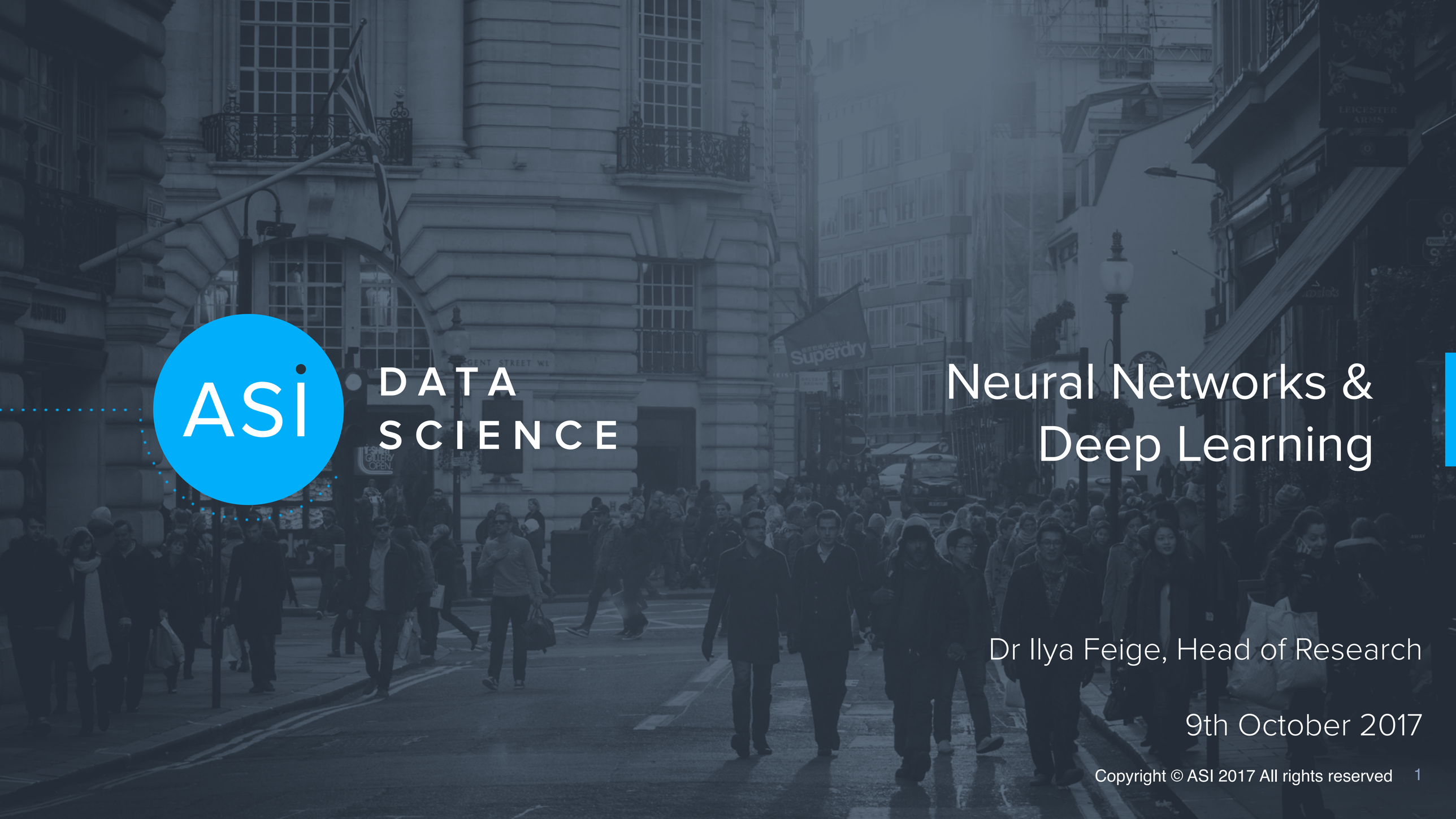


ASi

DATA SCIENCE

Neural Networks & Deep Learning

Dr Ilya Feige, Head of Research

9th October 2017

Neural Networks & Deep Learning: Into the black box

The goal of this course is to actually understand neural networks

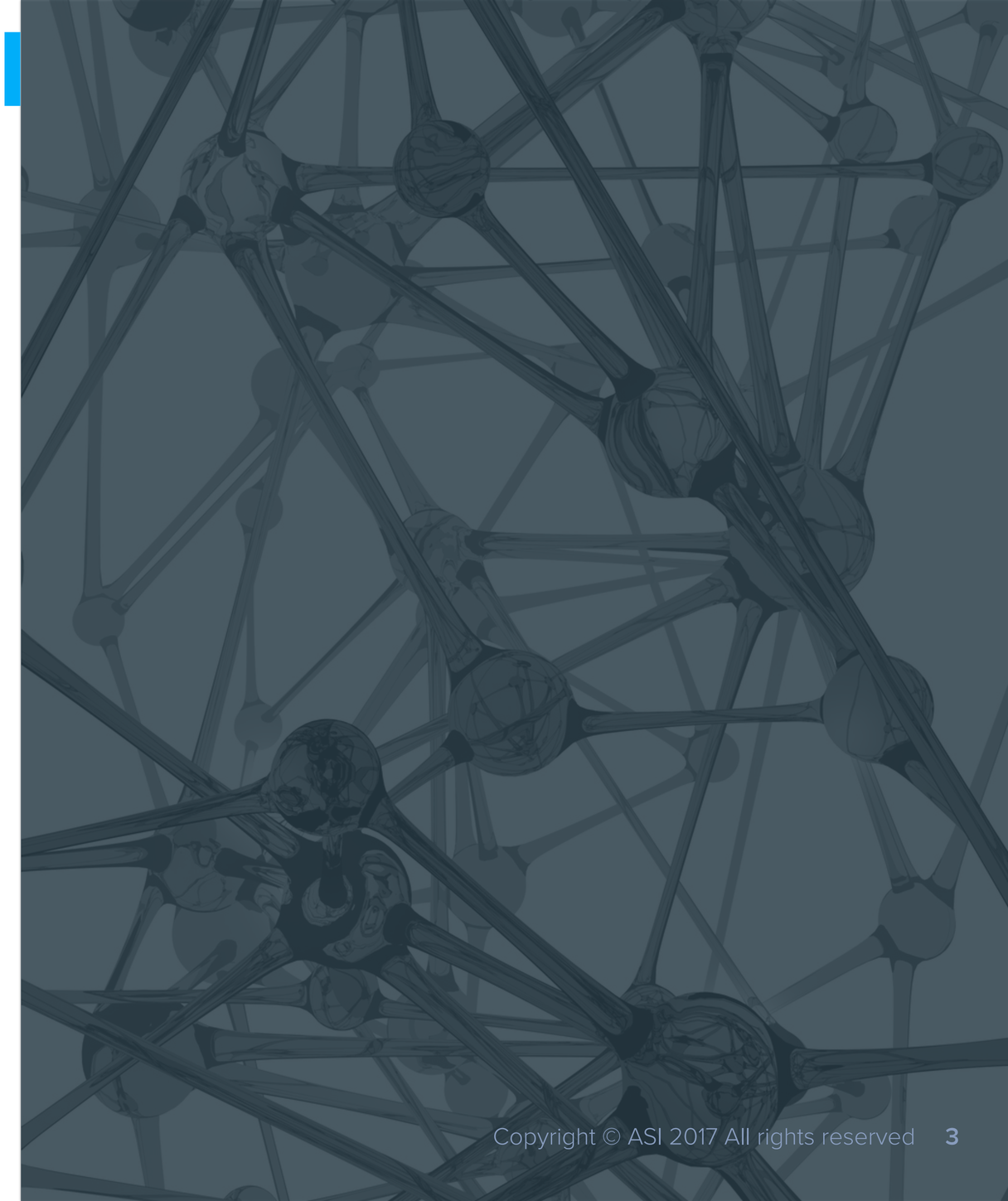
There are many black-boxes out there that are easy to use, but hard to understand

Course Outline

1. Neural Networks are familiar (1 hour)
2. Intro to Neural Networks as a framework (1 hour)
3. MNIST Exercise (1 hour)

Lunch Break

4. Learning / training networks (2 hours)
5. Advanced topics (1 hour)
6. Q&A (30 mins)



Neural Networks are Familiar

Neural networks are incredibly useful today

Neural networks:

A. can fit very general models to
B. highly non-linear data,
C. in very high dimensions

Today neural networks are used for:

- Images: high dimensional data -> number of pixels x colours
- NLP: sentence of length $N \implies (\text{vocab size})^n$ dimensions
- General data: financial systems, healthcare, marketing

Neural networks have the potential to do even more

The brain seems to use neural networks to “fit” our intelligence to the data of the world around us

- (e.g., artificial neural networks provide a predictive model for human intelligence)

