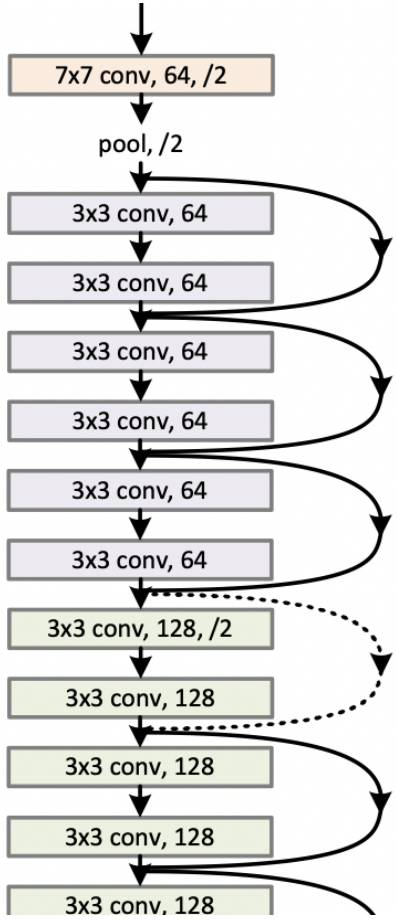
See paper by He, Zhang, Ren and Sun (2015)

# ResNets

ResNet block




The 34-layer residual (the dotted shortcuts increase dimensions)




See paper by He, Zhang, Ren and Sun (2015)

# Outline

# Language Models

A language model assigns a probability to a sequence of words, such that $\sum_{w \in \Sigma^*} p(w) = 1$

Given the observed training text, how probable is this new utterance?

Thus we can compare different orderings of words (e.g. Translation):

$$p(\text{he likes apples}) > p(\text{apples likes he})$$

or choice of words (e.g., Speech Recognition)

$$p(\text{he likes apples}) > p(\text{he licks apples})$$

Most language models employ the chain rule to decompose the joint probability into a sequence of conditional probabilities:

$$p(w_1, w_2, w_3, ..., w_N) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)\cdots\cdots p(w_N|w_1, w_2, ..., w_{N-1})$$

Note that this decomposition is exact and allows us to model complex joint distributions by learning conditional distributions over the next word $(w_n)$ given the history of words observed $(w_1, w_2, ..., w_{n-1})$

# Language Models

The simple objective of modelling the next word given the observed history contains much of the complexity of natural language understanding.

Consider predicting the extension of the utterance:

$$p(\cdot|\text{There she built a })$$

With more context we are able to use our knowledge of both language and the world to heavily constrain the distribution over the next word:

$$p(\cdot|\text{Alice went to the beach. There she built a })$$

There is evidence that human language acquisition partly relies on future prediction

Language modelling is a time series prediction problem in which we must be careful to train on the past and test on the future

With Recurrent Neural Networks we compress the entire history in a fixed length vector, enabling long range correlations to be captured.

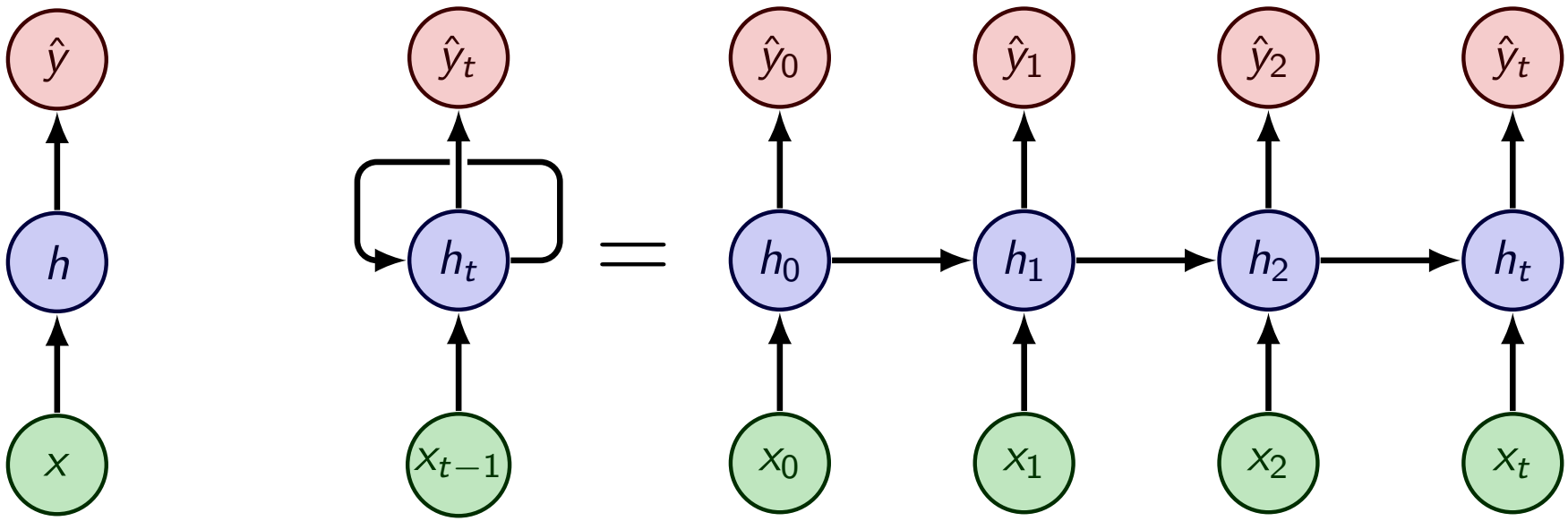
# Recurrent Neural Network Models

**Feed forward**

$$\mathbf{h} = \sigma(\mathbf{V}\mathbf{x} + \mathbf{c})$$
$$\hat{y} = \mathbf{W}\mathbf{h} + b$$

