The Krishnaswamy Laboratory

Yale Genetics and Yale SEAS present

Machine Learning

for

Single Cell Analysis

Online - May 20-29, 2020

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

Recap of Clustering, Trajectories, Gene relationships

An Impossibility Theorem for Clustering

Jon Kleinberg
Department of Computer Science
Cornell University
Ithaca NY 14853

Abstract

Although the study of *clustering* is centered around an intuitively compelling goal, it has been very difficult to develop a unified framework for reasoning about it at a technical level, and profoundly diverse approaches to clustering abound in the research community. Here we suggest a formal perspective on the difficulty in finding such a unification, in the form of an *impossibility theorem*: for a set of three simple properties, we show that there is no clustering function satisfying all three. Relaxations of these properties expose some of the interesting (and unavoidable) trade-offs at work in well-studied clustering techniques such as single-linkage, sum-of-pairs, k-means, and k-median.

1 Introduction

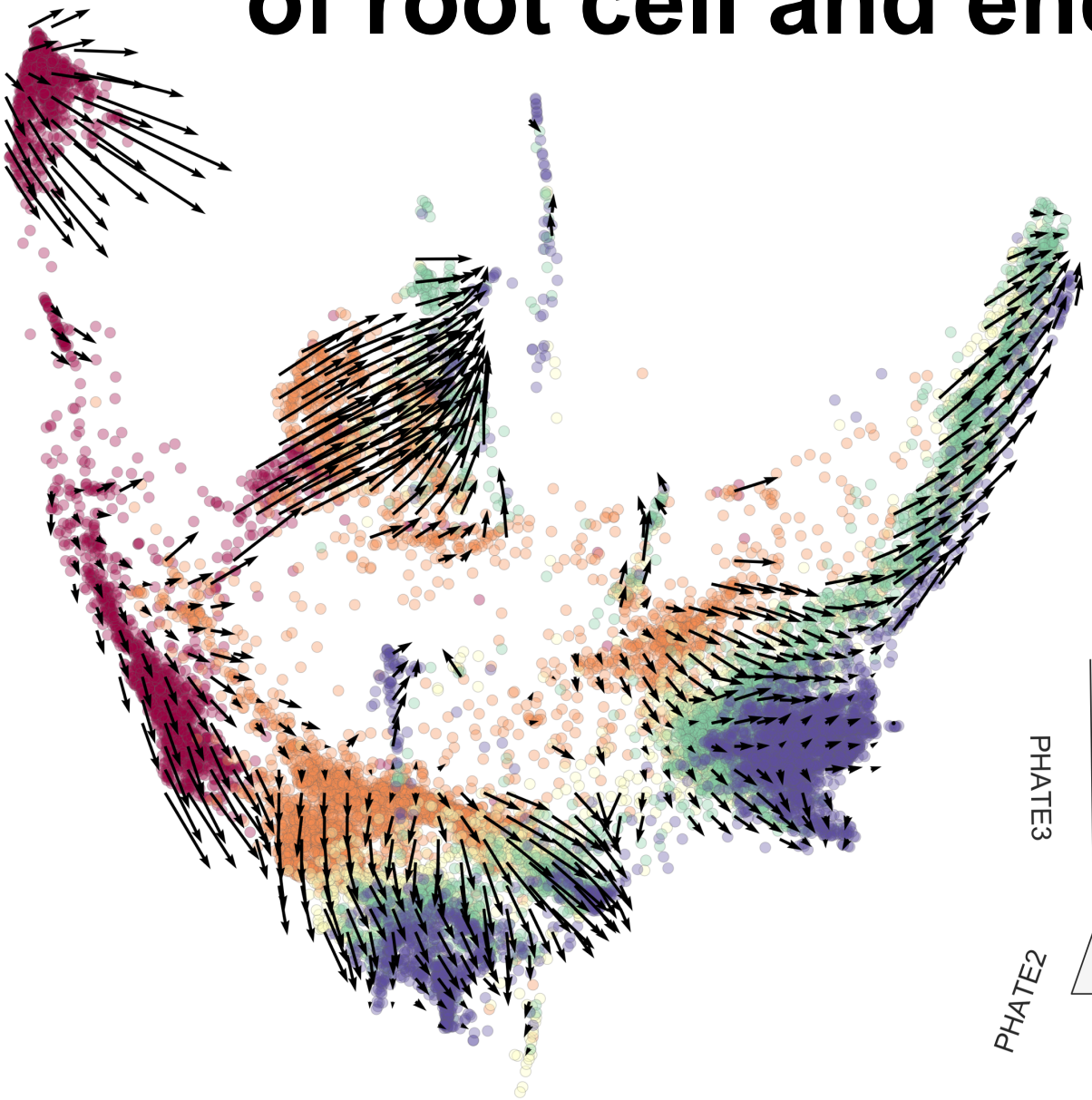
Clustering is a notion that arises naturally in many fields; whenever one has a heterogeneous set of objects, it is natural to seek methods for grouping them together based on an underlying measure of similarity. A standard approach is to represent the collection of objects as a set of abstract points, and define distances among the points to represent similarities — the closer the points, the more similar they are. Thus, clustering is centered around an intuitively compelling but vaguely defined goal: given an underlying set of points, partition them into a collection of *clusters* so that points in the same cluster are close together, while points in different clusters are far apart.

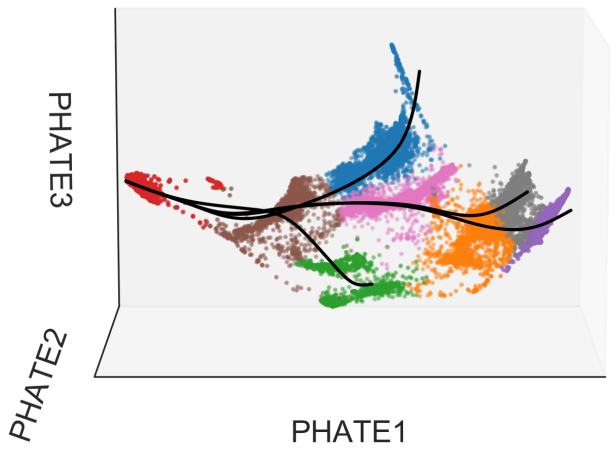
The study of clustering is unified only at this very general level of description, however; at the level of concrete methods and algorithms, one quickly encounters a bewildering array of different clustering techniques, including agglomerative, spectral, information-theoretic, and centroid-based, as well as those arising from combinatorial optimization and from probabilistic generative models. These techniques are based on diverse underlying principles, and they often lead to qualitatively different results. A number of standard textbooks [1, 4, 6, 9] provide overviews of a range of the approaches that are generally employed.

Given the scope of the issue, there has been relatively little work aimed at reasoning about clustering independently of any particular algorithm, objective function, or generative data model. But it is not clear that this needs to be the case. To take a motivating example from a technically different but methodologically similar set-

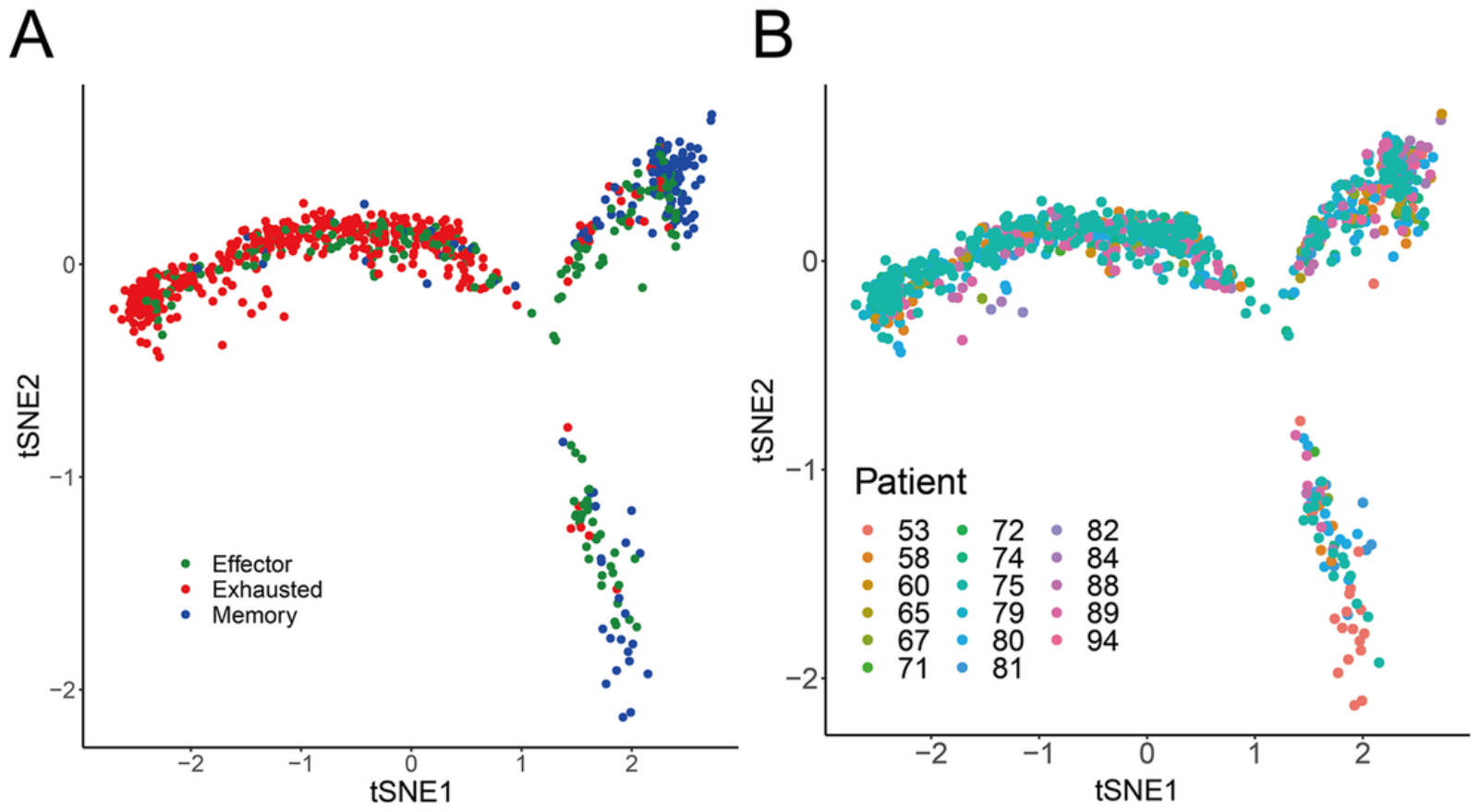
Kleinberg (2002)

RNA velocity assists selection of root cell and endpoints



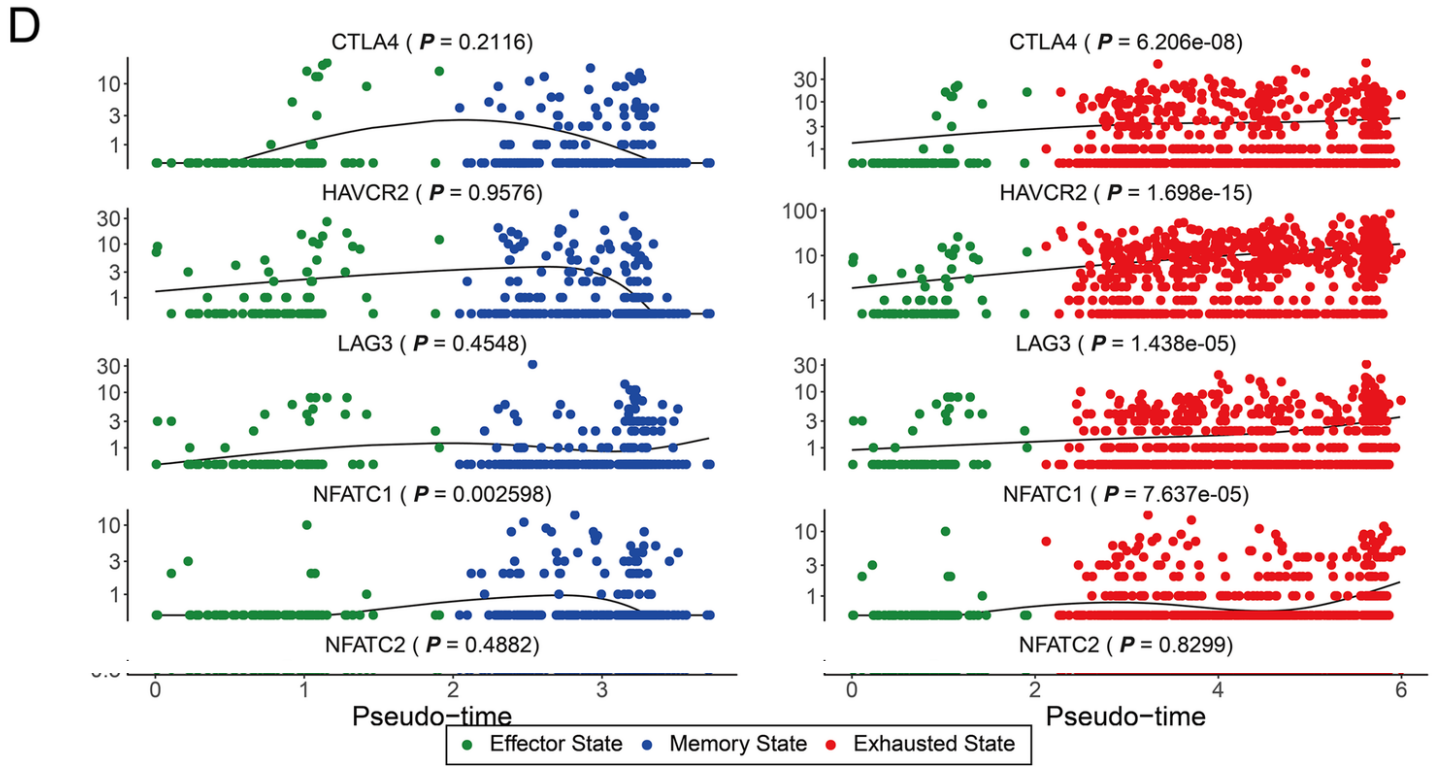


Pseudotime is not time series analysis



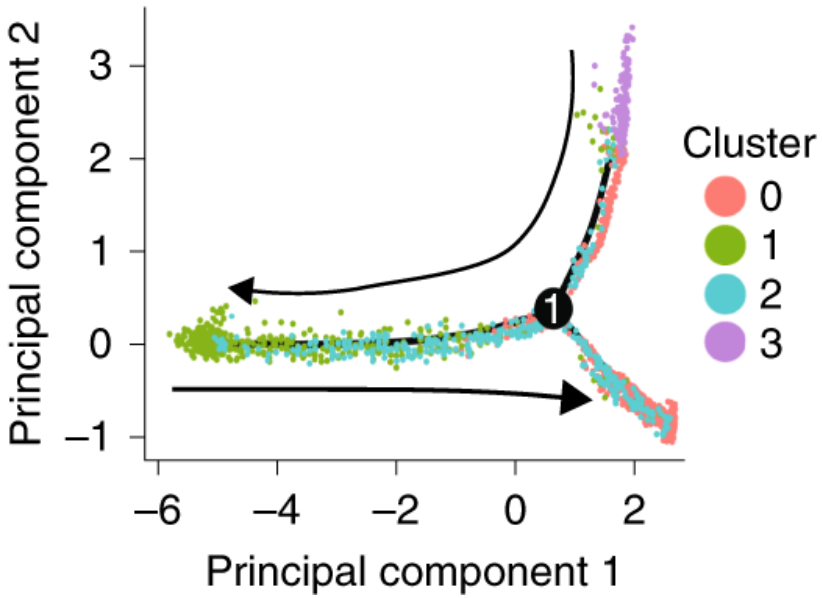
Kim et al. Genome Medicine (2020)
https://doi.org/10.1186/s13073-020-00722-9

