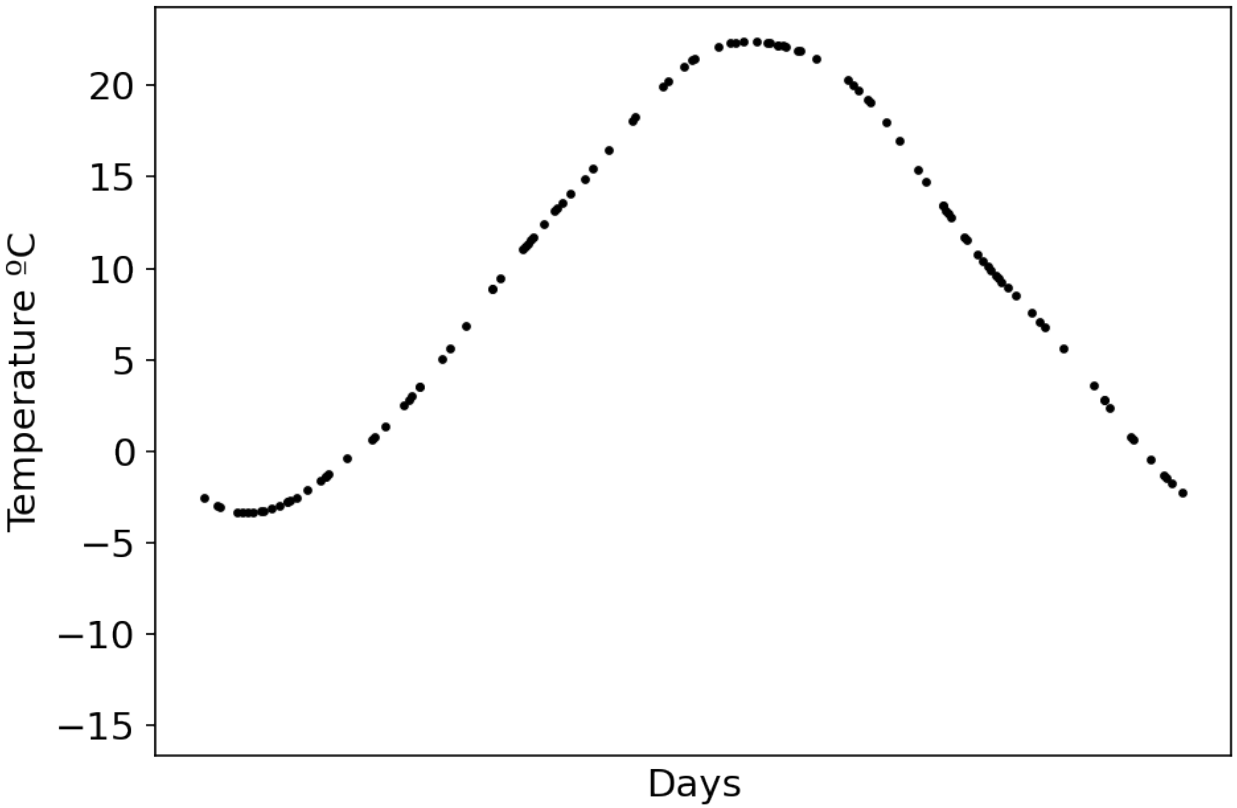
# What is a cool application for deep learning? (underscores for multi-word responses)

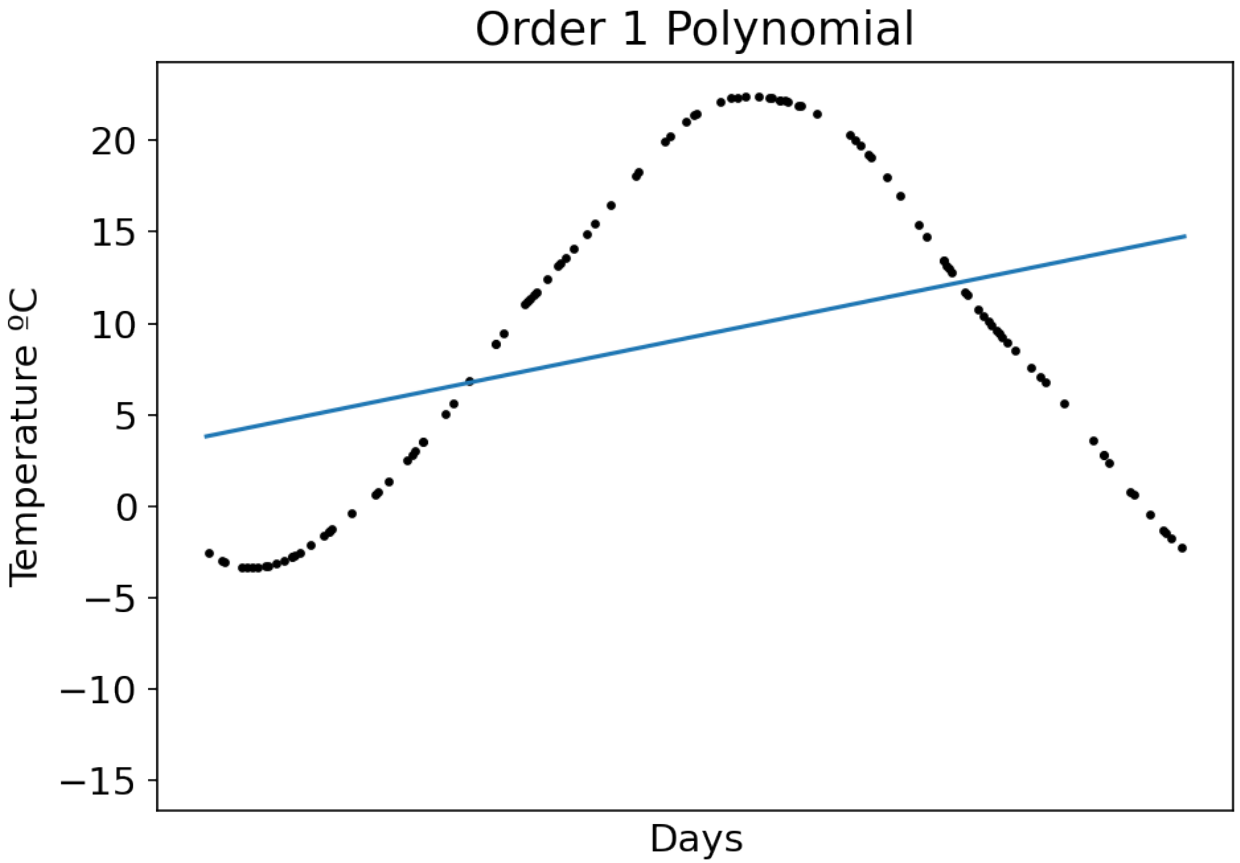
# Introduction to Deep Learning

# What is deep learning?

# How can we predict unseen data?

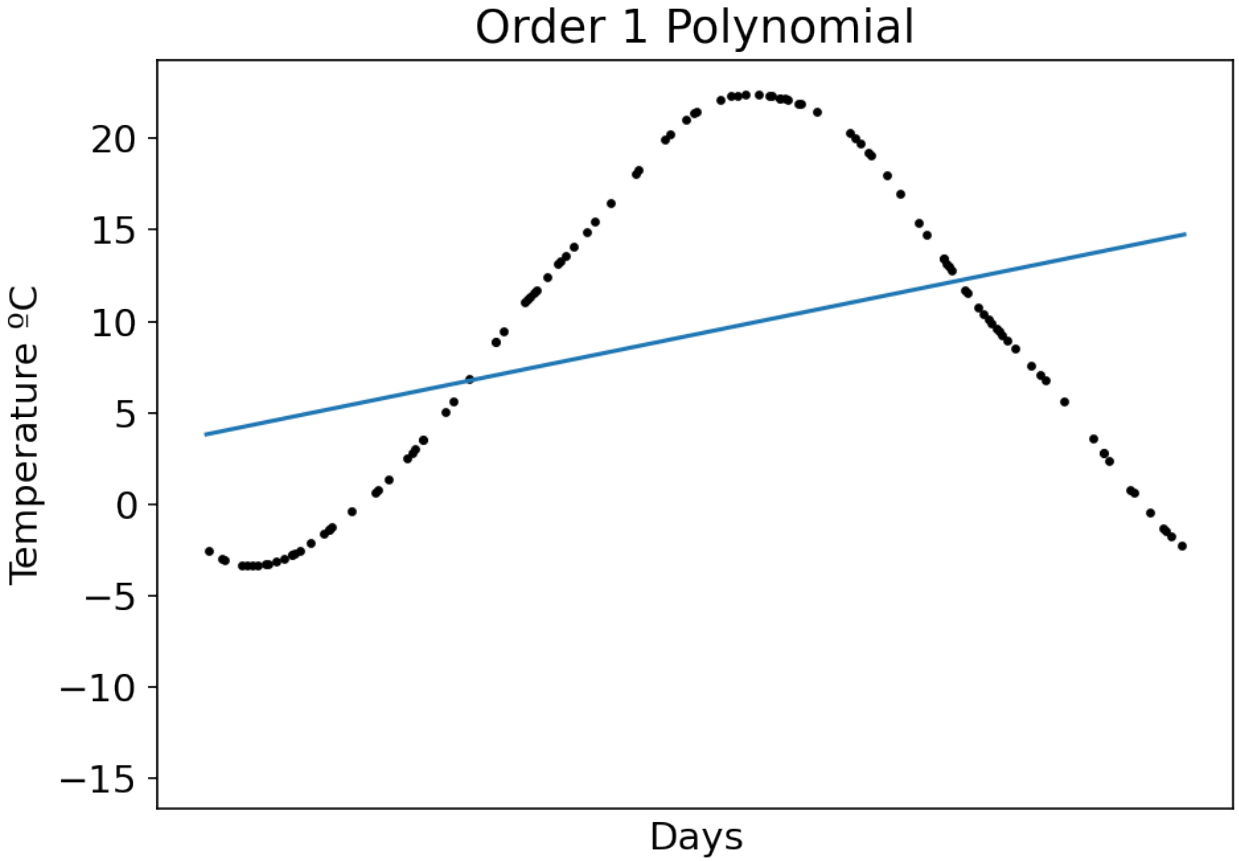


# How can we predict unseen data?




$$y = f(x, \theta)$$
$$y = \theta_0 + \theta_1 x$$
$$y = 3.81 + 0.03x$$

# How can we predict unseen data?



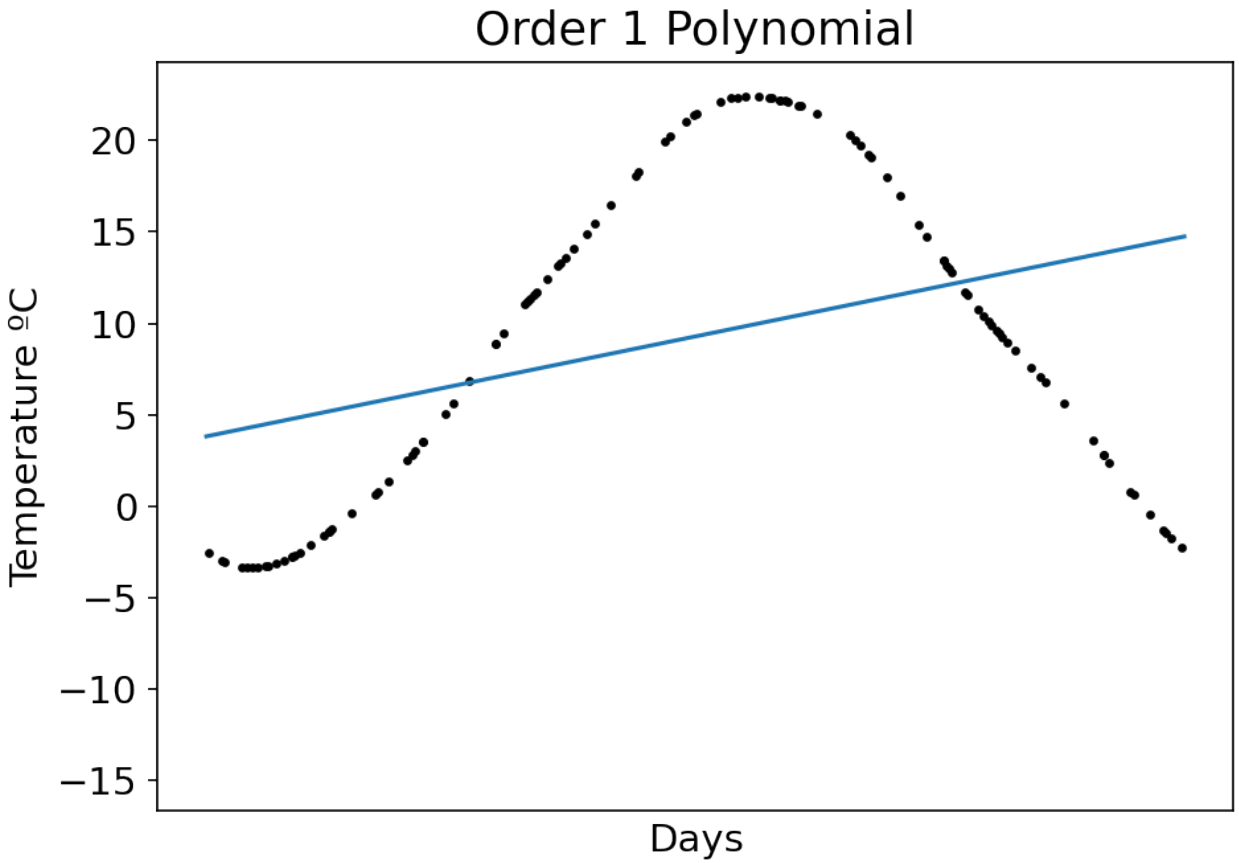

$$y = f(x, \theta)$$

$$y = \theta_0 + \theta_1 x$$

$$y = 3.81 + 0.03x$$

Bias Weight

# How can we predict unseen data?



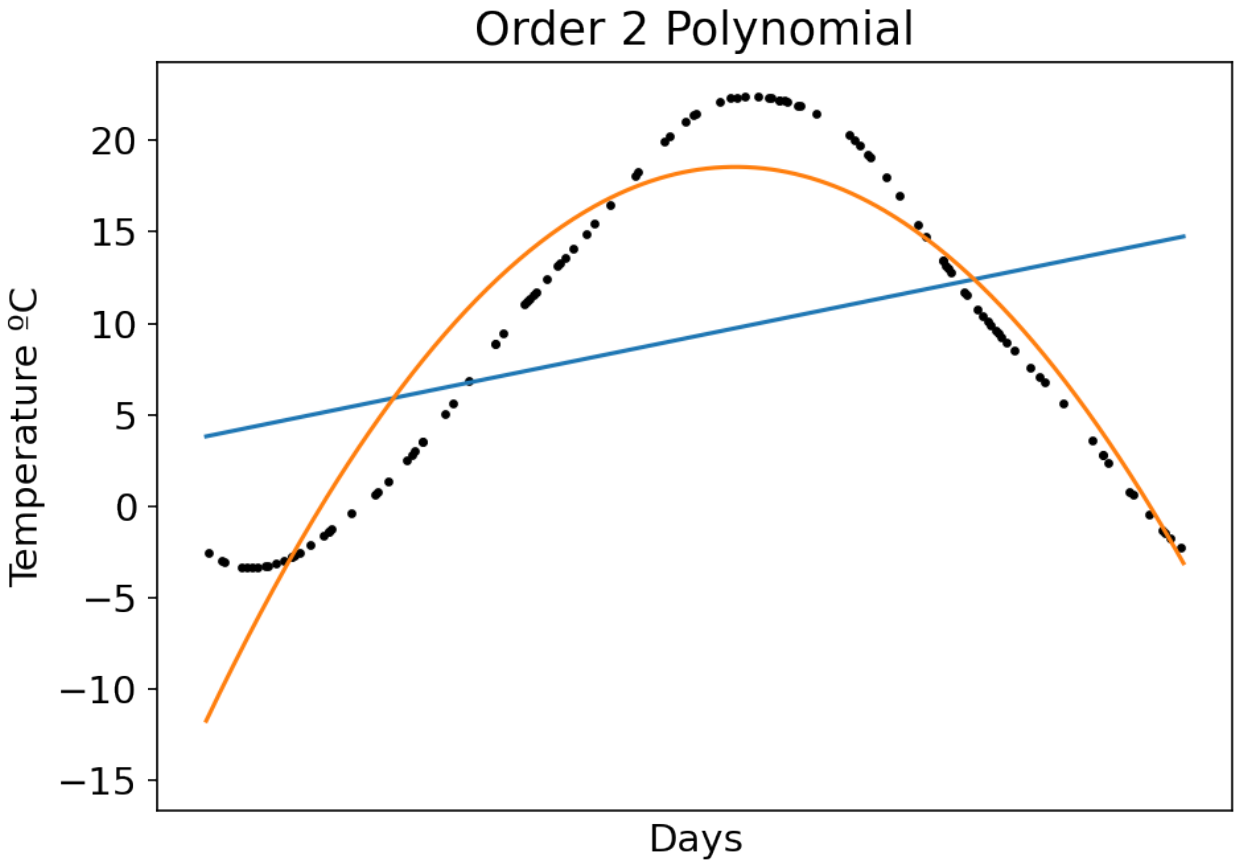

$$y = f(x, \theta)$$

$$y = \theta_0 + \theta_1 x$$

$$y = 3.81 + 0.03x$$

$$y = \theta_0 + \sum_i \theta_i x^i$$

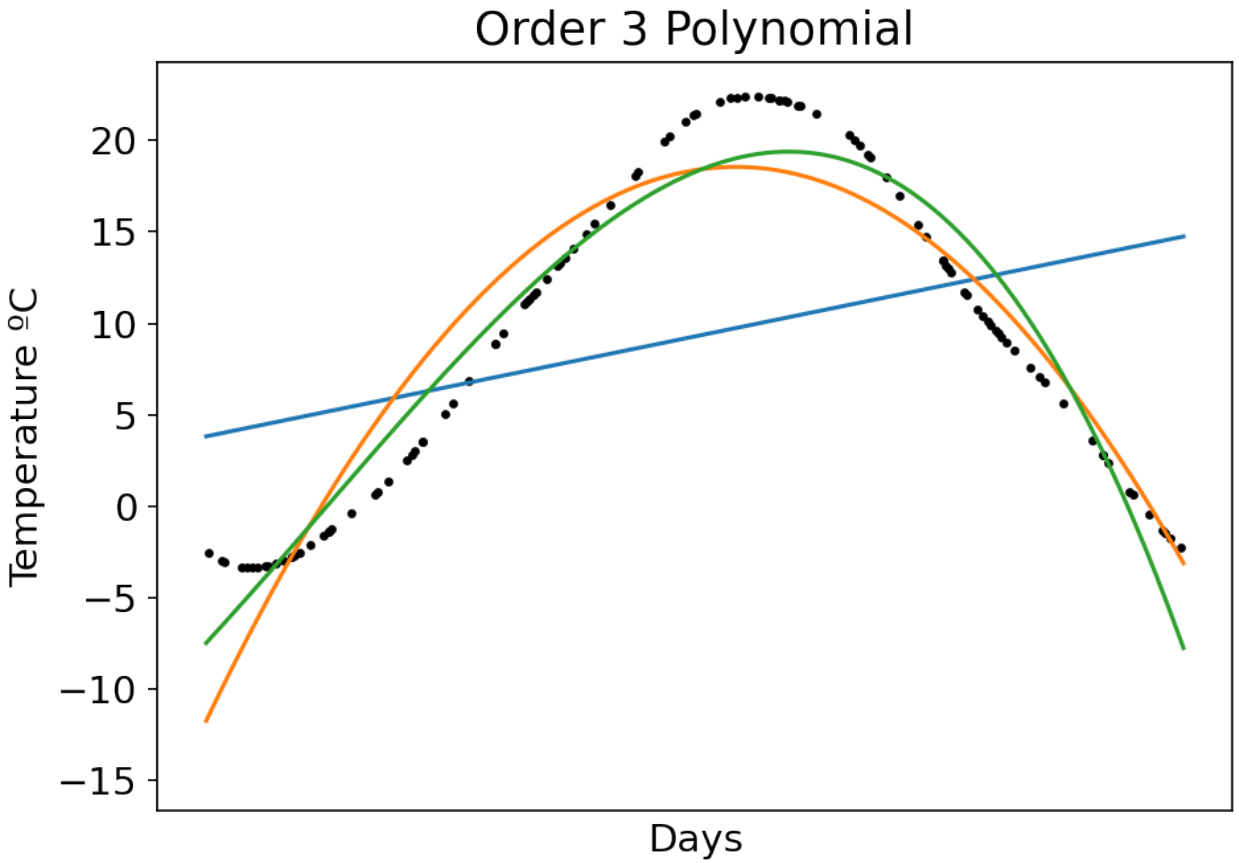
# How can we predict unseen data?




$$y = f(x, \theta)$$

$$y = \theta_0 + \theta_1 x + \theta_2 x^2$$

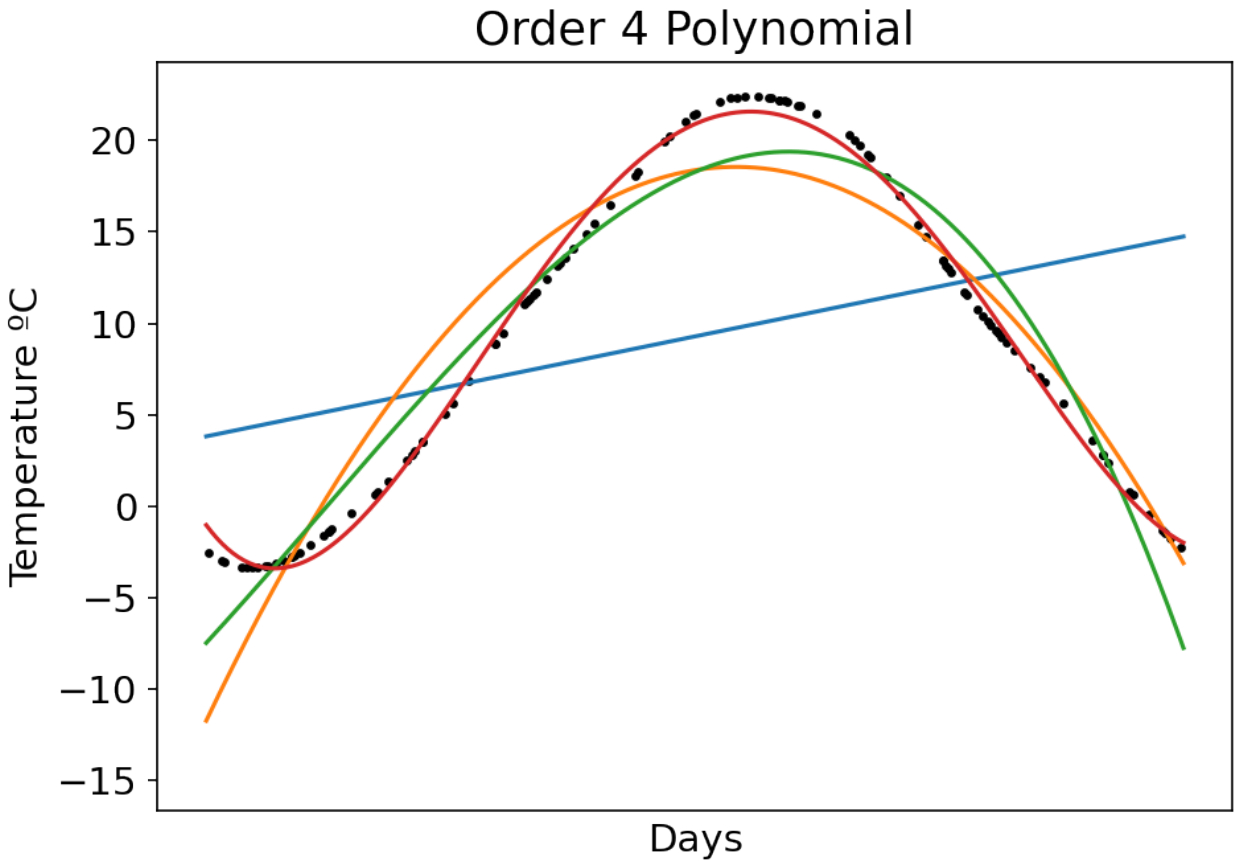
# How can we predict unseen data?