**Recurrent networks**

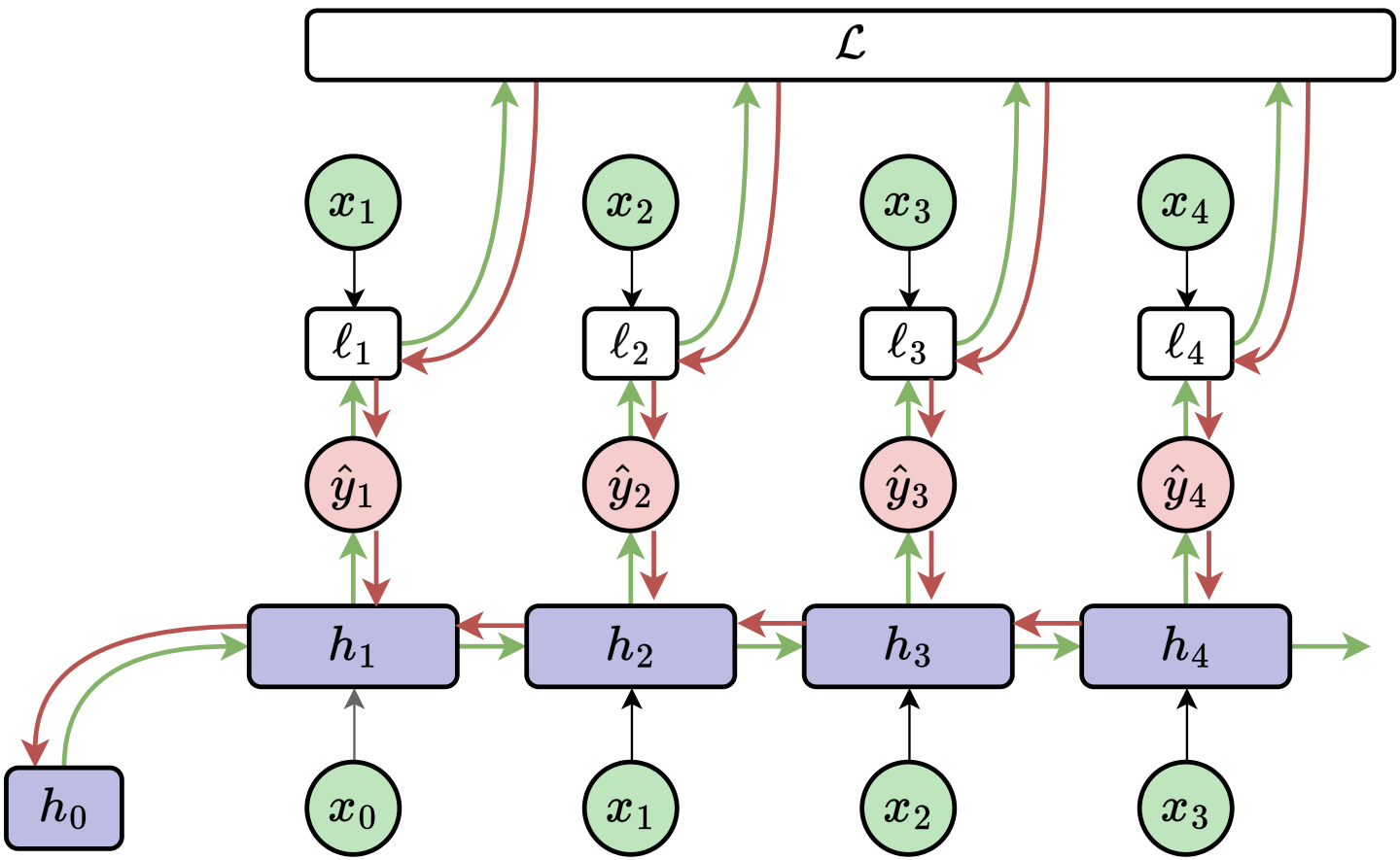
$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c})$$
$$\hat{y}_n = \mathbf{W}\mathbf{h}_n + b$$



# Recurrent Neural Network Models

The unrolled recurrent network is a directed acyclic computation graph. We can run backpropagation as usual:

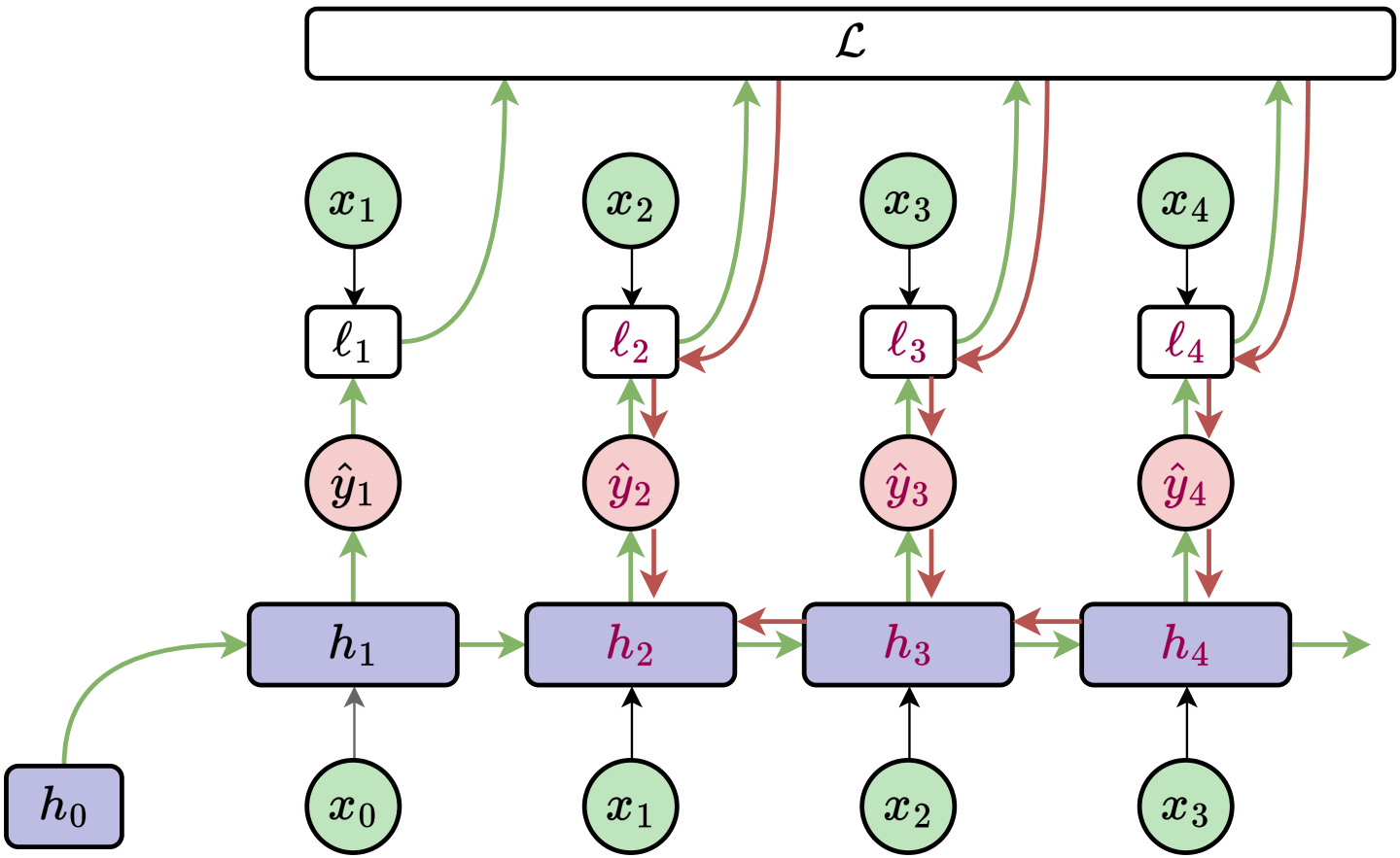
$$\mathcal{L} = -\frac{1}{N}\sum_{n=1} \ell_n(x_n, \hat{y}_n)$$