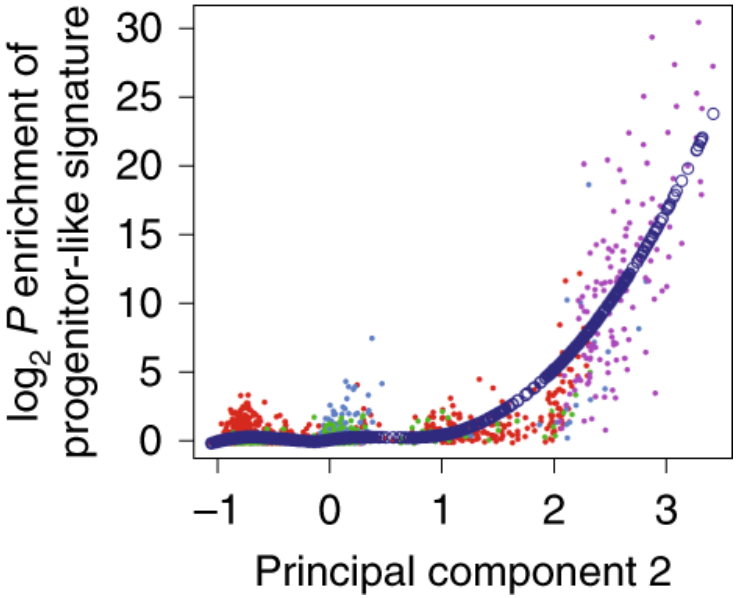
Pseudotime is not time series analysis



Kim et al. Genome Medicine (2020)
https://doi.org/10.1186/s13073-020-00722-9

Pseudotime is not time series analysis



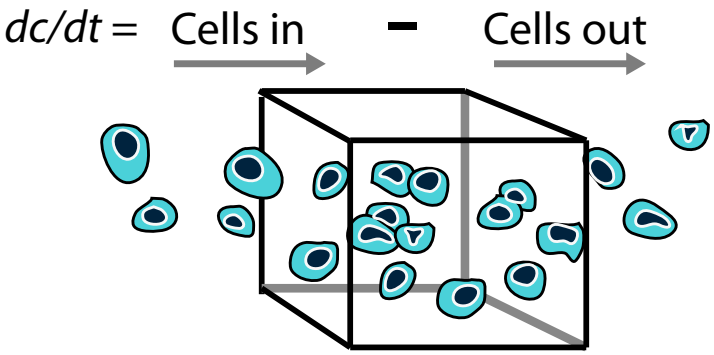


Yao et al. Nature Immunology (2019)

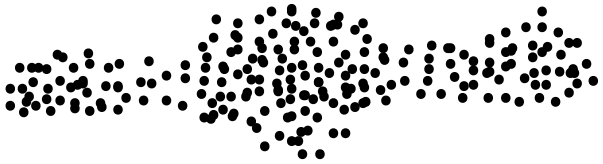
https://doi.org/10.1038/s41590-019-0403-4

Challenges in inferring cell state dynamics from snapshots

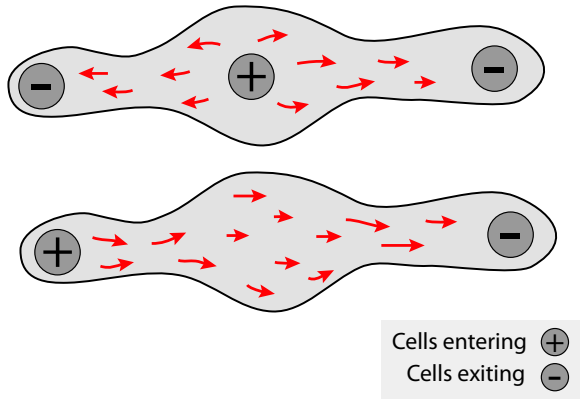
Population balance law



Single-cell snapshot



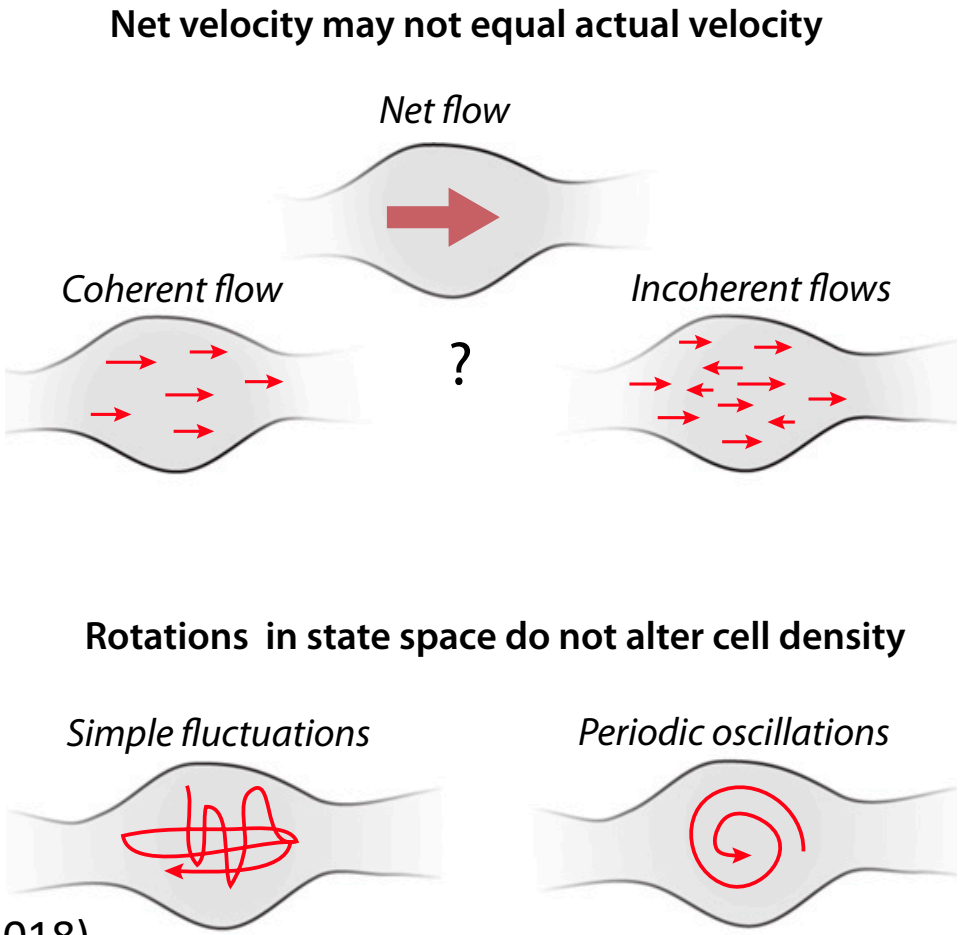
Entry and exit points direct the flow of cells



Weinreb et al. PNAS (2018)

https://doi.org/10.1073/pnas.1714723115

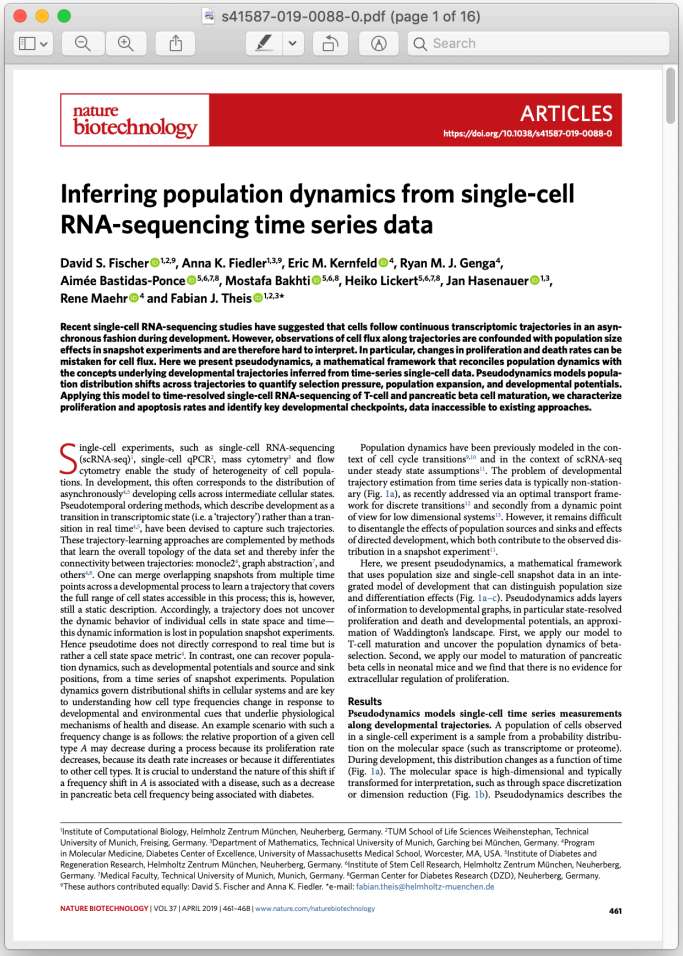
Challenges in inferring cell state dynamics from snapshots



Weinreb et al. PNAS (2018)

https://doi.org/10.1073/pnas.1714723115

Leveraging time series to reconstruct developmental transitions



s41587-019-0088-0.pdf (page 1 of 16)

nature biotechnology

ARTICLES

https://doi.org/10.1038/s41587-019-0088-0

Inferring population dynamics from single-cell RNA-sequencing time series data

David S. Fischer[1,2,9], Anna K. Fiedler[1,3,9], Eric M. Kernfeld[4], Ryan M. J. Genga[4], Aimée Bastidas-Ponce[5,6,7,8], Mostafa Bakhti[5,6,8], Heiko Lickert[5,6,7,8], Jan Hasenauer[1,3], Rene Maehr[4] and Fabian J. Theis[1,2,3*]

Recent single-cell RNA-sequencing studies have suggested that cells follow continuous transcriptomic trajectories in an asynchronous fashion during development. However, observations of cell flux along trajectories are confounded with population size effects in snapshot experiments and are therefore hard to interpret. In particular, changes in proliferation and death rates can be mistaken for cell flux. Here we present pseudodynamics, a mathematical framework that reconciles population dynamics with the concepts underlying developmental trajectories inferred from time-series single-cell data. Pseudodynamics models population distribution shifts across trajectories to quantify selection pressure, population expansion, and developmental potentials. Applying this model to time-resolved single-cell RNA-sequencing of T-cell and pancreatic beta cell maturation, we characterize proliferation and apoptosis rates and identify key developmental checkpoints, data inaccessible to existing approaches.

Single-cell experiments, such as single-cell RNA-sequencing (scRNA-seq)[1], single-cell qPCR[2], mass cytometry[3] and flow cytometry enable the study of heterogeneity of cell populations. In development, this often corresponds to the distribution of asynchronously[4,5] developing cells across intermediate cellular states. Pseudotemporal ordering methods, which describe development as a transition in transcriptomic state (i.e. a 'trajectory') rather than a transition in real time[4,5], have been devised to capture such trajectories. These trajectory-learning approaches are complemented by methods that learn the overall topology of the data set and thereby infer the connectivity between trajectories: monocle[2], graph abstraction[3], and others[6,4]. One can merge overlapping snapshots from multiple time points across a developmental process to learn a trajectory that covers the full range of cell states accessible in this process; this is, however, still a static description. Accordingly, a trajectory does not uncover the dynamic behavior of individual cells in state space and time—this dynamic information is lost in population snapshot experiments. Hence pseudotime does not directly correspond to real time but is rather a cell state space metric[4]. In contrast, one can recover population dynamics, such as developmental potentials and source and sink positions, from a time series of snapshot experiments. Population dynamics govern distributional shifts in cellular systems and are key to understanding how cell type frequencies change in response to developmental and environmental cues that underlie physiological mechanisms of health and disease. An example scenario with such a frequency change is as follows: the relative proportion of a given cell type *A* may decrease during a process because its proliferation rate decreases, because its death rate increases or because it differentiates to other cell types. It is crucial to understand the nature of this shift if a frequency shift in *A* is associated with a disease, such as a decrease in pancreatic beta cell frequency being associated with diabetes.

Population dynamics have been previously modeled in the context of cell cycle transitions[9,10] and in the context of scRNA-seq under steady state assumptions[11]. The problem of developmental trajectory estimation from time series data is typically non-stationary (Fig. 1a), as recently addressed via an optimal transport framework for discrete transitions[12] and secondly from a dynamic point of view for low dimensional systems[13]. However, it remains difficult to disentangle the effects of population sources and sinks and effects of directed development, which both contribute to the observed distribution in a snapshot experiment[11].

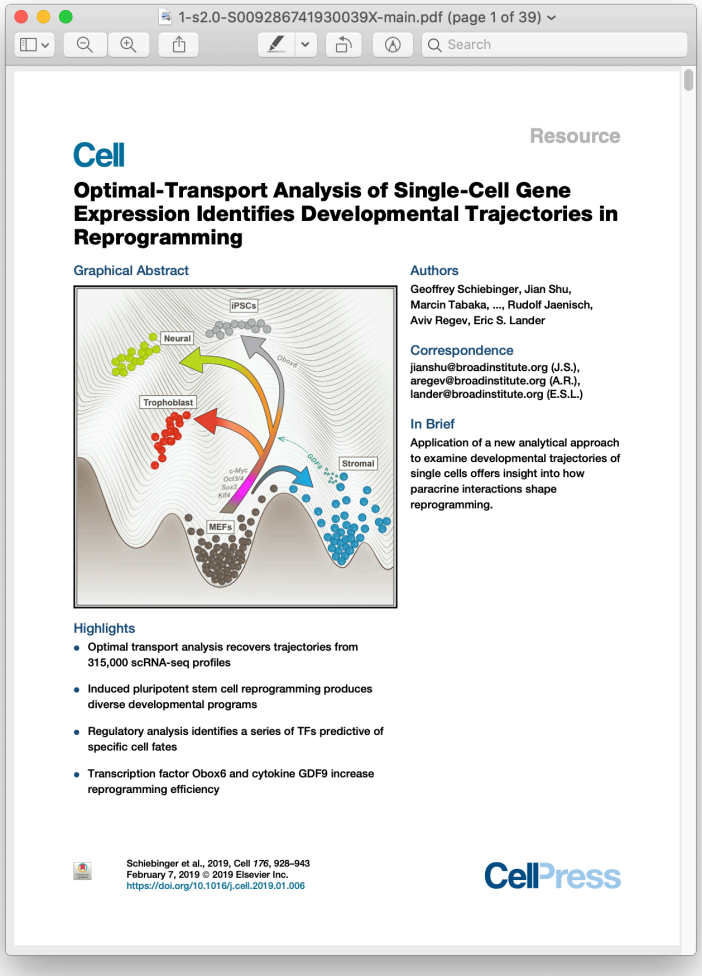
Here, we present pseudodynamics, a mathematical framework that uses population size and single-cell snapshot data in an integrated model of development that can distinguish population size and differentiation effects (Fig. 1a–c). Pseudodynamics adds layers of information to developmental graphs, in particular state-resolved proliferation and death and developmental potentials, an approximation of Waddington's landscape. First, we apply our model to T-cell maturation and uncover the population dynamics of beta-selection. Second, we apply our model to maturation of pancreatic beta cells in neonatal mice and we find that there is no evidence for extracellular regulation of proliferation.

Results

Pseudodynamics models single-cell time series measurements along developmental trajectories. A population of cells observed in a single-cell experiment is a sample from a probability distribution on the molecular space (such as transcriptome or proteome). During development, this distribution changes as a function of time (Fig. 1a). The molecular space is high-dimensional and typically transformed for interpretation, such as through space discretization or dimension reduction (Fig. 1b). Pseudodynamics describes the

[1]Institute of Computational Biology, Helmholz Zentrum München, Neuherberg, Germany. [2]TUM School of Life Sciences Weihenstephan, Technical University of Munich, Freising, Germany. [3]Department of Mathematics, Technical University of Munich, Garching bei München, Germany. [4]Program in Molecular Medicine, Diabetes Center of Excellence, University of Massachusetts Medical School, Worcester, MA, USA. [5]Institute of Diabetes and Regeneration Research, Helmholtz Zentrum München, Neuherberg, Germany. [6]Institute of Stem Cell Research, Helmholtz Zentrum München, Neuherberg, Germany. [7]Medical Faculty, Technical University of Munich, Munich, Germany. [8]German Center for Diabetes Research (DZD), Neuherberg, Germany. [9]These authors contributed equally: David S. Fischer and Anna K. Fiedler. *e-mail: fabian.theis@helmholtz-muenchen.de

Fisher et al. (2019)



1-s2.0-S009286741930039X-main.pdf (page 1 of 39)

Cell

Resource

Optimal-Transport Analysis of Single-Cell Gene Expression Identifies Developmental Trajectories in Reprogramming

Graphical Abstract

Authors

Geoffrey Schiebinger, Jian Shu, Marcin Tabaka, ..., Rudolf Jaenisch, Aviv Regev, Eric S. Lander

Correspondence

jianshu@broadinstitute.org (J.S.), aregev@broadinstitute.org (A.R.), lander@broadinstitute.org (E.S.L.)