$$y = f(x, \theta)$$

$$y = \theta_0 + \cdots + \theta_3 x^3$$

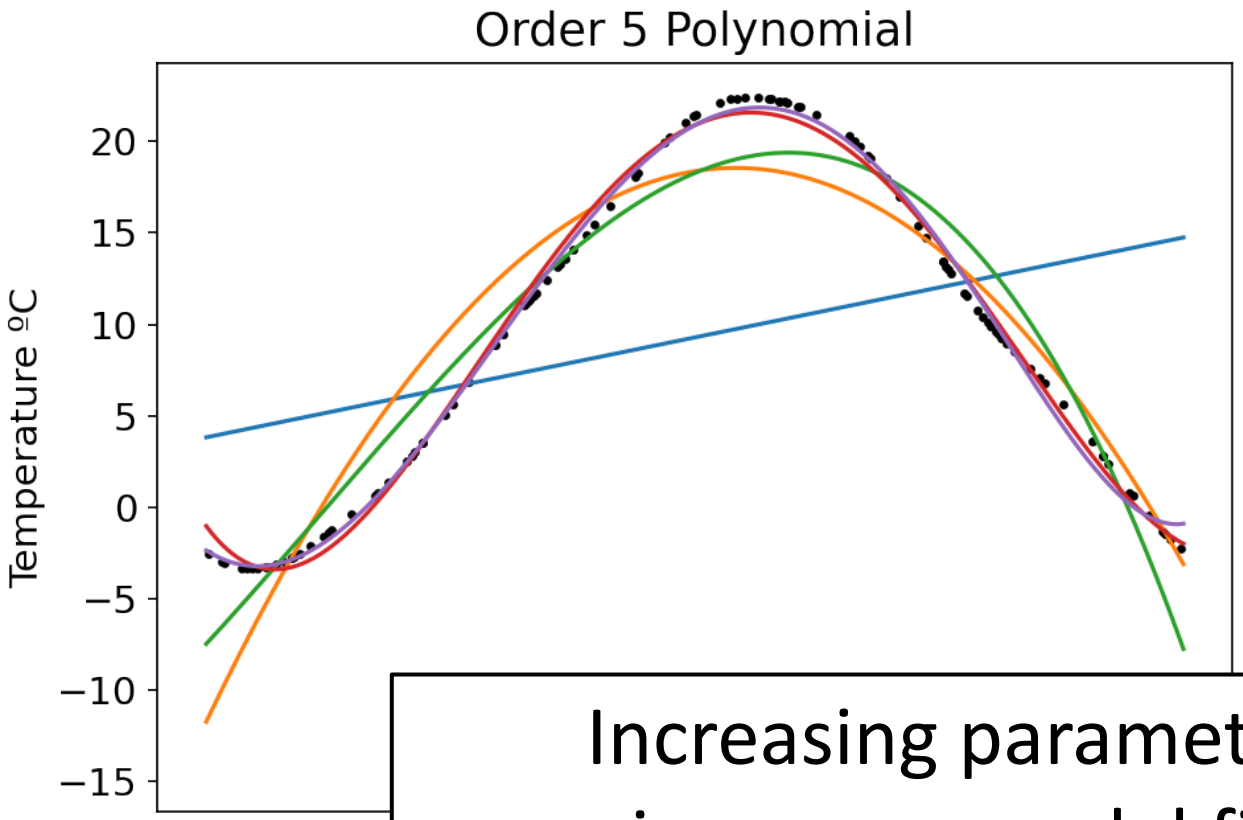
# How can we predict unseen data?




$$y = f(x, \theta)$$

$$y = \theta_0 + \cdots + \theta_4 x^4$$

# How can we predict unseen data?

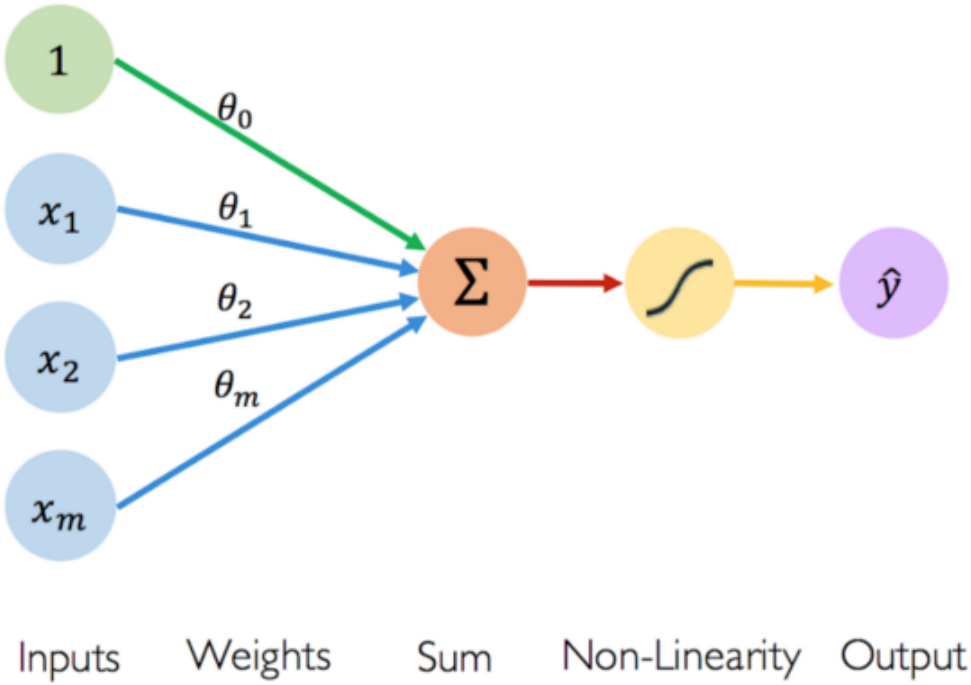



$$y = f(x, \theta)$$

$$y = \theta_0 + \cdots + \theta_5 x^5$$

Increasing parameters improves model fit!

# The heart of neural networks are weights and biases

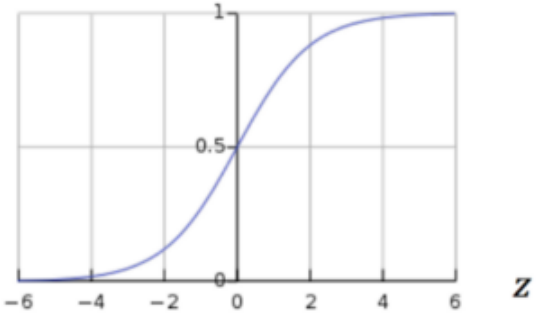



## Activation Functions

$$\hat{y} = g(\theta_0 + \boldsymbol{X}^T\boldsymbol{\theta})$$

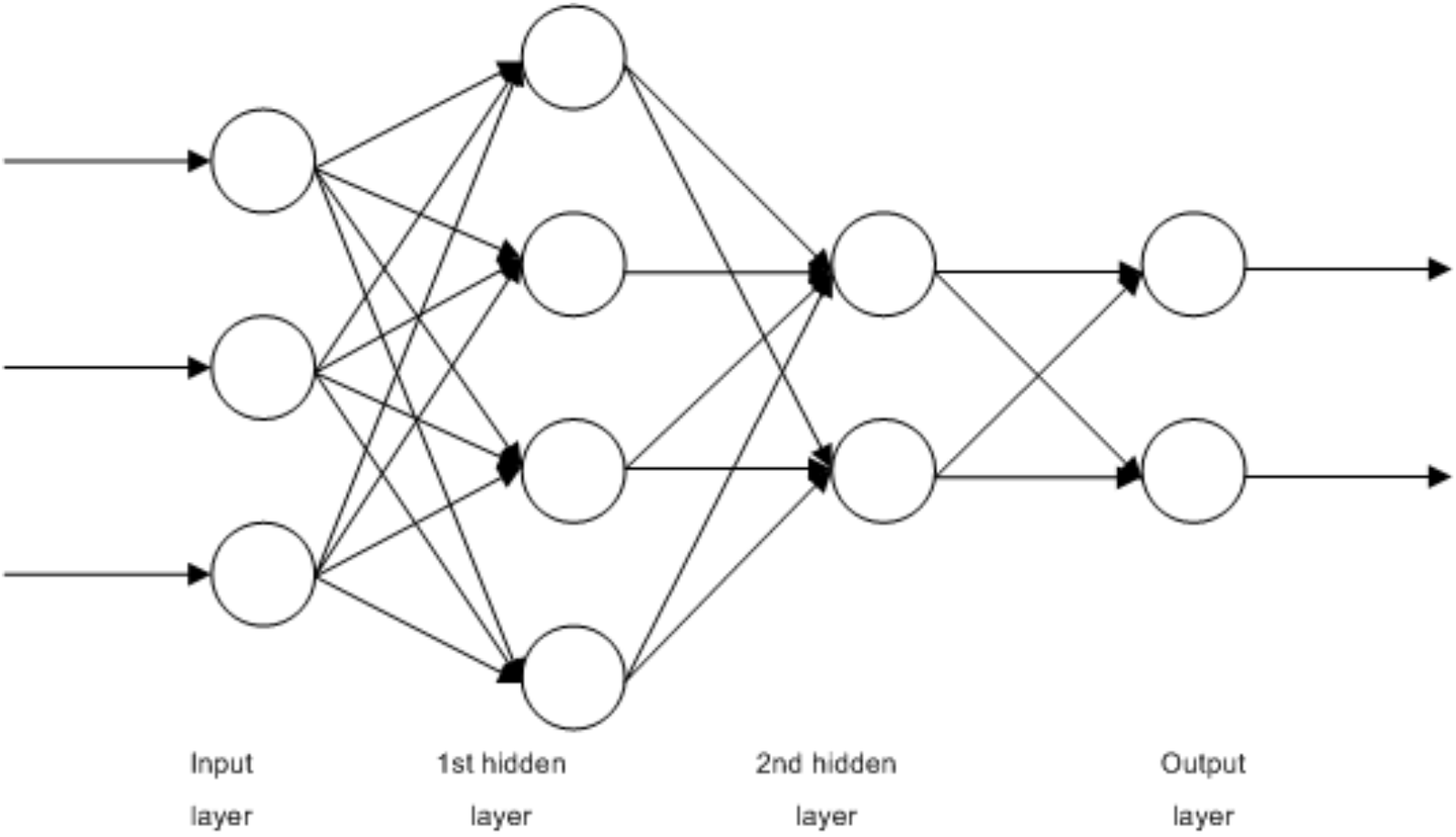
- Example: sigmoid function

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



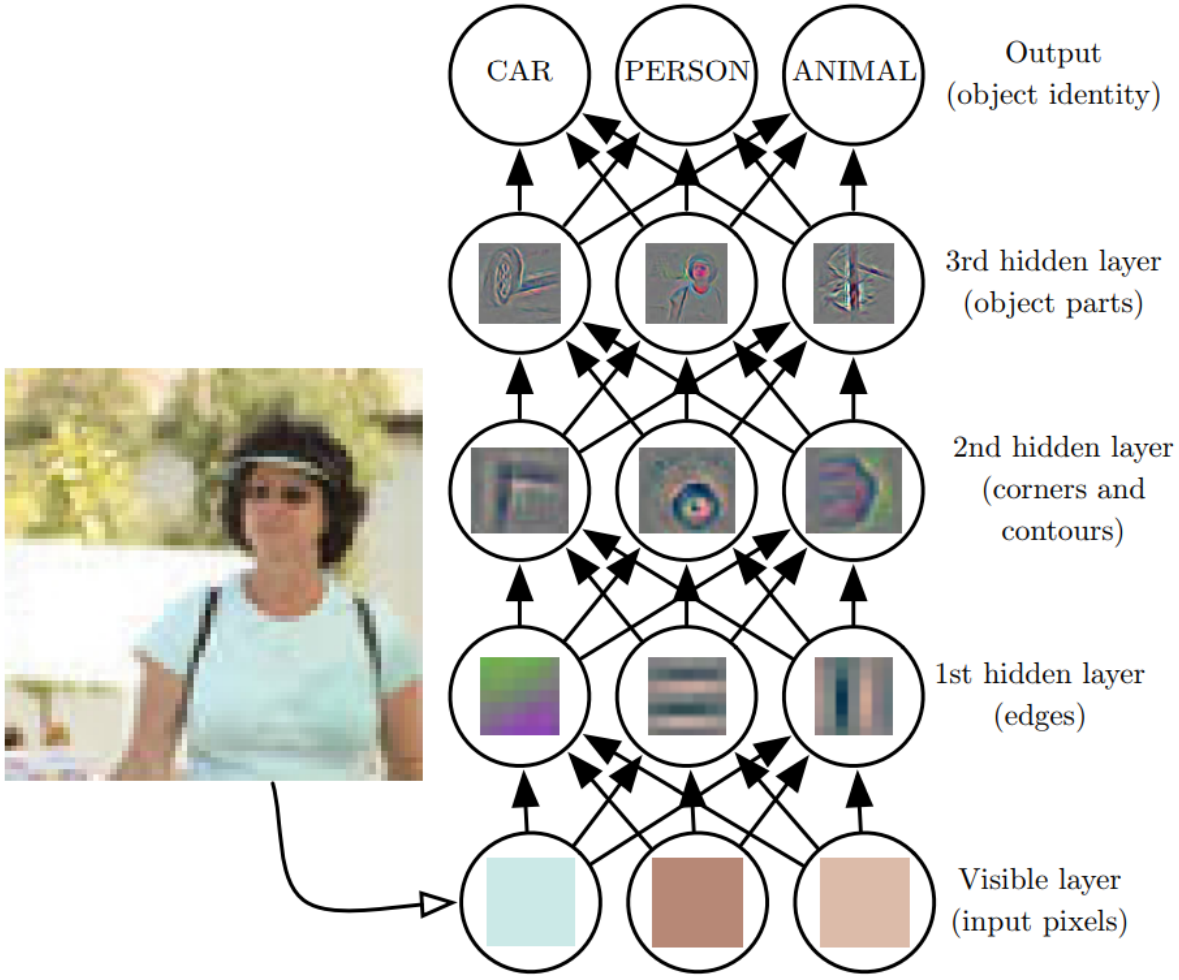

MIT: Alexander Amini, 2018 introtodeeplearning.com

# Nodes of a network are arranged in layers to form deep networks




WikiMedia

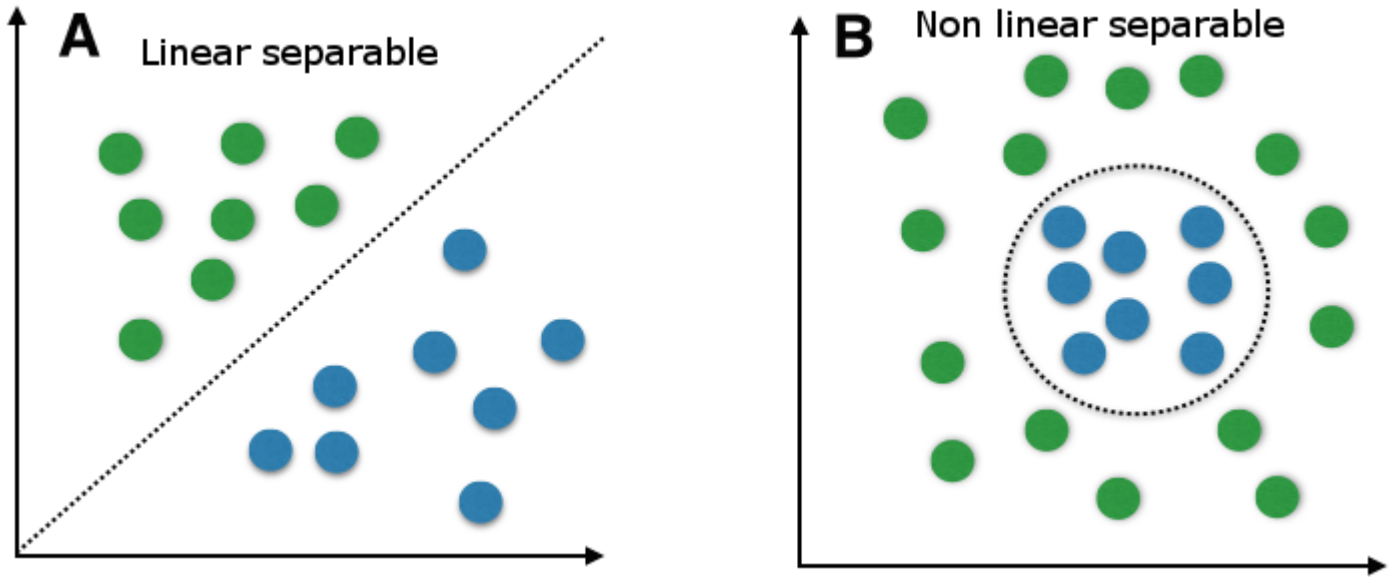
# Network depth creates a composition of functions




Goodfellow et al., 2016

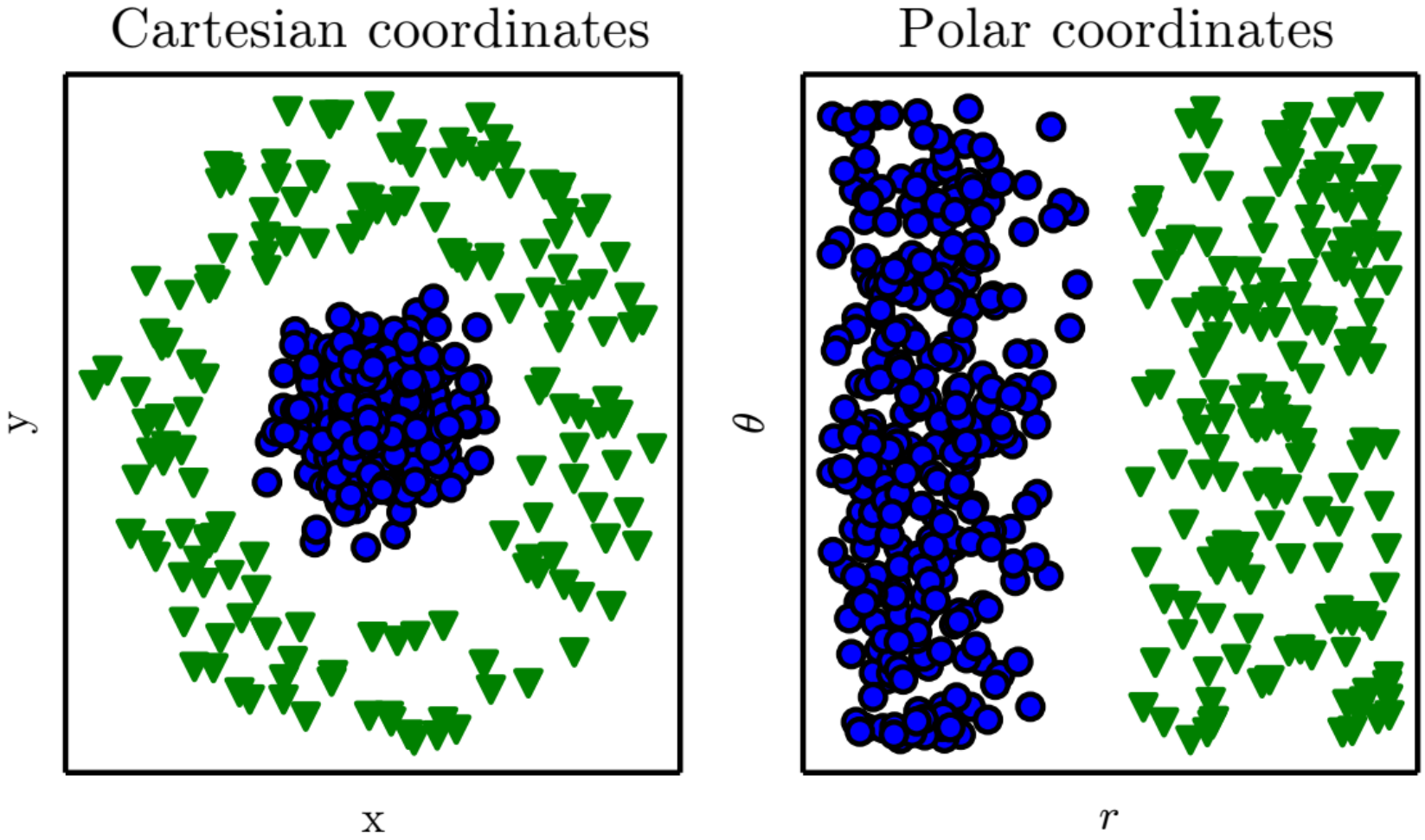
# Why go deep?

Depth = complexity



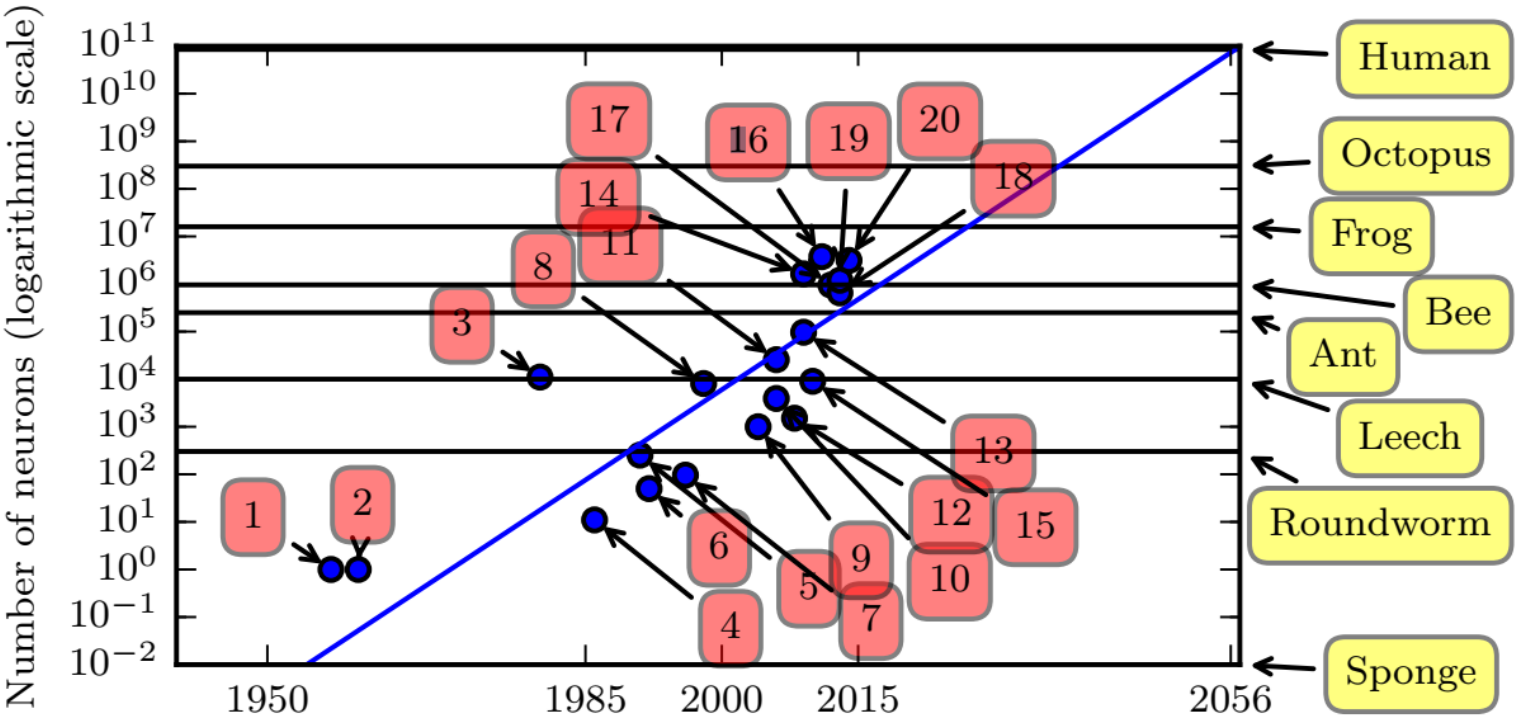

Single layer perceptrons can only handle linear separability

# Representations matter




Goodfellow et al., 2016

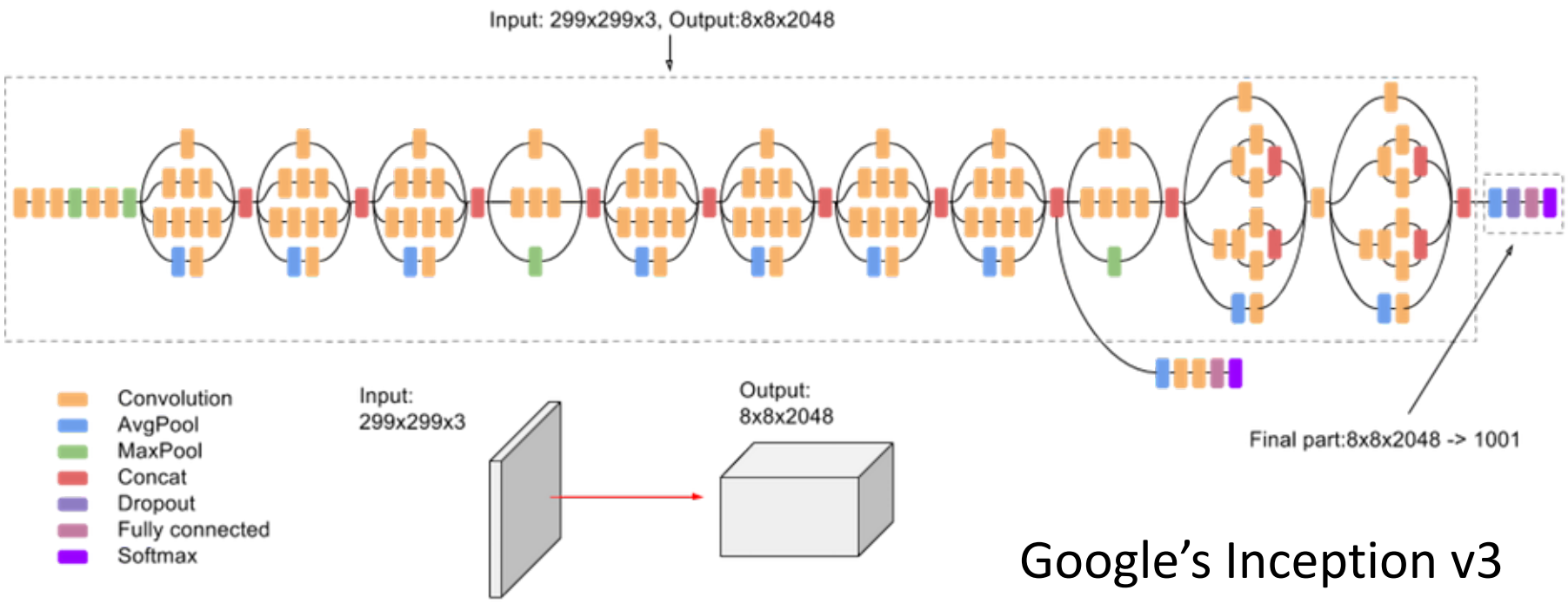
# Increasing # of neurons




1. Perceptron (Rosenblatt, 1958)
4. Early backpropagation network
6. MLP for speech recognition (Bengio et al, 1991)
11. GPU-accelerated convolutional network (Challeapilla et al., 2006)
20. GoogLeNet (Szegedy et al., 2014a)