# Recurrent Neural Network Models

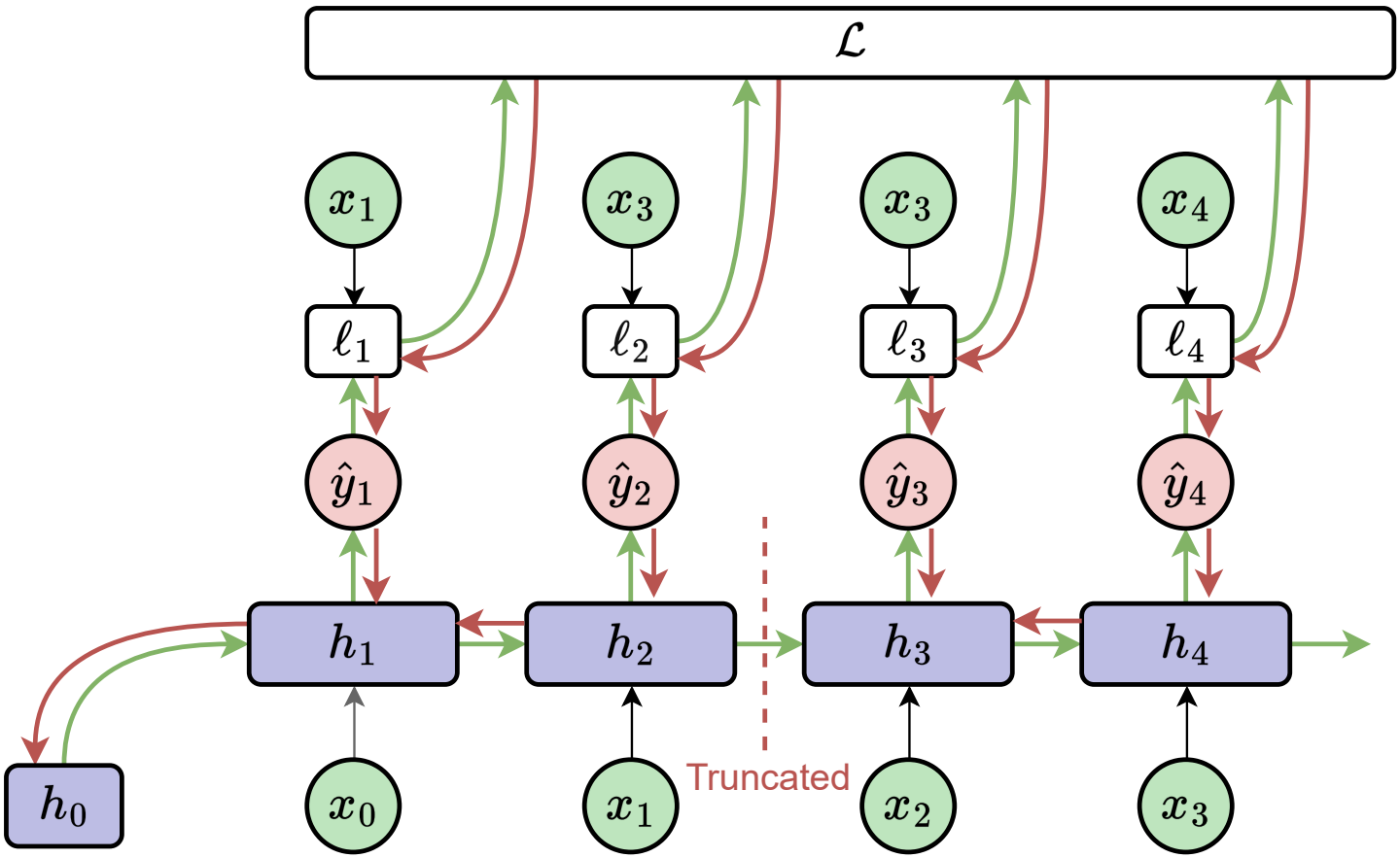
This algorithm is called Back Propagation Through Time (BPTT). Note the dependence of derivatives at time $n$ with those at time $n + \alpha$:

$$\frac{\partial \mathcal{L}}{\partial h_2} = \frac{\partial \mathcal{L}}{\partial \ell_2}\frac{\partial \ell_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial \mathbf{h}_2} + \frac{\partial \mathcal{L}}{\partial \ell_3}\frac{\partial \ell_3}{\partial \hat{\mathbf{y}}_3}\frac{\partial \hat{y}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} + \frac{\partial \mathcal{L}}{\partial \ell_4}\frac{\partial \ell_4}{\partial \hat{y}_4}\frac{\partial \hat{y}_4}{\partial \mathbf{h}_4}\frac{\partial \mathbf{h}_4}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} + \dots$$