In Brief

Application of a new analytical approach to examine developmental trajectories of single cells offers insight into how paracrine interactions shape reprogramming.

Highlights

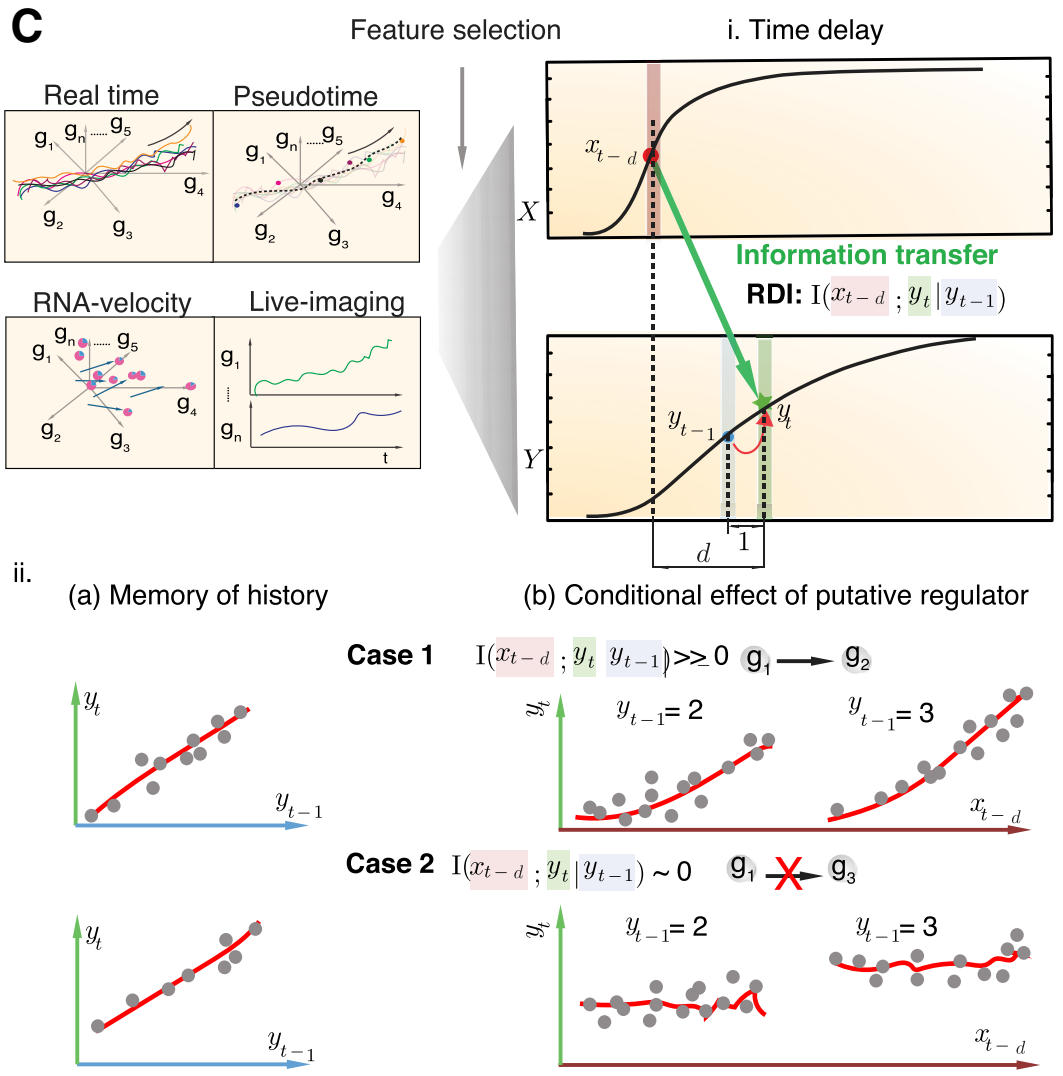
- Optimal transport analysis recovers trajectories from 315,000 scRNA-seq profiles
- Induced pluripotent stem cell reprogramming produces diverse developmental programs
- Regulatory analysis identifies a series of TFs predictive of specific cell fates
- Transcription factor Obox6 and cytokine GDF9 increase reprogramming efficiency

Schiebinger et al., 2019, Cell 176, 928–943
February 7, 2019 © 2019 Elsevier Inc.
https://doi.org/10.1016/j.cell.2019.01.006

CellPress

Schiebinger et al. (2019)

Leveraging time series to reconstruct gene-gene relationships

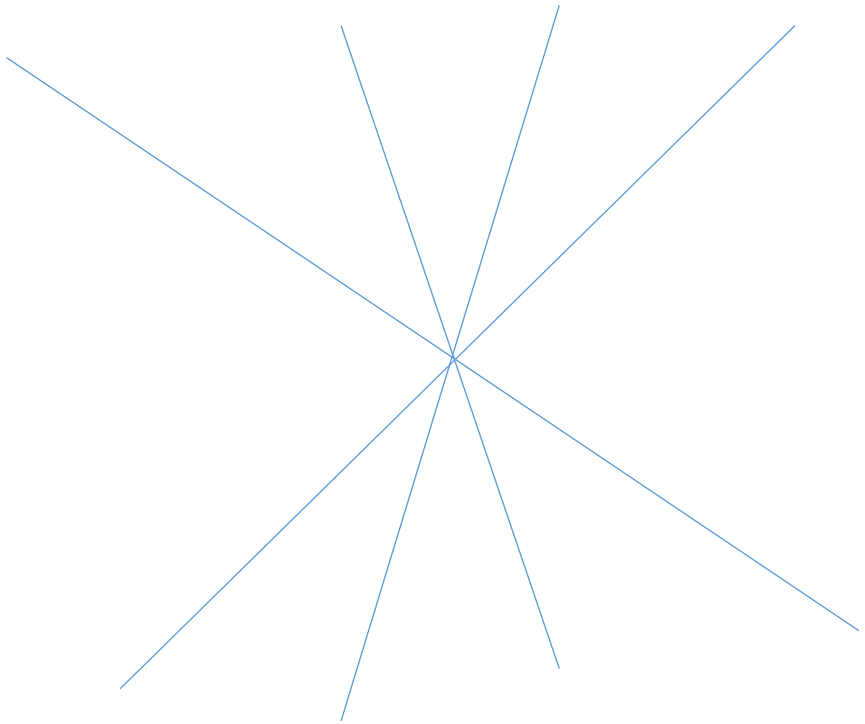


Qiu et al. Cell Systems (2020)
https://doi.org/10.1016/j.cels.2020.02.003

Introduction to Deep Learning

What is deep learning?

2-parameter function

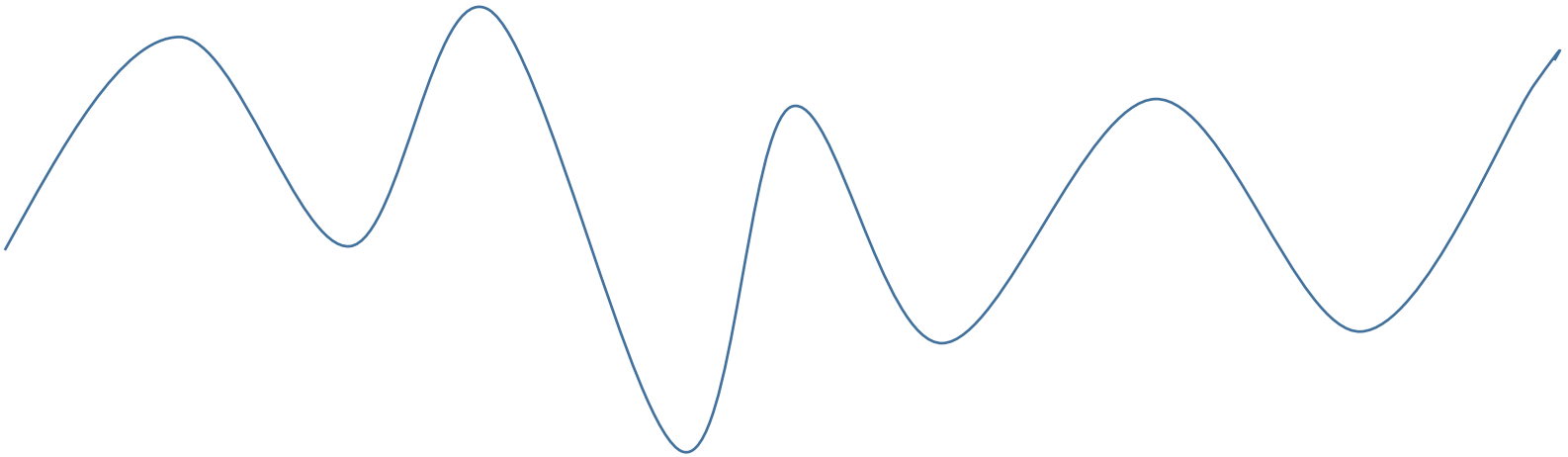


Can learn any line

K parameter function

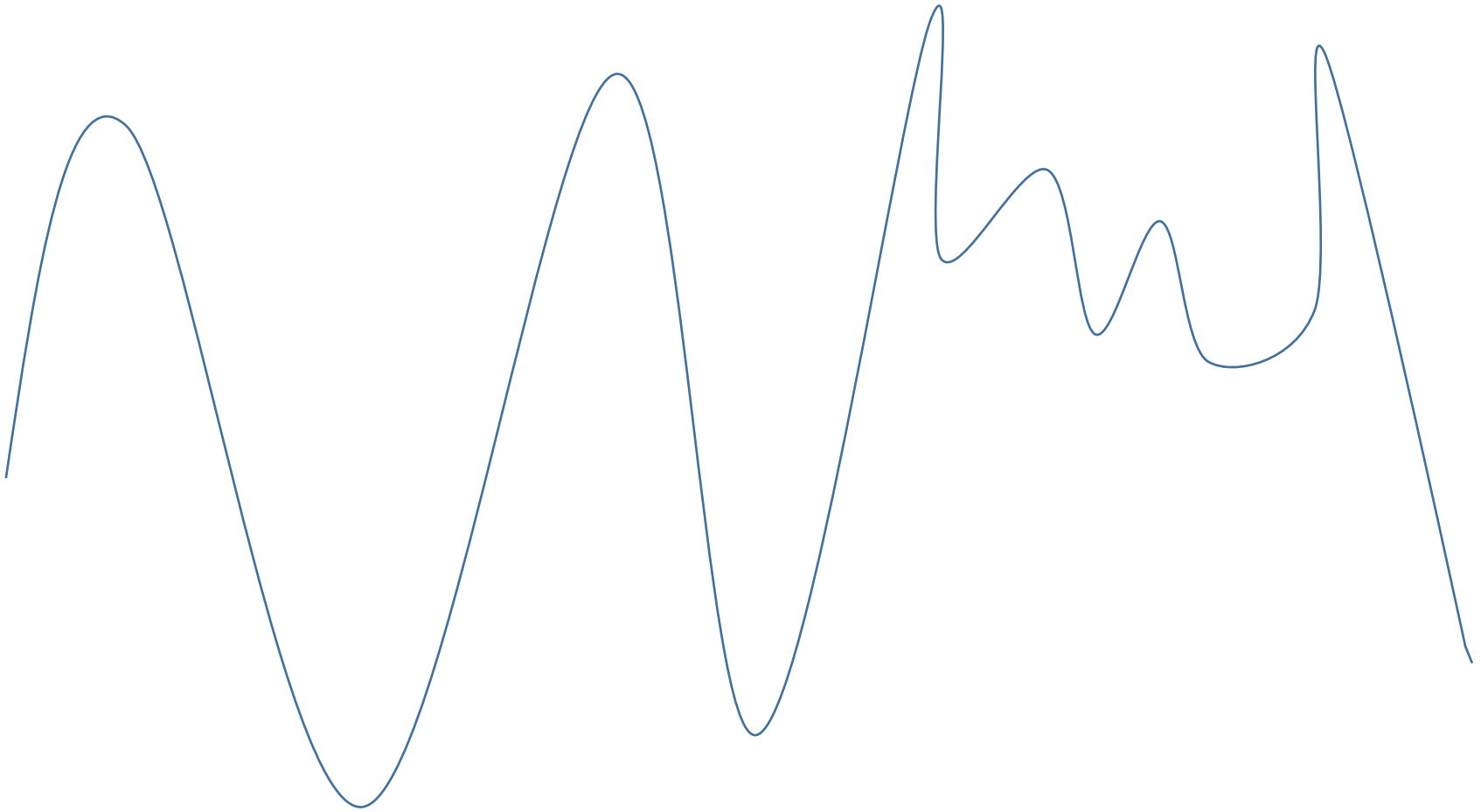
- Parameters add power!

Any kth degree polynomial



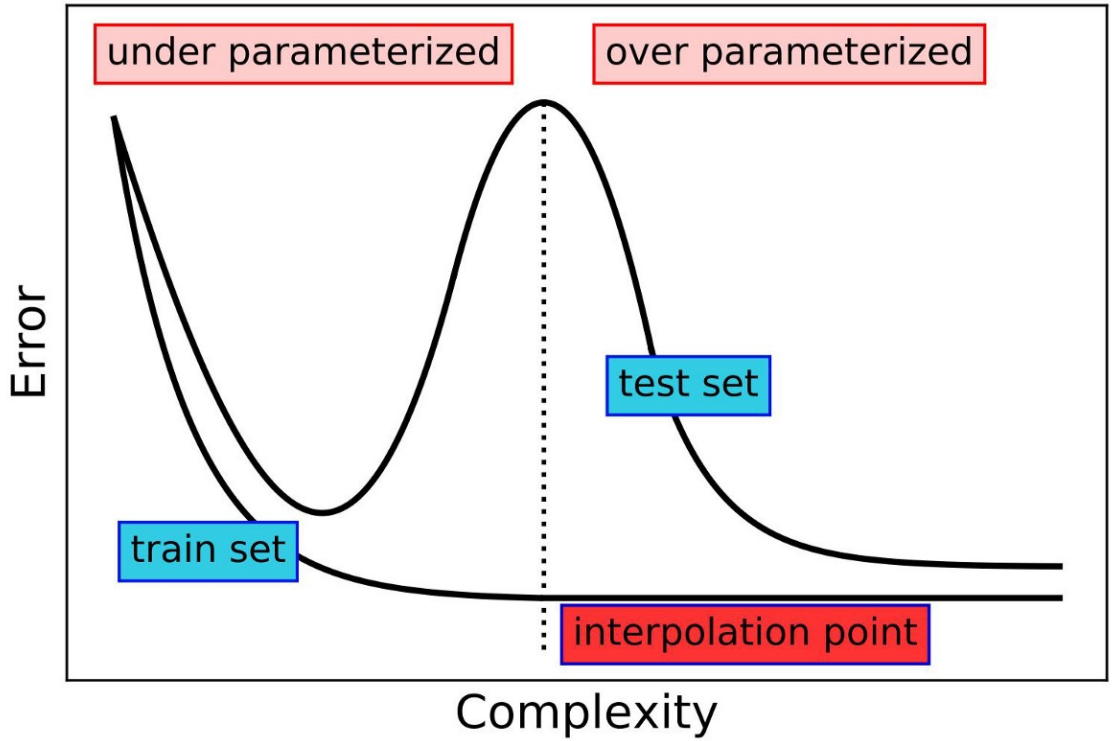
More parameters

- Allow you to approximate any function



Neural Networks

- Highly parameterized networks, very flexible
- Universal function approximators