Goodfellow et al., 2016

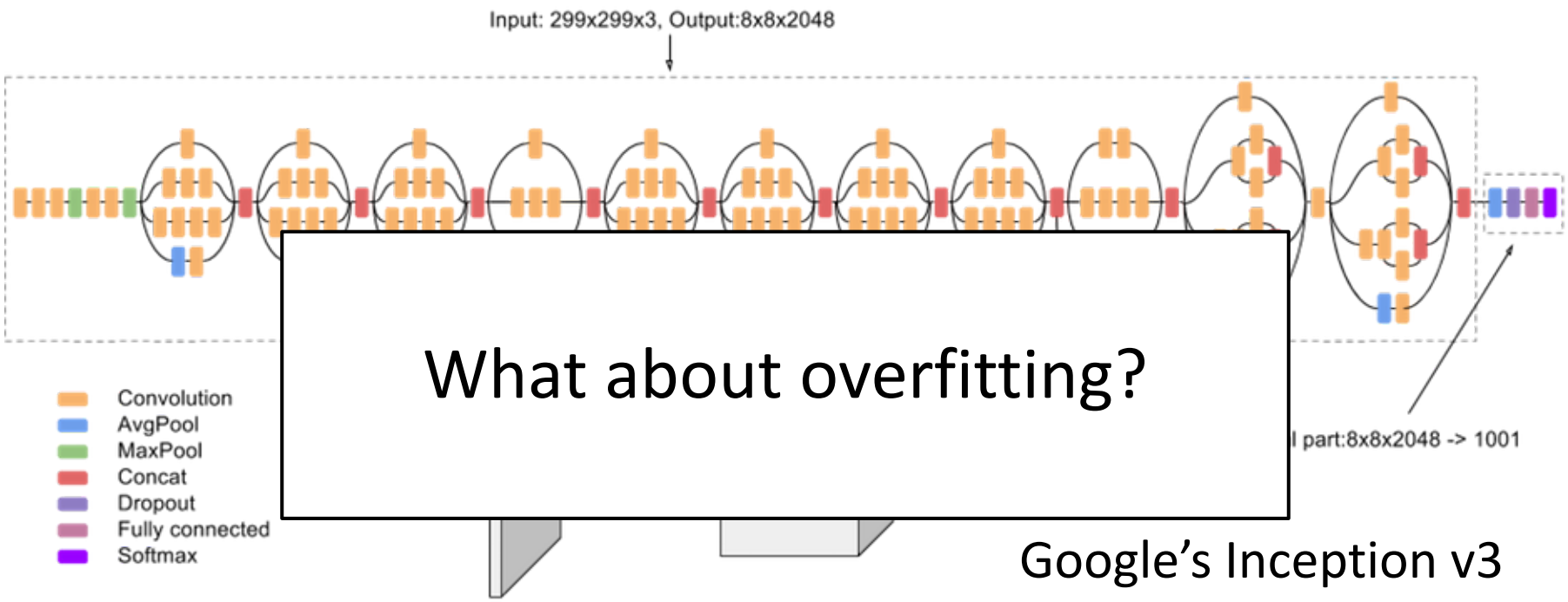
# Layers can be arranged in arbitrary configurations to increase model power




Google's Inception v3 Image Recognition network has 24M parameters!

Google

# Layers can be arranged in arbitrary configurations to increase model power

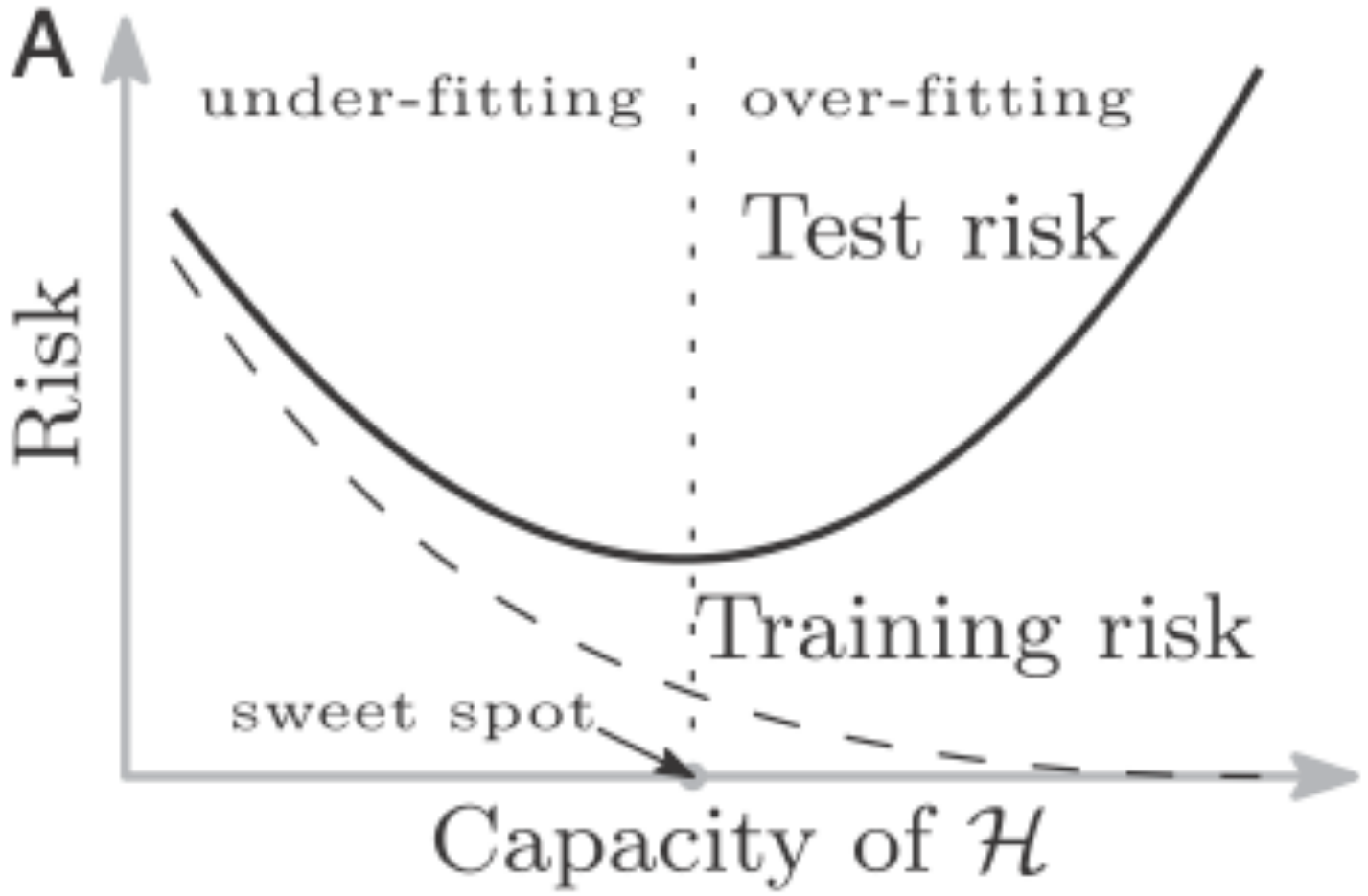



What about overfitting?

Google's Inception v3 Image Recognition network has 24M parameters!

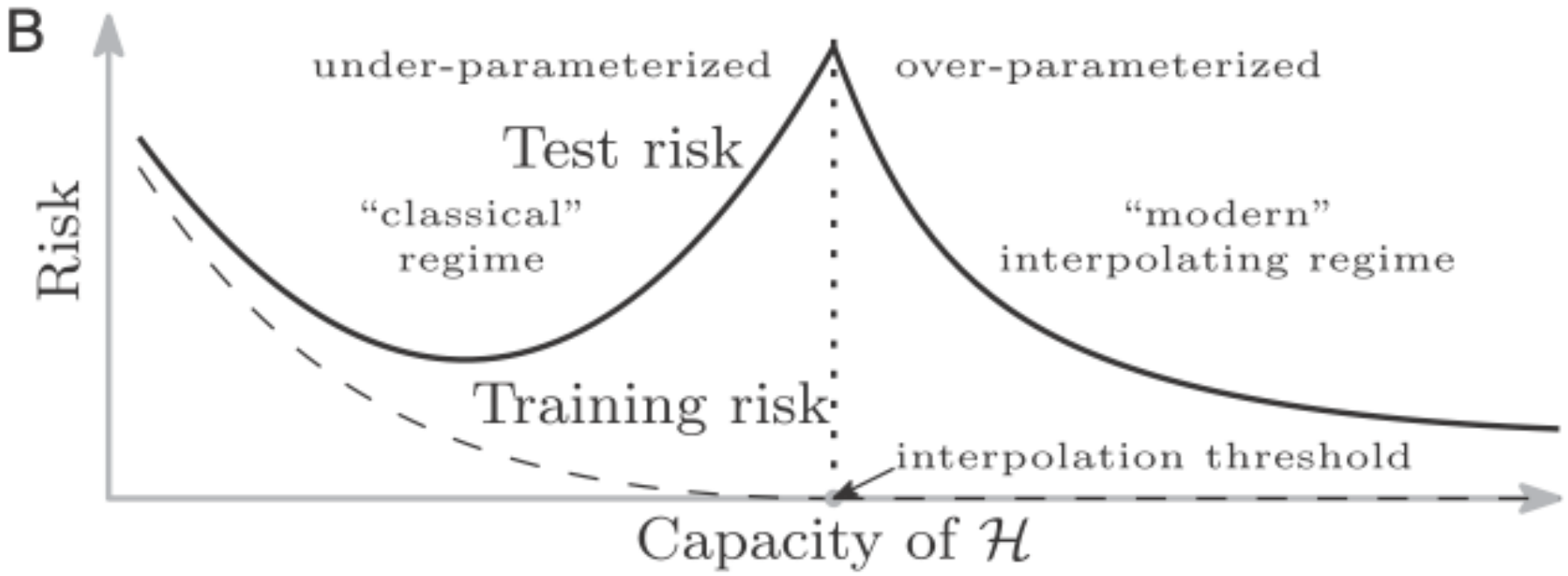
Google

# Classic risk curve



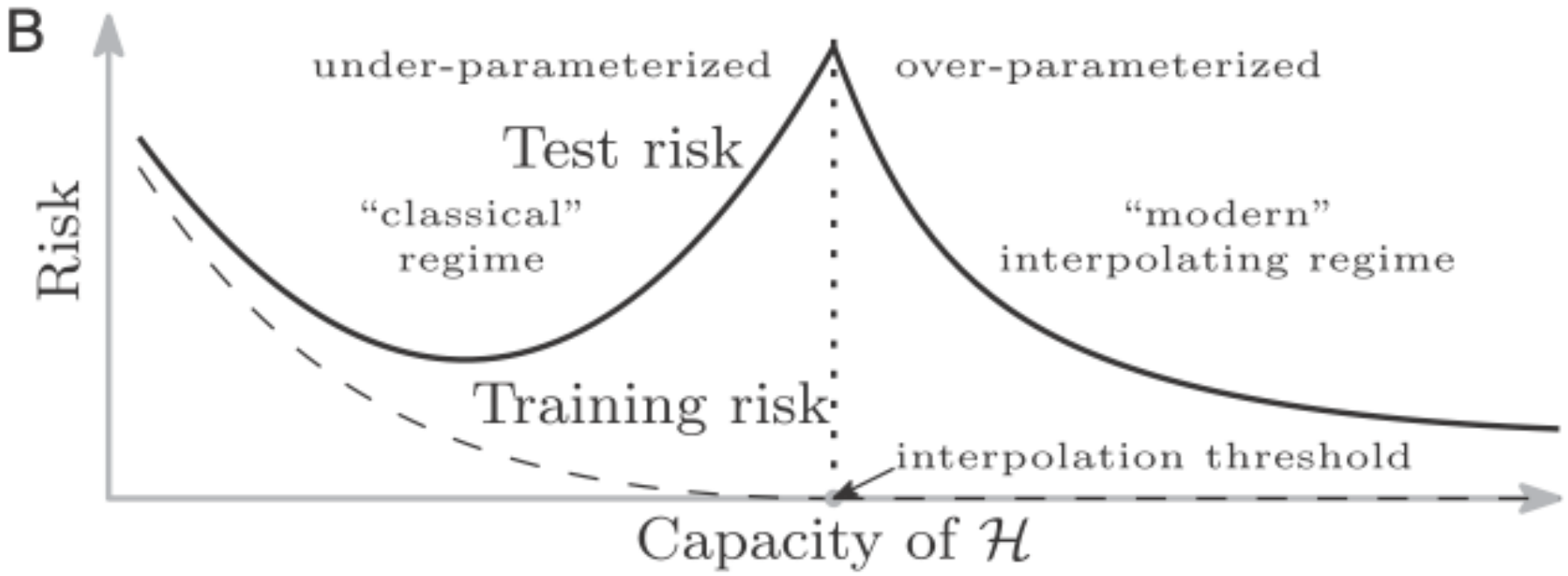

Belkin et al. PNAS (2019)

# Double Descent



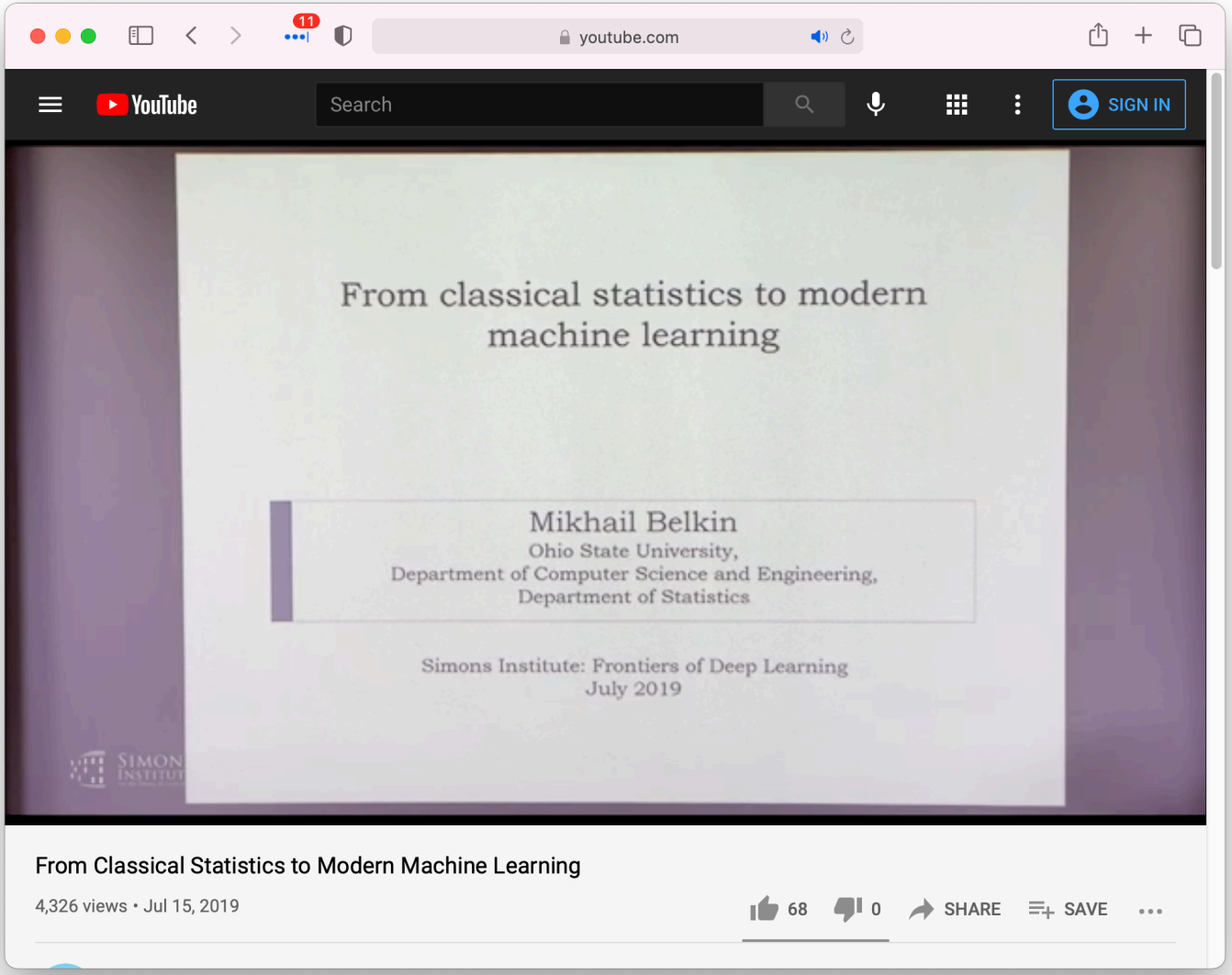

Belkin et al. PNAS (2019)

# Double Descent




Belkin et al. PNAS (2019)

# Double Descent




https://www.youtube.com/watch?v=OBCciGnOJVs&feature=youtu.be

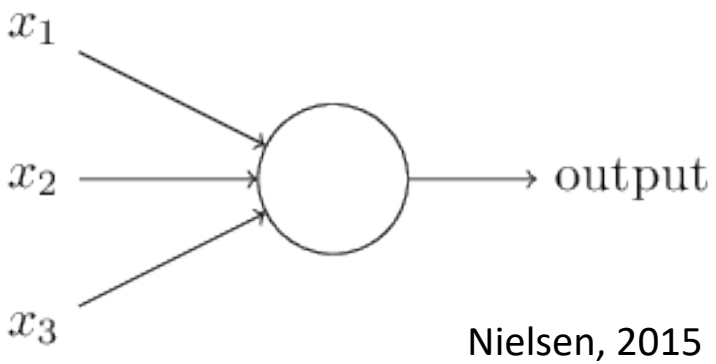
# Differentiable Computing

*The important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization….It's really very much like a regular program, except it's parameterized, automatically differentiated, and trainable/optimizable.*

*- Yann LeCun, Director of FAIR*

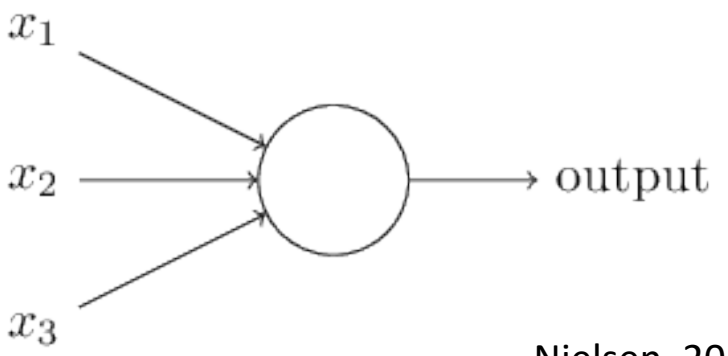
# The perceptron

- Developed in 1950's and 1960's by Frank Rosenblatt
- Binary inputs
- Single binary output
- Example:




Nielsen, 2015

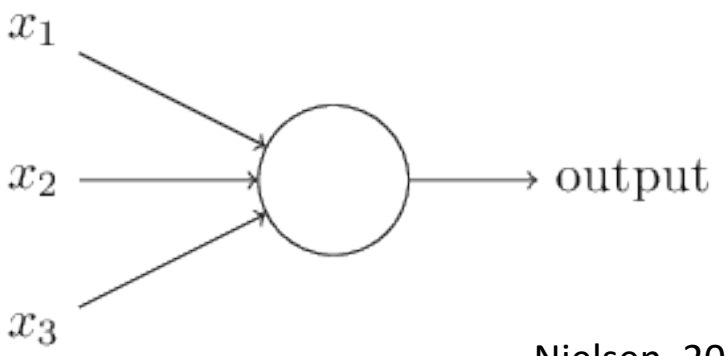
# The perceptron




Nielsen, 2015

- Computing the output:
  - Assign weights to each input
  - Determine if weighted sum of inputs is greater than some threshold

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$




Nielsen, 2015

- Example: Decide whether to attend a cheese festival
- Three factors:
  1. Is the weather good? $x_1$
  2. Does your friend want to accompany you? $x_2$
  3. Is the festival near public transit? (you don't own a car) $x_3$

$$x_j = \begin{cases} 0, \text{if no} \\ 1, \text{if yes} \end{cases}$$

- Example: Decide whether to attend a cheese festival
- Three factors:
  1. Is the weather good? $x_1$
  2. Does your friend want to accompany you? $x_2$
  3. Is the festival near public transit? (you don't own a car) $x_3$
- **Case 1**: Love cheese but hate bad weather
  - $w_1 = 6$
  - $w_2 = 2$
  - $w_3 = 2$
  - Threshold$= 5$
  - $\sum_j w_j x_j >$ threshold whenever weather is **good** ($x_1 = 1$)
  - $\sum_j w_j x_j <$ threshold whenever weather is **bad** ($x_1 = 0$)

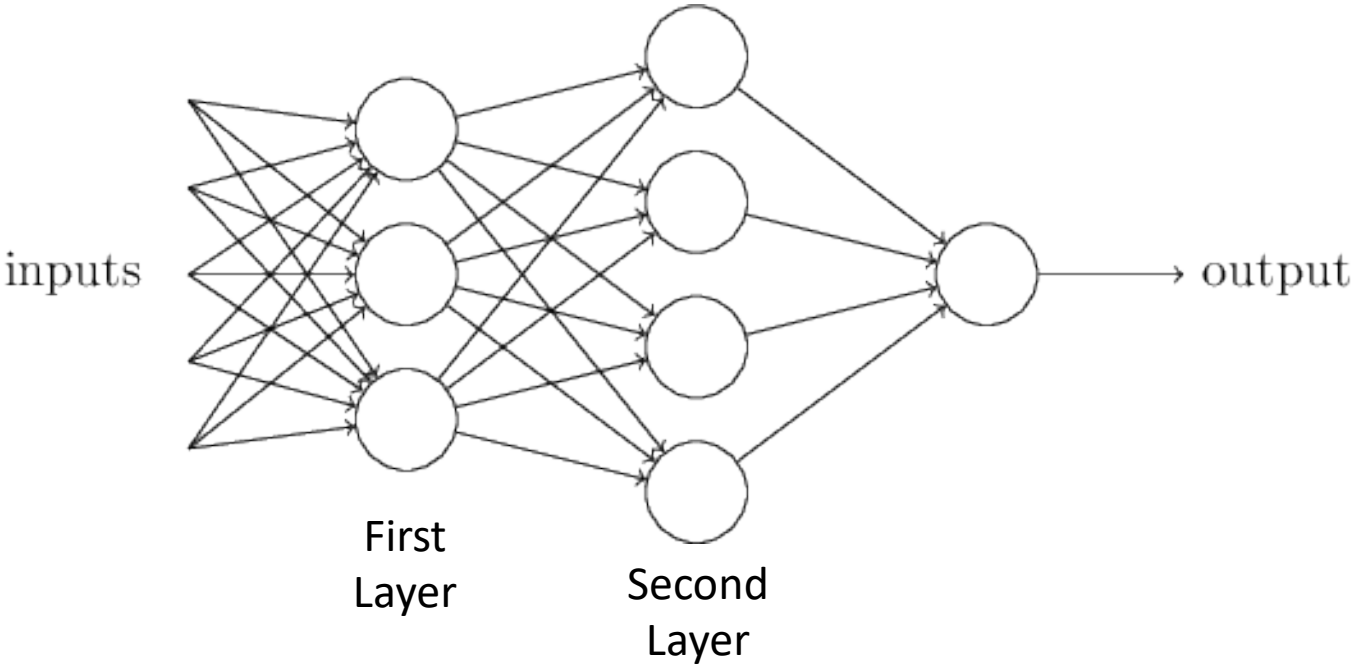
- Example: Decide whether to attend a cheese festival
- Three factors:
  1. Is the weather good? $x_1$
  2. Does your friend want to accompany you? $x_2$
  3. Is the festival near public transit? (you don't own a car) $x_3$
- **Case 2**: Love cheese but don't hate bad weather as much
  - $w_1 = 6$
  - $w_2 = 2$
  - $w_3 = 2$
  - Threshold$= 3$
  - $\sum_j w_j x_j >$ threshold whenever weather is **good** ($x_1 = 1$) *or* friend will go ($x_2 = 1$) *and* when the festival is near public transit ($x_3 = 1$)