# Recurrent Neural Network Models

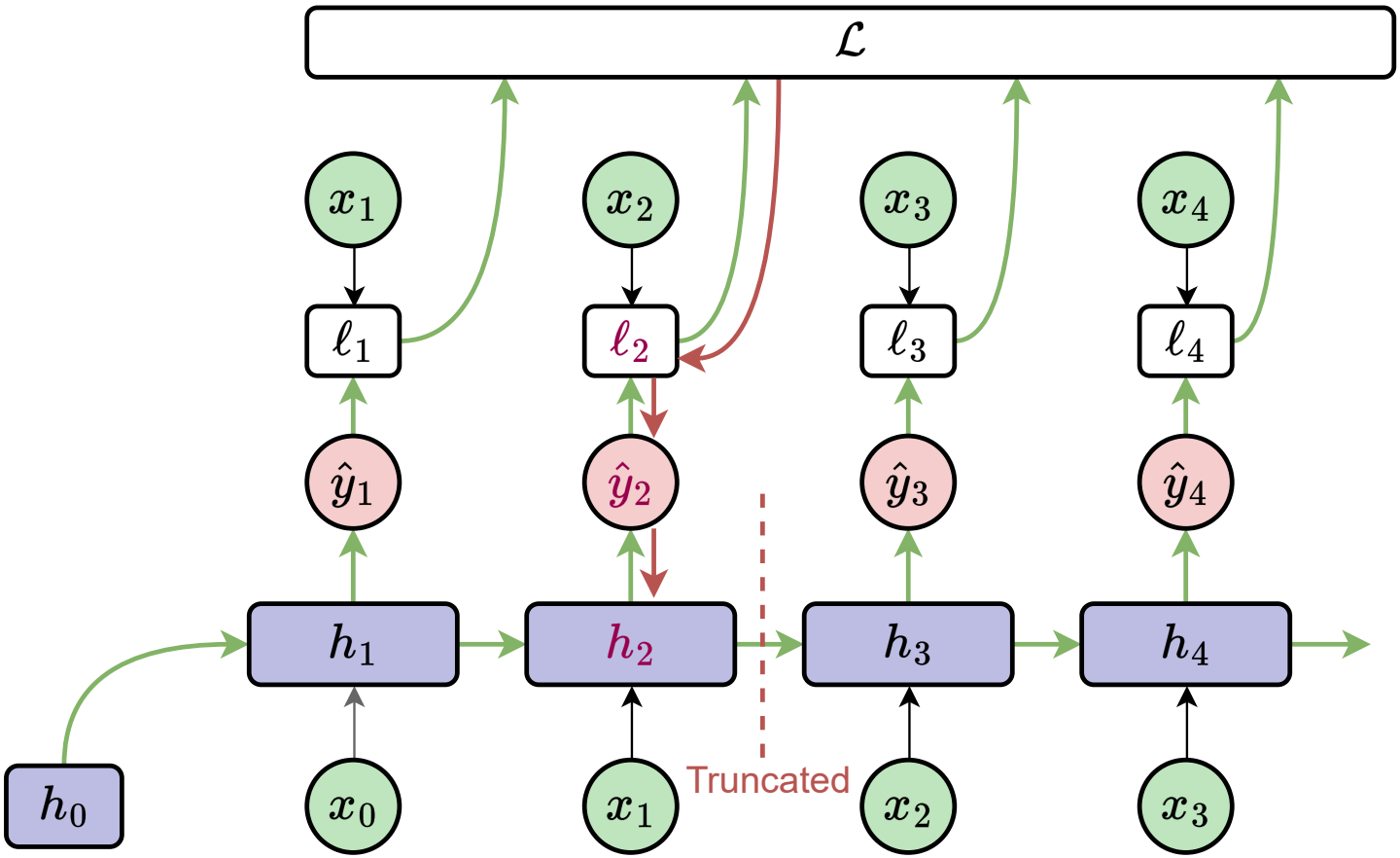
If we break these dependencies after a fixed number of timesteps we get Truncated Back Propagation Through Time (TBPTT):



# Recurrent Neural Network Models

If we break these dependencies after a fixed number of timesteps we get Truncated Back Propagation Through Time (TBPTT):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_2} \approx \frac{\partial \mathcal{L}}{\partial \ell_2} \frac{\partial \ell_2}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial \mathbf{h}_2}$$

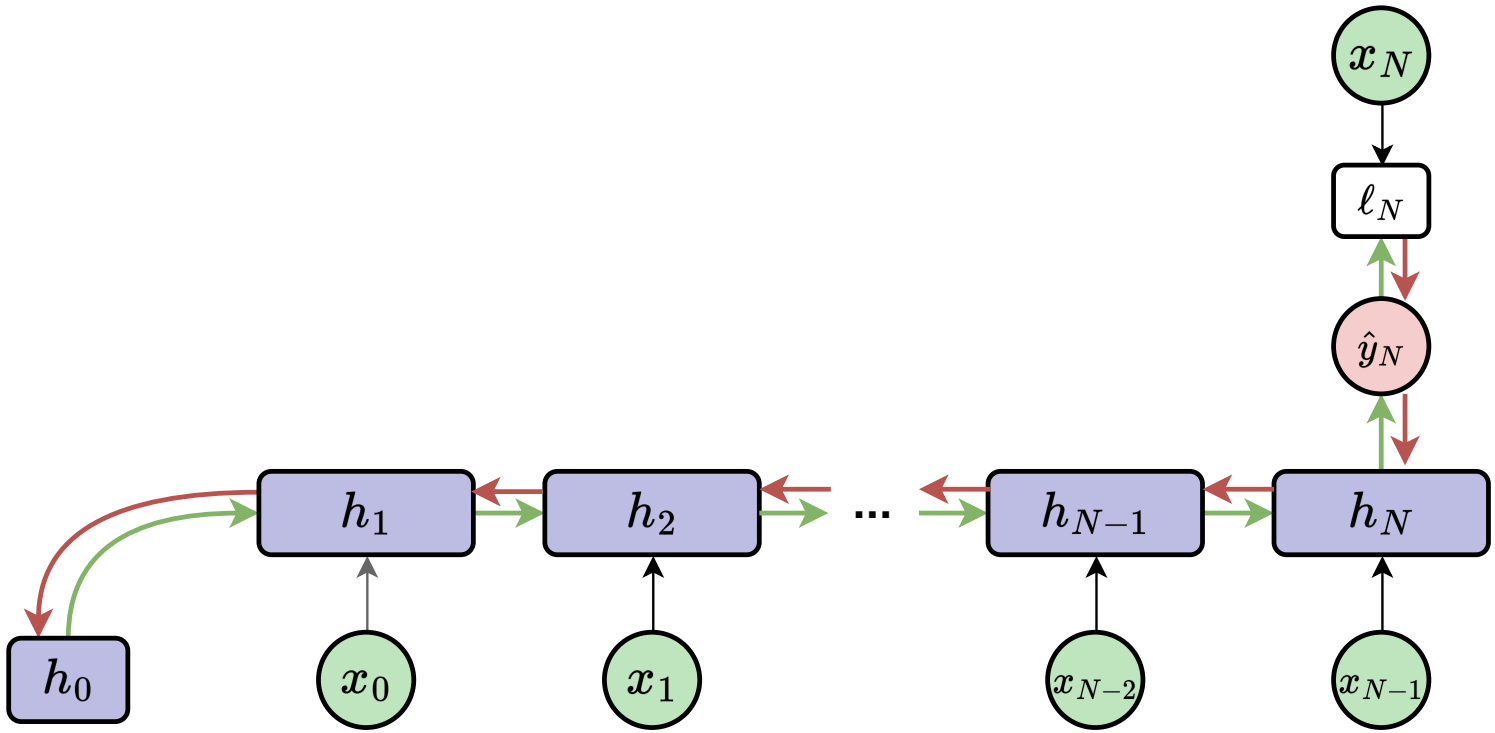


# Capturing Long Range Dependencies

An RNN Model needs to discover and represent long range dependencies:

$$p(\text{sandcastle} | \text{Alice went to the beach. There she built a })$$

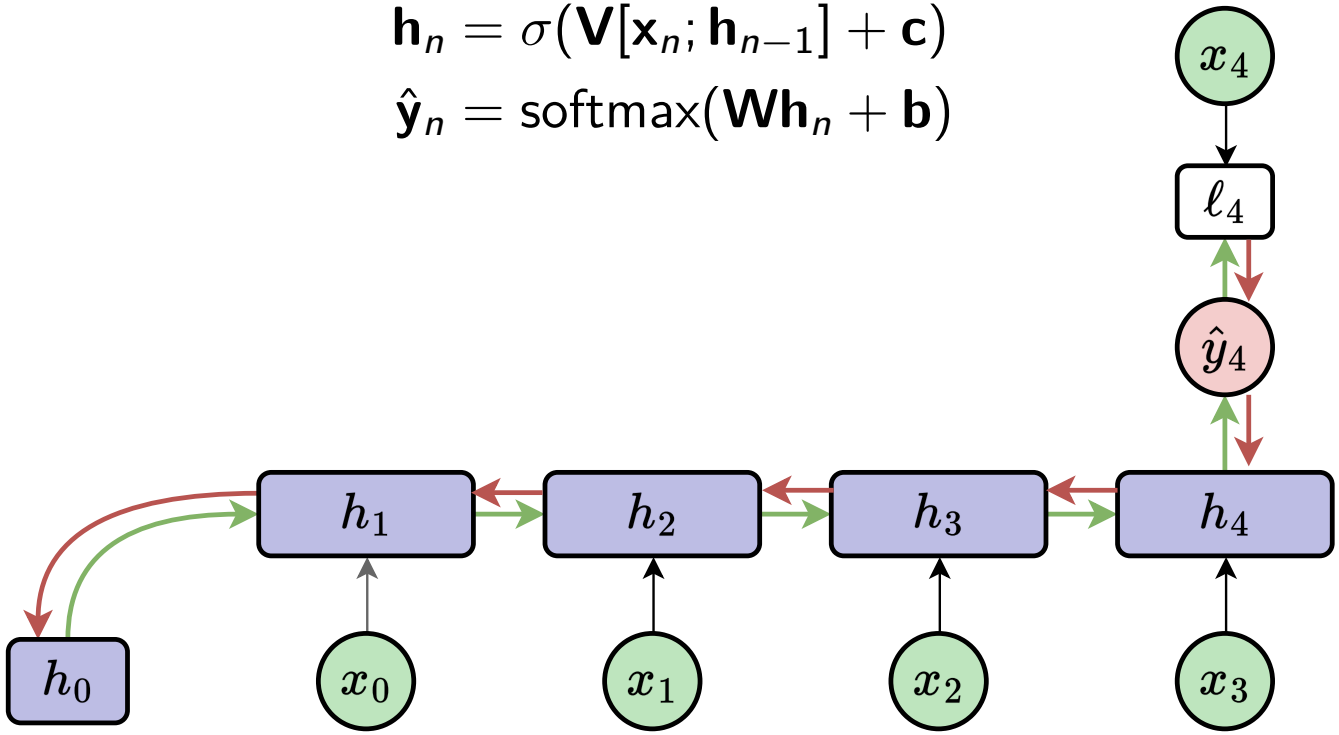
While a simple RNN model can represent such dependencies in theory, can it learn them?



# RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in $\ell_4$ to changes in $h_1$:

$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c})$$
$$\hat{\mathbf{y}}_n = \text{softmax}(\mathbf{W}\mathbf{h}_n + \mathbf{b})$$

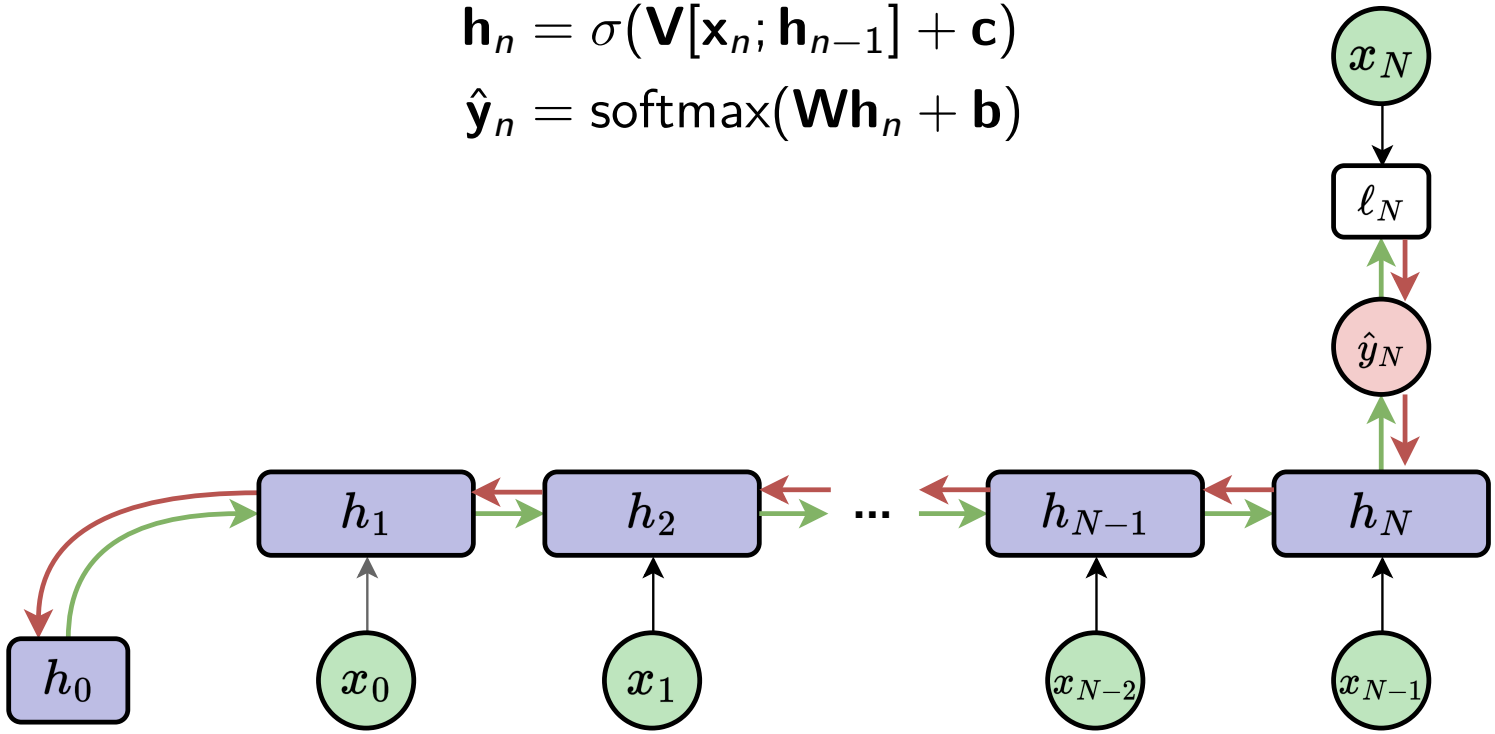



$$\frac{\partial \ell_4}{\partial \mathbf{h}_1} = \frac{\partial \ell_4}{\partial \hat{y}_4} \frac{\partial \hat{y}_4}{\partial \mathbf{h}_4} \frac{\partial \mathbf{h}_4}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}$$

# RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in $\ell_N$ to changes in $\mathbf{h}_1$:

$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c})$$
$$\hat{\mathbf{y}}_n = \text{softmax}(\mathbf{W}\mathbf{h}_n + \mathbf{b})$$




$$\frac{\partial \ell_N}{\partial \mathbf{h}_1} = \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left( \prod_{n \in \{N,\ldots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \right)$$

# RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in $\ell_N$ to changes in $\mathbf{h}_1$:

$$\frac{\partial \ell_N}{\partial \mathbf{h}_1} = \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left( \prod_{n \in \{N,\dots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \right)$$

$$\mathbf{h}_n = \sigma(\mathbf{V}[\mathbf{x}_n; \mathbf{h}_{n-1}] + \mathbf{c}) = \sigma(\underbrace{\mathbf{V}_x \mathbf{x}_n + \mathbf{V}_h \mathbf{h}_{n-1} + \mathbf{c}}_{\mathbf{z}_n})$$

$$\frac{\partial \mathbf{h}_n}{\partial \mathbf{z}_n} = \operatorname{diag}\left(\sigma'(\mathbf{z}_n)\right), \qquad \frac{\partial \mathbf{z}_n}{\partial \mathbf{h}_{n-1}} = \mathbf{V}_h$$

$$\begin{aligned}
\frac{\partial \ell_N}{\partial \mathbf{h}_1} &= \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left( \prod_{n \in \{N,\dots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \right) \\
&= \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left( \prod_{n \in \{N,\dots,2\}} \frac{\partial \mathbf{h}_n}{\partial \mathbf{z}_n} \frac{\partial \mathbf{z}_n}{\partial \mathbf{h}_{n-1}} \right) \\
&= \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left( \prod_{n \in \{N,\dots,2\}} \operatorname{diag}\left(\sigma'(\mathbf{z}_n)\right) \mathbf{V}_h \right)
\end{aligned}$$

# RNNs: Exploding and Vanishing Gradients

$$\frac{\partial \ell_N}{\partial \mathbf{h}_1} = \frac{\partial \ell_N}{\partial \hat{y}_N} \frac{\partial \hat{y}_N}{\partial \mathbf{h}_N} \left( \prod_{n \in \{N, \ldots, 2\}} \text{diag}\left(\sigma'(\mathbf{z}_n)\right) \mathbf{V}_h \right)$$

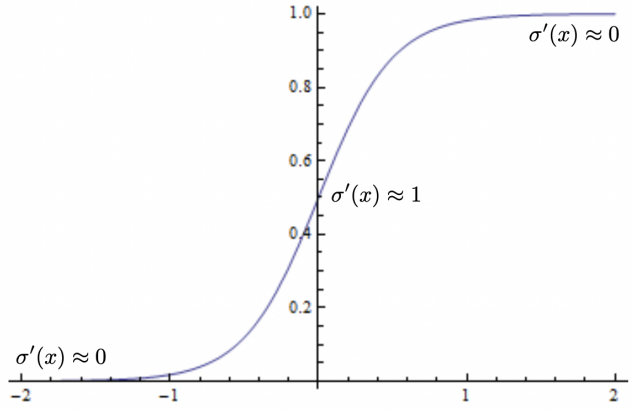
The core of the recurrent product is the repeated multiplication of $\mathbf{V}_h$. If the largest eigenvalue of $\mathbf{V}_h$ is

- 1, then gradient will propagate,
- $> 1$, the product will grow exponentially (explode),
- $< 1$, the product shrinks exponentially (vanishes).

# RNNs: Exploding and Vanishing Gradients

Most of the time the spectral radius of $\mathbf{V}_h$ is small. The result is that the gradient vanishes and long range dependencies are not learnt.

Many non-linearities ($\sigma(\cdot)$) can also shrink the gradient.




Second order optimizers ((Quasi-)Newtonian Methods) can overcome this, but they are difficult to scale. Careful initialization of the recurrent weights can help[5].

The most popular solution to this issue is to change the network architecture to include gated units[6].

[5] Stephen Merity: Explaining and illustrating orthogonal initialization for recurrent neural networks. `https://smerity.com/articles/2016/orthogonal_init.html`

[6] Christopher Olah: Understanding LSTM Networks (`https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

# Outline

# Issues with recurrent models

Lack of parallelizability: Forward and backward passes have $\mathcal{O}$(sequence length) unparallelizable operations

- GPUs can perform a bunch of independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!

If not recurrence, then what? How about attention?

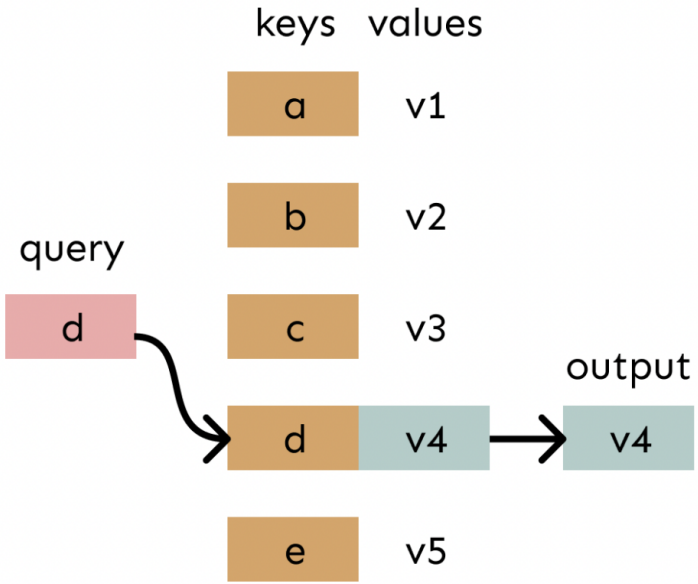
**Attention** treats each timestep as a query to access and incorporate information from a set of values.