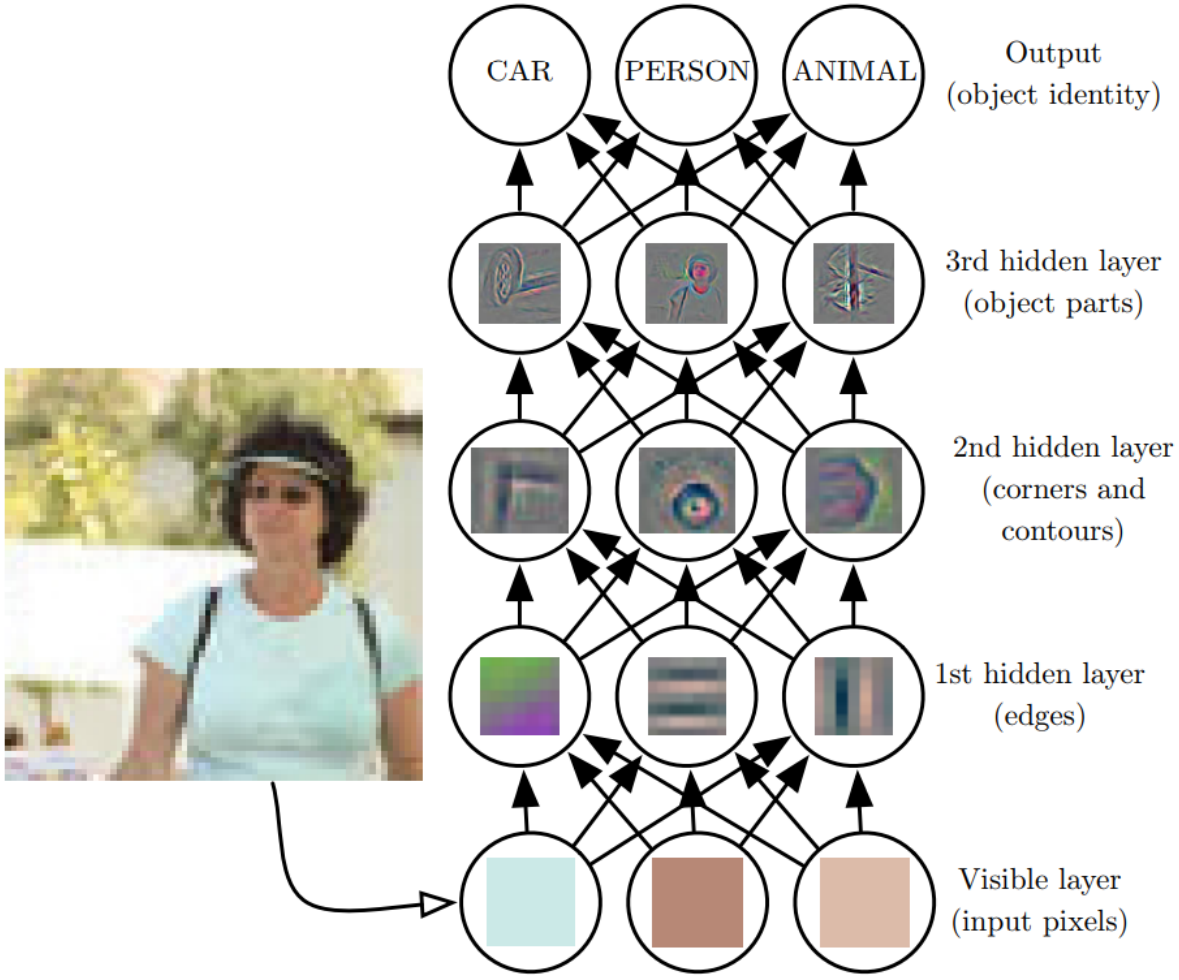


https://medium.com/@LightOnIO/beyond-overfitting-and-beyond-silicon-the-double-descent-curve-18b6d9810e1b

Depth

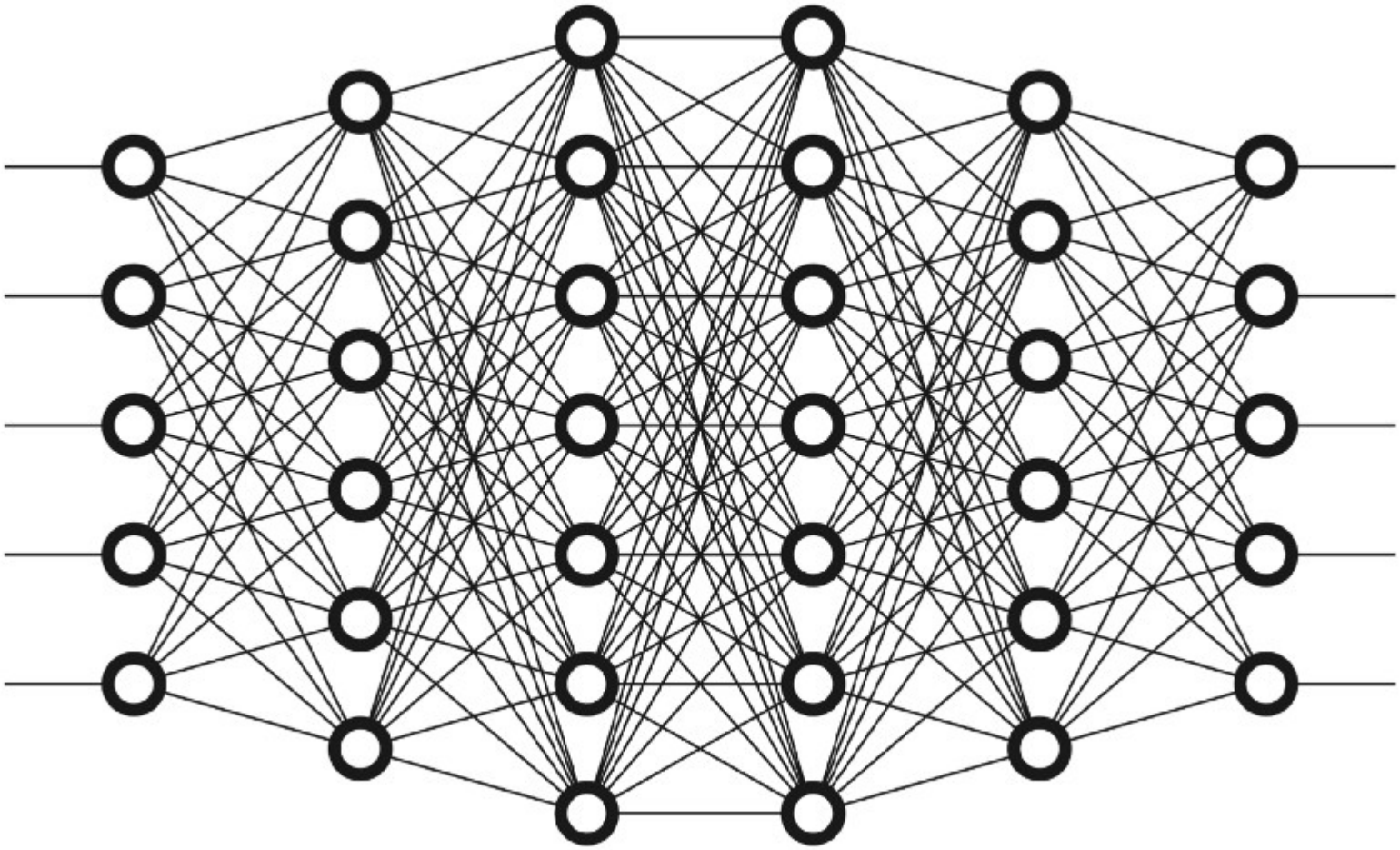
- **Deep learning** comes from the composition of many layers of multiparameter functions
- Why is function composition useful?
- Imagine image data, input variables are single pixels
- First layer computes functions of pixels
- Second layer can computes functions of groups of nearby pixels (edges, lines)
- Later layers can recognize complex objects

Why go deep?

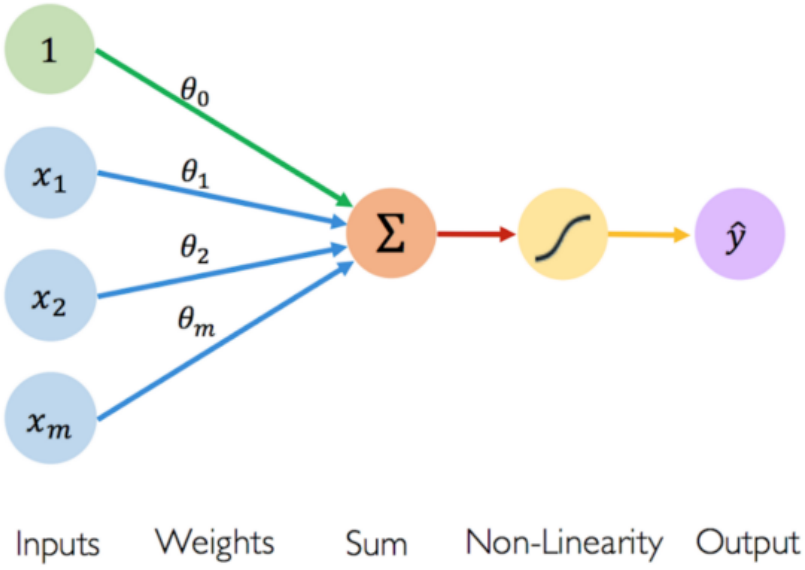


Goodfellow et al., 2016

Neural Network Picture



Cascades of Neurons

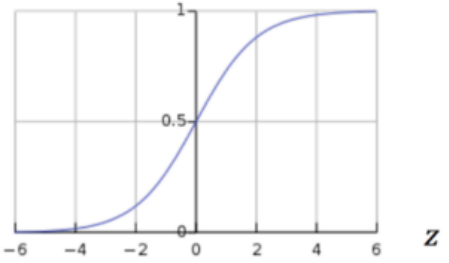


Activation Functions

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

- Example: sigmoid function

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



MIT: Alexander Amini, 2018 introtodeeplearning.com

Inspired by Biological Neurons

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.

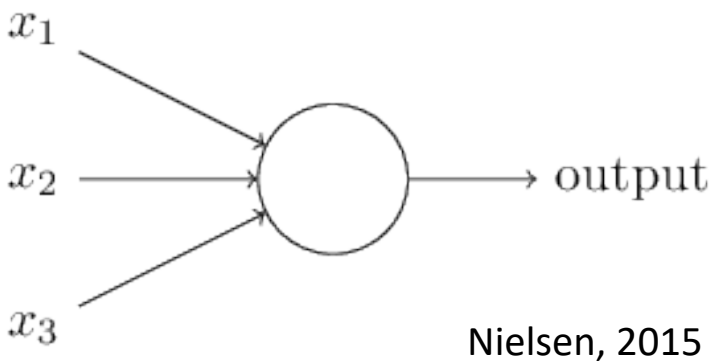
- Yann LeCun, Director of FAIR

Artificial Neural Networks

- A computing system made up of simple, interconnected processing elements that process information by their dynamic state response to external inputs
- Loosely inspired by neural computation, hence processing elements are called “neurons”
- Trained with instead of programmed
- Capacity to encode vast array of functions and learn makes them capable of learning complicated functions of inputs amenable to various tasks

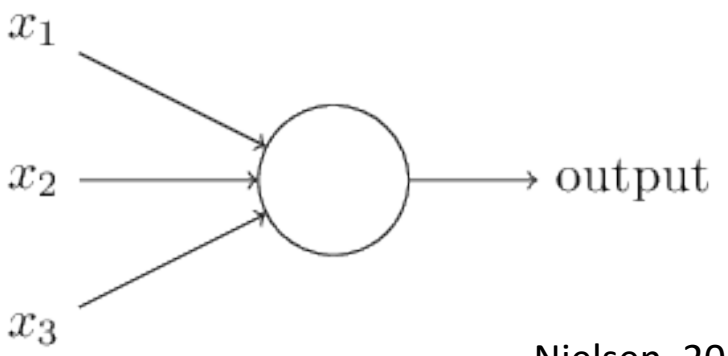
The perceptron

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



Nielsen, 2015

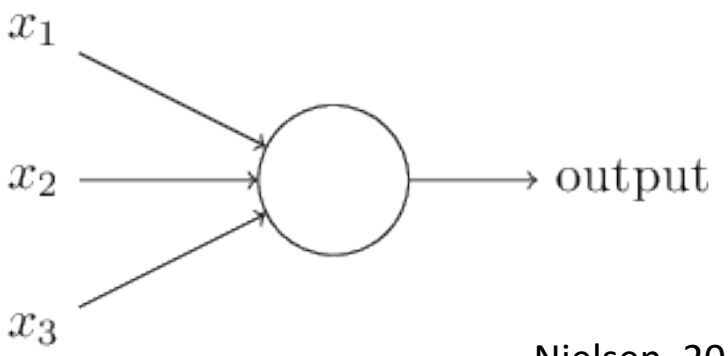
The perceptron



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

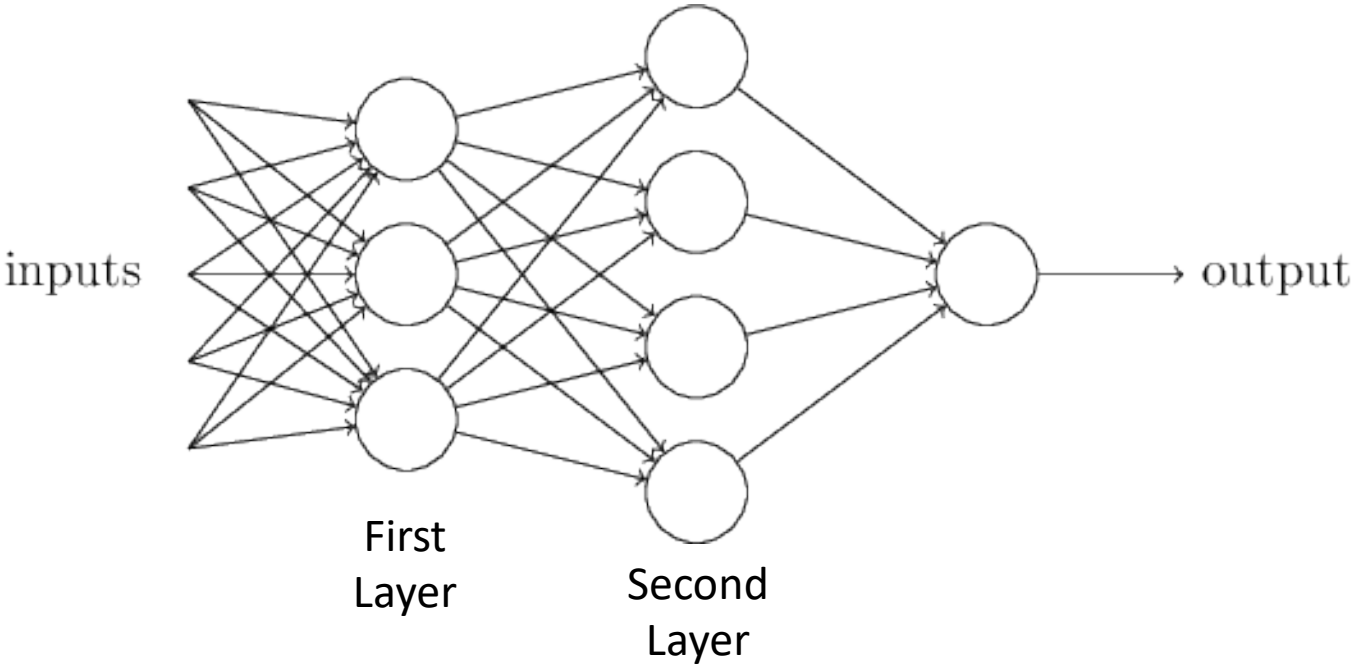
$$x_j = \begin{cases} 0, \text{if no} \\ 1, \text{if yes} \end{cases}$$

- 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 ($x_3 = 1$)

The multilayer perceptron (MLP)

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



Nielsen, 2015

Bias

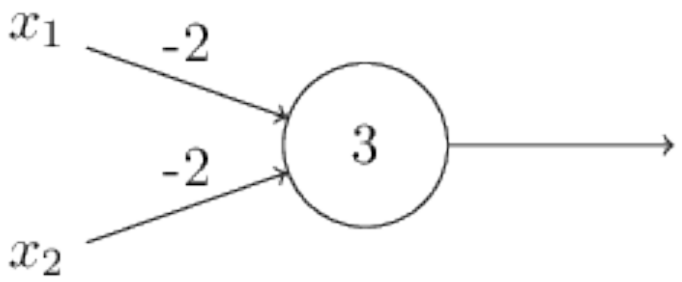
- $w \cdot x = \sum_j w_j x_j$
  - w and x are the weight and input vectors, respectively
- Replace the threshold with perceptron *bias*
  - Bias $b = -$threshold