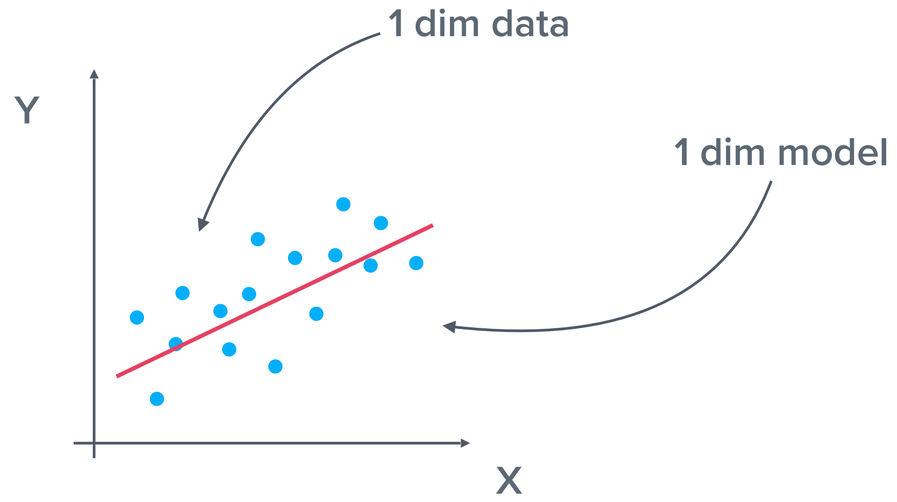
Potential: getting closer to building a fully functional AI

Neural nets are familiar: Linear regression

Linear Regression:

This is the simplest version of a neural net

- $\text{Model}(x) = m * x + b$
- $\text{Data} = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$
- Model has parameters m (slope) and b (y-intercept) that need to be learned



Neural nets are familiar: Linear regression

Learning parameters:

$$C = \sum_{i=1}^{n} (y_i - (mx_i + b))^2$$

Minimising cost function C gives best-fit parameters, since it makes

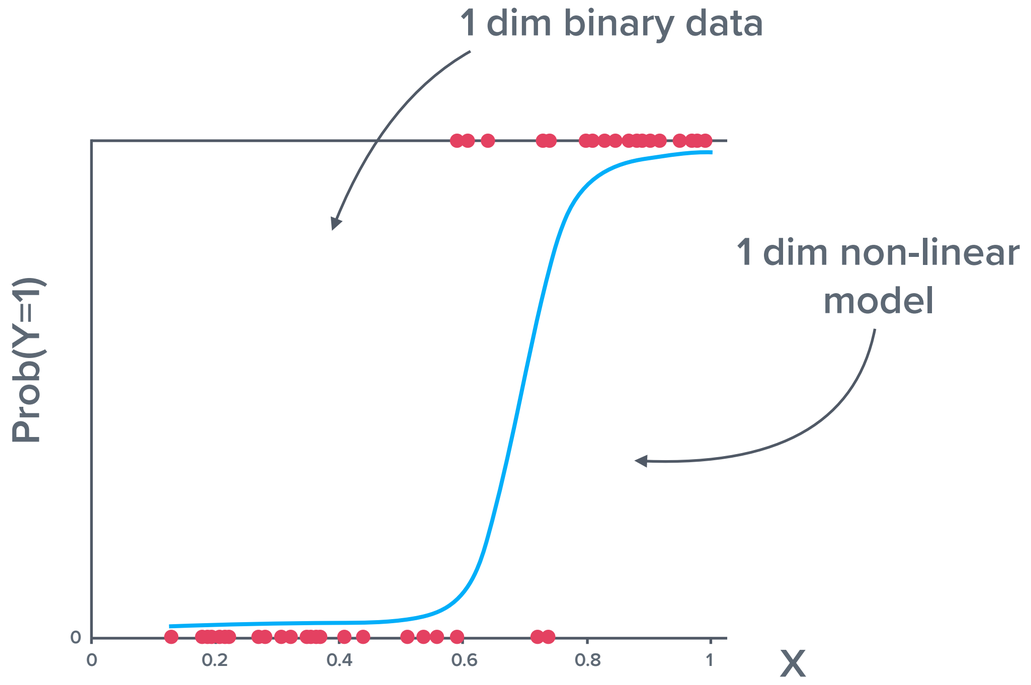
$\mathrm{Model}(x_i)$ close to y_i on average

Neural nets are familiar: Logistic regression

Logistic Regression (binary classification):

This is the second simplest version of a neural net

- Model (x) = $\sigma(mx+b) = \frac{1}{1+e^{-(mx+b)}}$
- Remember: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Exercise: check limits of sigmoid



Neural nets are familiar: Logistic regression

Learning parameters:

$$C = \sum_{i=1}^{n} (y_i - \sigma(mx_i + b))^2$$

Minimising cost function C gives best-fit parameters, since it makes $\mathrm{Model}(x_i)$ close to y_i on average

Neural nets are familiar: Adding a layer

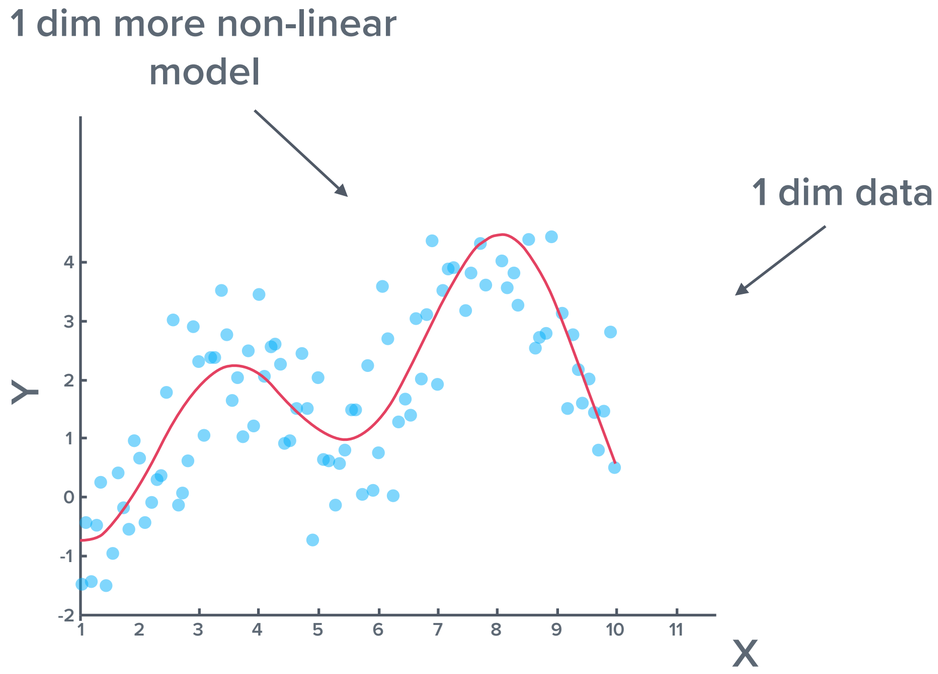
Adding a (hidden) layer:

We've seen models of the form:

$$\text{Model}(x) = f(mx + b) \text{ for } f = 1, \sigma()$$

- What if we used: $\text{Model}(\text{Model}(x))$
- Or: $\text{Model}(\text{Model}(\text{Model}(x)))$
- You get a more non-linear function with 4, or 6 parameters respectively, e.g.,

$$\text{2layer-model}(x) = f(m_2 f(m_1 x + b_1) + b_2)$$



Neural nets are familiar: Adding a layer

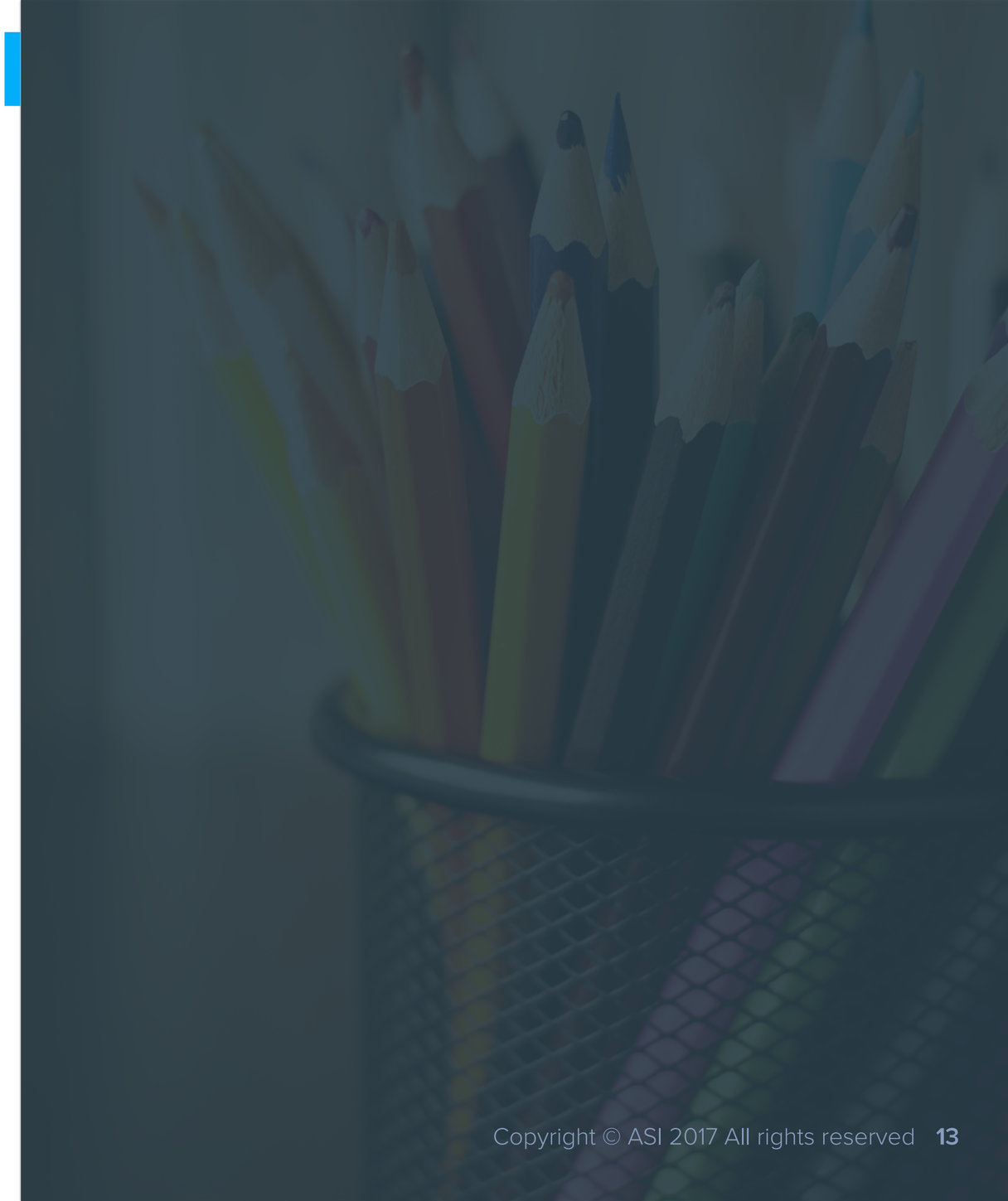
Learning parameters:

$$C = \sum_{i=1}^{n} (y_i - \text{2layer-model}(x_i))^2$$

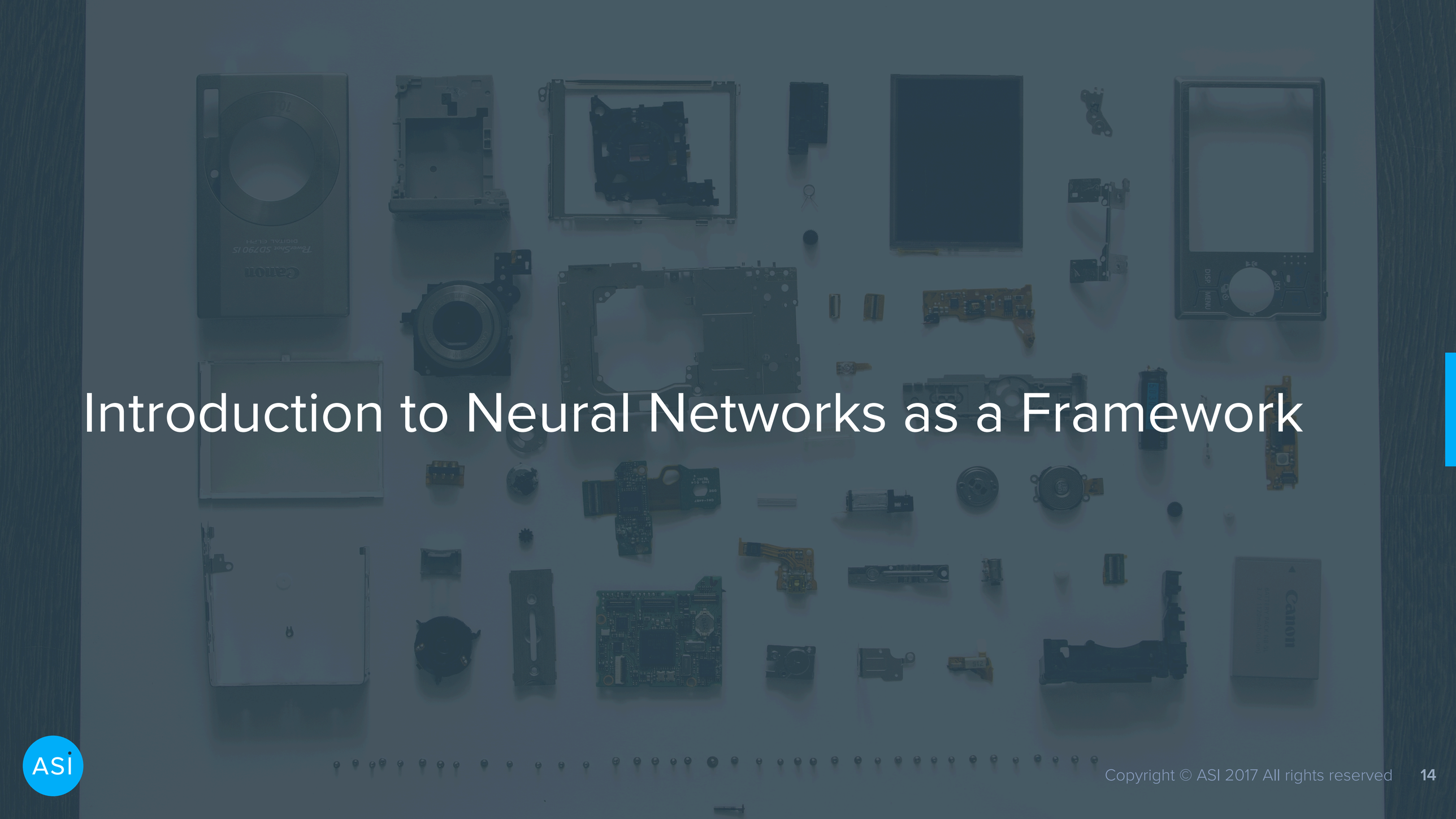
Minimising cost function C gives best-fit parameters (e.g., m1, m2, b1, b2), since it makes $\text{Model}(x_i)$ close to y_i on average

EXERCISE

- Login to SherlockML
- Fit a 1-dim neural network to data

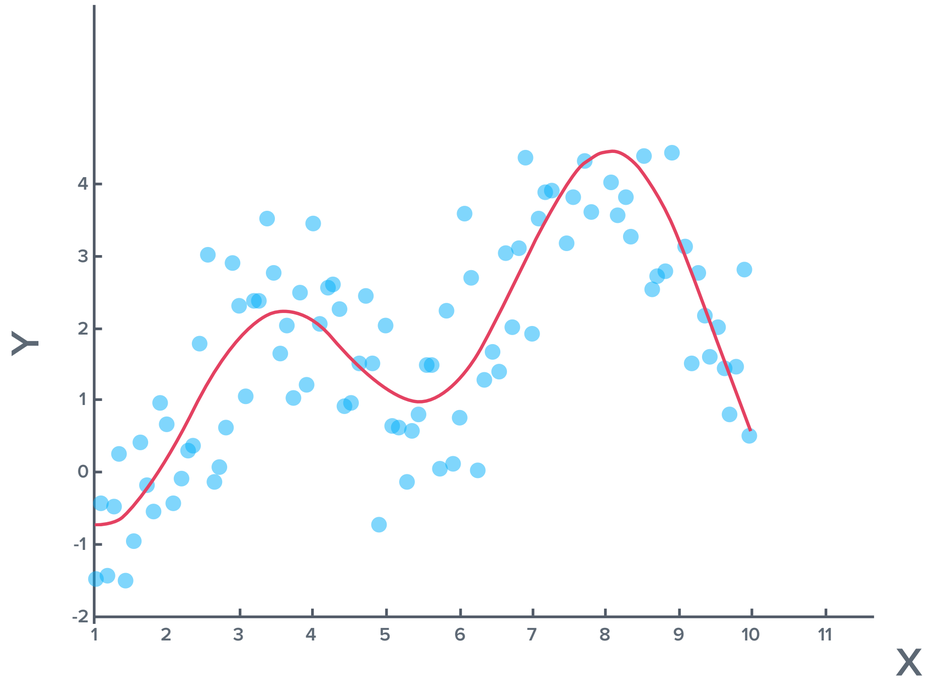


Introduction to Neural Networks as a Framework



Neural networks provide a general framework

Which non-linear function would you fit to this data?



Neural networks provide a general framework

Choosing a non-linear function to fit complex data can be an arbitrarily highly redundant problem

We need a framework for doing this, that allows complete generalisability (flexibility) over space of all functions, without sacrificing interpretability and simplicity for optimisation

This is a big ask!

Neural networks provide a general framework

Solution: Neural Networks!

Neural Networks provide a framework for fitting arbitrary high-dimensional data with general non-linear models

Example: Feed Forward Neural Network in 1 Dimension

Feed forward neural network on 1-dimensional data:

Just a generalisation of:

$$\text{Model}(\text{Model}(\ldots\text{Model}(x)))$$

$$\vec{h_1} = f(\vec{W_1}\,x + \vec{b_1})$$

$$\vec{h_2} = f(W_2 \cdot \vec{h_1} + \vec{b_2})$$

$$\vdots$$

$$\vec{h_N} = f(W_N \cdot \vec{h}_{N-1} + \vec{b_N})$$

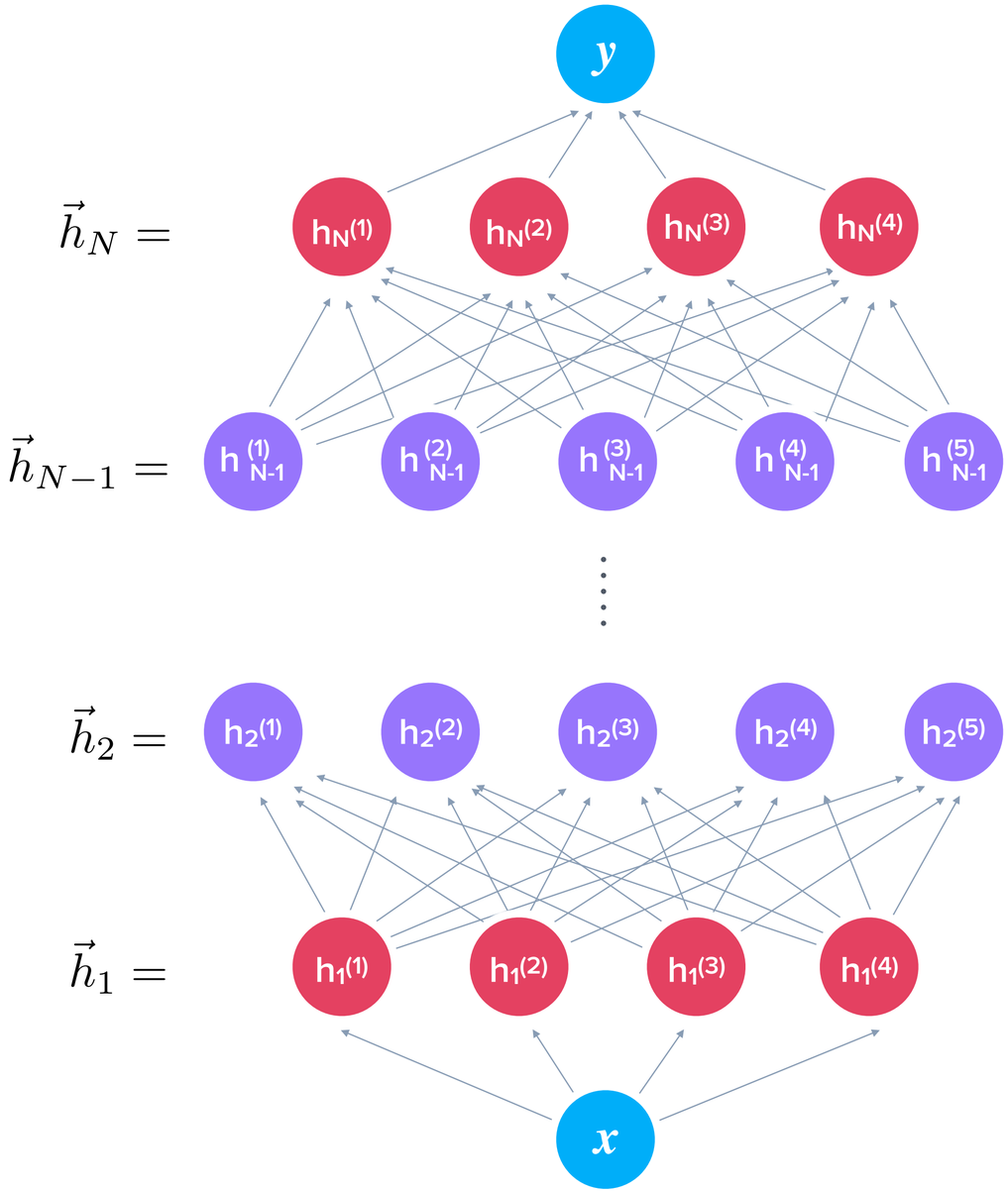
$$\text{Model}(x) = f(\vec{W_{N+1}} \cdot \vec{h^N} + b_{N+1})$$

This is a very non-linear function with a lot of parameters, but it simply takes x -> y!

Example: Feed Forward Neural Network in 1 Dimension

Feed forward neural network on 1-dimensional data:

We normally write these models as pictures (networks) rather than write all the equations



Neural networks provide a general framework

Situation: you have some data and no idea what the underlying model should be (e.g., probability distribution of images of cats on the internet)….

Choose architecture:

A. What activation functions, f, to use? They can all be different! (e.g. linear, sigmoid, reLU, softmax, tanh)

B. Width of each layer? (i.e., number of parameters used, or the capacity of the layer)

C. Depth of network? (i.e. number of layers, this is the depth as in “deep” learning, deeper networks can sequentially learn features)

Neural networks provide a general framework

With the model built, learn parameters, W_j and b_j, by minimising the cost function:

$$C = \sum_{i=1}^{n} (y_i - \text{Model}(x_i))^2$$

There are many other cost functions that can be used:

$$C = \sum_{i=1}^{n} c(y_i, \text{Model}(x_i))$$

Feed forward neural networks for high-dimensional data look similar

Finally, for higher dimensional data, nothing really changes:

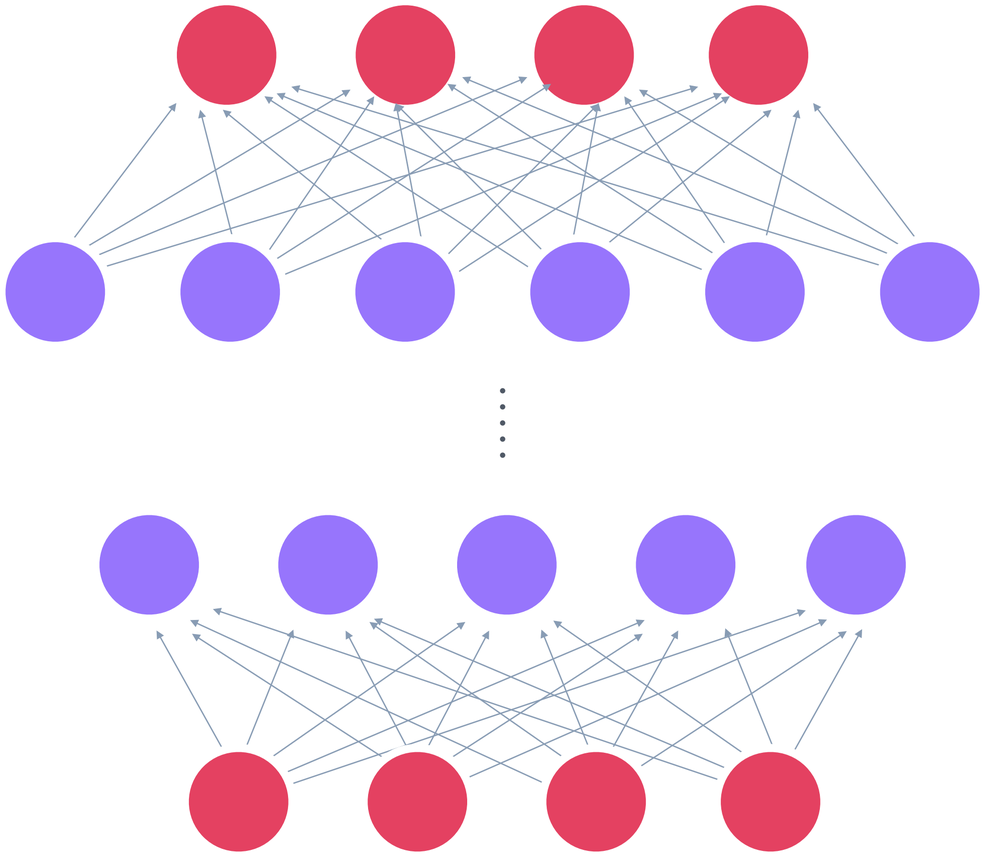
$$\vec{y} = \text{Model}(x) = f_{N+1}(W_{N+1} \cdot \vec{h}_N + b_{N+1})$$

$$\vec{h}^N = f_N(W_N \cdot \vec{h}_{N-1} + \vec{b}_N)$$

$$\vdots$$

$$\vec{h}^1 = f_1(W_1 \cdot \vec{x} + \vec{b}_1)$$

$$\vec{x}$$



MNIST Example

MNIST Example

MNIST dataset	Images of hand-written digits from US social workers and high school students

Data consist of square, 28x28 pixel B&W images:

x1 = label = 5 

x2 = label = 0 

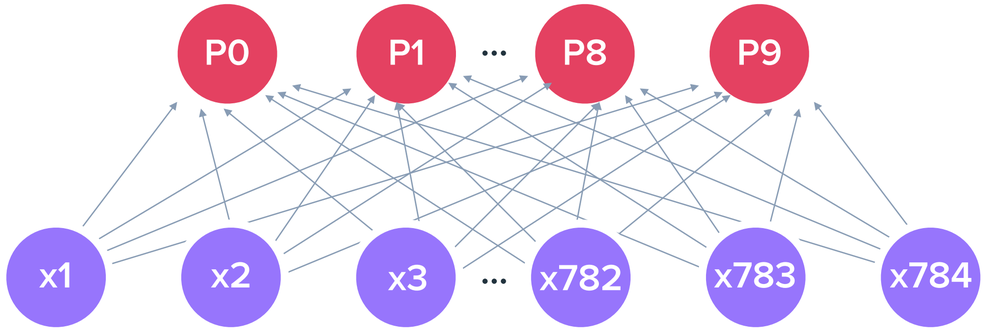
x3 = label = 4 

- Each pixel has one number $\in [0, 255]$
- We "flatten" images into 784 (=28*28) dim vectors

MNIST Example

Dataset consists of 60k training images and 10k test images

We will use a feed forward network without any hidden layers:



$\vec{y} \in \mathbb{R}^{10}$ Probabilities of each label

$\vec{x} \in \mathbb{R}^{784}$ Pixels in image

- Our model we use: $\text{Model}(\vec{x}) = \text{softmax}(W \cdot \vec{x} + \vec{b})$
- Categorical generalisation of sigmoid function:

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

MNIST Example

For dim 2, i.e. $x \in \mathbb{R}^2$

$$\text{softmax}(x)_1 = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{1}{1 + e^{-(x_1 - x_2)}} = \sigma(x_1 - x_2)$$

For $x \in \mathbb{R}^n$

$$\sum_{i=1}^{n} \text{softmax}(x)_i = \frac{\sum_{i=1}^{n} e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} = 1$$

- So, softmax is a categorical probability model
- In our model: $P(\text{image digit} = j) = \text{softmax}(W \cdot x + b)_j$