# Conclusions

- Neural networks are collections of individual neurons or nodes that multiple input values by weights and apply a non-linearity
- Layers of nodes allow for increased complexity of calculations (functions of functions)
- Neural networks can theoretically approximate any function (given appropriate size)
- Creative design of network architecture admits many practical applications across domains

# The multilayer perceptron (MLP)

- A single perceptron is pretty simple
- A complex network of perceptrons can make subtle decisions




Nielsen, 2015

# Bias

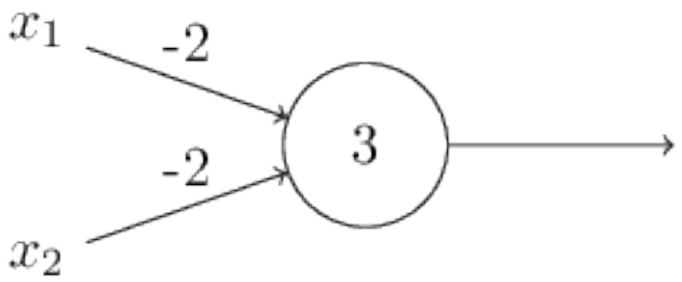
- $w \cdot x = \sum_j w_j x_j$
  - $w$ and $x$ are the weight and input vectors, respectively
- Replace the threshold with perceptron *bias*
  - Bias $b = -$threshold

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

- Bias is a measure of ease in *firing* the perceptron

# Logic circuits with perceptrons

- $w_1, w_2 = -2, b = 3$

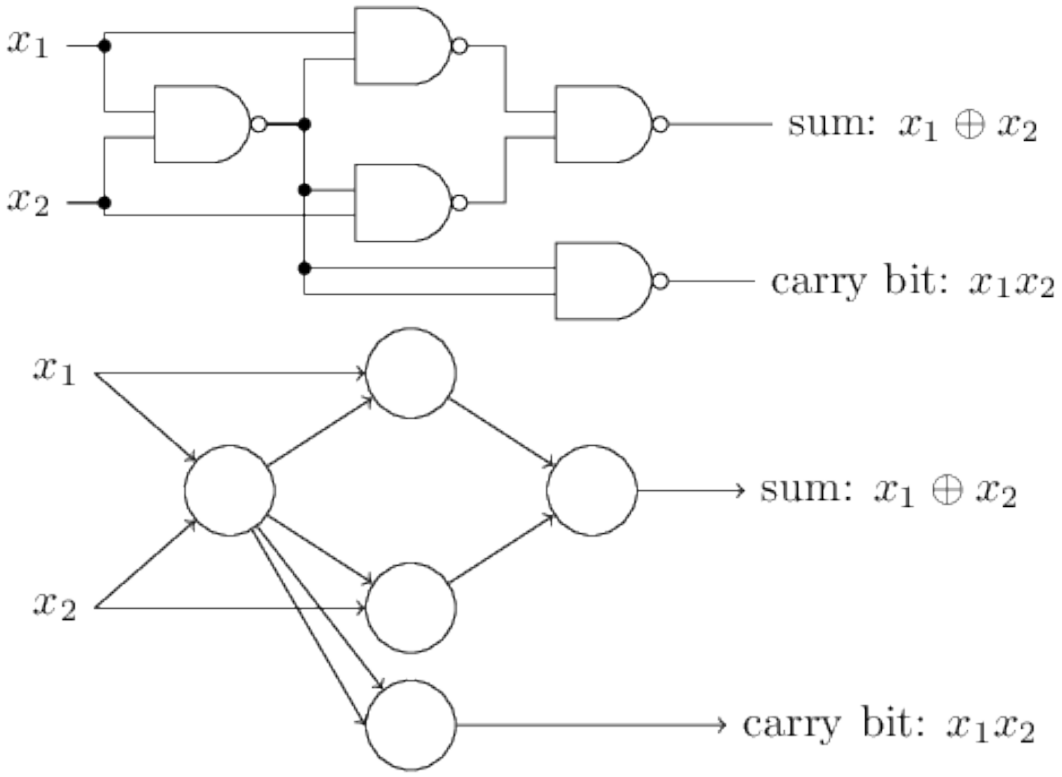



Nielsen, 2015

- What is the output of this perceptron for each possible input?
- What logic circuit is this?
- Input 00 produces 1
- Input 01 or 10 produce 1
- Input 11 produces 0
- This is a NAND gate!

# Logic circuits with perceptrons

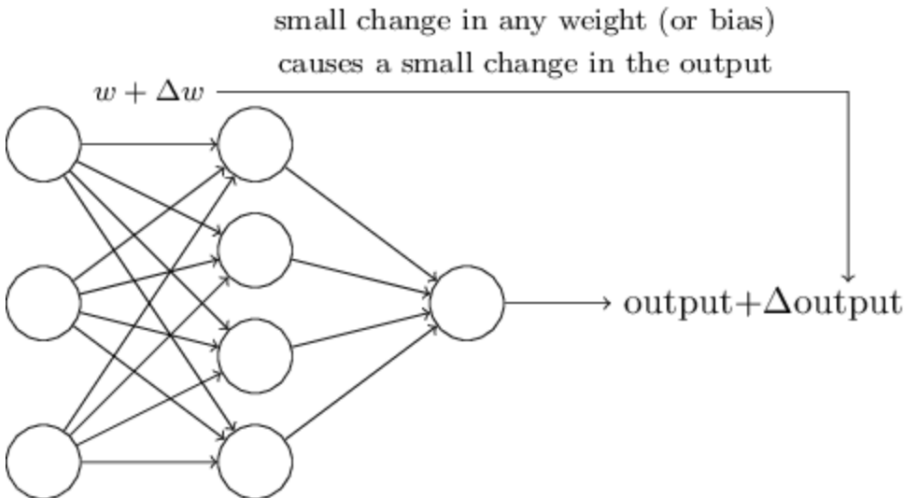
- NAND gates are universal for computation
  - Any computation can be built from NAND gates
  - Therefore, perceptrons are universal for computation
- Bitwise addition:



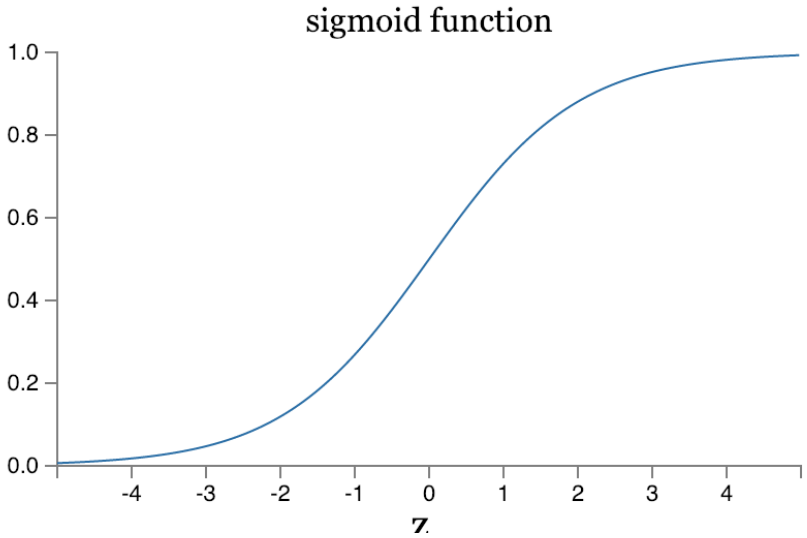

Nielsen, 2015

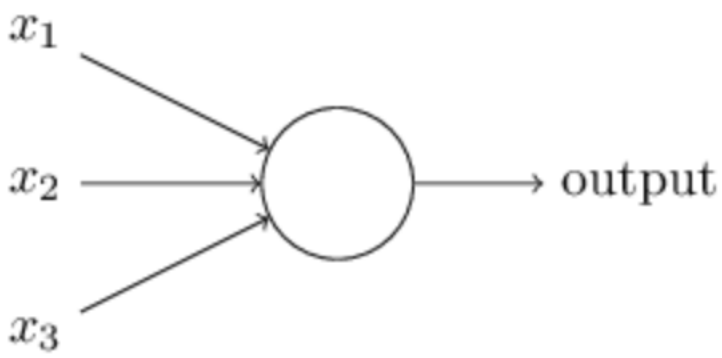
# Problem with Logical Functions

- Hard to “tune” in a traditional sense
- Small change in weight can lead to large changes in conditions (or on change at all)
  - Think the volume knob in your car behaving this way
- For this to happen we need neurons to perform a continuous *differentiable* function



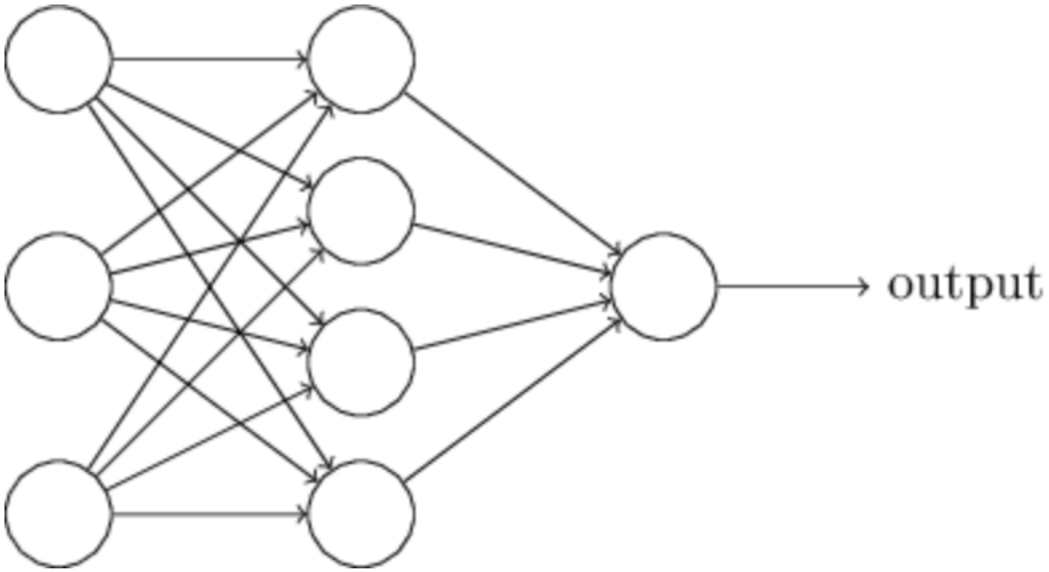
# Sigmoidal Neuron



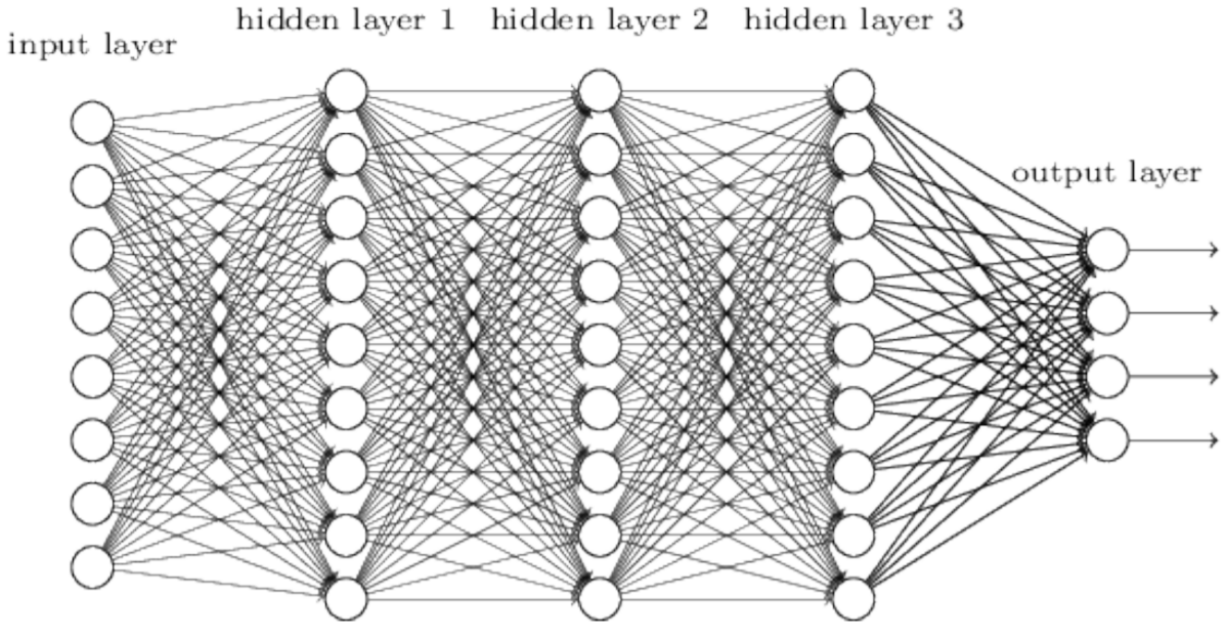

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

# Multi-layers of Neurons



# Differentiable Computing

- We can create *learning algorithms* that automatically tune the weights and biases with *Gradient Descent*
  - Tuning occurs in response to external stimuli and w/o direct intervention
  - Creates a circuit designed for the problem at hand



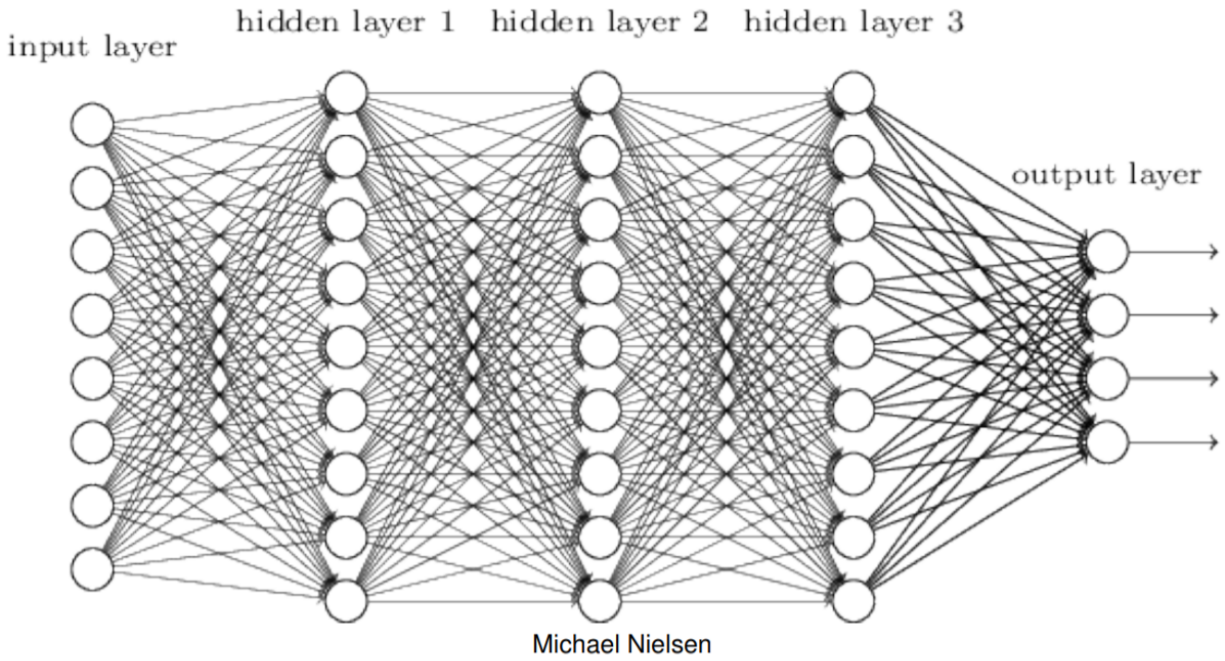

Michael Nielsen

# Design choices for an ANN

- Activation function (e.g. threshold)
- Cost functions
- Number and dimension of layers
- Connections between layers
- Regularizations
  - Layers
  - Batches
- More…

# Fully connected network

- Every feature interacts with every other feature
- Weight matrix at every level allowed to be dense



# Training Neural Networks

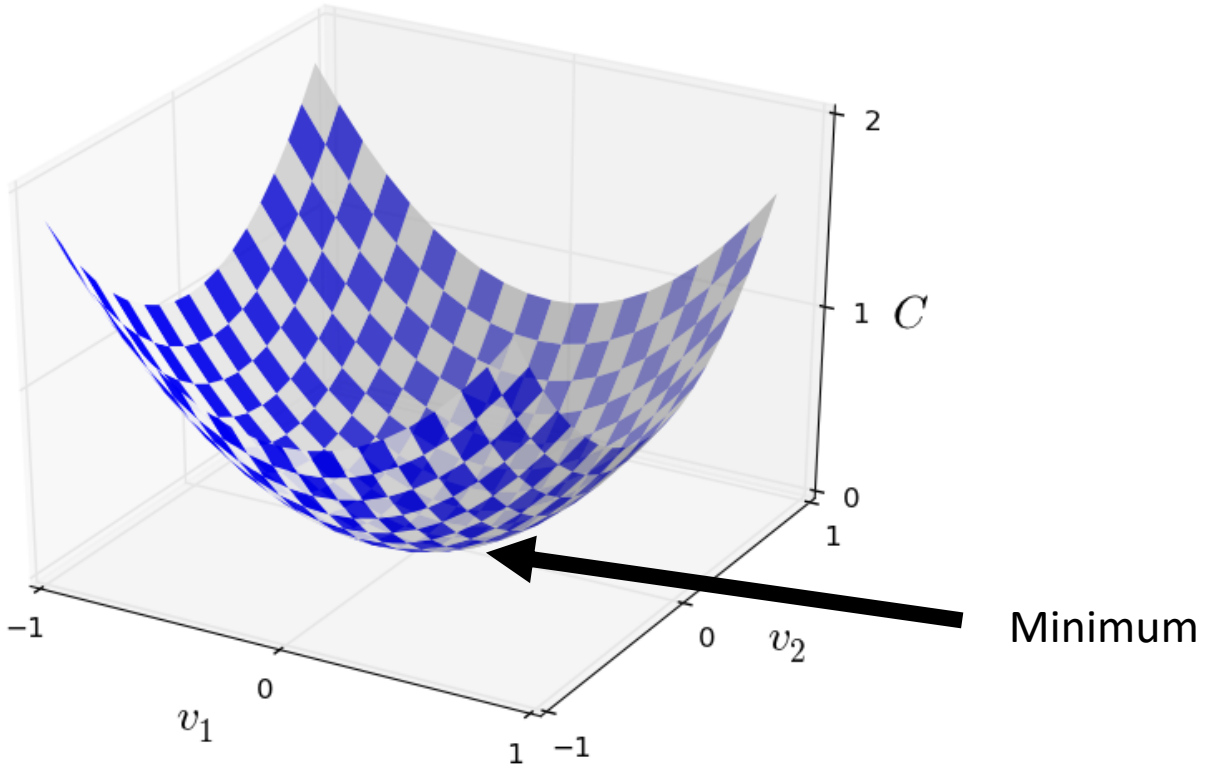
# Review: Supervised Machine Learning

- Learning where there are training data with labels
- Two major types of supervised learning tasks
  - Regression tasks
  - Classification tasks
- Classification tasks have discrete labels
  - Ex: Digit type classification
- Regression tasks have continuous labels
  - Ex: Linear regression

# Cost functions

- Labels are in $\mathcal{Y}$
  - Discrete (classification)
  - Continuous (regression)
- *Cost function* $C: \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$
- $C$ maps decisions/predictions to a cost.
  - $C(\hat{y}, y)$ is the penalty for predicting $\hat{y}$ when the true label is $y$
- Standard choice for regression is *mean squared error* (MSE)
  - $C(\hat{y}, y) = (\hat{y} - y)^2$
- Absolute cost: $C(\hat{y}, y) = |\hat{y} - y|$

# Cost/Loss Minimization




Convex cost functions can be solved by differentiation, at the point where cost is minimum the derivative wrt to parameters should be 0!

# Ex: Linear Regression

- We want to fit a linear function to dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n)\}$
- $\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + \mathrm{w}_0$
- How do we determine $\boldsymbol{w}$?
- Based on optimizing for the performance measure $P$

# Linear Regression Optimization

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} (\boldsymbol{w}^T \boldsymbol{x}_i + w_0 - y_i)^2$$

$$= \arg\min_{\boldsymbol{w}} L(\boldsymbol{w}; \mathcal{D})$$

- Set $\frac{\partial L(\boldsymbol{w};\mathcal{D})}{\partial w_i} = 0$ for each $i$
- Results in $d + 1$ equations and $d + 1$ unknowns

# Mean squared error cost

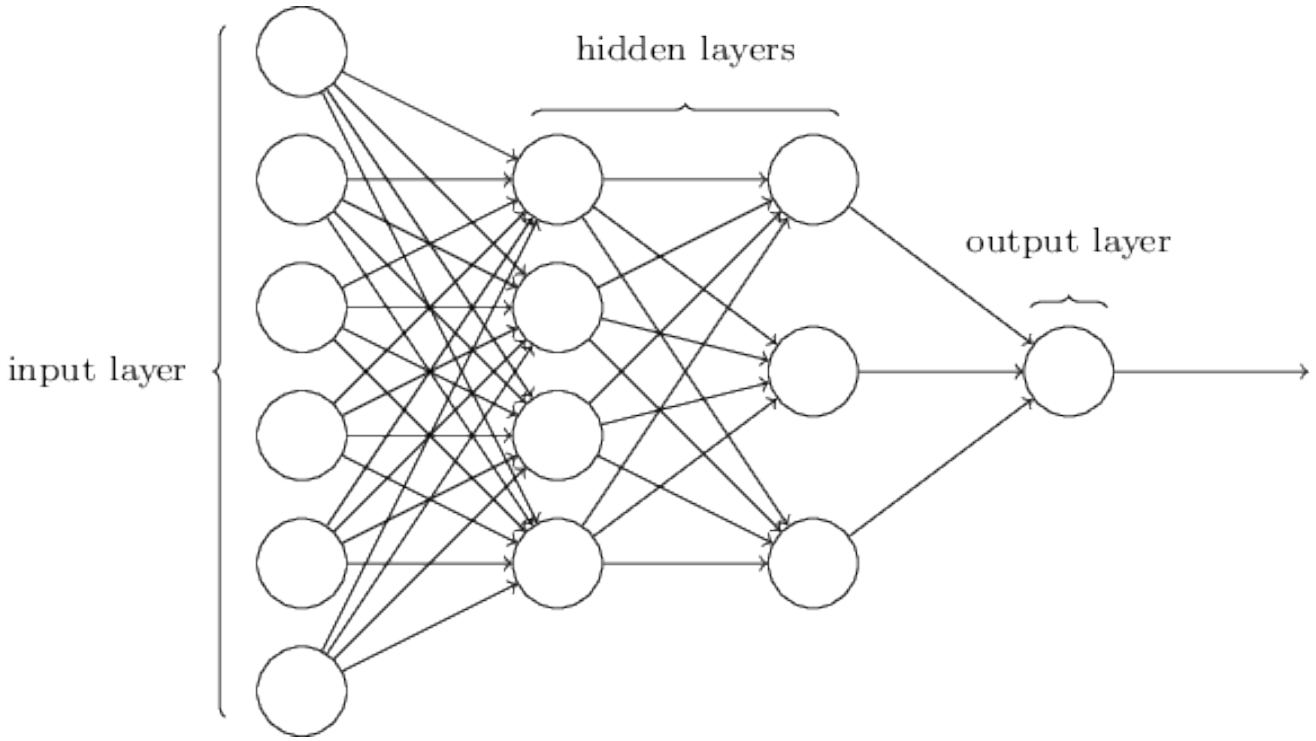
$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} (\boldsymbol{w}^T \boldsymbol{x}_i + w_0 - y_i)^2$$