# Attention

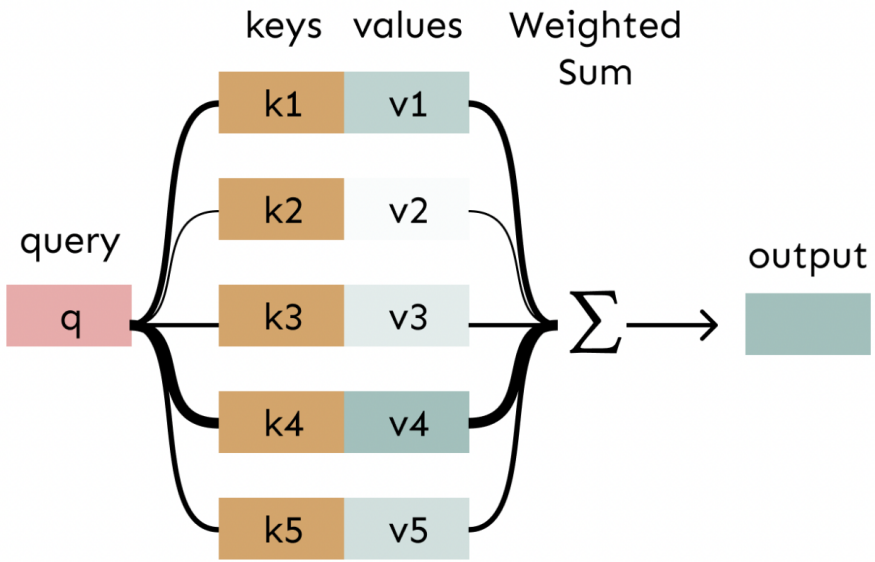
## Attention as a soft, averaging lookup table

We can think of attention as performing fuzzy lookup in a key-value store.

In a lookup table, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.




In attention, the **query** matches all **keys** softly, to a weight between 0 and 1. The keys' **values** are multiplied by the weights and summed.



# Attention

The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$.

Compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values

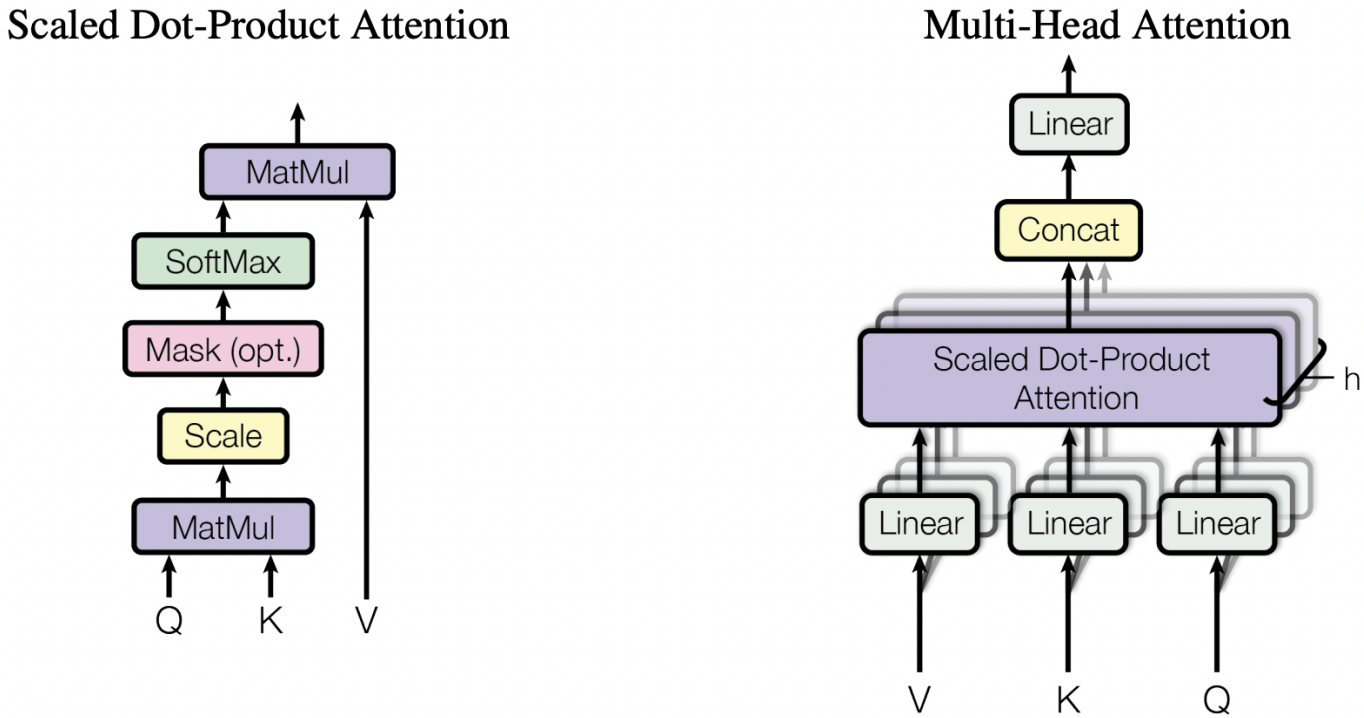
Compute the attention function on a set of queries simultaneously, packed together into a matrix $\mathbf{Q}$

Keys and values are also packed together into matrices $\mathbf{K}$ and $\mathbf{V}$