MNIST Example

Label probabilities = model() = (0.2, 0.0, 0.0, 0.0, 0.0, 0.4, 0.0, 0.2, 0.2)

Predicted label = Argmax(model ()) = 6

MNIST Example

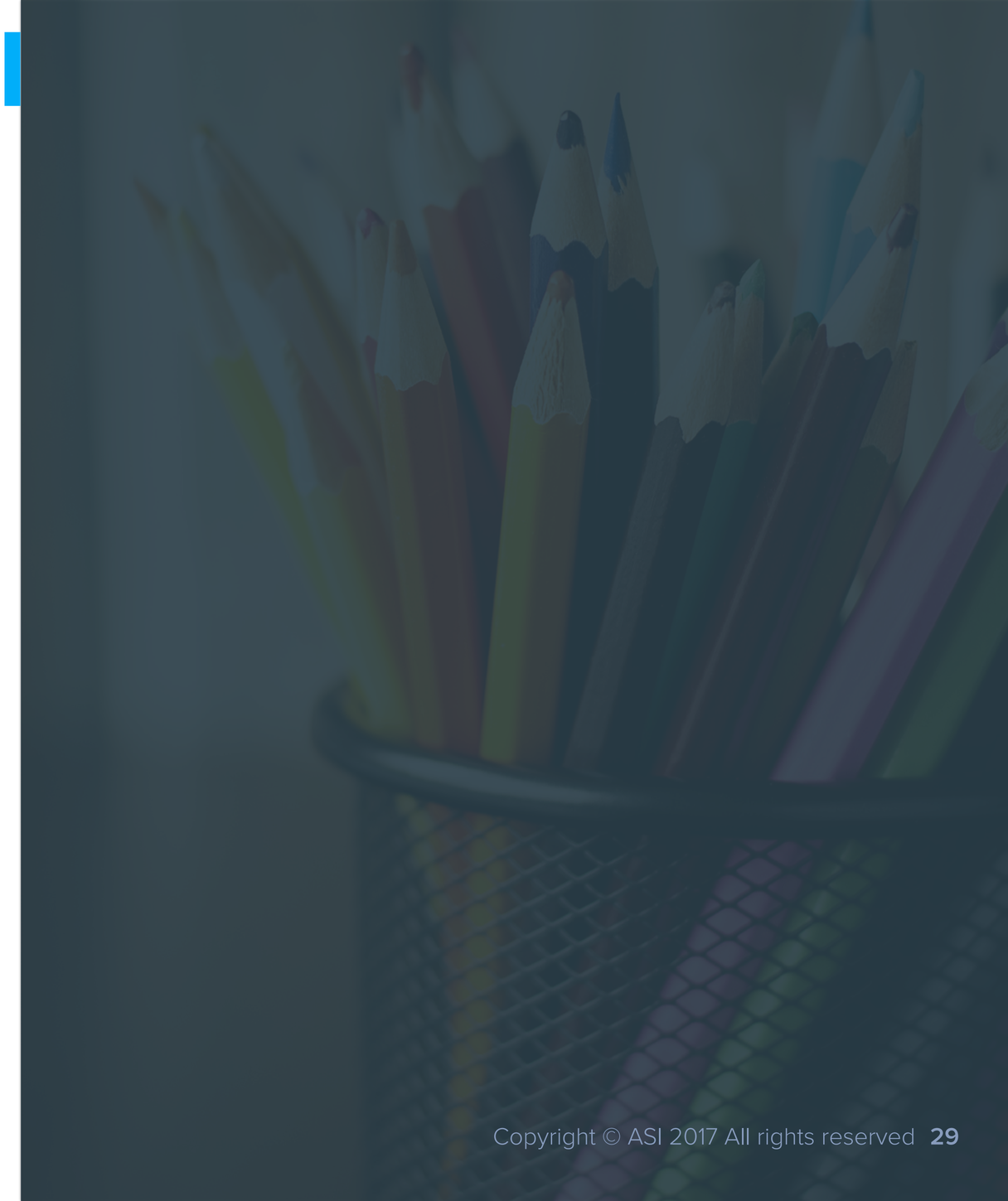
Cross checks: how much accuracy do you think you can get?

Worst = 10%

World record (huge NN) = 99.7%

EXERCISE

- Build neural network for learning MNIST data



Learning / Training Networks

Training networks

We have discussed how to construct general feedforward neural networks, $\mathrm{Model}(x;\theta)$, where θ = {W's, b's}

But how do we learn the parameters?

Only trained models, $\mathrm{Model}(x;\theta^*)$, where $\theta^* = \mathrm{argmin}\big(C(\theta)\big)$ can make useful predictions

Gradient descent can minimise high-dimensional functions

But how do we find the minimum of the cost function, $C(\theta)$?

- Remember $\frac{df(x)}{dx}$ Is the slope of f
- What is $\vec{\nabla} f(\vec{x}) = \left(\frac{\partial f(\vec{x})}{\partial x_1}, \frac{\partial f(\vec{x})}{\partial x_2}, ..., \frac{\partial f(\vec{x})}{\partial x_m} \right)$?

It is the direction of the fastest ascent!

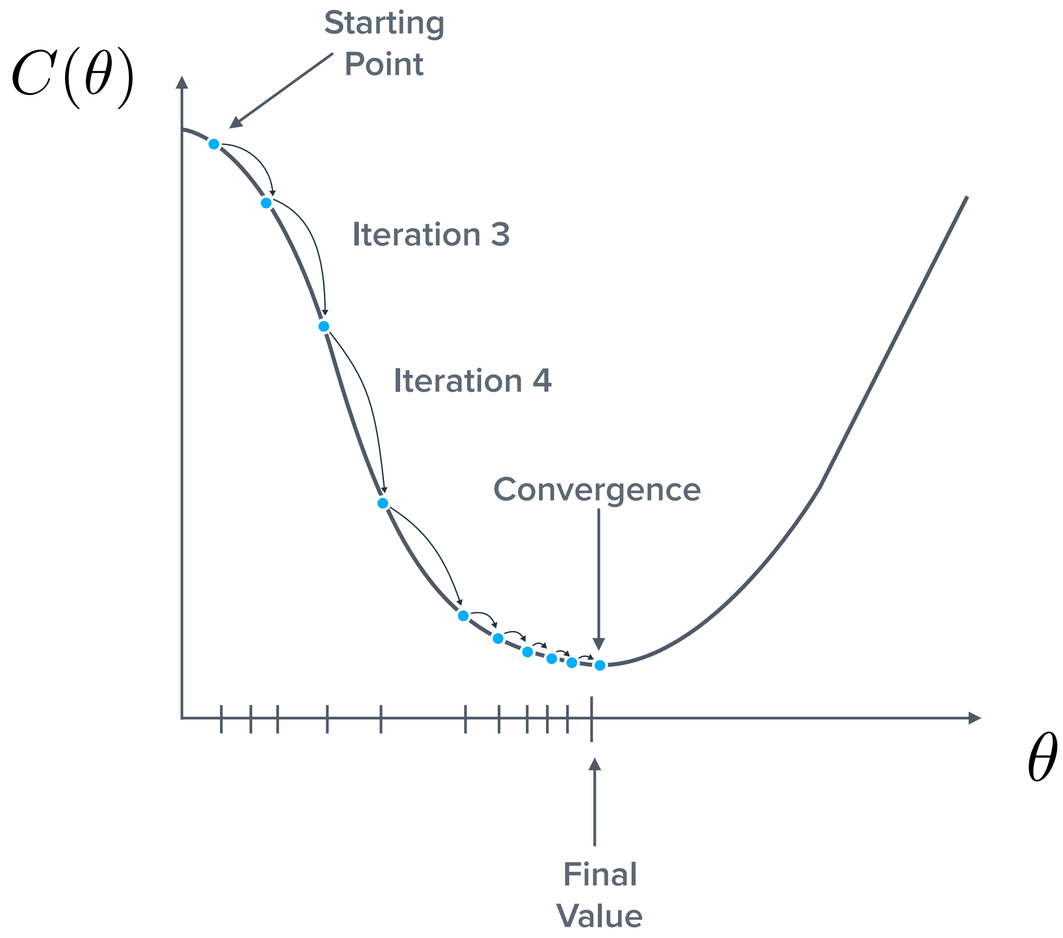
Gradient descent can minimise high-dimensional functions

So if you are at $C(\vec{\theta}\,)$ and you want to move toward C's minimum, you should travel in the direction: $-\vec{\nabla} C(\vec{\theta}\,)$

Hence, the gradient descent algorithm:

- Set the learning rate (e.g., $\alpha = 0.1$)
- Initialise parameters θ
- While not converged: $\theta \leftarrow \theta - \alpha \vec{\nabla} C(\vec{\theta})$
- return θ

Gradient Descent can minimise high-dimensional functions



Back propagation allows efficient computation of gradients

Back prop is an efficient algorithm for calculating gradients of neural network models via the chain rule of calculus