Rewrite:

$$\begin{aligned}(X\mathbf{w} - \mathbf{y})^T (X\mathbf{w} - \mathbf{y}) &= (\mathbf{w}^T X^T - \mathbf{y}^T)(X\mathbf{w} - \mathbf{y}) \\ &= \mathbf{w}^T X^T X\mathbf{w} - \mathbf{w}^T X^T \mathbf{y} - \mathbf{y}^T X\mathbf{w} + \mathbf{y}^T \mathbf{y} \\ &= \mathbf{w}^T X^T X\mathbf{w} - 2\mathbf{w}^T X^T \mathbf{y} + \mathbf{y}^T \mathbf{y}.\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial w} \mathbf{w}^T X^T X\mathbf{w} - 2\mathbf{w}^T X^T \mathbf{y} + \mathbf{y}^T \mathbf{y} &= 0 \\ 2X^T X\mathbf{w} - 2X^T \mathbf{y} &= 0 \\ X^T X\mathbf{w} &= X^T \mathbf{y} \\ \mathbf{w} &= (X^T X)^{-1} X^T \mathbf{y}\end{aligned}$$

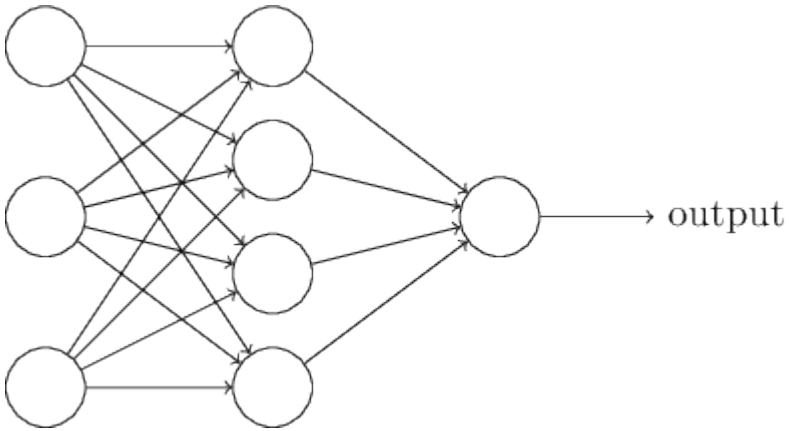
# Classification/Regression Network




- Sometimes called multi-layer perceptron (although sigmoid neurons are used)
- Output from one layer is used as input for the next (feedforward network)

# Classification with sigmoid neurons

- Sigmoid neuron output = any real number between 0 and 1
- How do we do classification?
- Threshold the final output
  - E.g., a value above 0.5 indicates a “9” while a value below 0.5 does not



# Weights, bias, and output

- Change in weight $\Delta w_j$, change in bias $\Delta b$

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

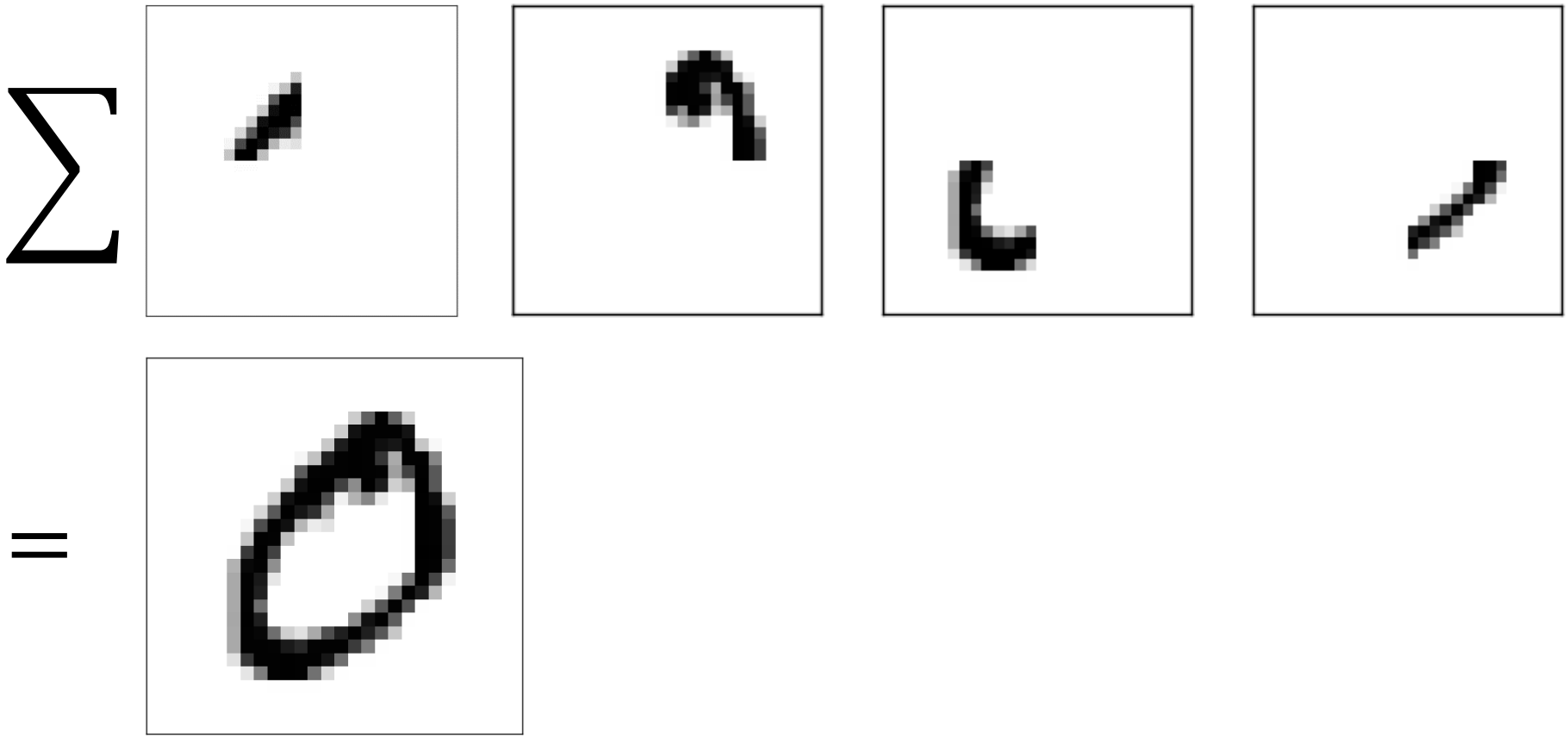
- $\Delta$output is a *linear function* of the changes $\Delta w_j$ and $\Delta b$ in the weights and bias
  - Linearity makes it easier to choose small changes in weights and biases to achieve the desired small change in output
- Thus sigmoid neurons have similar behavior as perceptrons but are easier to use

# Handwritten digit recognition

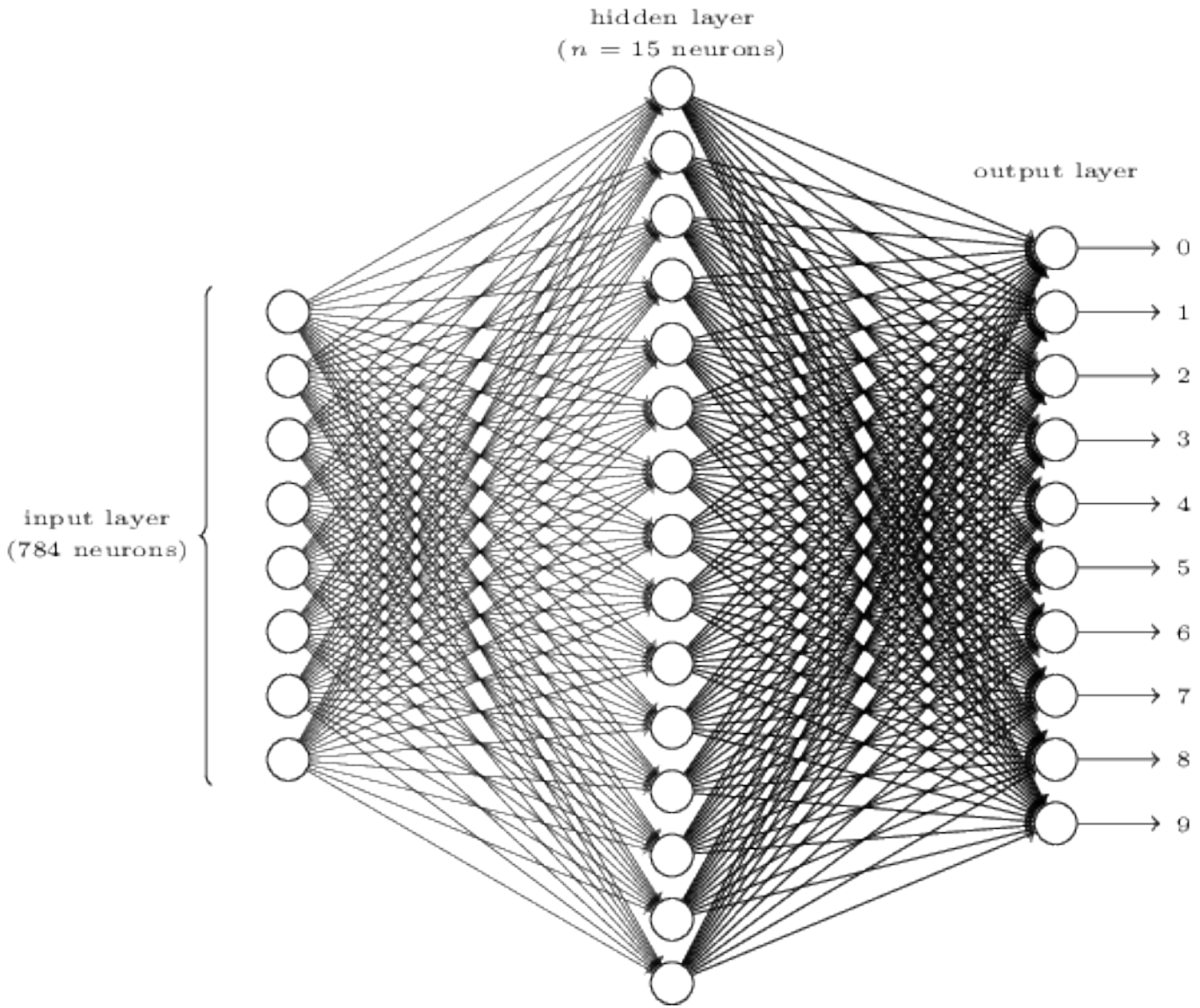
# How would you write an algorithm for this?

# Number of output neurons

- Possible heuristic:



# A simple network



# Forward Propagation

- Store weights and biases as matrices
- Suppose we are considering the weights from the second (hidden) layer to the third (output) layer
  - $w$ is the weight matrix with $w_{jk}$ the weight for the connection between the $k$th neuron in the second layer and the $j$th neuron in the third layer
  - $b$ is the vector of biases in the third layer
  - $a$ is the vector of activations (output) of the 2nd layer
  - $a'$ the vector of activations (output) of the third layer

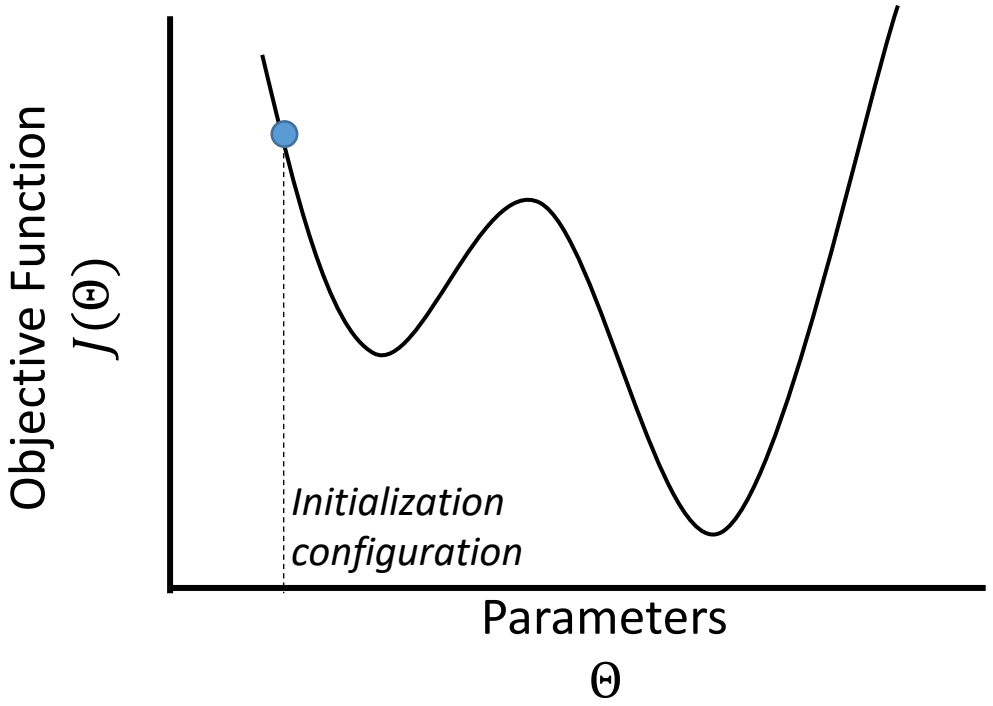
$$\boldsymbol{a' = \sigma(wa + b)}$$

# How training works

1. In each ***epoch***, randomly shuffle the training data
2. Partition the shuffled training data into ***mini-batches***
3. For each mini-batch, apply a single step of **gradient descent**
   - **Gradients** are calculated via ***backpropagation*** (the next topic)
4. Train for multiple epochs

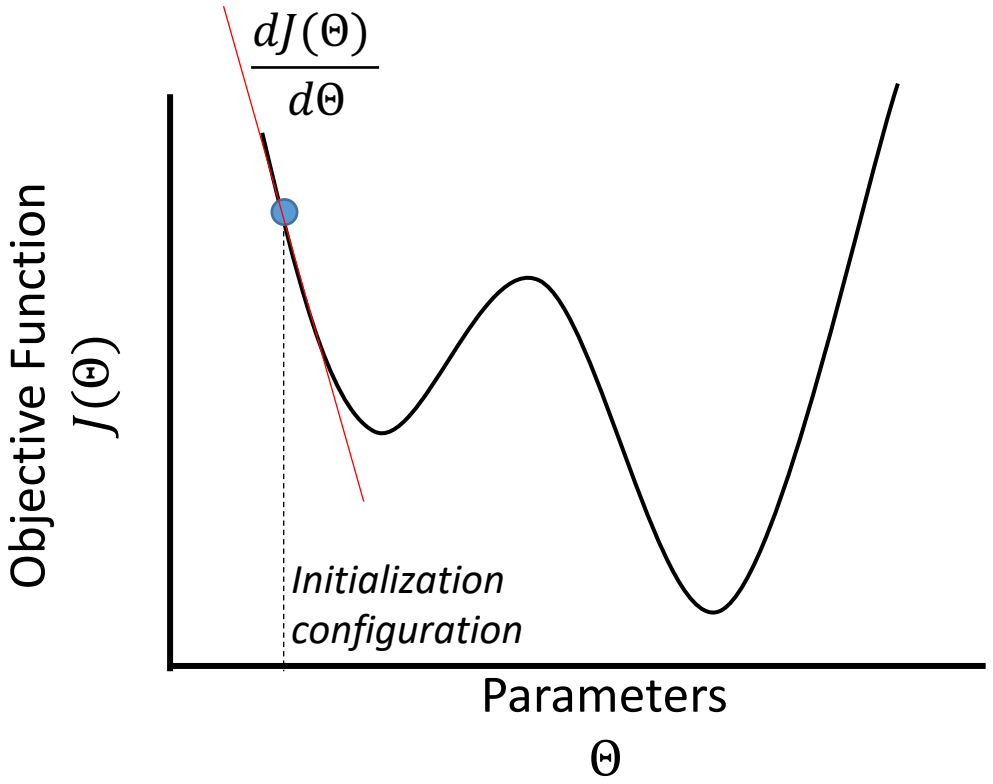
# Gradient Descent

- Gradient descent refers to taking a step in the direction of the *gradient (partial derivative)*
- *Gradient is the direction of steepest change*
- Gradient wrt weight or bias with respect to the cost function



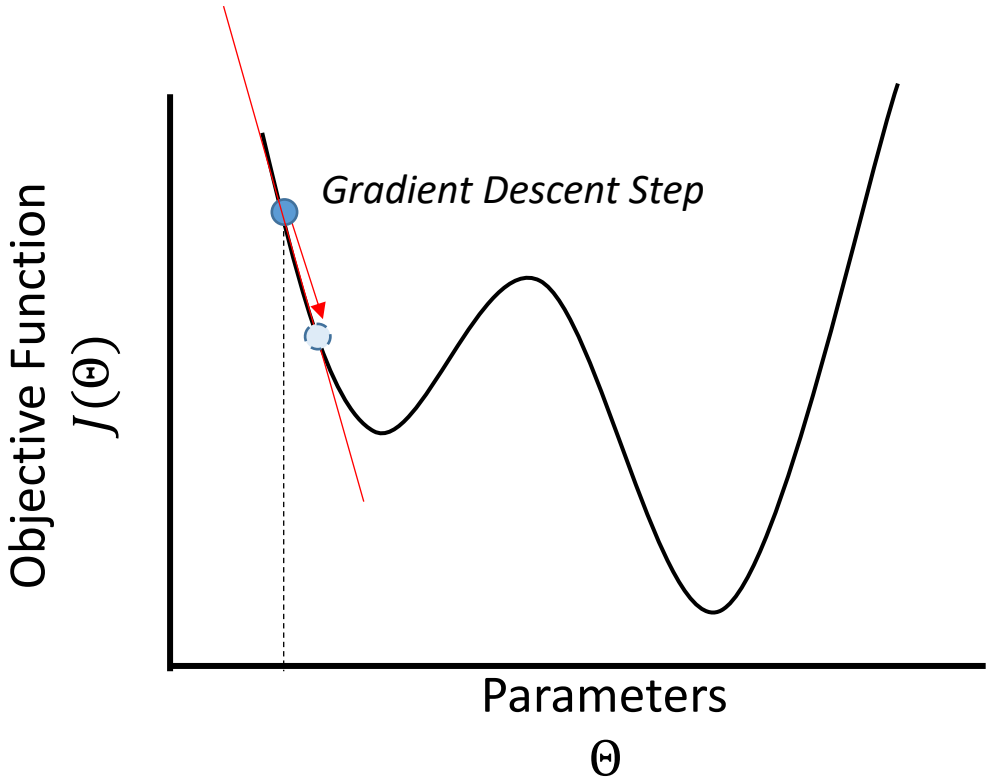
# Gradient Descent

- Gradient descent refers to taking a step in the direction of the *gradient (partial derivative)*
- *Gradient is the direction of steepest change*
- Gradient wrt weight or bias with respect to the cost function



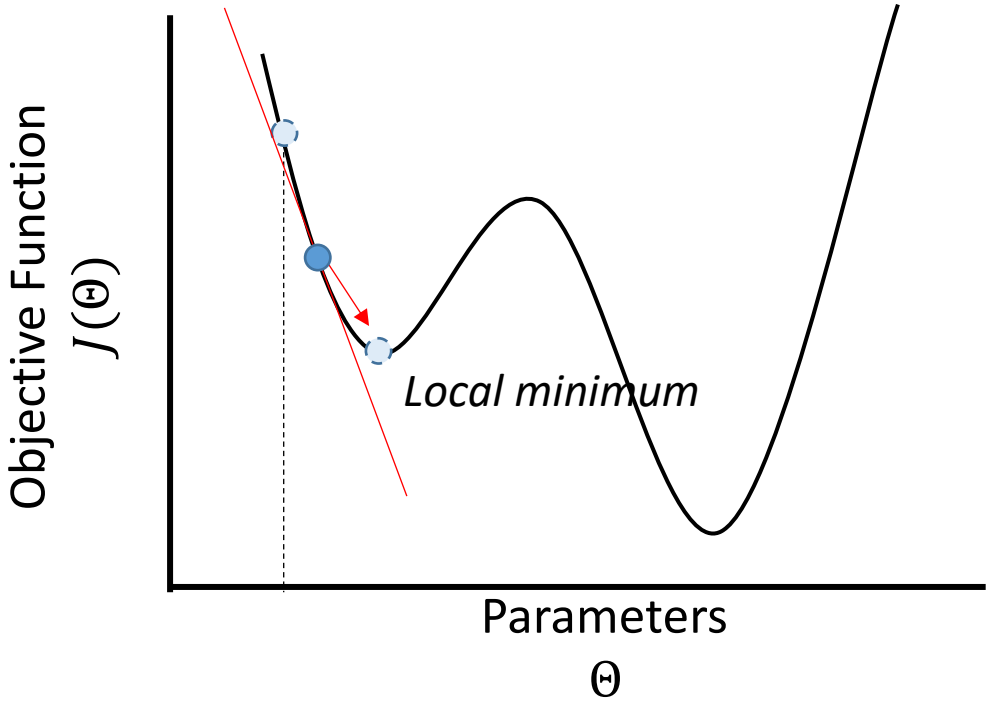
# Gradient Descent

- Gradient descent refers to taking a step in the direction of the *gradient (partial derivative)*
- *Gradient is the direction of steepest change*
- Gradient wrt weight or bias with respect to the cost function



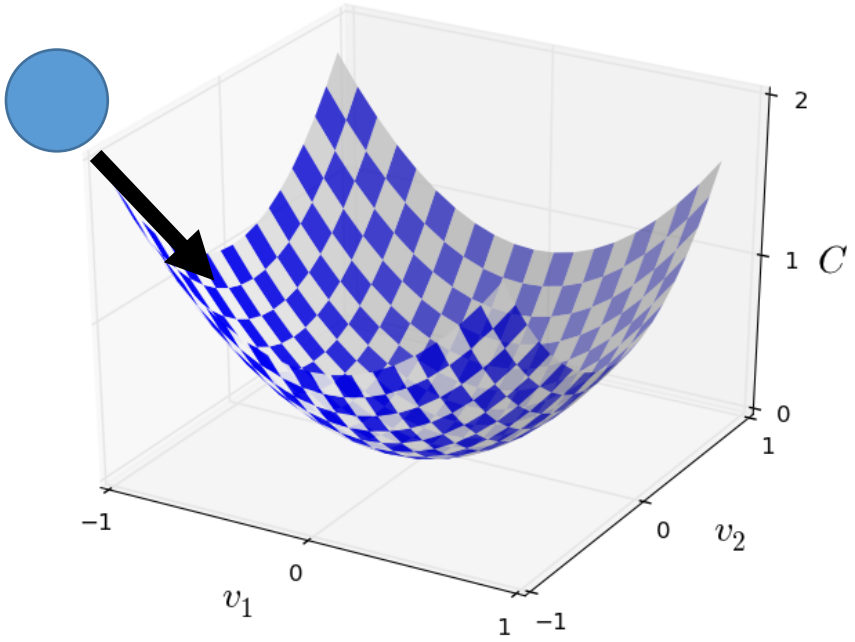
# Gradient Descent

- Gradient descent refers to taking a step in the direction of the *gradient (partial derivative)*
- *Gradient is the direction of steepest change*
- Gradient wrt weight or bias with respect to the cost function



# Gradient descent: Rolling ball analogy

- Choose a random start point
- Ball rolls to the bottom of the valley
- Can simulate this by computing 1st and 2nd derivatives of $C$



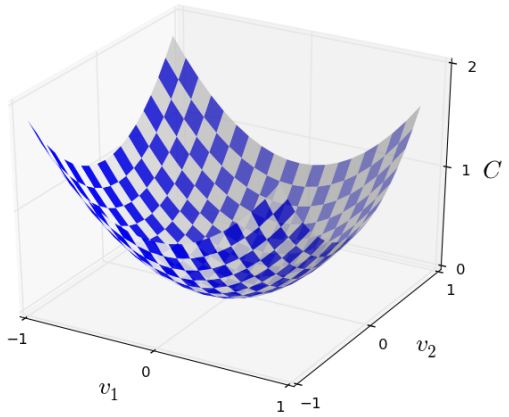
# The gradient

- Choose a random starting point
- Move small amounts $\Delta v_1$ in the $v_1$ direction and $\Delta v_2$ in the $v_2$ direction