Attention is computed as

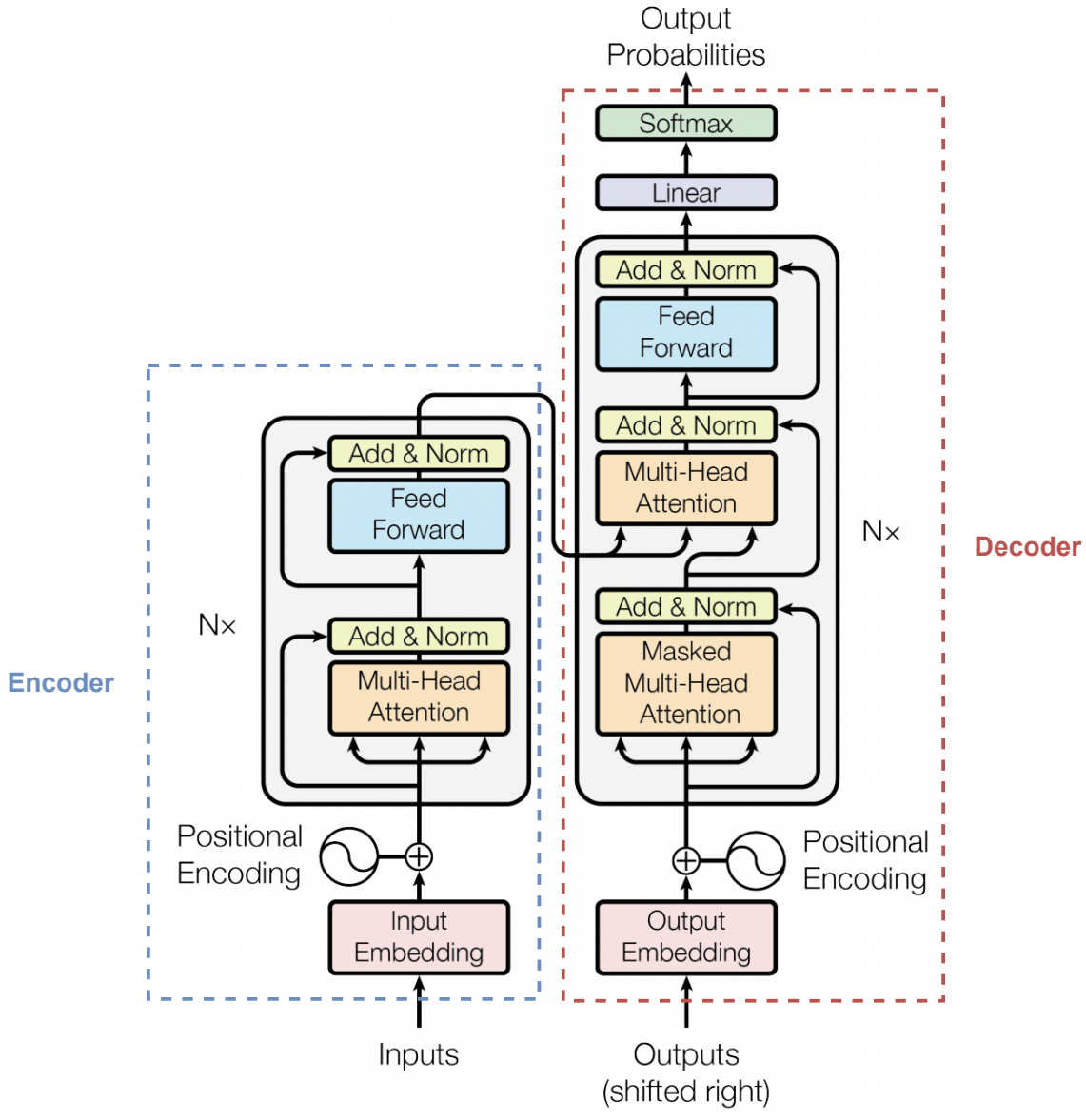
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

# Multi-head Attention




See paper by Vaswani et al. (2017)

# Transformers

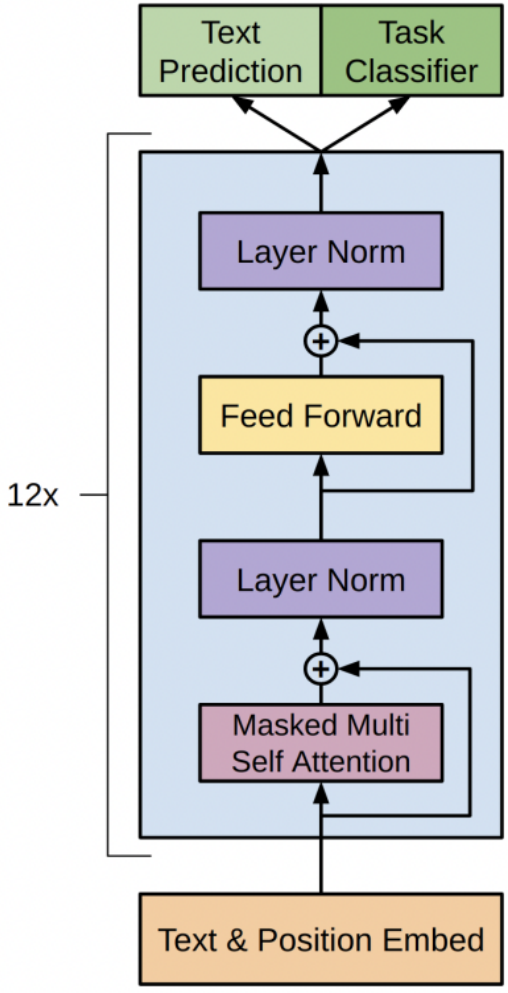



See paper by Vaswani et al. (2017)

# GPT and BERT

## Decoder part

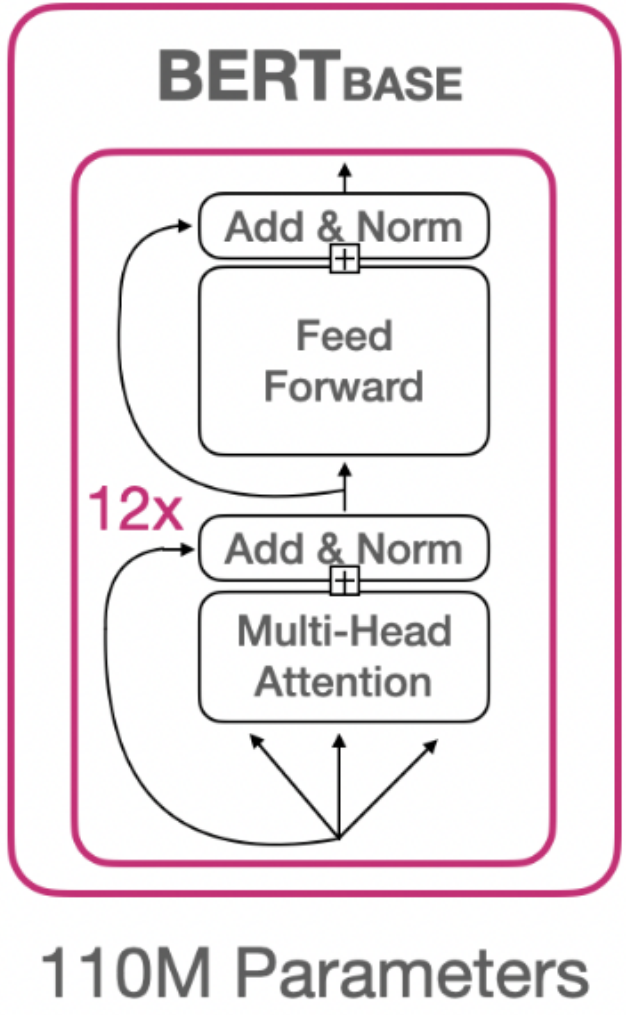
GPT: Generative Pre-trained Transformers




See paper by Radford et al. (2018)

## Encoder part

BERT: Bidirectional Encoder Representations from Transformers




[a]

[a] https://huggingface.co/blog/bert-101

# Recap

Neural Networks

Training Deep Neural Networks

Convolution Neural Networks

Recurrent Neural Networks

Transformers

END LECTURE