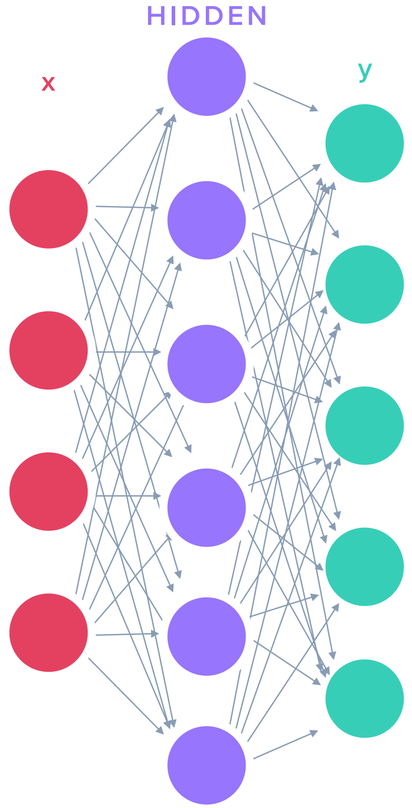
Consider the neural network

$$\vec{h} = f_1(W_1 \cdot \vec{x} + \vec{b}_1)$$

$$\vec{y} = f_2(W_2 \cdot \vec{h} + \vec{b}_2)$$

We need to calculate gradients w.r.t. W's & b's of:

$$C(\{\text{data}\}, \{W_i, b_i\}) = \sum_{i=1}^{n} c(y_i, y(x_i; \{W_i, b_j\}))$$



Back propagation allows efficient computation of gradients

To understand back prop, we adopt simplified notation:

Network equations

$$\vec{h} = f_1(W_1 \cdot \vec{x} + \vec{b}_1)$$
$$\vec{y} = f_2(W_2 \cdot \vec{h} + \vec{b}_2)$$

$$C(\{\text{data}\}, \{W_i, b_i\}) = \sum_{i=1}^{n} c(y_i, y(x_i))$$

Notation:

$$u_1 = W_1 \cdot \vec{x} + \vec{b}_1$$
$$u_2 = W_2 \cdot \vec{h} + \vec{b}_2$$
$$C(y) = \sum_{i=1}^{n} c(y_i, y(x_i))$$

Simplified equations:

$$y = f_2(u_2)$$
$$h = f_1(u_1)$$
$$C(y)$$

Back propagation allows efficient computation of gradients

Simplified equations:

$$
\begin{aligned}
y &= f_2(u_2) \\
h &= f_1(u_1) \\
C(y) & \\
u_2 &= W_2\, h + b_2 \\
u_1 &= W_1\, x + b_1
\end{aligned}
$$

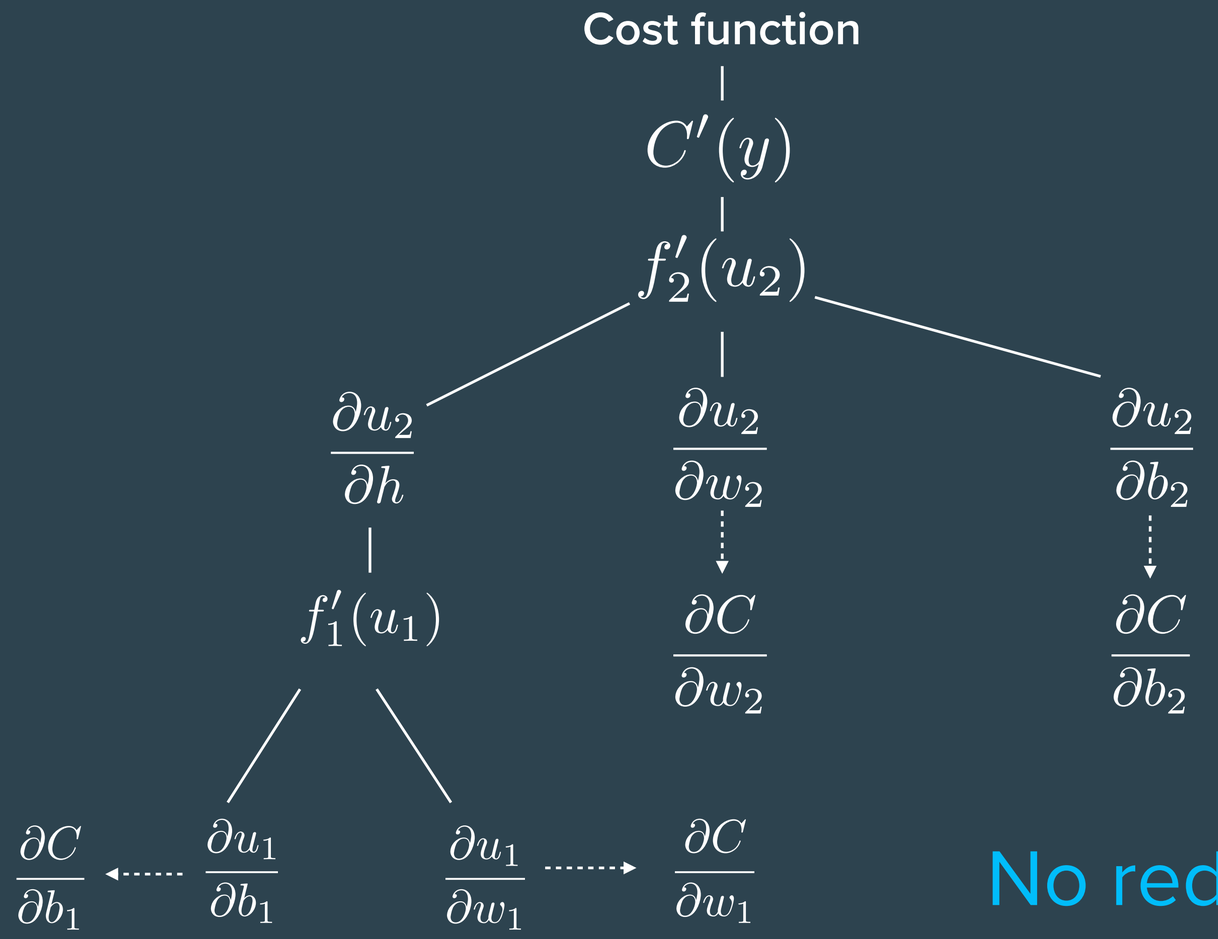
Then gradients from chain rule are:

$$
\begin{aligned}
\frac{\partial C}{\partial b_2} &= C'(y)\, f_2'(u_2)\, \frac{\partial u_2}{\partial b_2} \\
\frac{\partial C}{\partial w_2} &= C'(y)\, f_2'(u_2)\, \frac{\partial u_2}{\partial w_2} \\
\frac{\partial C}{\partial b_1} &= C'(y)\, f_2'(u_2)\, \frac{\partial u_2}{\partial h}\, f_1'(u_1)\, \frac{\partial u_1}{\partial b_1} \\
\frac{\partial C}{\partial w_1} &= C'(y)\, f_2'(u_2)\, \frac{\partial u_2}{\partial h}\, f_1'(u_1)\, \frac{\partial u_1}{\partial w_1}
\end{aligned}
$$

So much redundancy!

Back propagation allows efficient computation of gradients

The gradients from back propagation are:



No redundancy!

Stochastic gradient descent is more efficient

Cost function on n data points: $C(\{\text{data}\};\theta) = \sum_{i=1}^{n} c(y_i, y(x_i;\theta))$

Cost function has gradient: $\frac{\partial C}{\partial \theta} = \sum_{i=1}^{n} \frac{\partial}{\partial \theta} c(y_i, y(x_i;\theta))$

- You have to run over all n data points in order to do one step in gradient descent
- Running over an entire dataset can take hours, and a model may require 1 million steps to converge, that’s millions of hours!

Stochastic gradient descent is more efficient

Solution is “batching”: break dataset into sets of size m (batch size):

$$\{x_i|i=1,...,n\} = \{x_i|i=1,..,m\} \cup \{x_i|i=m+1,\dots 2m\} \cup \dots$$

Stochastic gradient descent algorithm:

$$\frac{\partial C}{\partial \theta} = \sum_{i=1}^{m} \frac{\partial}{\partial \theta} c(y_i, y(x_i;\theta)) \rightarrow \text{step}$$