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

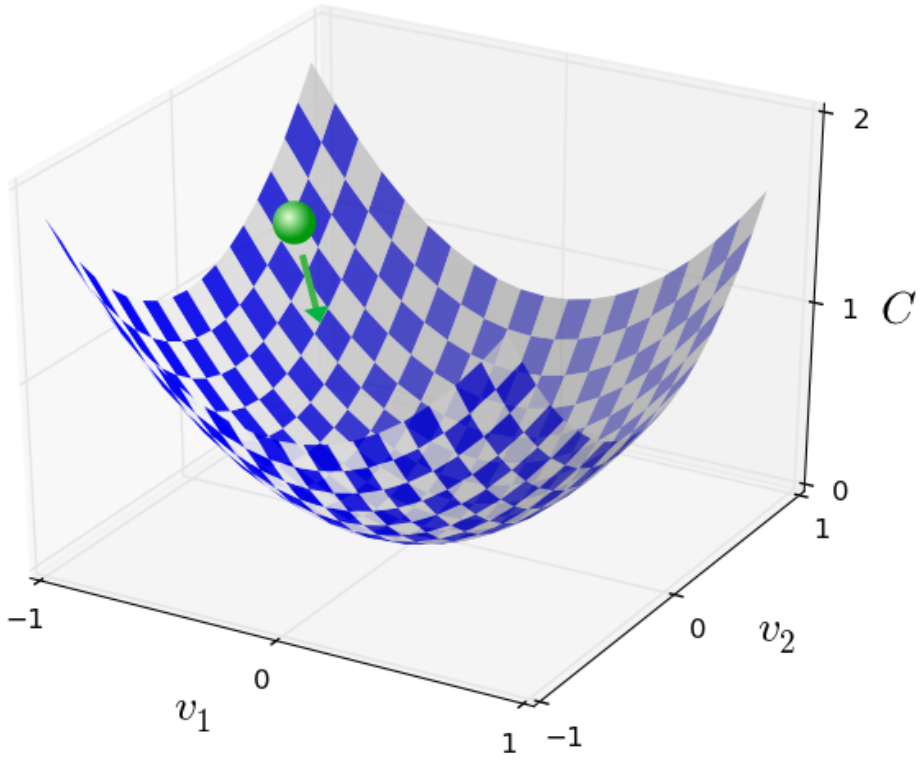
- The ***gradient*** of $C$ is $\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- $\Delta C \approx \nabla C \cdot \Delta v$
- Choose where $\eta$ is learning rate

  $\Delta v = -\eta \nabla C$



# Summary of gradient descent

- Compute the gradient $\nabla C$
- Move in the *opposite* direction
  - It's like falling down the slope of the valley



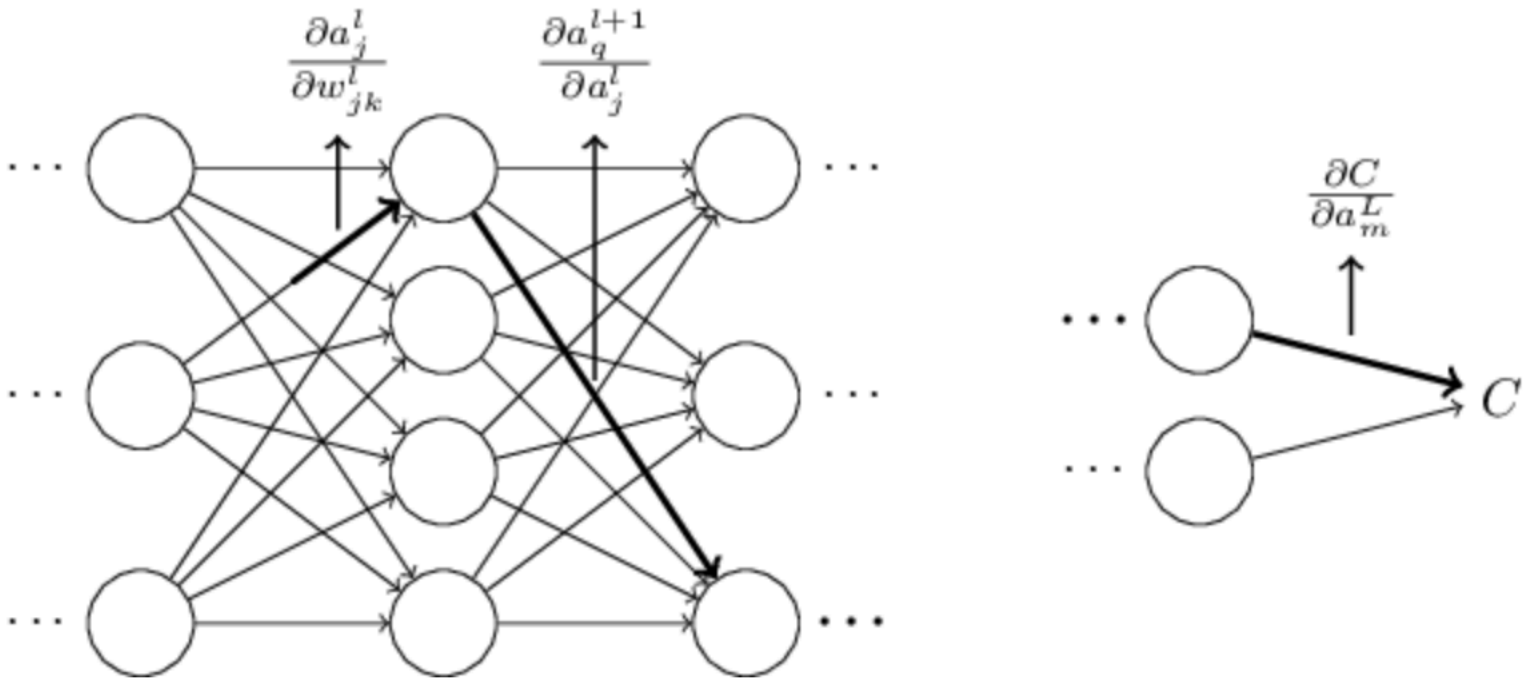
# Gradient of a sigmoidal neuron

- $f(z) = \frac{1}{1+e^{-z}}$
- Use chain rule with g(z) = $e^{-z}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{\delta(f(g(z)))}{\delta(g(z))} \frac{\delta(g(z))}{\delta(z)}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{e^{-z}}{(1+e^{-z})^2}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{1+e^{-z}-1}{(1+e^{-z})^2}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{(1+e^{-z})}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2}$

- Nice form of answer is $\frac{\delta(f(g(z)))}{\delta(z)}$=f(z) (1-f(z))
- How do you get this in terms of weights and biases?

# Backpropagation

- Propagates gradient backward through the circuit, by using the chain rule of differentiation
- f(g(x))=f′(g)**g′(x)** "opens out g"



# First attempt

- 30 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\boldsymbol{\eta = 3.0}$

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

- Best accuracy at epoch 28: 95.42%

# Second attempt

- 100 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\eta = 3.0$
- Improves accuracy to 96.59%
  - Although depends on initialization: some runs give worse results

# Third attempt

- 100 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\boldsymbol{\eta = 0.001}$

```
Epoch 0: 1139 / 10000
Epoch 1: 1136 / 10000
Epoch 2: 1135 / 10000
...
Epoch 27: 2101 / 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

# Debugging a neural network

- What do we do if the output is essentially noise?
- Suppose we ran the following as our first attempt:
- 30 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\eta = 100.0$

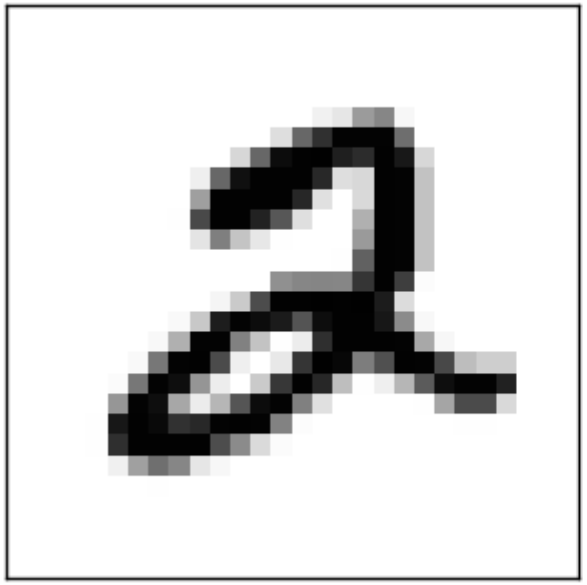
```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

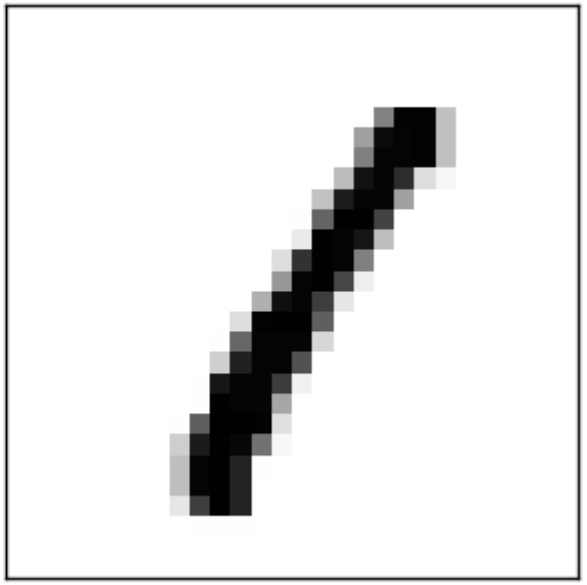
# Debugging a neural network

- What can we do?
  - Should we change the learning rate?
  - Should we initialize differently?
  - Do we need more training data?
  - Should we change the architecture?
  - Should we run for more epochs?
  - Are the features relevant for the problem (i.e. is the Bayes error rate reasonable)?
- Debugging is an art
  - We'll develop good heuristics for choosing good architectures and hyper parameters

# How well does our network do?

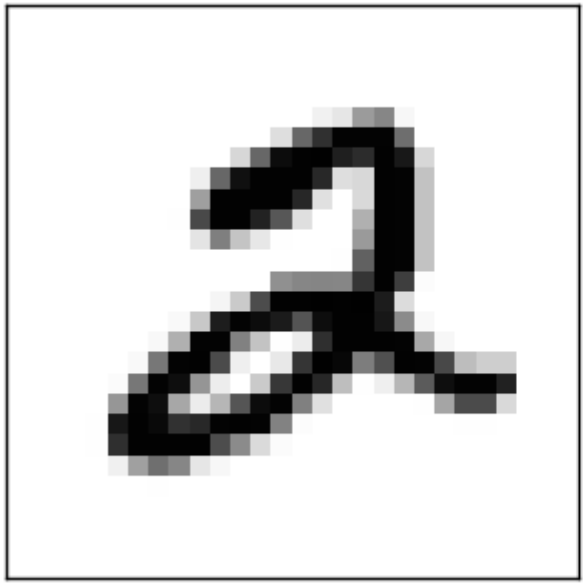
- Need to compare to some baselines
- Random guessing: 10% accuracy
  - Our network does much better
- Simple idea: How dark is the image?
  - E.g. a 2 will typically be darker than a 1

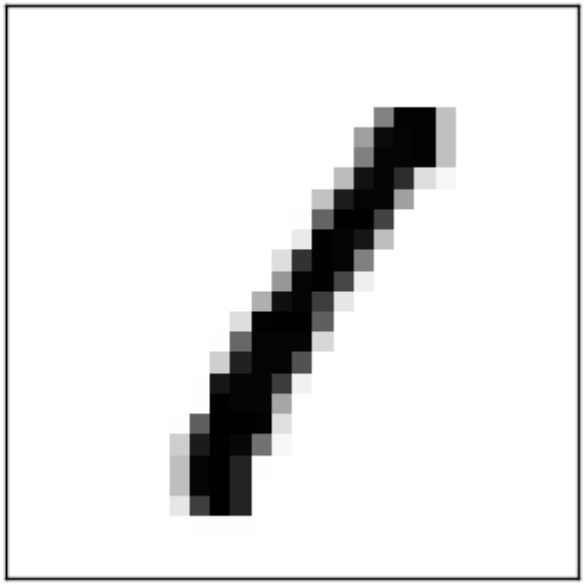




# Average darkness

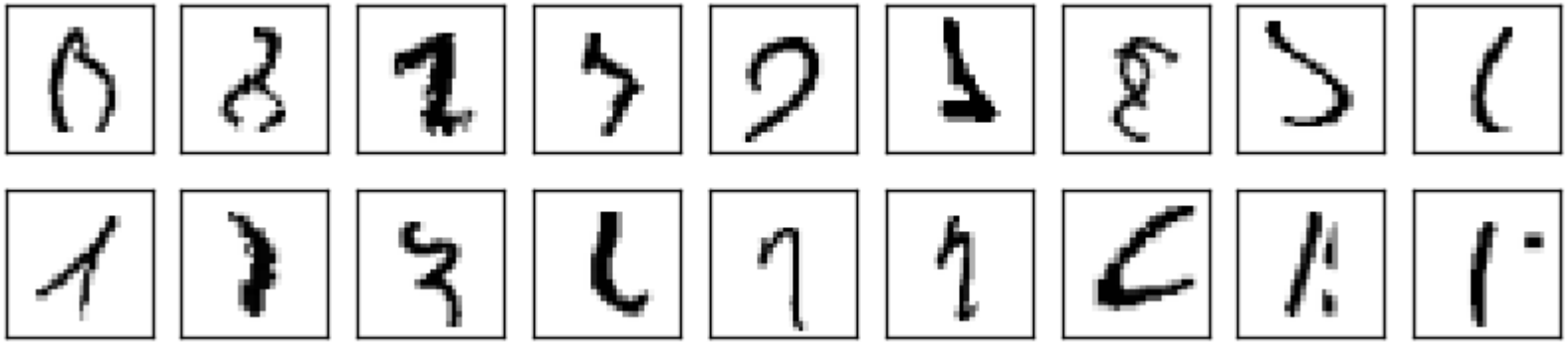
- Compute average darkness for each digit from the training data
- Classify an image based on the digit with the closest average darkness
- This gives 22.25% accuracy
  - Better than random guessing





# Support Vector Machine (SVM)

- Using default settings in SVM scikit-learn (a Python library) gives 94.35% accuracy
  - It's possible to optimize tuning parameters to achieve 98.5% accuracy
- Can neural networks do better?
- Yes!
  - Record set in 2013 was 99.79% accuracy (only 21 wrong in the test data)
  - Can you do better?



# Regularization

# Regularization in Linear Regression

- Ridge regression: penalize with L2 norm

$$\boldsymbol{w}^* = \arg\min \sum_i C(f(\boldsymbol{x}_i; \boldsymbol{w}), y_i) + \lambda \sum_{j=1}^{m} w_j^2$$

  - Closed form solution exists $\boldsymbol{w}^* = (\lambda I + X^T X)^{-1} X^T \boldsymbol{y}$
- LASSO regression: penalize with L1 norm

$$\boldsymbol{w}^* = \arg\min \sum_i C(f(\boldsymbol{x}_i; \boldsymbol{w}), y_i) + \lambda \sum_{j=1}^{m} |w_j|$$

  - No closed form solution but still convex (optimal solution can be found)

# Regularization

- In neural networks regularizations express a preference for simpler solutions
- Yields an inductive bias in the solution space
- Sometimes can get solution out of local minima
- How do we define “simple” ?
- Regularizations are not the same as in low-parameters where capacity is restricted

# Weight decay/L2 regularization

- Can write regularized cost function as

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

  - $C_0$ is the unregularized cost
- First term is the cross-entropy cost function
- Second term is the regularization term
  - Scaled by factor $\frac{\lambda}{2n}$, $\lambda$ the ***regularization parameter***

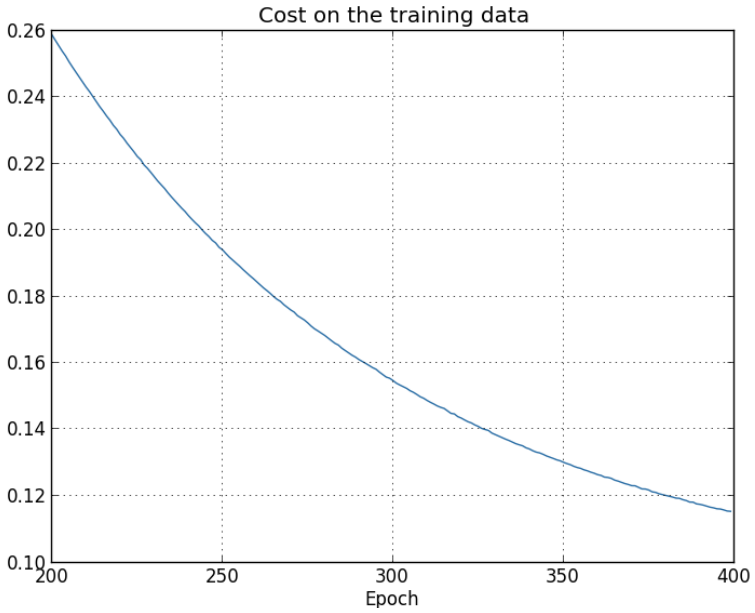
# Weight decay/L2 regularization

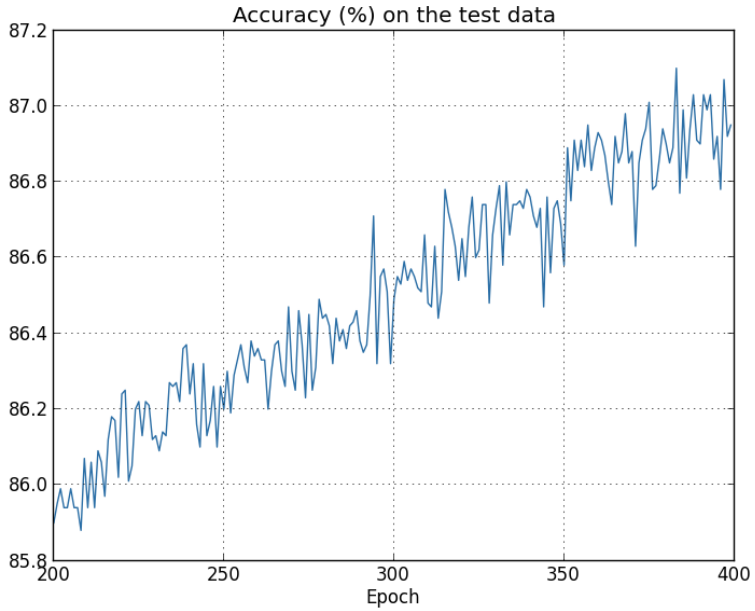
$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

- Regularization forces the weights to be small
  - Large weights allowed only if they considerably improve $C_0$
- I.e., regularization is a compromise between finding small weights and minimizing the cost function $C_0$
  - $\lambda$ controls this compromise
  - Small $\lambda \Rightarrow$ we prefer to minimize $C_0$
  - Large $\lambda \Rightarrow$ we prefer small weights
- How do we apply regularization in gradient descent?

# L2 regularization applied to MNIST

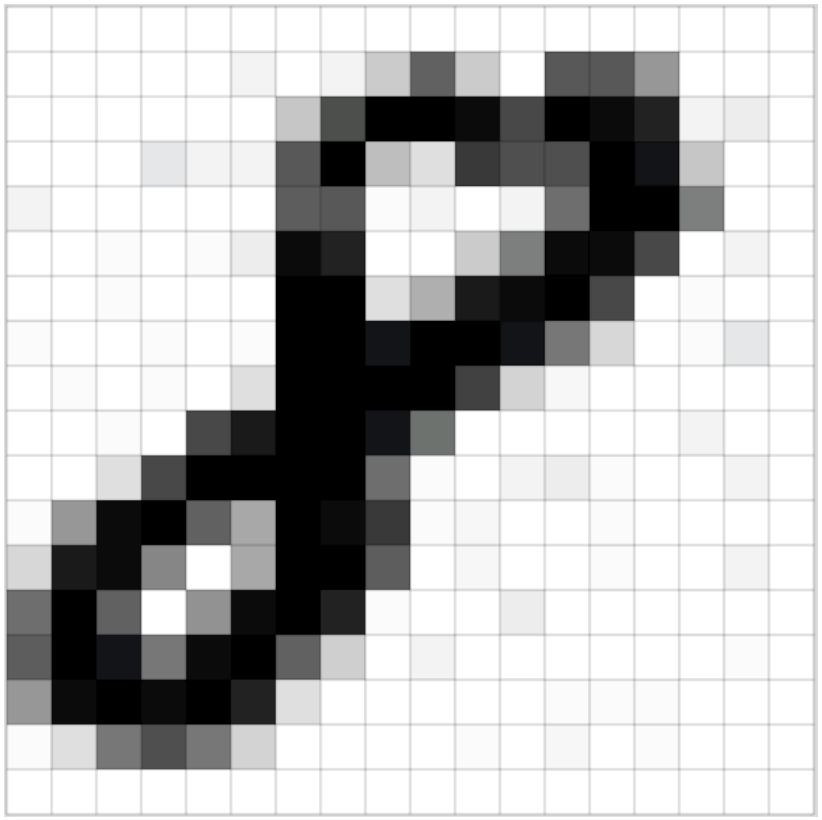
- 30 hidden neurons
- Train with first 1,000 training images
  - Cross-entropy cost
  - Learning rate $\eta = 0.5$
  - Mini-batch size $10$
  - $\lambda = 0.1$
- Training cost decreases with each epoch
- Test accuracy continues to increase
  - More epochs would likely improve results further
  - Regularization improves generalization in this case



# Convolutions

# Images are a series of Pixel Values



Grayscale images:
0=Black
255 = White

Spatial locality structure

# Local receptive fields

Make connections in small, localized regions of the input image

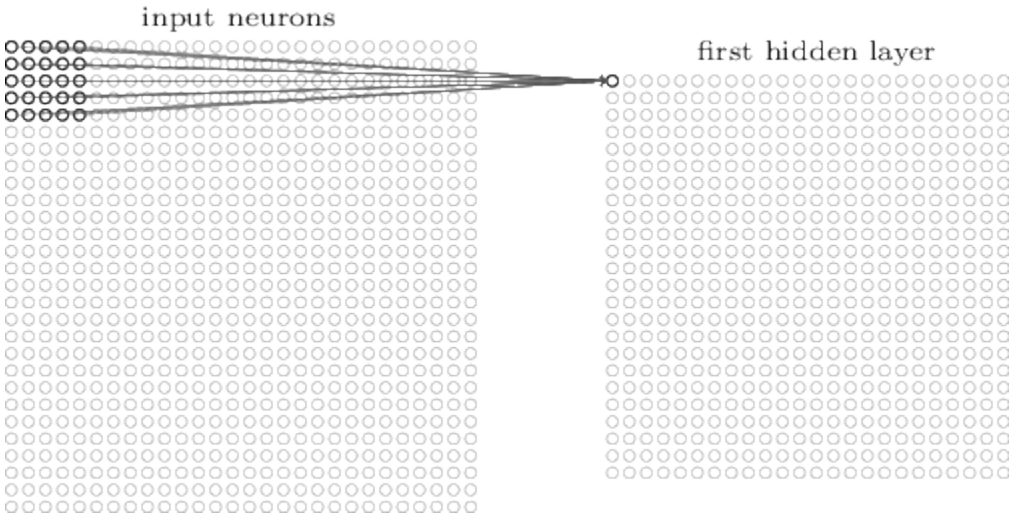



Image taken from Michael Neilsen's book "Neural Networks and Deep Learning"

# Local receptive fields

Slide the local receptive field over by one (or more) pixel and repeat

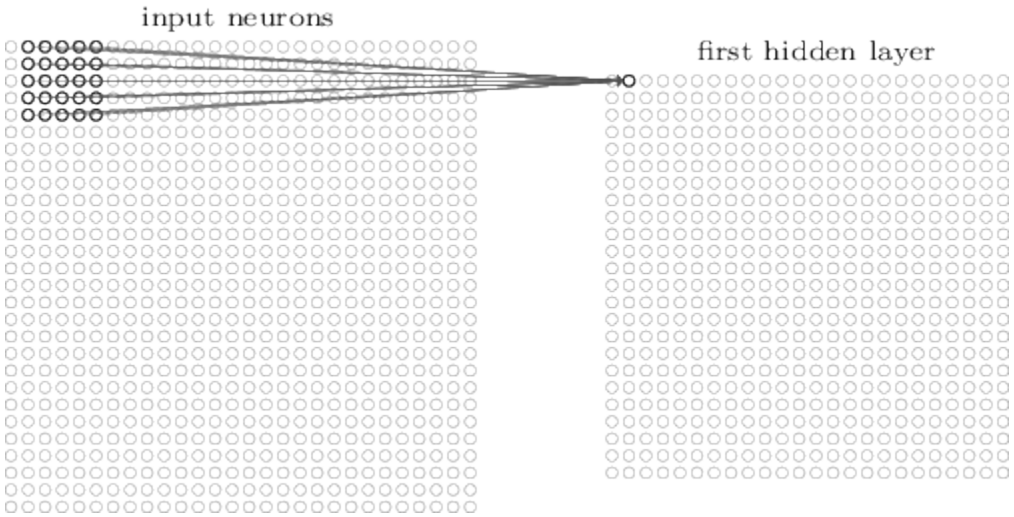
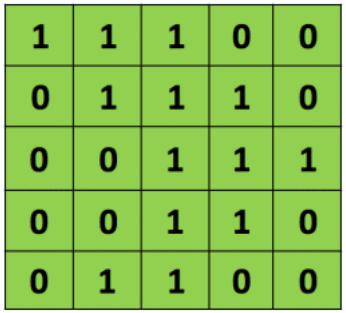
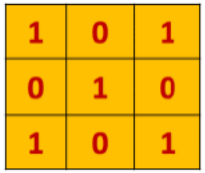