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

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

Logic circuits with perceptrons

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

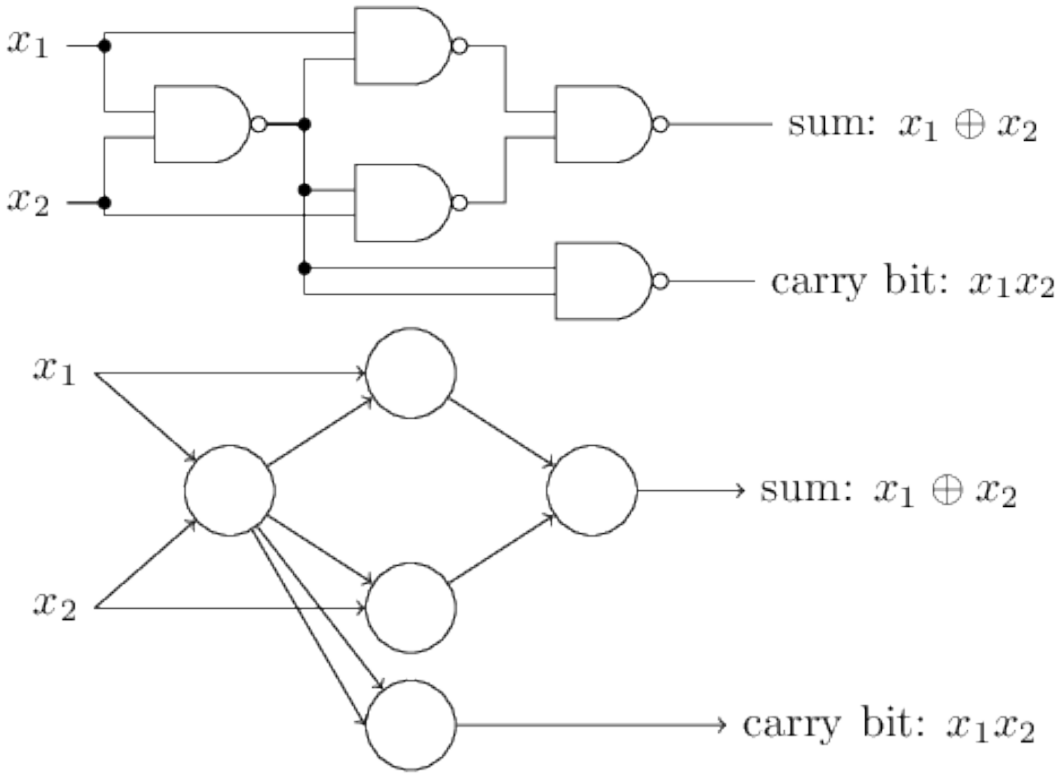


Nielsen, 2015

- 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

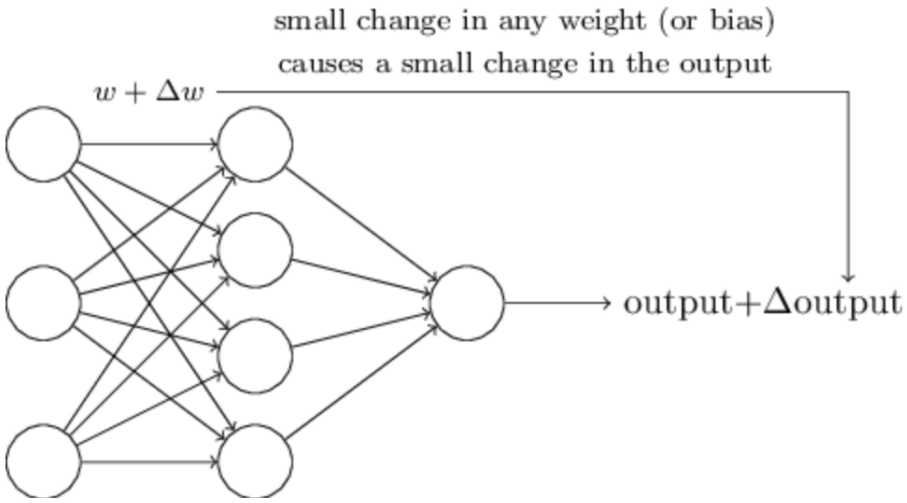
- NAND gates are universal for computation
  - Any computation can be built from NAND gates
  - Therefore, perceptrons are universal for computation
- Bitwise addition:



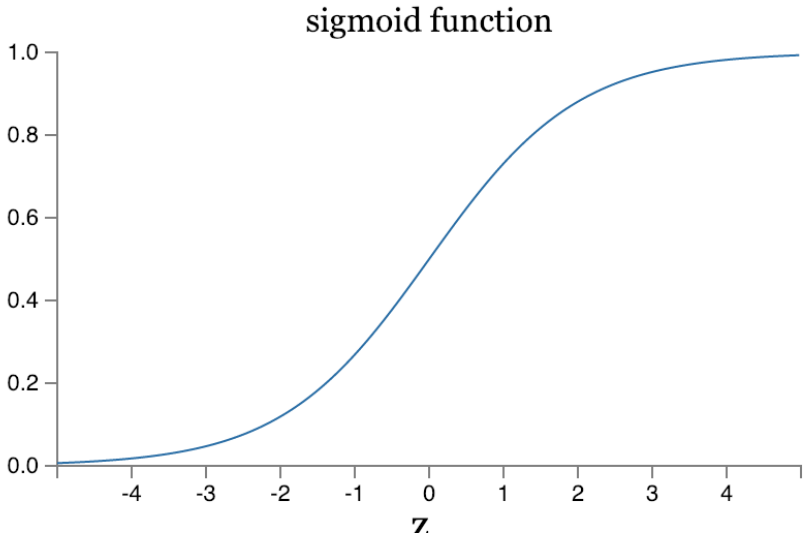
Nielsen, 2015

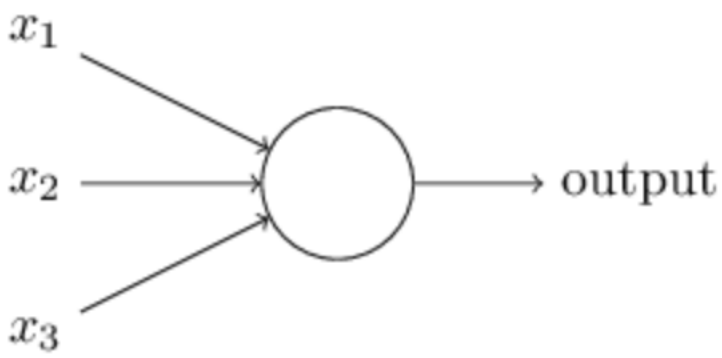
Problem with Logical Functions

- 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



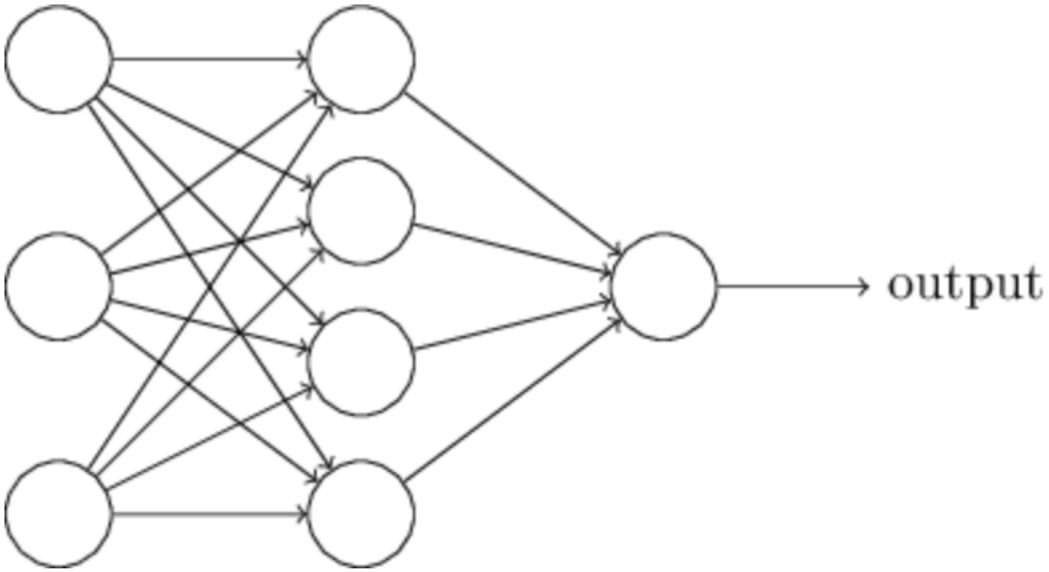
Sigmoidal Neuron





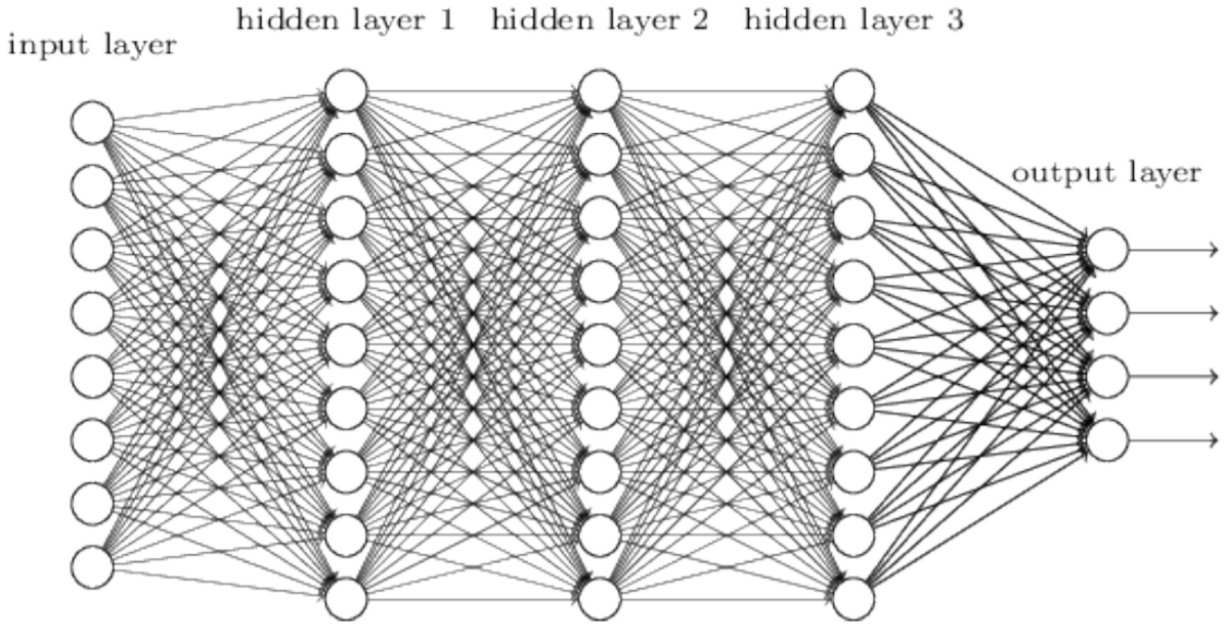
$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

Multi-layers of Neurons



Differentiable Computing

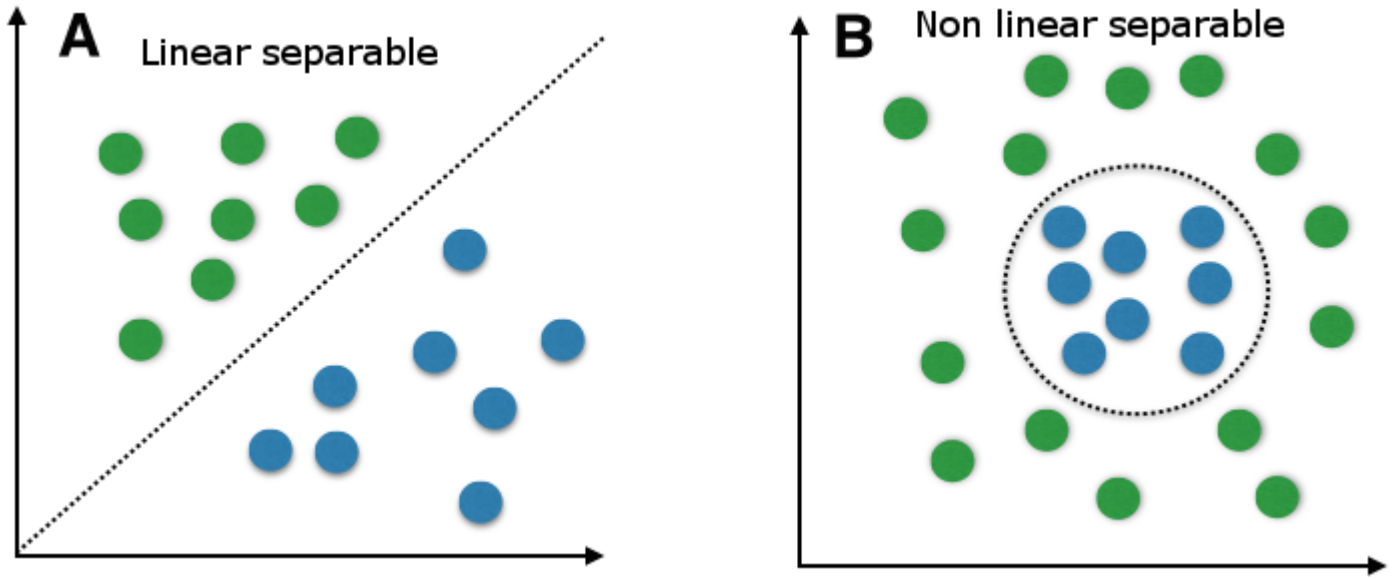
- We can create *learning algorithms* that automatically tune the weights and biases with *Gradient Descent*
  - Tuning occurs in response to external stimuli and w/o direct intervention
  - Creates a circuit designed for the problem at hand



Michael Nielsen

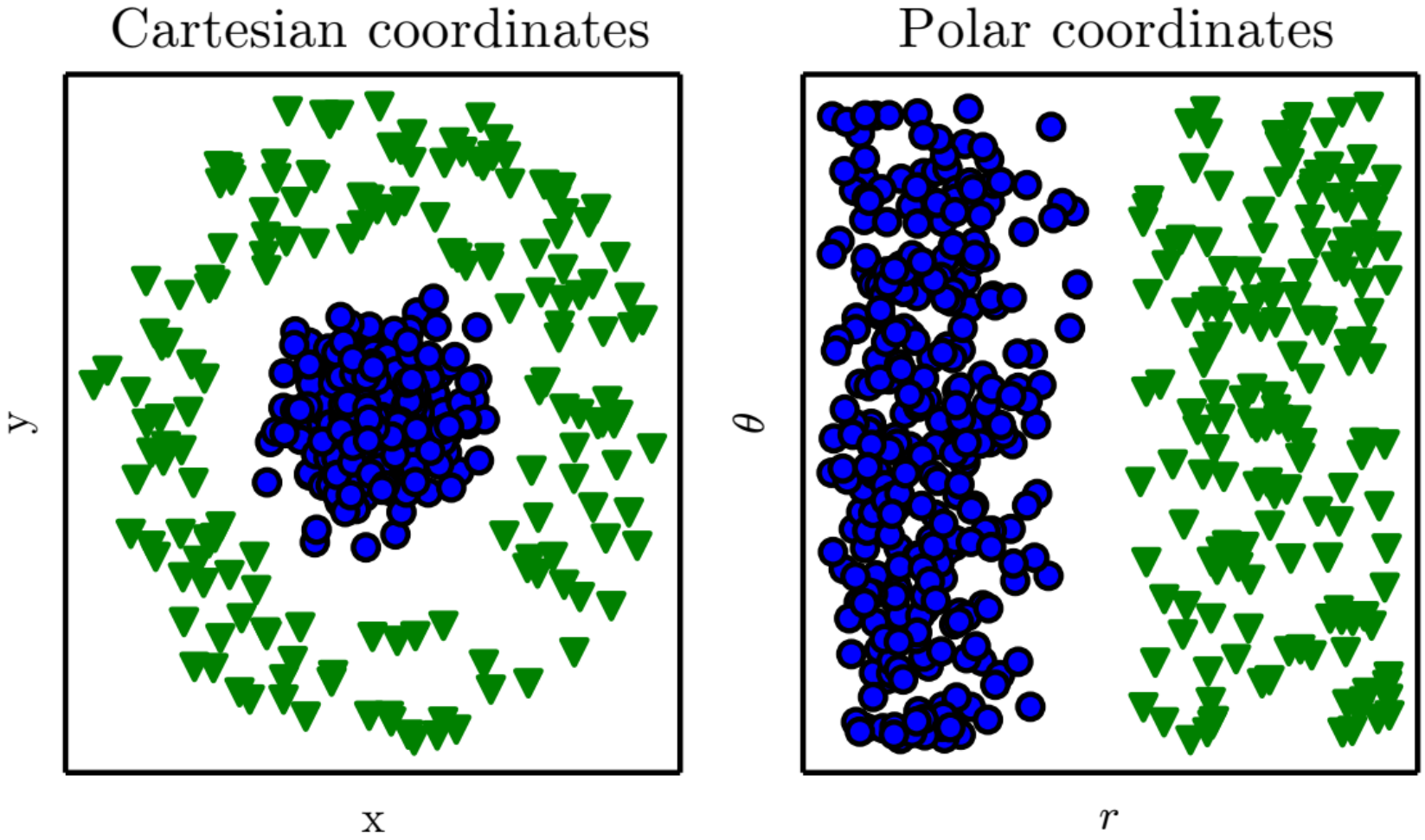
Why go deep?