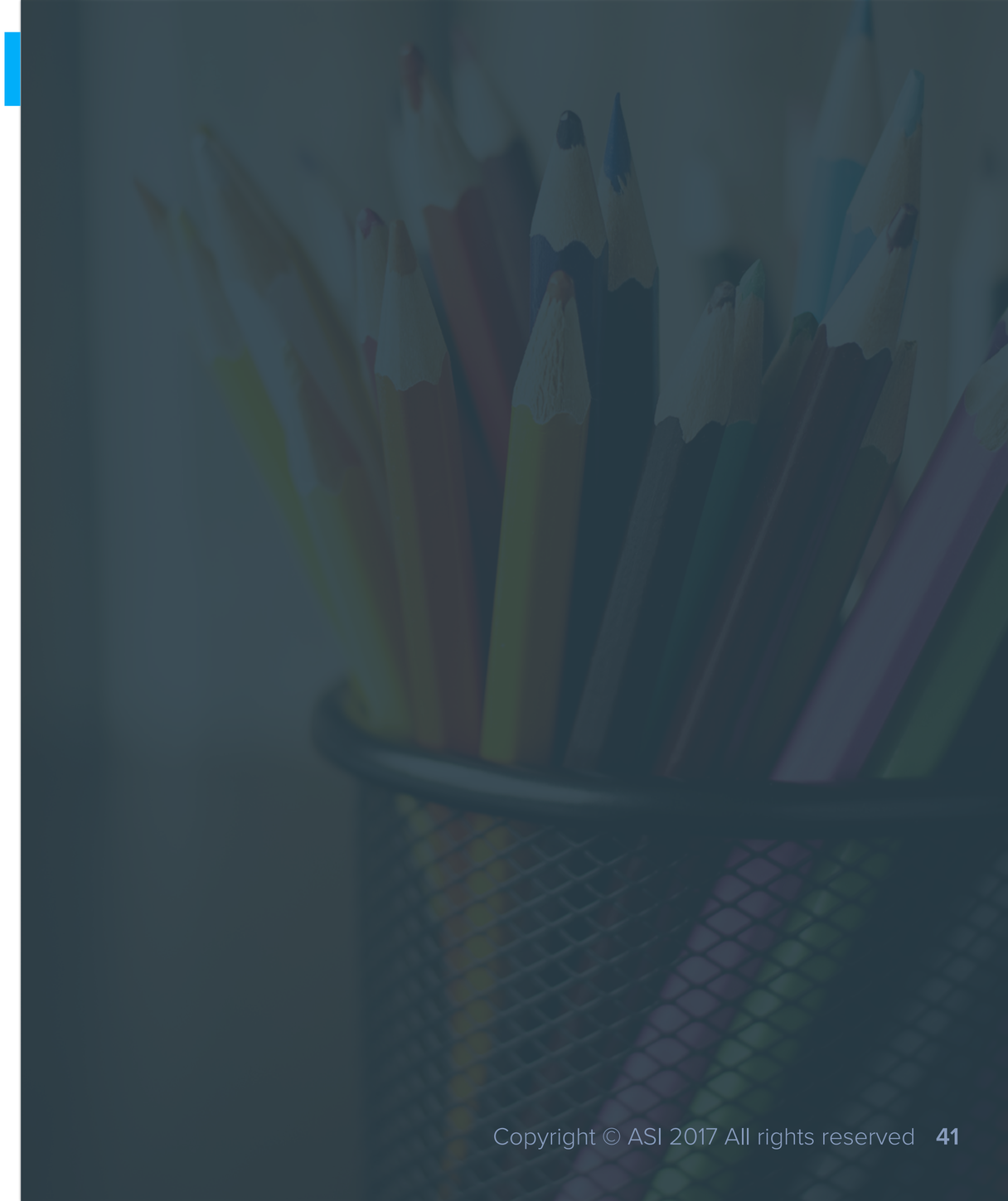
$$\frac{\partial C}{\partial \theta} = \sum_{i=m+1}^{2m} \frac{\partial}{\partial \theta} c(y_i, y(x_i;\theta)) \rightarrow \text{step}$$

$$\frac{\partial C}{\partial \theta} = \sum_{i=2m+1}^{3m} \frac{\partial}{\partial \theta} c(y_i, y(x_i;\theta)) \rightarrow \text{step}$$

ASI

EXERCISE

- Build your own function to learn MNIST parameters with stochastic gradient descent



Advanced Topics

Advanced topics

There are 3 main families of Neural Network architectures

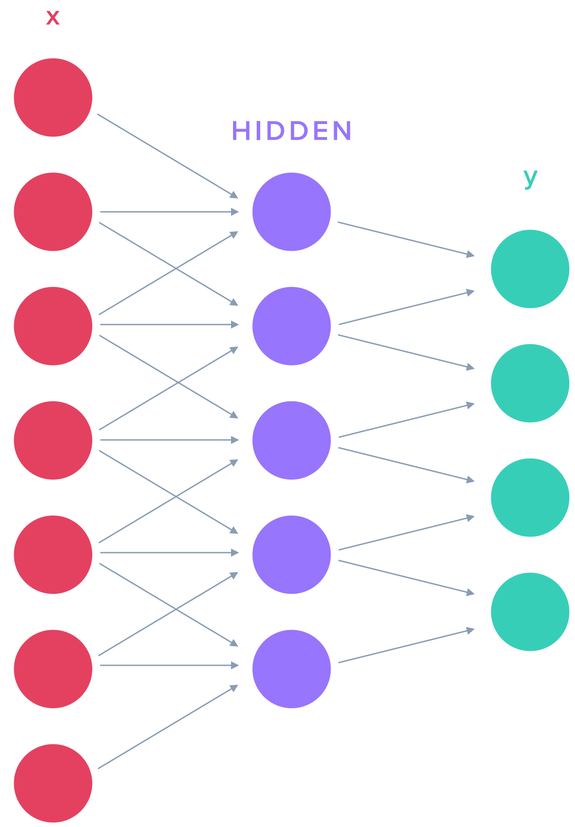
1. Feed forward
2. Convolutional
3. Recurrent

Convolutional Neural Networks

- Feed Forward Neural Networks: this is what we saw today, used for fitting non-linear functions to general data of fixed length

- Convolutional Neural Networks: used for fitting non-linear functions to data with spatial locality (e.g., edges in images)

Example Convolutional Neural Network



Convolutional Neural Networks

Equations for FFNN:

$$\vec{h} = f_1(\vec{x} \cdot W_1 + \vec{b}_1)$$

$$\vec{y} = f_2(\vec{h} \cdot W_2 + \vec{b}_2)$$

Equations for CNN:

$$\vec{h} = f_1(\vec{x} * W_1 + \vec{b}_1)$$

$$\vec{y} = f_2(\vec{h} * W_2 + \vec{b}_2)$$

Scalar product compares all values of x through W (e.g., it is non-local):

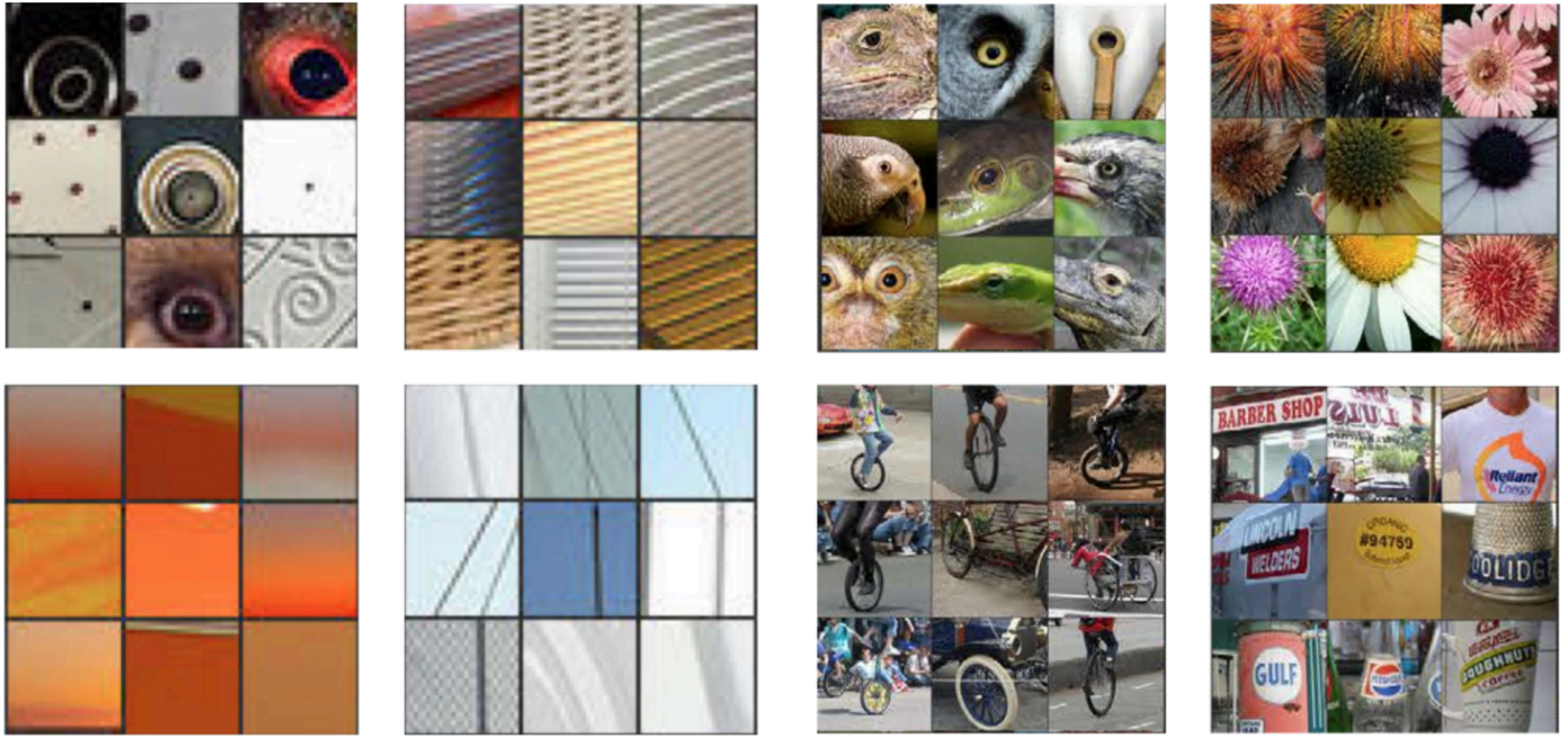
$$(x \cdot W)_j = \sum_{k=1}^{\text{dim of } x} x_k W_{kj}$$

Convolution only compares nearby values of x depending on size of filter, W:

$$(x * W_1)_{ij} = \sum_{k=1}^{\text{Filter size}} x_{i+k}(W_1)_{kj}$$

Since Filter size << dim of x, convolutions use many fewer parameters; they have to fit to all the data points, versus just 1 for FFNN