Image taken from Michael Neilsen's book "Neural Networks and Deep Learning"

# The convolution operation

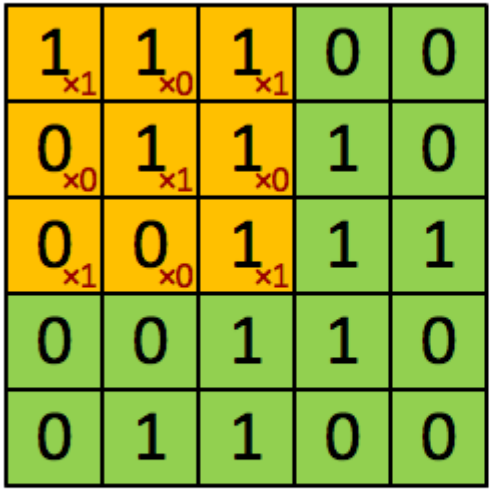



Image

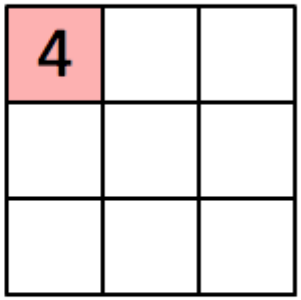



Filter/
Feature detector

1. Pointwise multiply
2. Add results
3. Translate filter

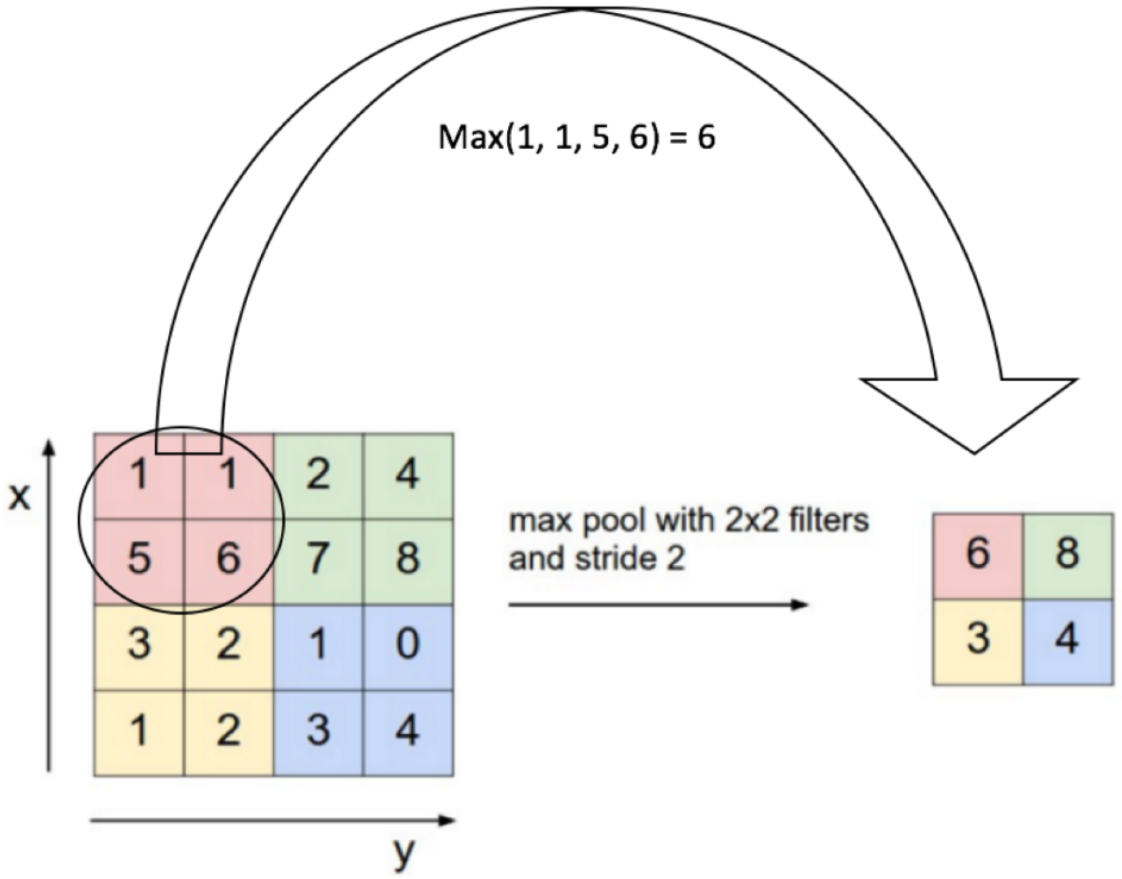



Image




Convolved
Feature

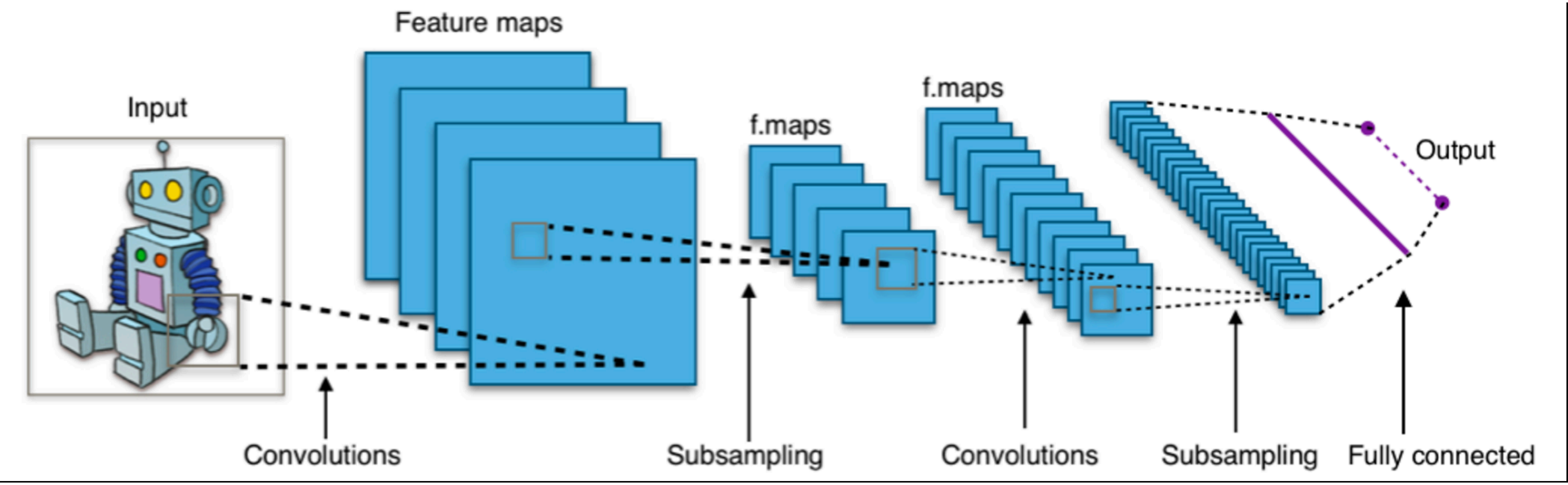
# Pooling




Downsampling the results by pooling values by averaging or max

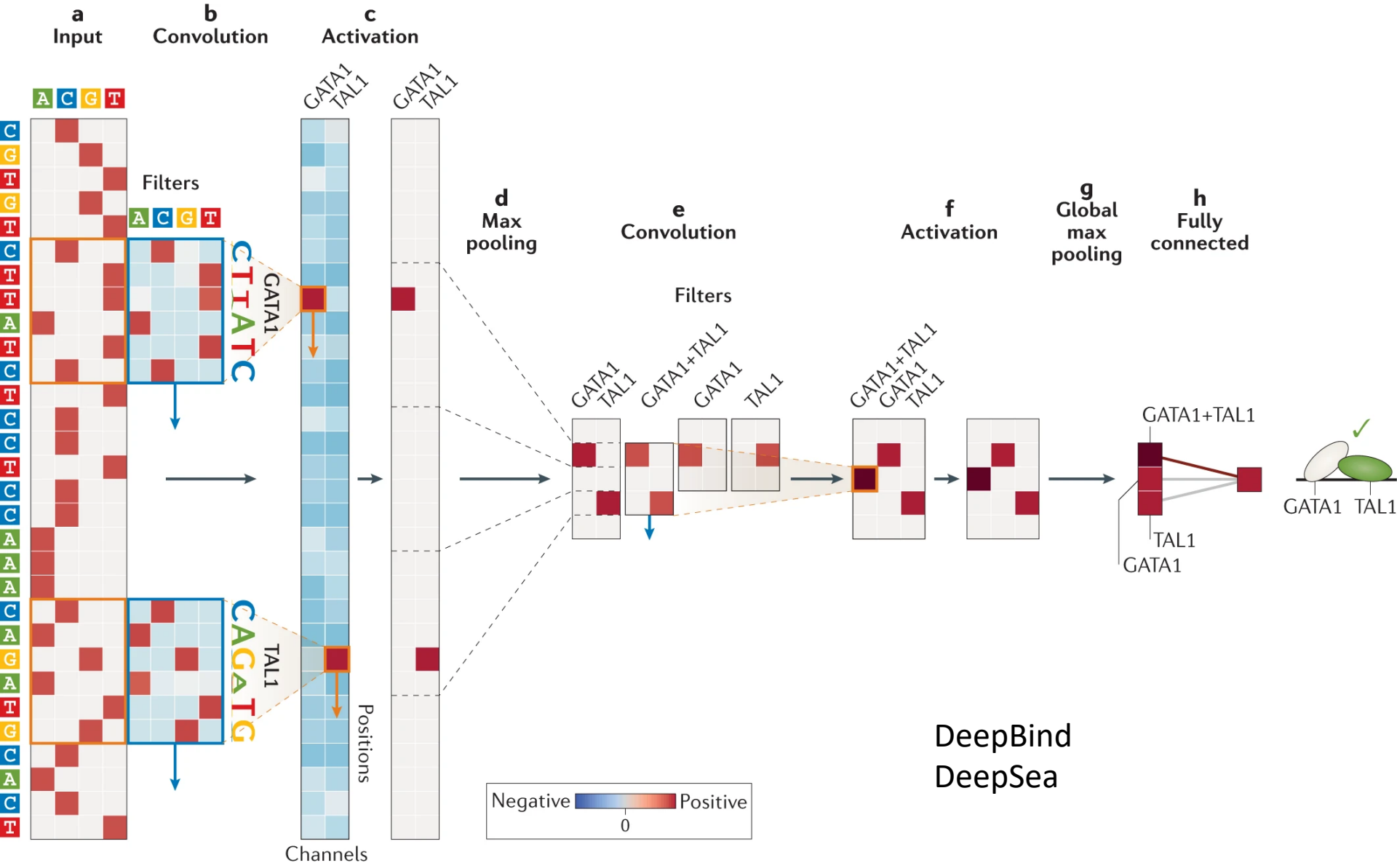
Max pool is used more commonly

# CNN Architecture




Each convolutional unit has the same weight.

# Binding Motif Prediction




DeepBind
DeepSea

# Autoencoders

# Unsupervised learning

- Dataset contains only features and no labels
- Goal is to find hidden patterns in the unlabeled data
- Examples
  - Density estimation
  - Data generation
  - Clustering
  - Data denoising
  - Representation learning
- *Semi-supervised learning* attempts to combine information from both labeled and unlabeled data to deduce information

# Representation learning

- Find the “best” representation of the data
  - I.e., find a representation of the data that preserves as much information as possible while obeying some penalty or constraint that simplifies the representation
- What information do we want to preserve?
- How do we define simple?
  - Low-dimensional
  - Sparse
  - Independent components
- Above criteria aren’t necessarily mutually exclusive
  - E.g., low-dimensional representations often have independent or weakly dependent components

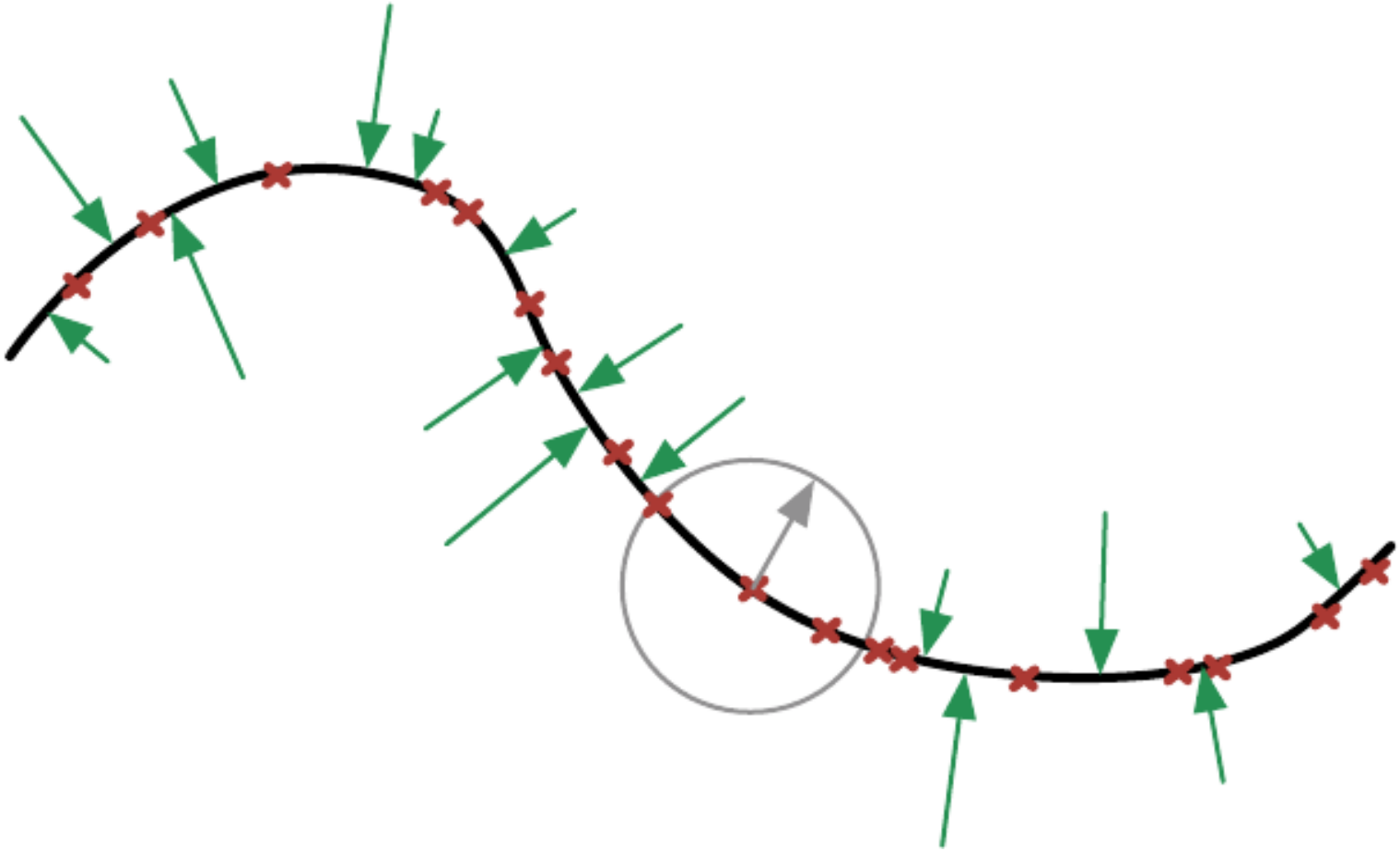
# Representation learning

Why is a simple representation useful?

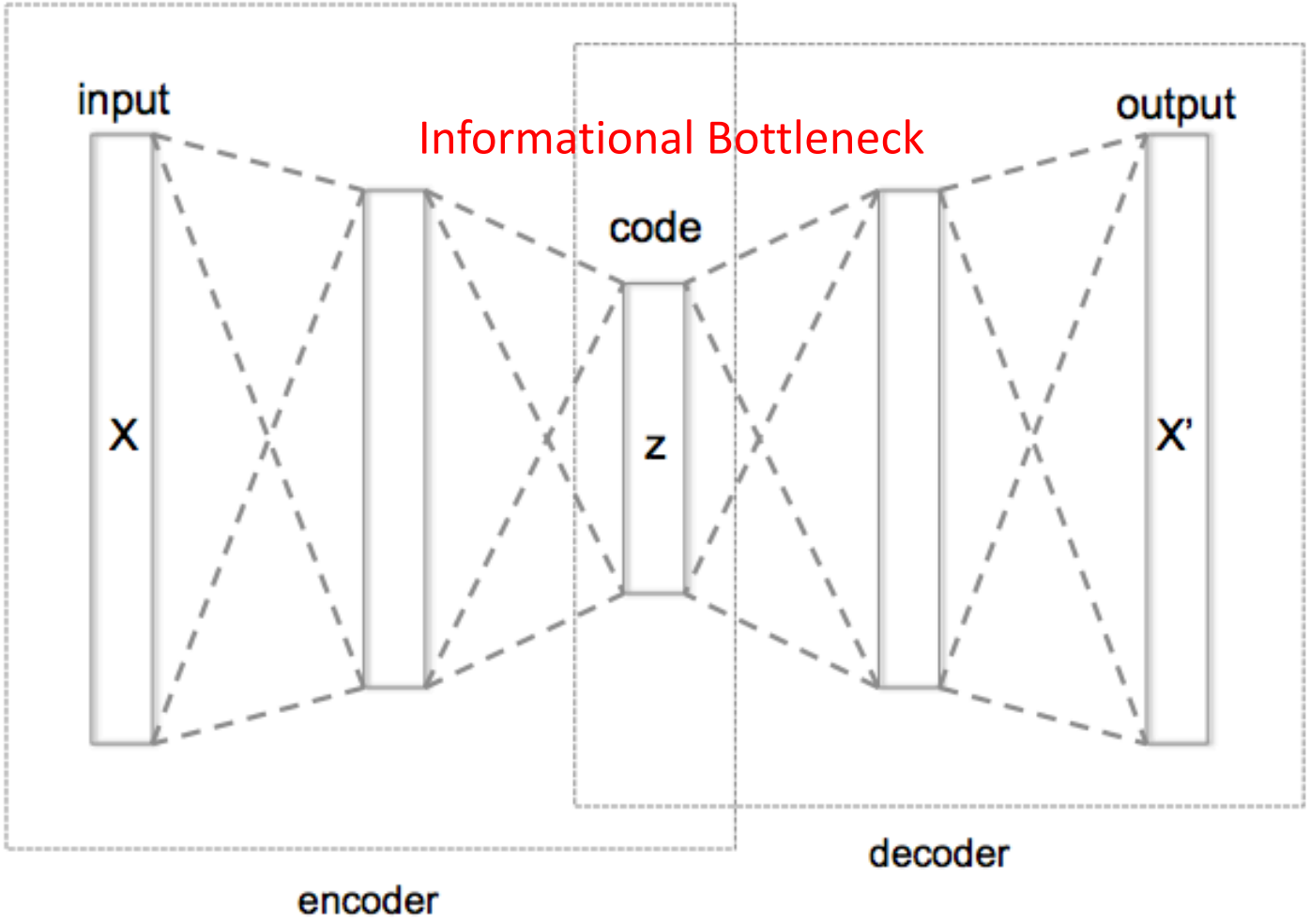
- Interpretability
  - E.g. visualization
- Computational cost
  - Compression
- Performance
  - Preprocessing

# Manifold assumption

- Data may be modeled as lying on a low-dimensional manifold



# Autoencoder




[Hinton, Salakhutdinov, Science 2006]

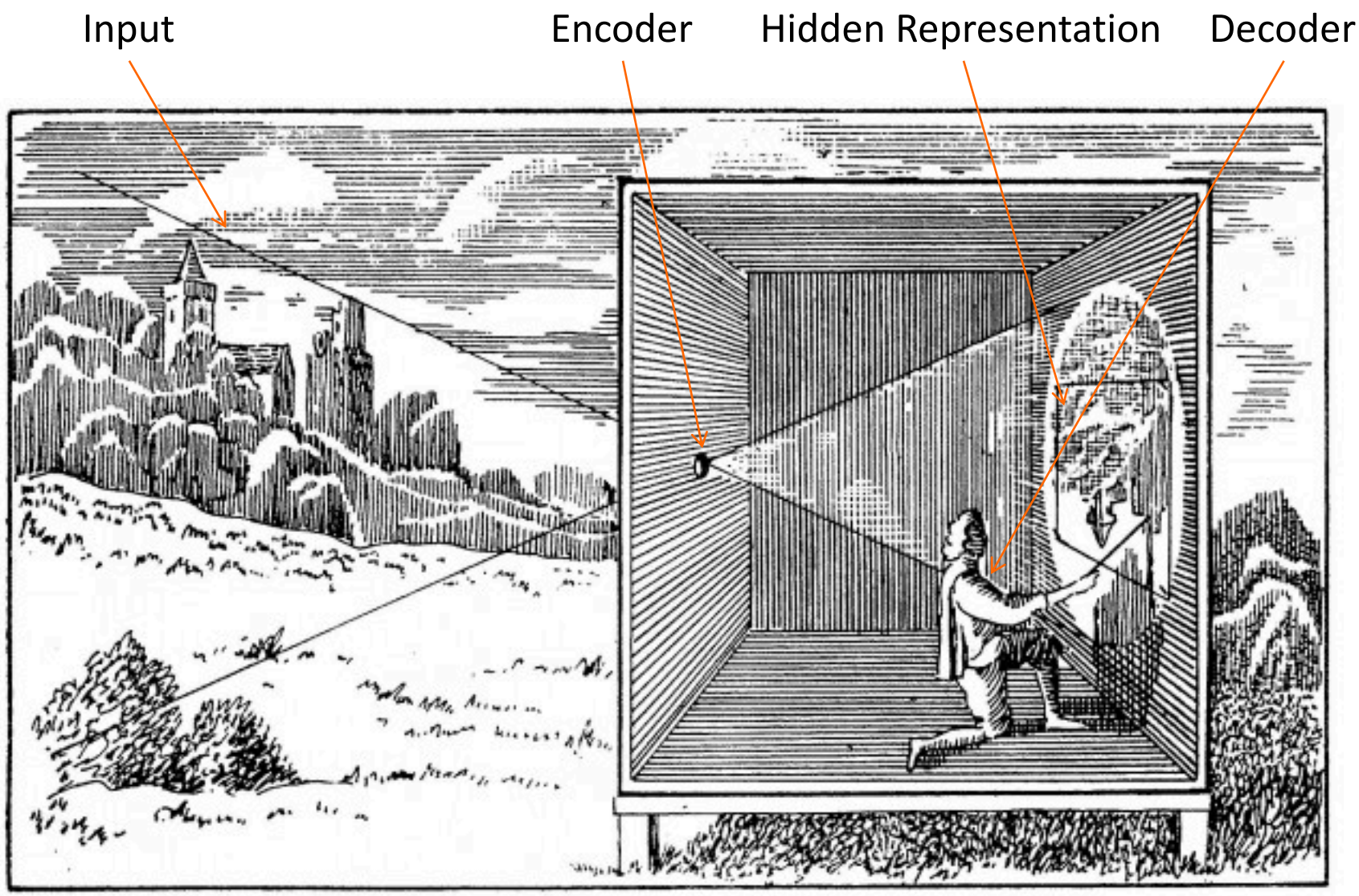
# Autoencoder (AE)

- Neural network trained to copy its input $x$ to its output
- Hidden layer $z$ describes a ***code*** to represent the input
- Two parts
  - Encoder: $\mathrm{z} = f(x)$
  - Decoder: $x' = g(z)$
- Goal: minimize $\mathrm{C}\Big(x, g\big(f(x)\big)\Big)$
  - $C$ penalizes $g\big(f(x)\big)$ for being dissimilar from $x$
  - Example: mean squared error

# Are Autoencoders useful?

- Can't we just set $g\big(f(x)\big) = x$ everywhere?
  - Not very useful
- Design the AE so it can't learn to copy the input perfectly
  - Restrict the AE to copy only approximately
  - Can prioritize certain aspects of the input
- Examples
  - Restrict the size of the code (i.e. add a bottleneck)
  - Add regularization

# Camera Obscura



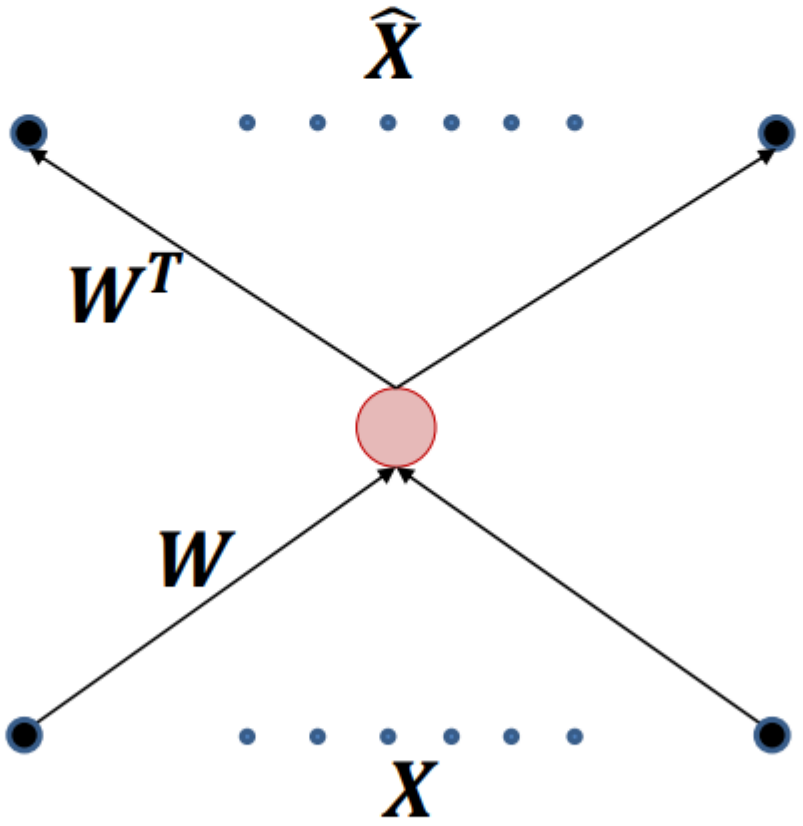

***Camera Obscura, the renaissance autoencoder***

# Undercomplete Autoencoders

- Hope: training the autoencoder will result in $z$ having useful properties
- $\text{Dim}(z) < \text{Dim}(x) \Rightarrow$ ***undercomplete*** autoencoder
  - Forces the AE to capture the most salient features of the training data
  - The AE only approximately copies the input (lossy compression)
- Forced to learn a meaningful non-linear dimensionality reduction of the data!