Depth = complexity



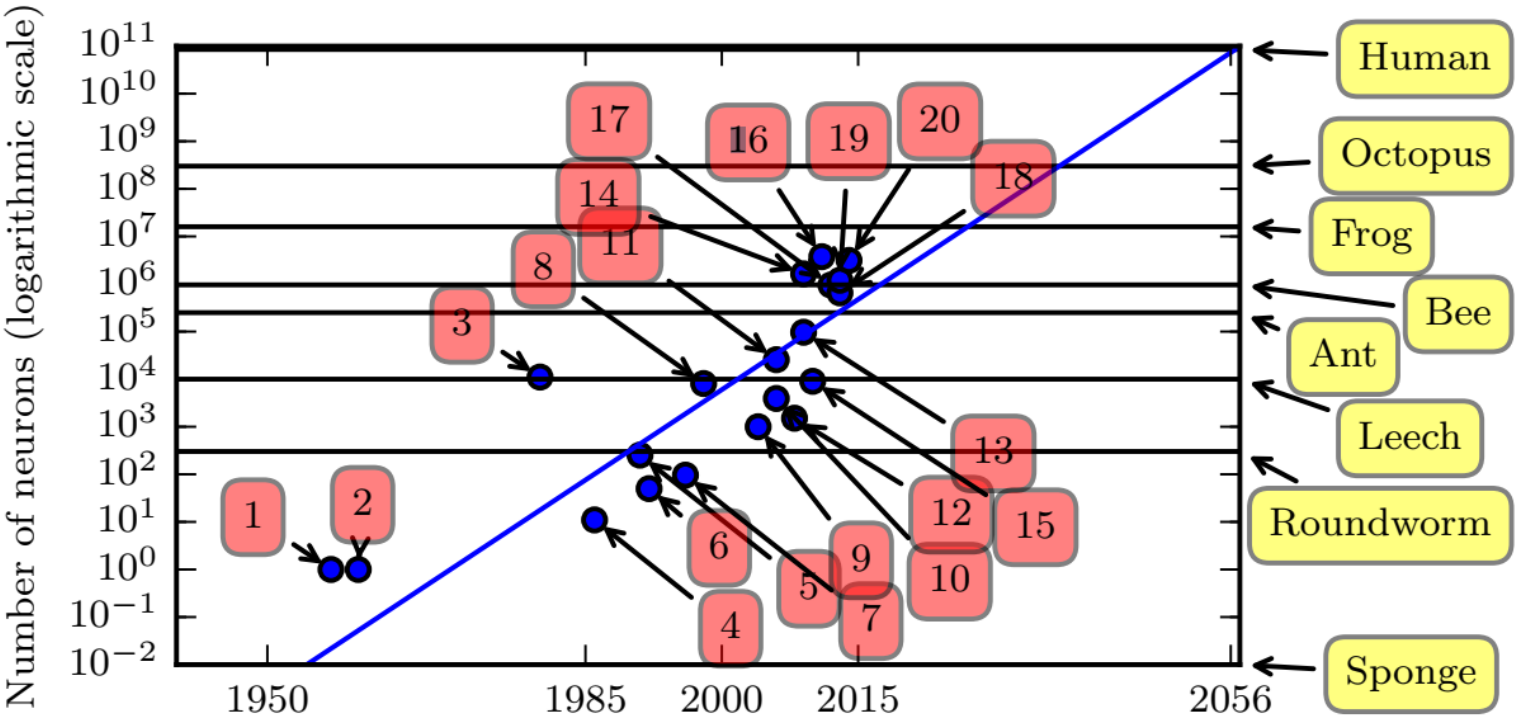
Single layer perceptrons can only handle linear separability

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)

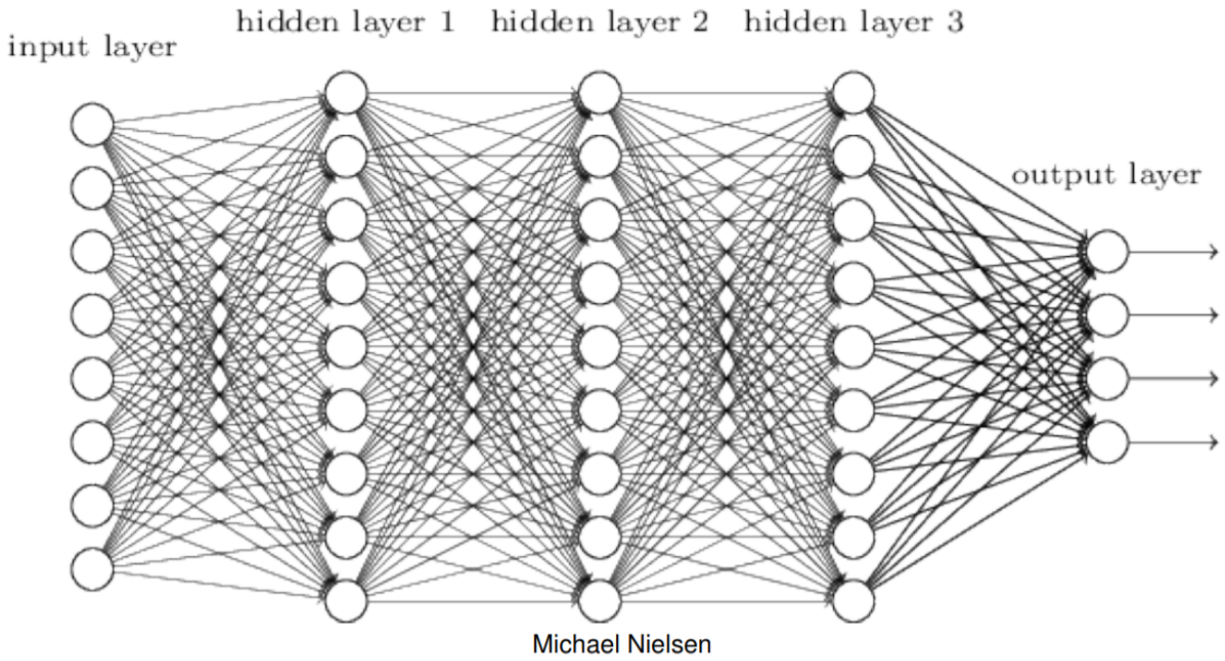
Goodfellow et al., 2016

Design choices for an ANN

- Learning algorithms
  - Backpropagation
  - Stochastic gradient descent (SGD)
- 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

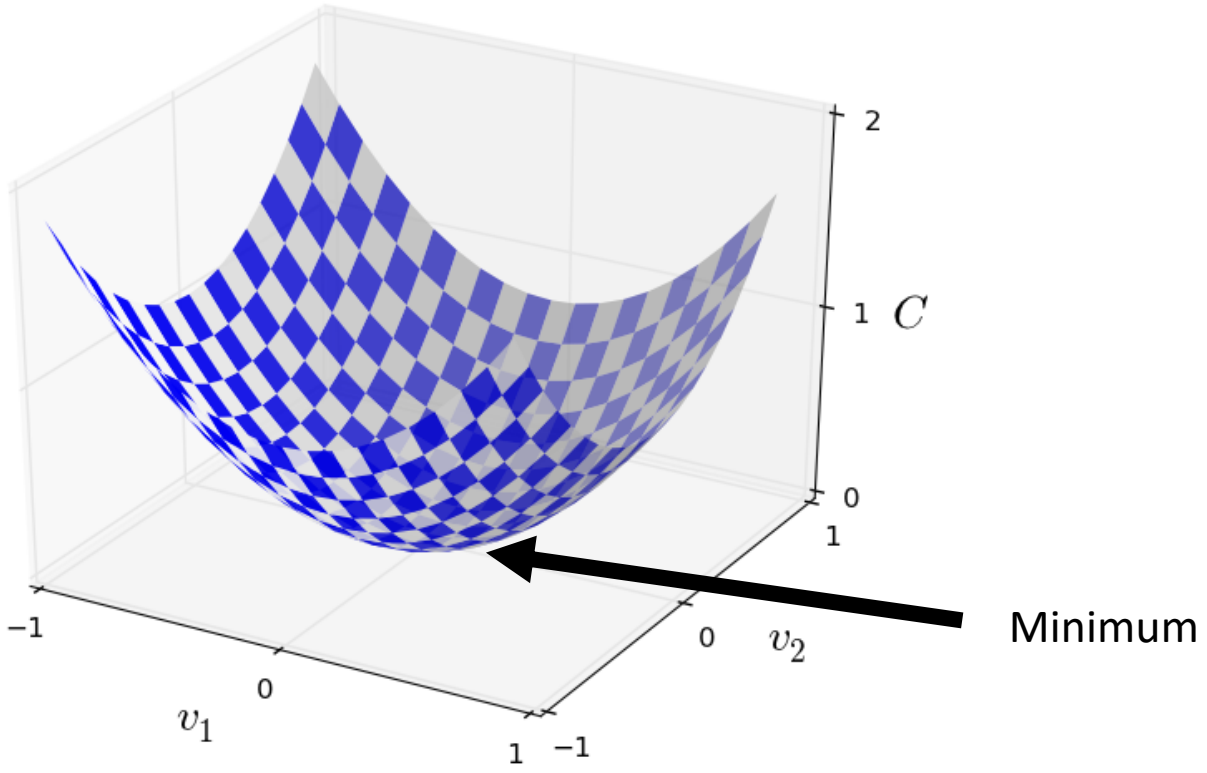
Model Parameter Optimization

- A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E
- Usually in machine learning you pick a model f with parameters w
- These problems **Optimize** w such that the training data is best fit, this fit is measured using some loss function L
  - **Some types of optimization can be solved analytically!**

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:

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

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

Regularization

- Ridge regression: penalize with L2 norm

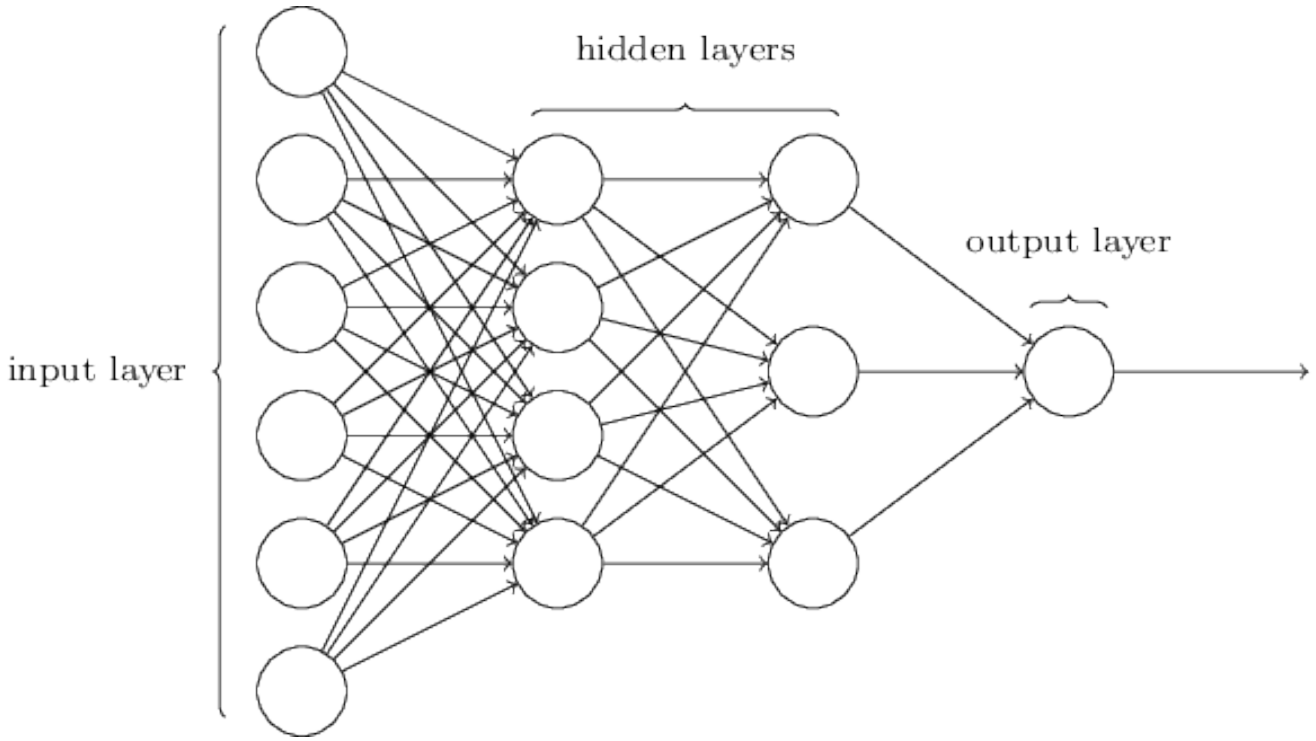
$$\boldsymbol{w}^* = \arg\min \sum_i C(f(\boldsymbol{x}_i; \boldsymbol{w}), y_i) + \lambda \sum_{j=1}^{m} w_j^2$$

  - 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)

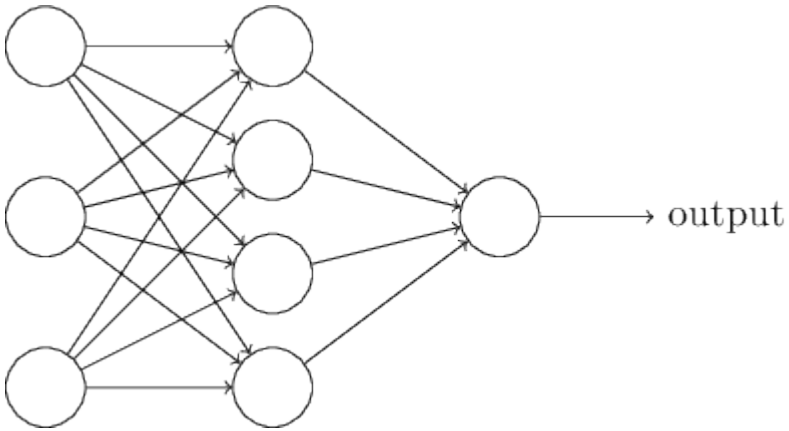
Classification/Regression Network



- 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 Δw_j, change in bias Δb

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

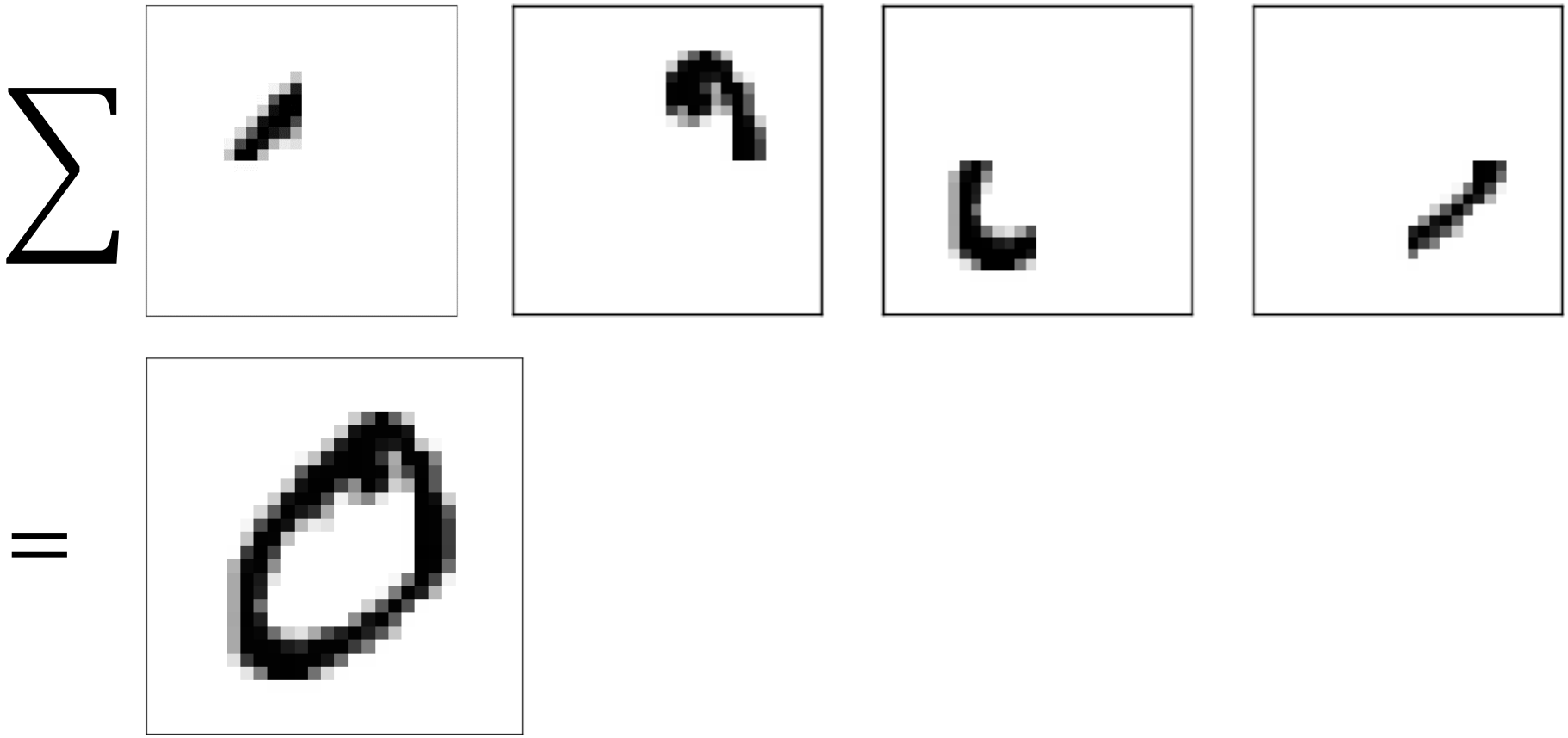
- Δoutput is a *linear function* of the changes Δw_j and Δb in the weights and bias
  - Linearity makes it easier to choose small changes in weights and biases to achieve the desired small change in output
- Thus sigmoid neurons have similar behavior as perceptrons but are easier to use