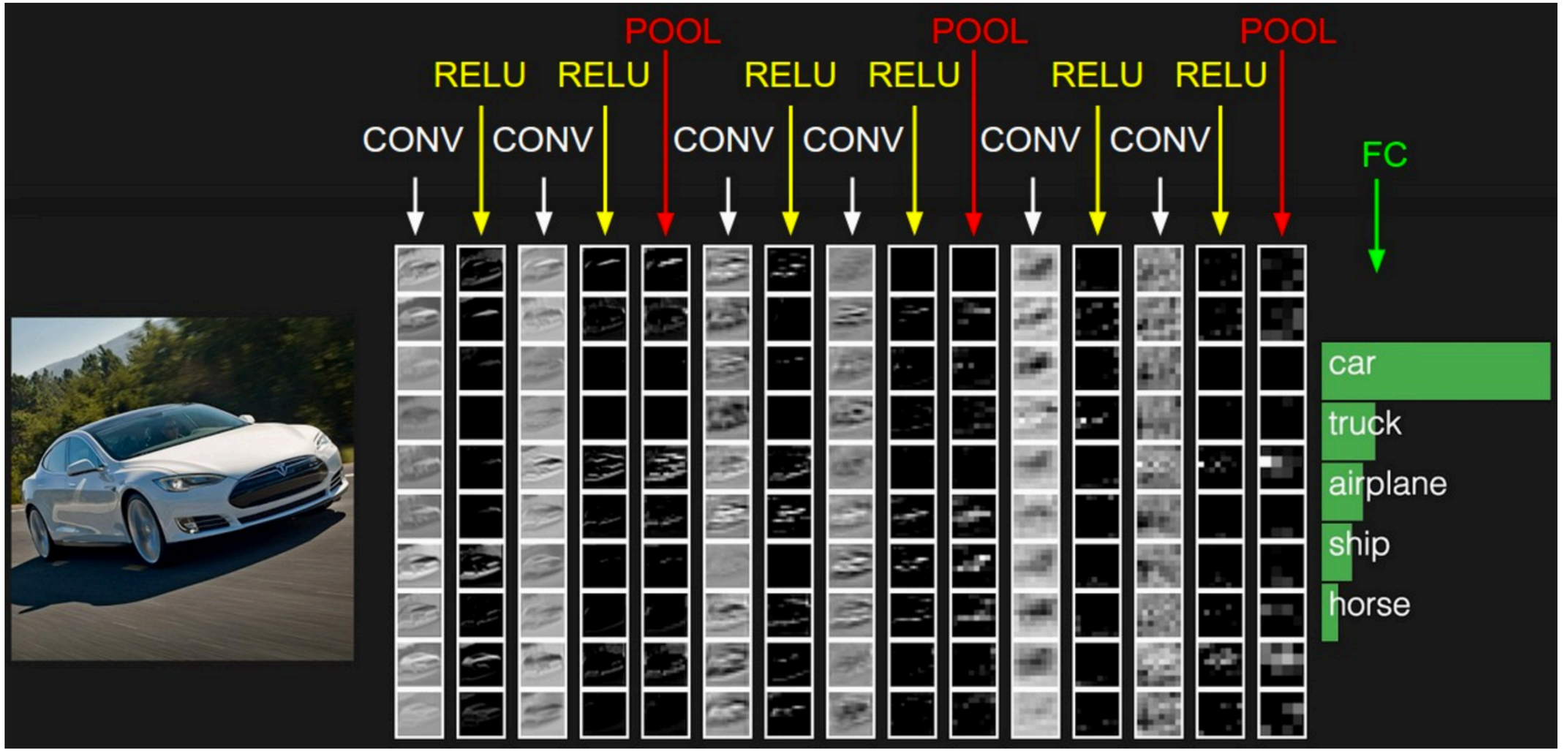
Convolutional Neural Networks



Low level features (e.g., what CNN filters learn - h's)

High level features (e.g., what is in images)

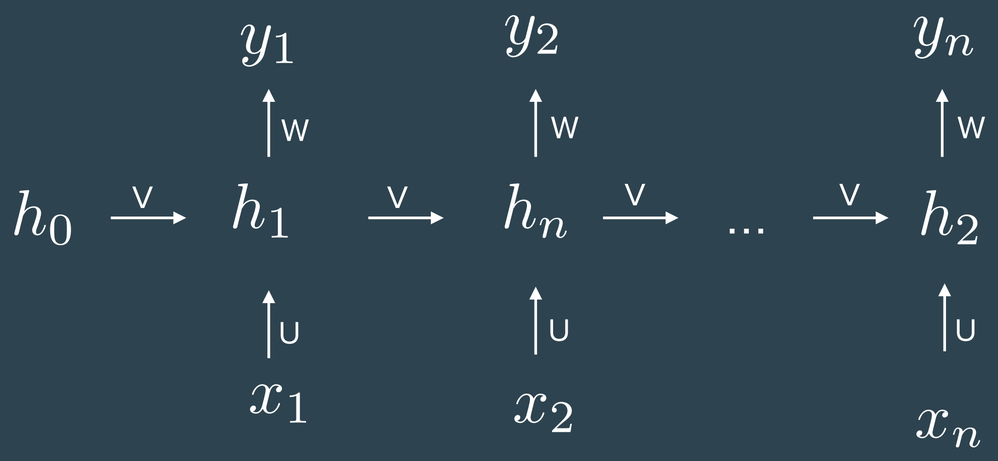
Convolutional Neural Networks



Volume of activations along the processing path

Recurrent Neural Networks

Basic RNN architecture:



$$\vec{h}_0 = \vec{0}$$

$$\vec{h}_t = \tanh(U \cdot \vec{x}_t + V \cdot \vec{h}_{t-1} + \vec{b}_t)$$

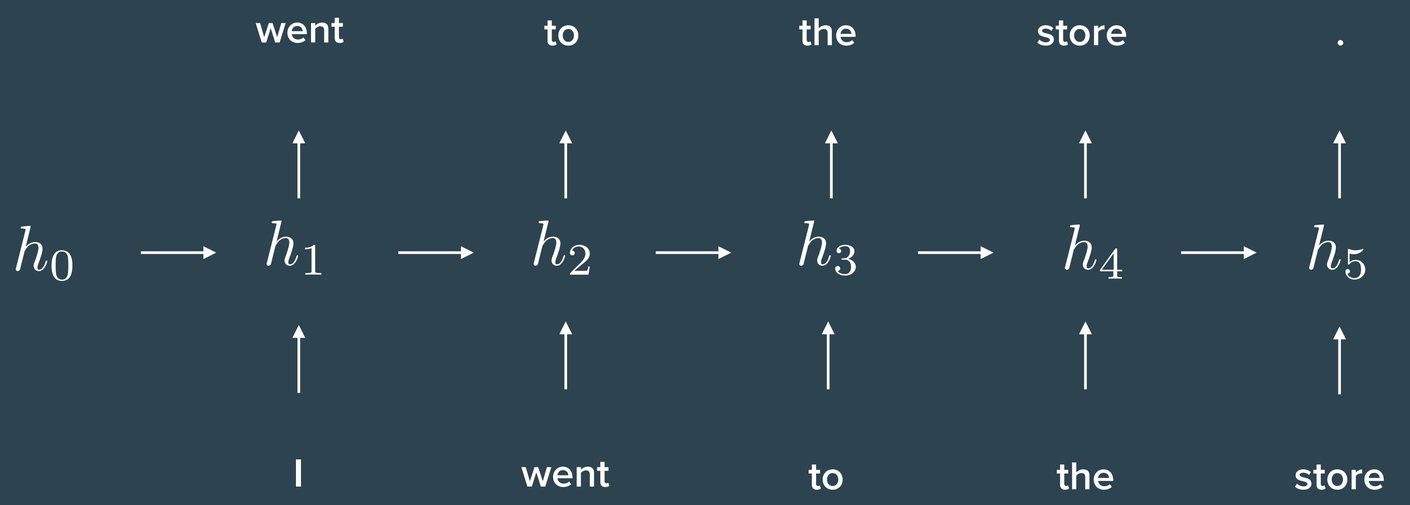
$$\vec{y}_t = \text{softmax}(W \cdot \vec{h}_t + \vec{c}_t)$$

Attributes of RNNs

- Uses same parameters everywhere, so learns underlying “laws” in data
- Learns local relationships between data points - just like CNNs
- Can deal with data of any length (e.g., time series, sentences)

Recurrent Neural Networks

Unsupervised training:



- Minimise the distance between x = word(t), and y = word(t+1), for all words in each sentence, and all sentences in corpus
- Learns the structure of language

Recurrent Neural Networks

Example:

A RNN writing Shakespeare

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```


Recurrent Neural Networks

Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{array}{ccc} \mathcal{S} & \longrightarrow & \\ \downarrow & & \\ \xi & \longrightarrow & \mathcal{O}_{X'} \\ & & \uparrow \\ \mathrm{gor}_s & & \end{array} \qquad \begin{array}{ccc} = \alpha' & \longrightarrow & \\ \updownarrow & & \\ = \alpha' & \longrightarrow & \alpha \end{array} \qquad \begin{array}{c} X \\ \downarrow \\ \mathrm{d}(\mathcal{O}_{\mathcal{X}_{X/k}}, \mathcal{G}) \end{array}$$

$$\mathrm{Spec}(K_\psi) \qquad \mathrm{Mor}_{Sets}$$

is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type f_*. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \mathrm{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.
A *reduced above* we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\overline{x}} \quad \text{-1}(\mathcal{O}_{X_{\acute{e}tale}}) \longrightarrow \mathcal{O}_{X_\ell}^{-1}\mathcal{O}_{X_\lambda}(\mathcal{O}_{X_\eta}^{\overline{v}})$$

is an isomorphism of covering of $\mathcal{O}_{X_i}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_\lambda}$ is a closed immersion, see Lemma ??. This is a sequence of $\mathcal{F}$ is a similar morphism.

Advanced topics

Other topics to consider:

- Tuning hyperparameters
- Regularisation
- Early stopping
- Dropout
- LSTMs & other RNN cells

Q & A