# Undercomplete Autoencoders

- Special case: Linear decoder, mean squared error loss
- Example: single hidden unit
- What will this learn?

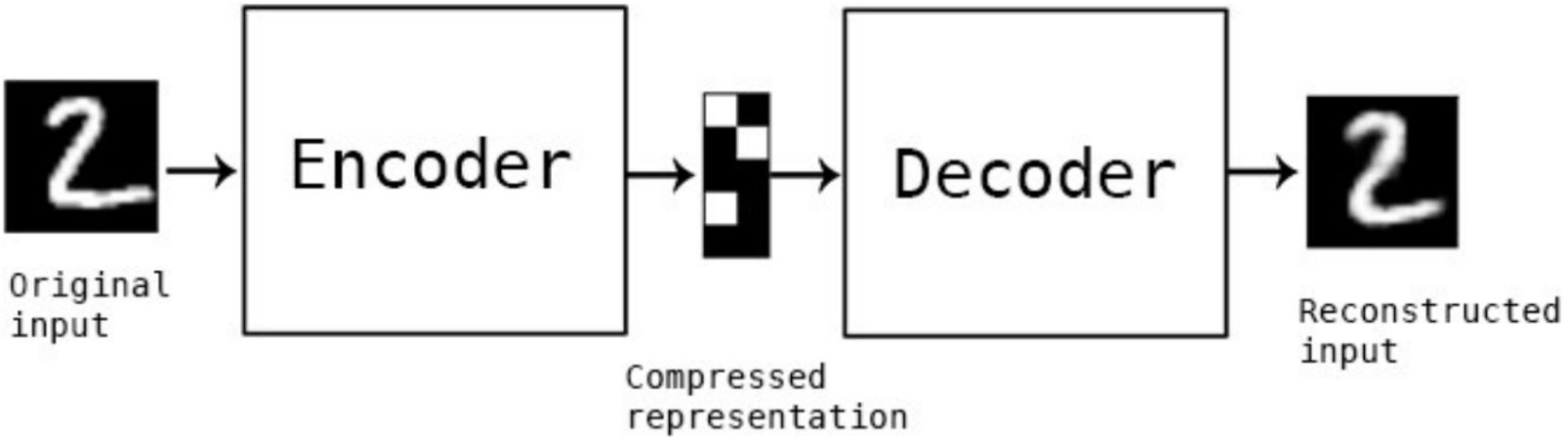



- Minimize reconstruction error:

$$\hat{w} = \arg\min_{w} \mathbb{E}[\|x - w^T w x\|^2]$$

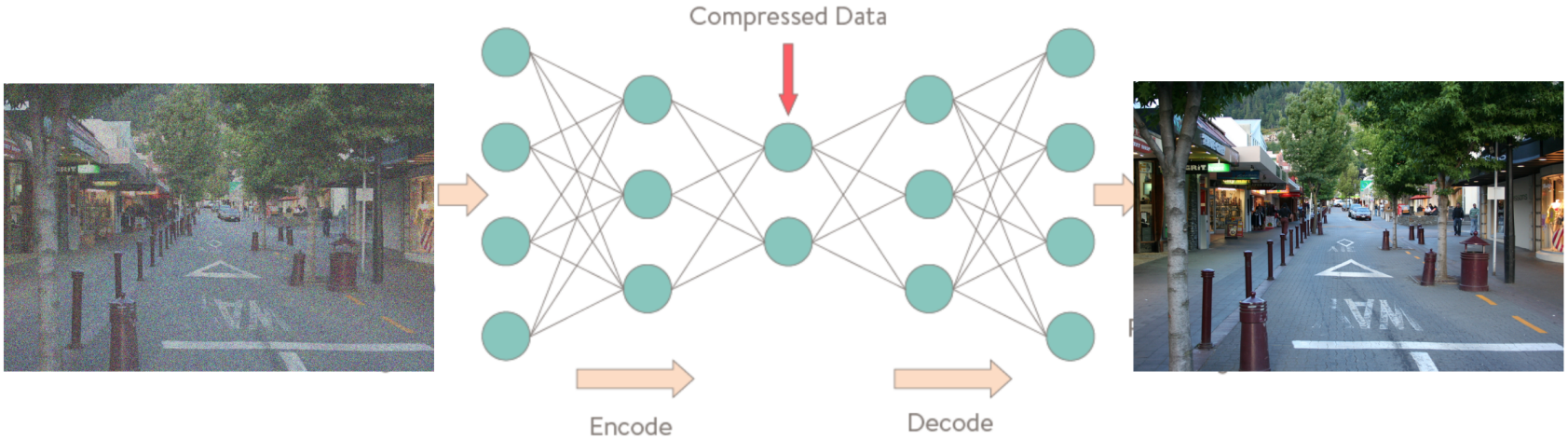
- Equivalent to PCA!

# Autoencoding MNIST




Moves to a continuous space where distances are meaningful

# Application to Image Denoising

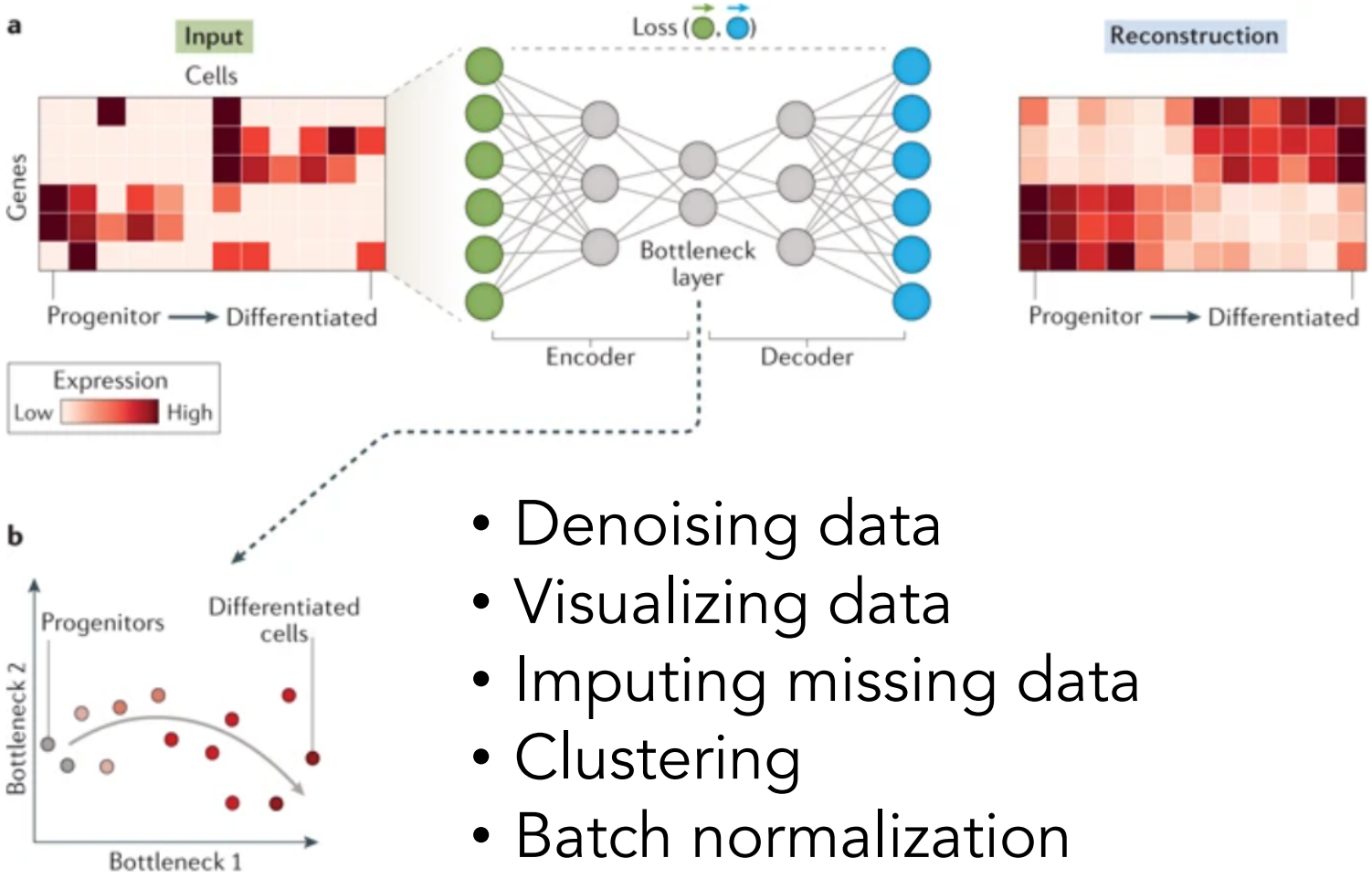



Compression layer automatically denoises

# Denoising Autoencoder

- Artificially add noise to input, train network to take it off

# Uses in Single Cell




- Denoising data
- Visualizing data
- Imputing missing data
- Clustering
- Batch normalization
- Data generation

# AutoEncoders in Single Cell Genomics


25. Ding, J., Condon, A. & Shah, S. P. Interpretable dimensionality reduction of single cell transcriptome data with deep generative models. *Nat. Commun.* **9**, 2002 (2018).

PubMed Central PubMed Google Scholar

26. Cho, H., Berger, B. & Peng, J. Generalizable and scalable visualization of single-cell data using neural networks. *Cell Syst.* **7**, 185–191 (2018).

CAS PubMed Central PubMed Google Scholar

27. Deng, Y., Bao, F., Dai, Q., Wu, L. & Altschuler, S. Massive single-cell RNA-seq analysis and imputation via deep learning. Preprint at *bioRxiv* https://doi.org/10.1101/315556 (2018).

Article Google Scholar

28. Talwar, D., Mongia, A., Sengupta, D. & Majumdar, A. AutoImpute: autoencoder based imputation of single-cell RNA-seq data. *Sci. Rep.* **8**, 16329 (2018).

PubMed Central PubMed Google Scholar

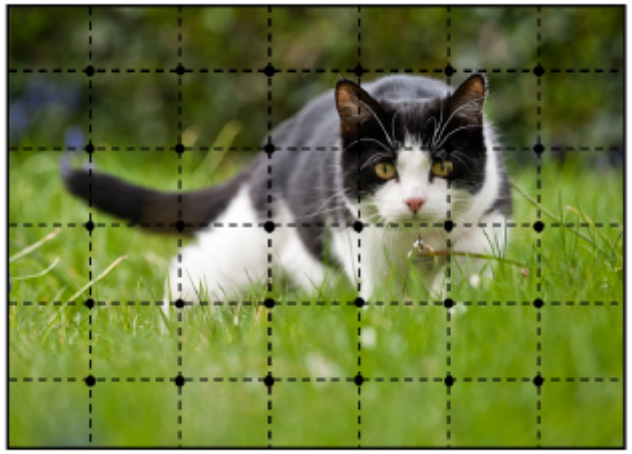
29. Amodio, M. et al. Exploring single-cell data with deep multitasking neural networks. Preprint at *bioRxiv* https://doi.org/10.1101/237065 (2019).

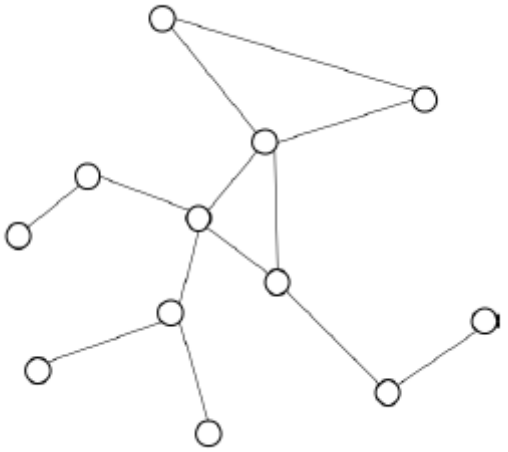
Article Google Scholar

30. Eraslan, G., Simon, L. M., Mircea, M., Mueller, N. S. & Theis, F. J. Single-cell RNA-seq denoising using a deep count autoencoder. *Nat. Commun.* **10**, 390 (2019)

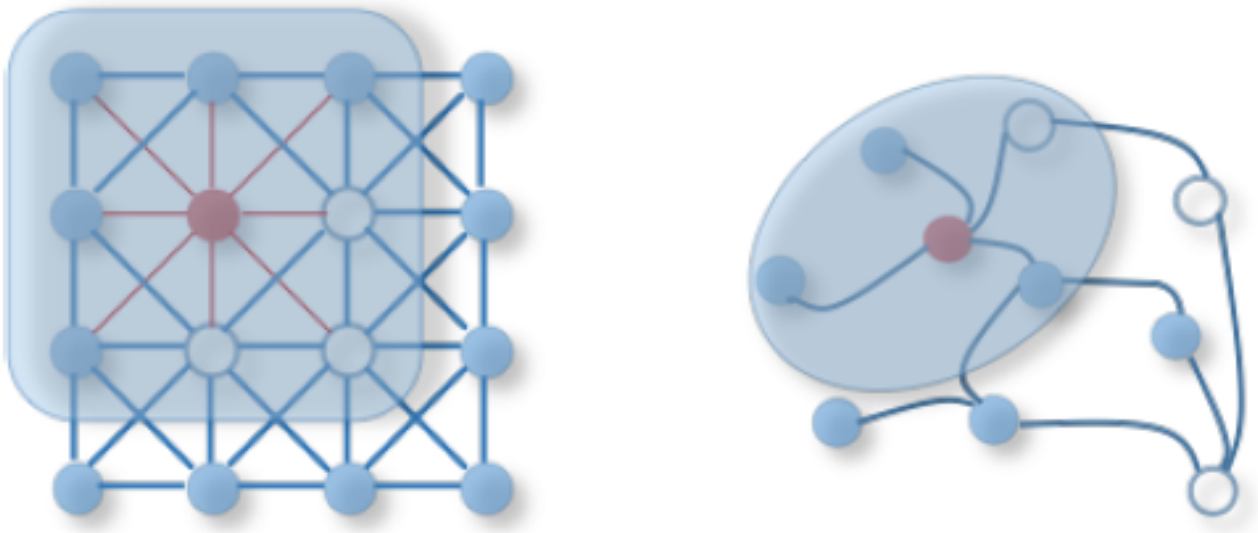

https://www.nature.com/articles/s41576-019-0122-6

# Images are a specific sort of graph—grid graph



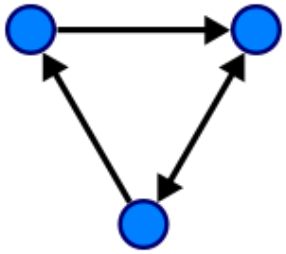


# Graphs



# Graph and node features

$G = (V, E)$



Each node v has node features $\mathbf{x}_v$

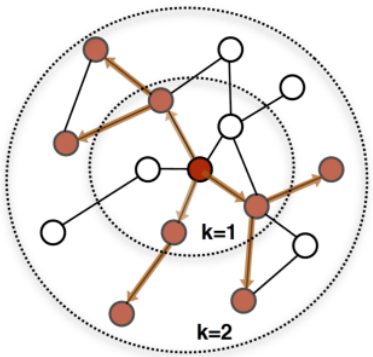
Each node v adjacent edges $co[v]$

Connecting to neighbors $ne[v]$

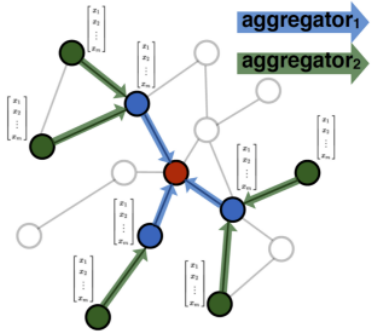
New node features are computed like this:

$$\mathbf{h}_v = f\left(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}\right)$$

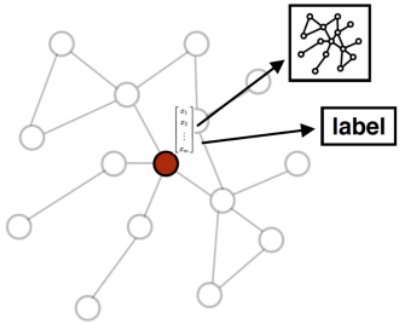
# GraphSAGE Network




1. Sample neighborhood




2. Aggregate feature information from neighbors




3. Predict graph context and label using aggregated information

# Aggregation

1. Mean aggregation:

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$$

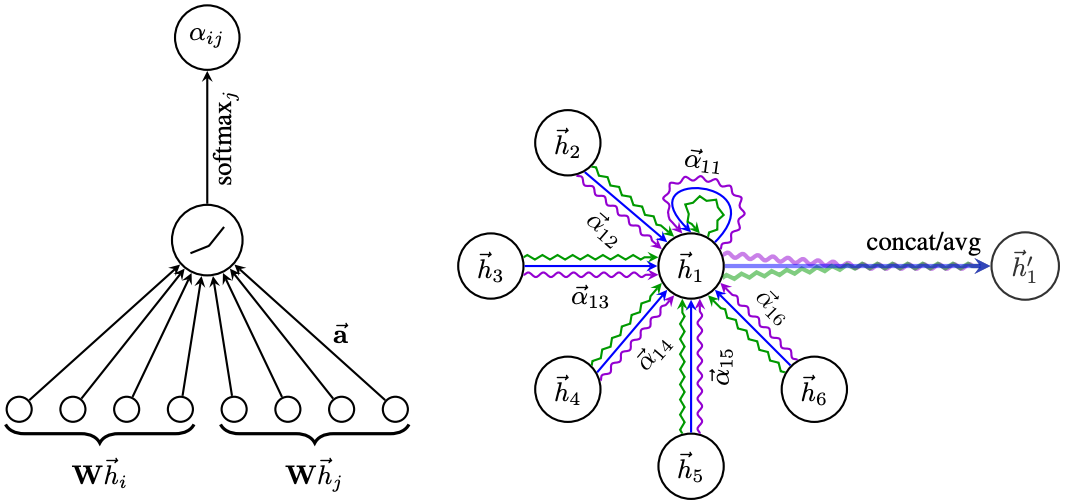
Neighborhood aggregation based on the mean of all nodes in the neighborhood

2. Max-pooling aggregation:

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\})$$

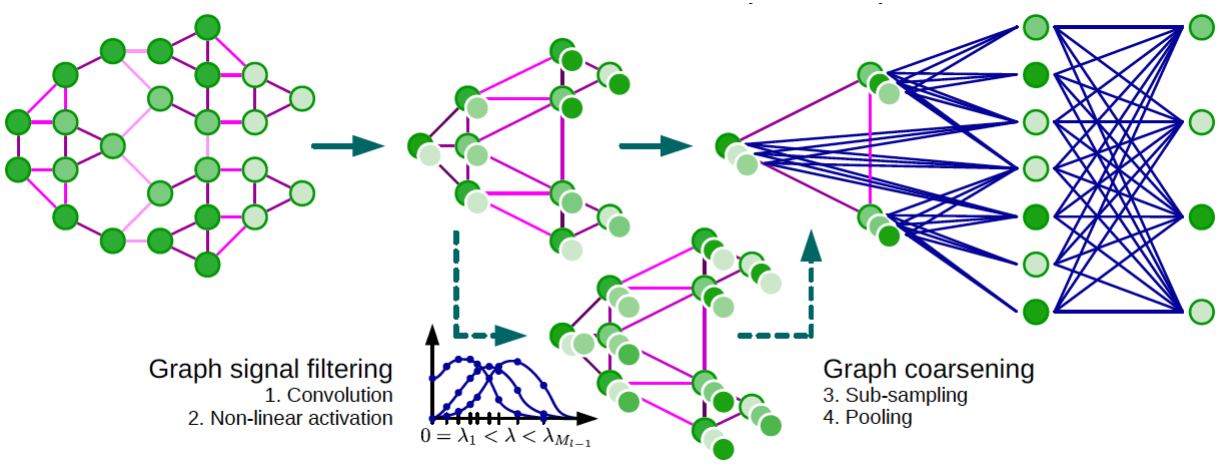
3. LSTM aggregation

# Graph Attention



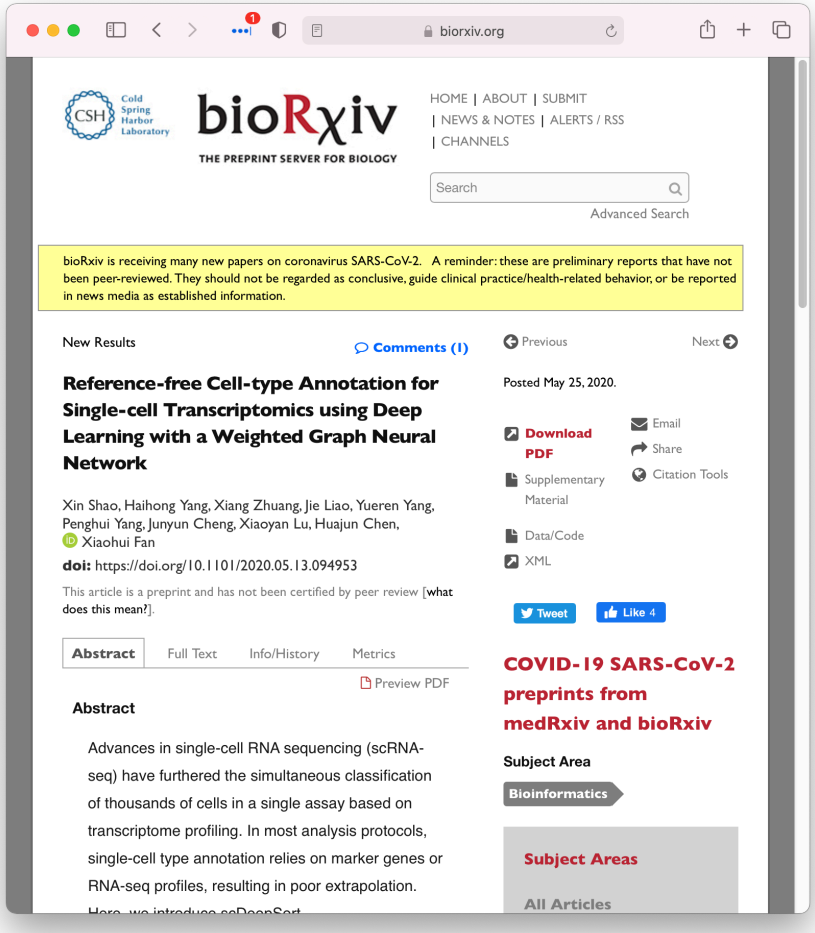
# Spectral Graph Convolutional Networks

- We can also regard features of vertices as *graph signals*
- Can use the graph Laplacian to filter signals




Defferard et al.

# GNNs in single cell biology





https://www.biorxiv.org/content/10.1101/2020.05.13.094953v2

https://www.biorxiv.org/content/10.1101/2020.08.02.233569v1