Handwritten digit recognition

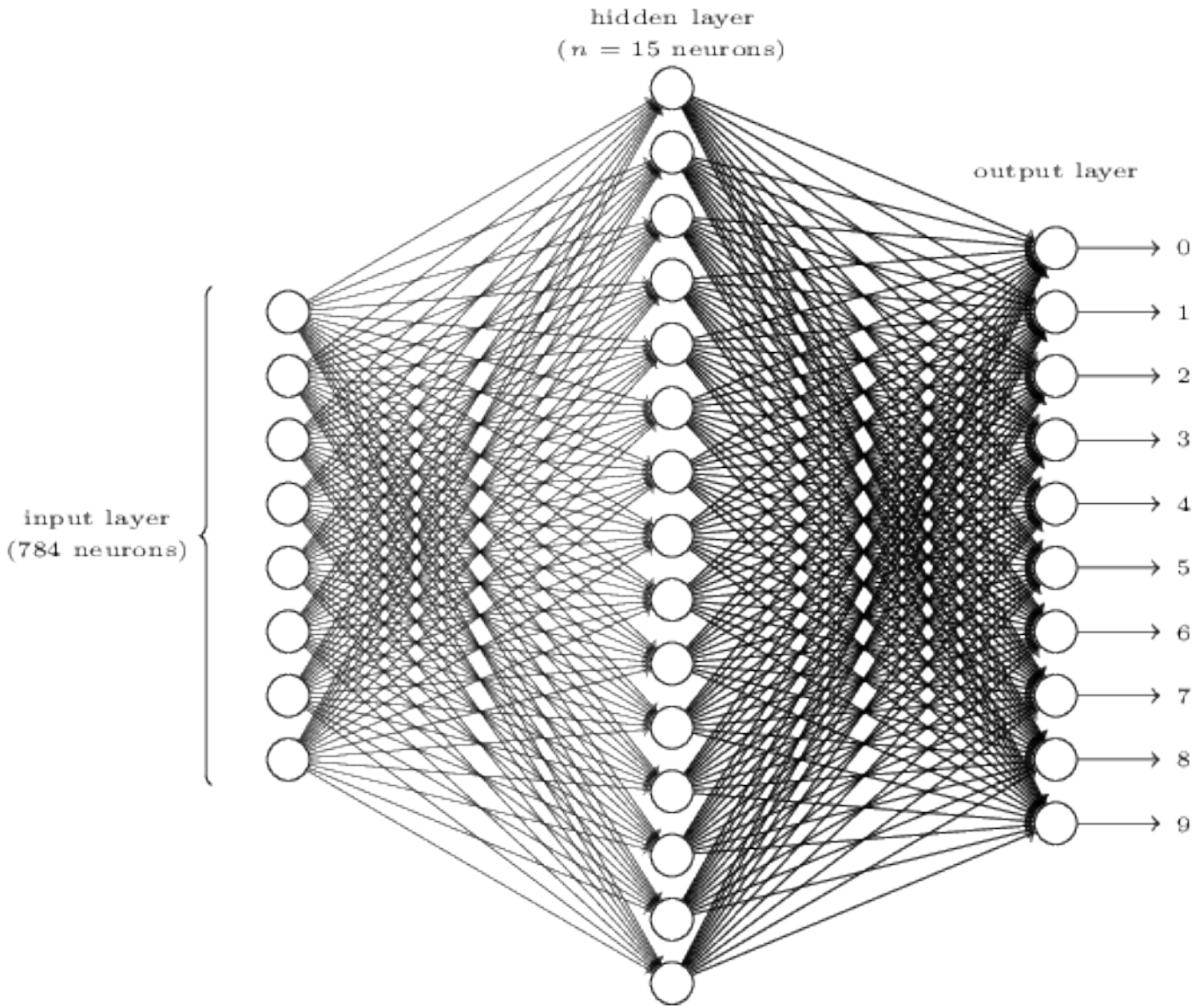
How would you write an algorithm for this?

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 kth neuron in the second layer and the jth 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 & Backprop

- Gradient descent refers to taking a step in the direction of the *gradient (partial derivative)* of a weight or bias with respect to the cost function
- Gradients are propagated backwards through the network in a process known as *backpropagation*
- The size of the step taken in the direction of the gradient is called the *learning rate*

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 $\eta = 100.0$

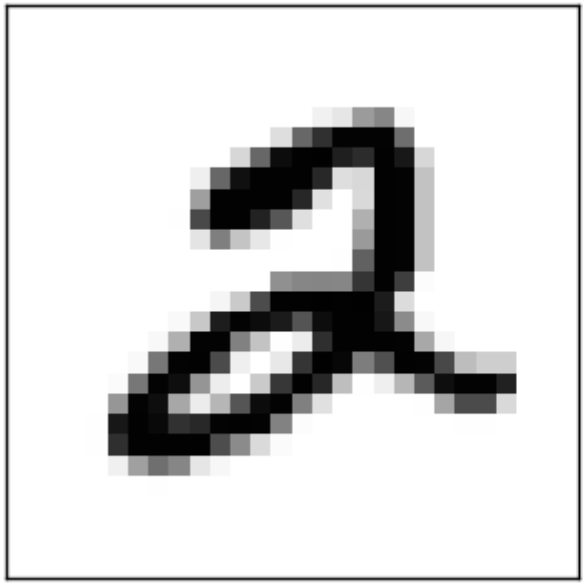
```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

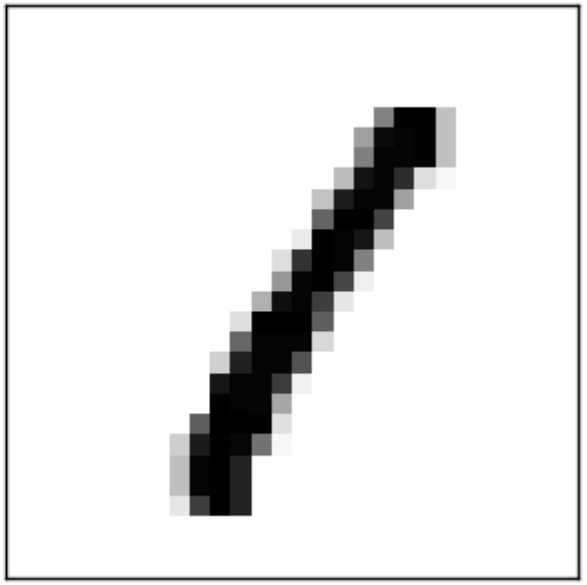
Debugging a neural network

- 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?

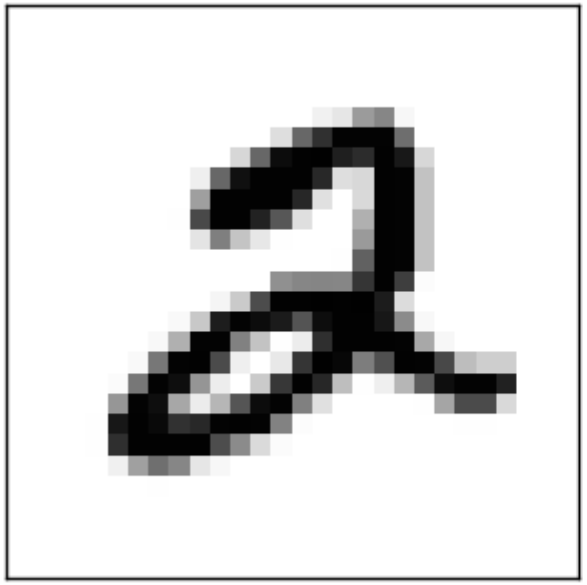
- Need to compare to some baselines
- Random guessing: 10% accuracy
  - Our network does much better
- Simple idea: How dark is the image?
  - E.g. a 2 will typically be darker than a 1

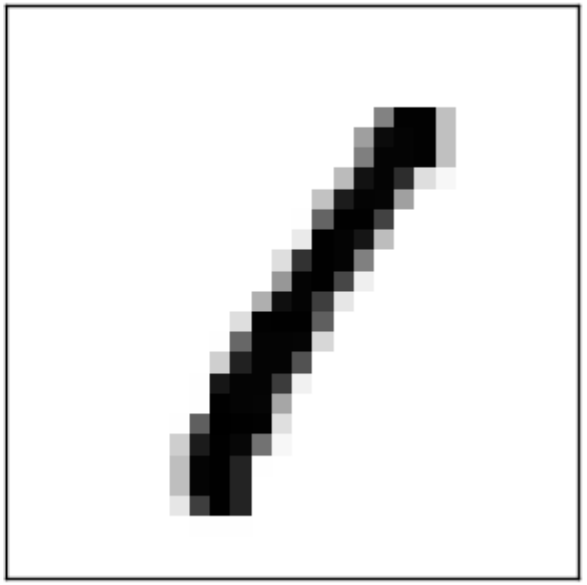




Average darkness

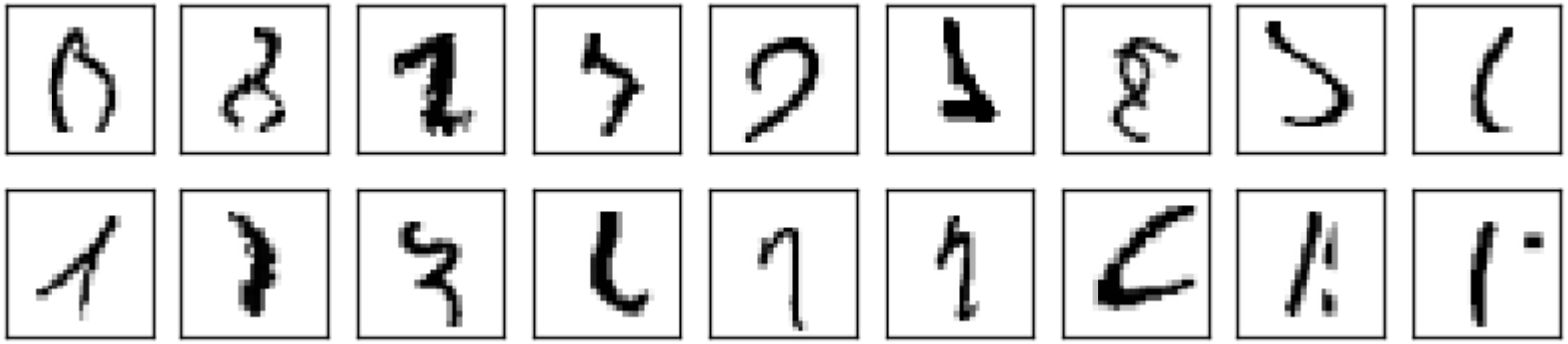
- Compute average darkness for each digit from the training data
- Classify an image based on the digit with the closest average darkness
- This gives 22.25% accuracy
  - Better than random guessing





Support Vector Machine (SVM)

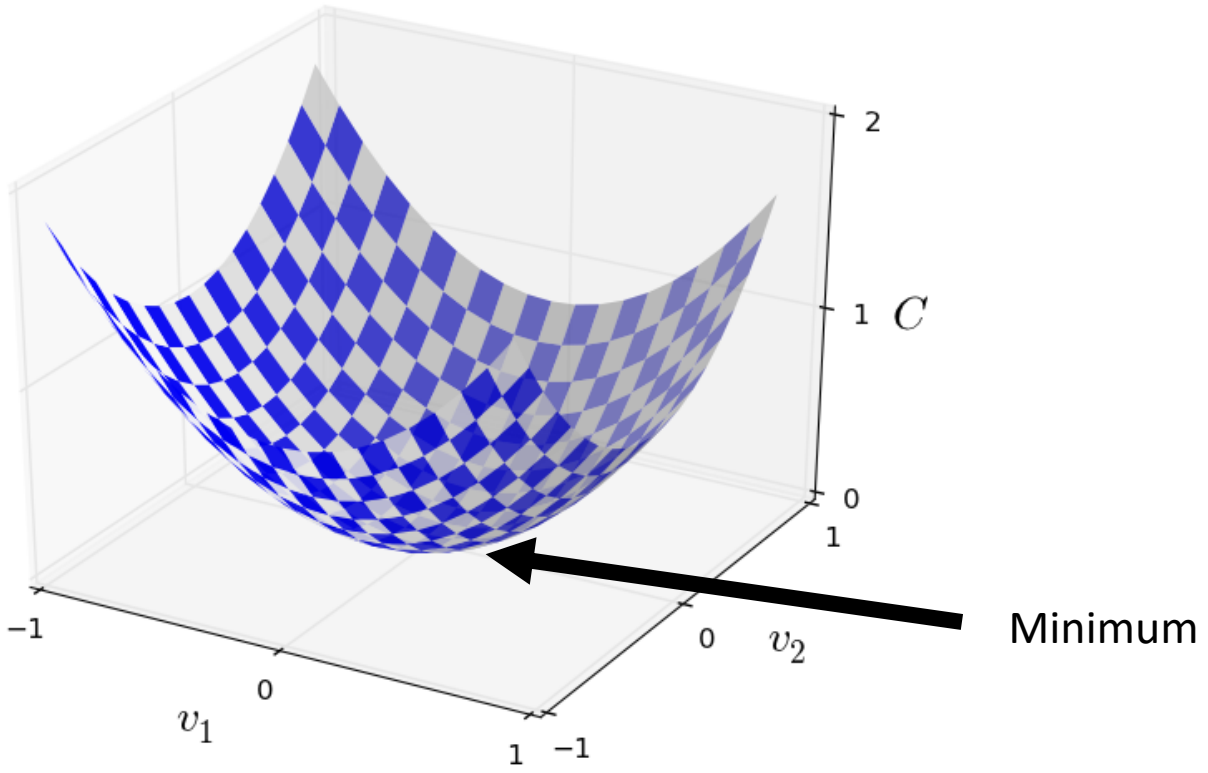
- Using default settings in SVM scikit-learn (a Python library) gives 94.35% accuracy
  - It's possible to optimize tuning parameters to achieve 98.5% accuracy
- Can neural networks do better?
- Yes!
  - Record set in 2013 was 99.79% accuracy (only 21 wrong in the test data)
  - Can you do better?



Innards of Neural Networks

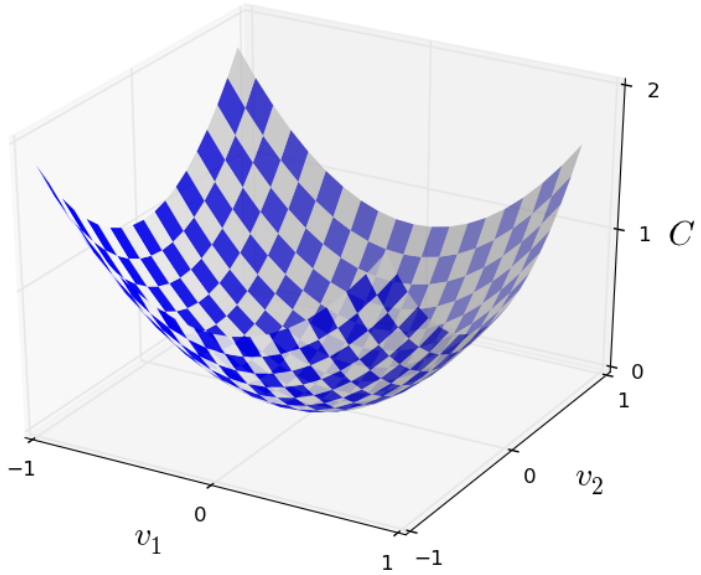
Gradient descent

- How do we minimize a cost function in general?
- Suppose we want to minimize some function $C(v)$
  - $v = v_1, v_2, \ldots$



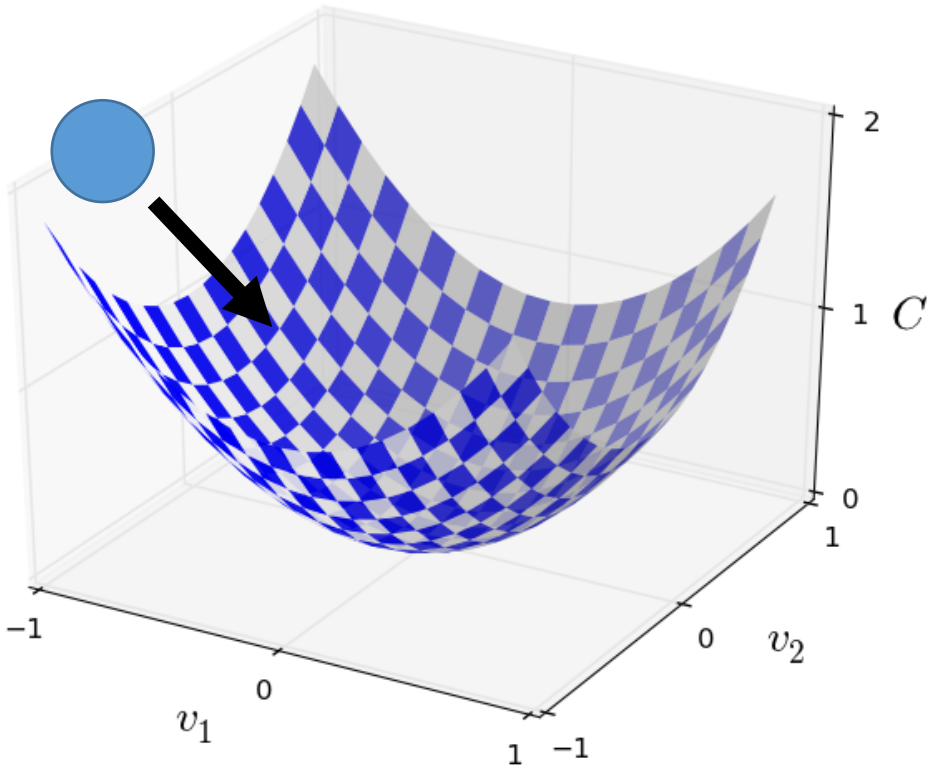
Gradient descent

- What if we have more variables?
- Could try calculus to find extremum of C
  - Difficult when we have lots of variables
  - Largest neural networks have billions of weights and biases



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

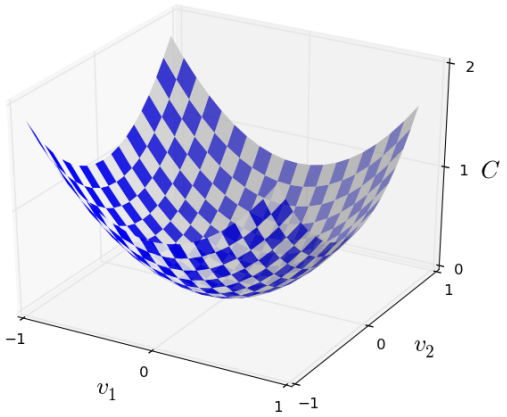


The gradient

- Choose a random starting point
- Move small amounts Δv_1 in the v_1 direction and Δv_2 in the v_2 direction

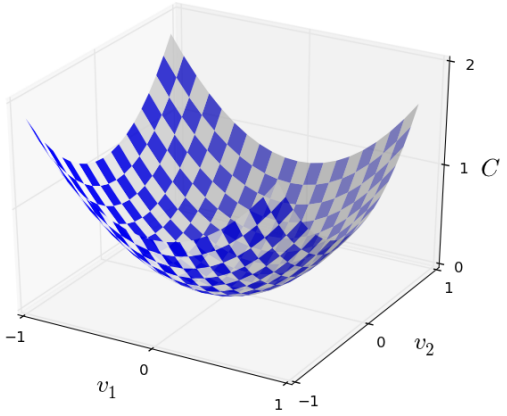
$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- Goal is to minimize C
  - Choose Δv_1 and $\Delta \mathrm{v}_2$ so that ΔC is negati
  - I.e. move in a direction that decreases C
- Define $\Delta v = (\Delta v_1, \Delta v_2)^T$
- The ***gradient*** of C is $\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- $\Delta C \approx \nabla C \cdot \Delta v$



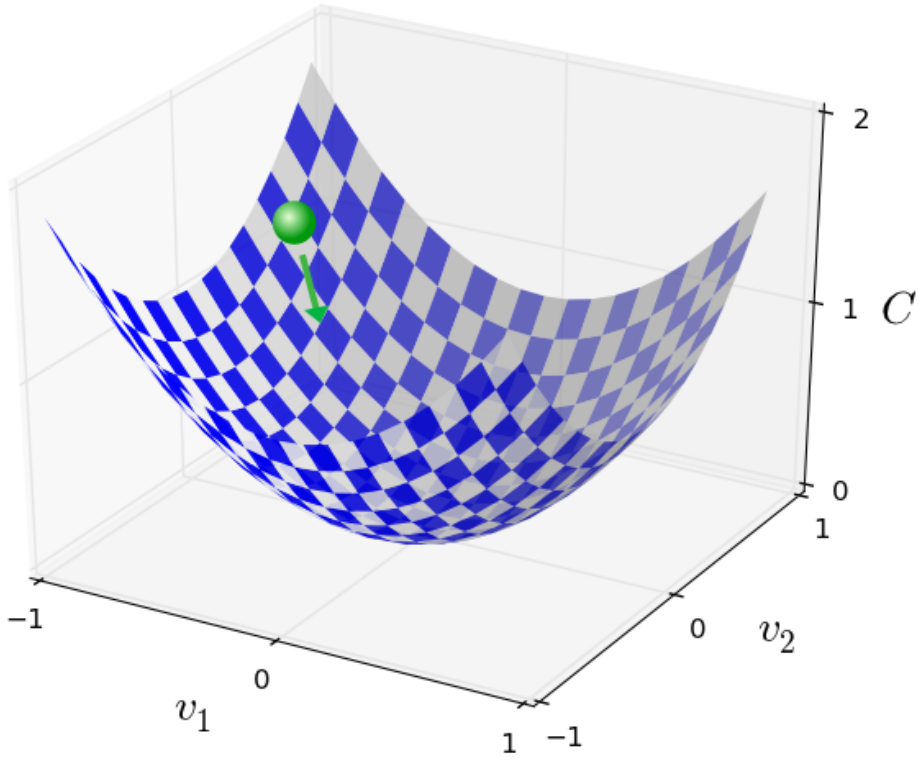
The gradient

- Define $\Delta v = (\Delta v_1, \Delta v_2)^T$
- The ***gradient*** of C is $\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- $\Delta C \approx \nabla C \cdot \Delta v$
- How should we choose Δv to ensure ΔC is negative?
- Choose $\Delta v = -\eta \nabla C$
  - η is a small, positive parameter (known as the ***learning rate***)
  - $\Rightarrow \Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$
  - Since $\|\nabla C\|^2 \geq 0$, $\Delta C \leq 0$
  - Thus C will always decrease and never increase
- Thus, choose new position $v \rightarrow v - \eta \nabla C$
- Repeat until we reach the minimum



Summary of gradient descent

- Compute the gradient ∇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
- $f(g(x))=f'(g(x))g'(x)$
- So if output node is $f(W_o,Y)$ and we "open out Y" into $g(W_1,X)$ where X is the input of the previous layer then we have to chain the derivatives as $f'(W_o,g(W_1,X))\ g'(W_1,X)$
- Remember these derivatives are wrt to the weights of the network!
- Detailed derivations are in Nielsen book

Regularization

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

- Add a ***regularization term*** to the cost function:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + \left(1 - y_j\right)\ln\left(1 - a_j^L\right)\right] + \frac{\lambda}{2n}\sum_{w} w^2$$

- First term is the cross-entropy cost function
- Second term is the regularization term
  - Scaled by factor $\frac{\lambda}{2n}$, λ the ***regularization parameter***
- Can write regularized cost function as

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

  - C_0 is the unregularized cost

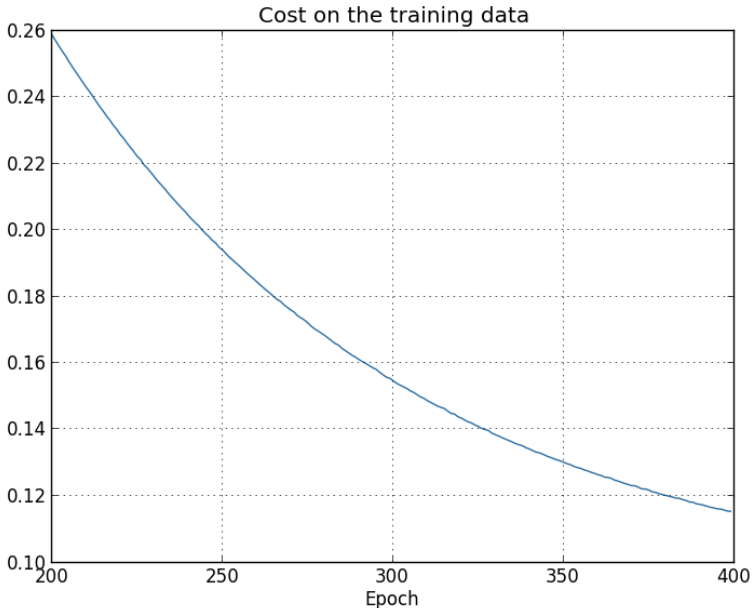
Weight decay/L2 regularization

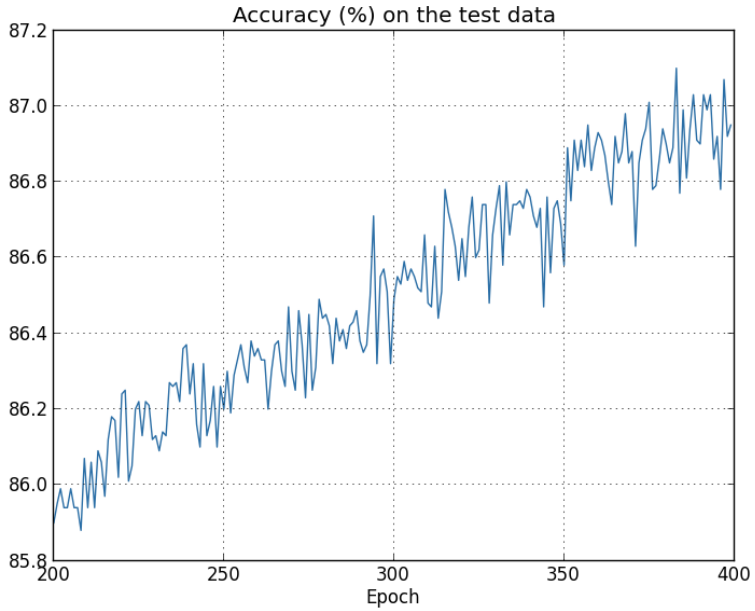
$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

- 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
  - λ 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

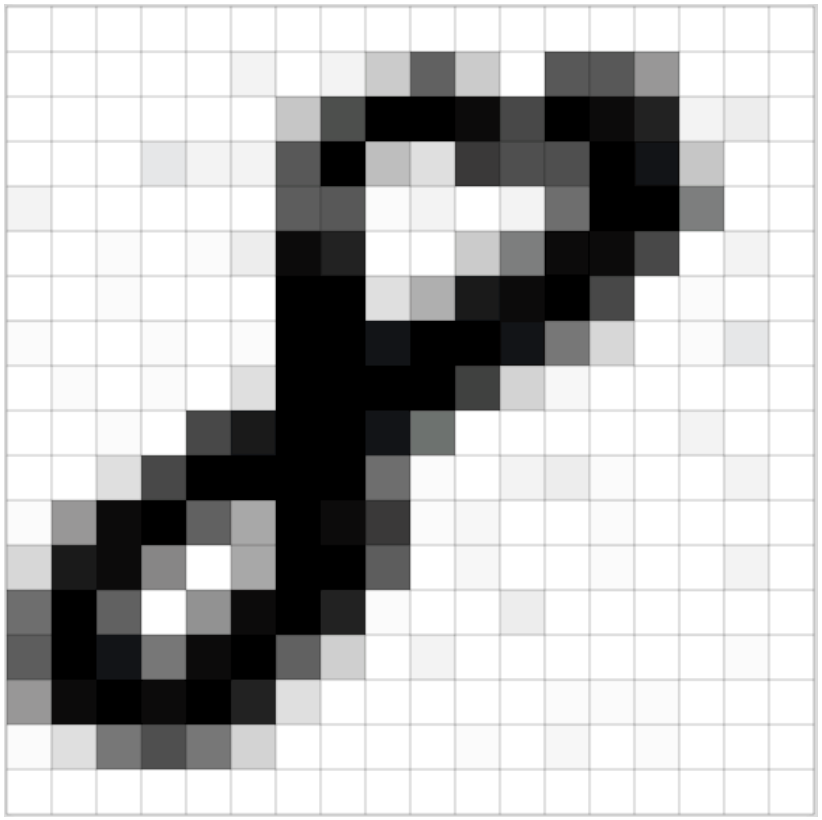
- 30 hidden neurons
- Train with first 1,000 training images
  - Cross-entropy cost
  - Learning rate $\eta = 0.5$
  - Mini-batch size 10
  - $\lambda = 0.1$
- Training cost decreases with each epoch
- Test accuracy continues to increase
  - More epochs would likely improve results further
  - Regularization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

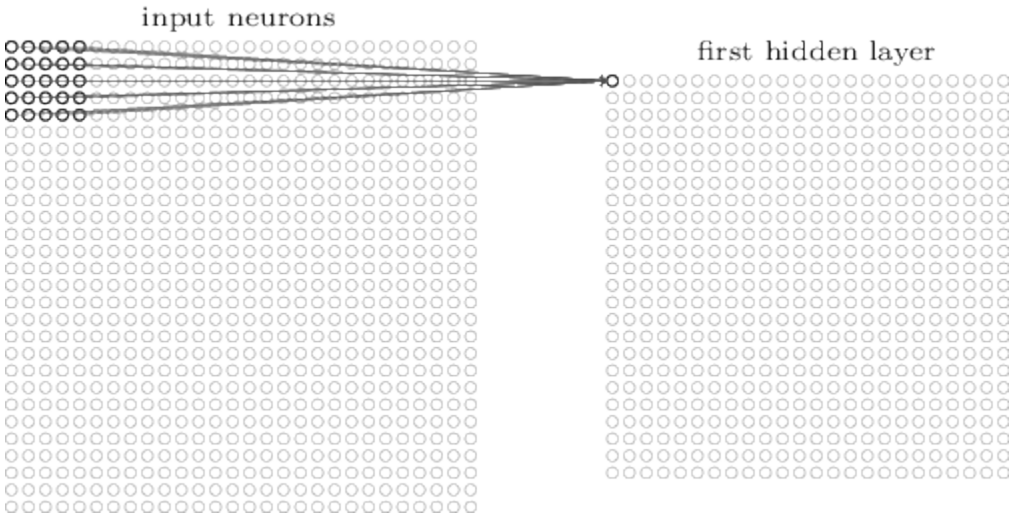


Image taken from Michael Neilsen's book "Neural Networks and Deep Learning"

Local receptive fields

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

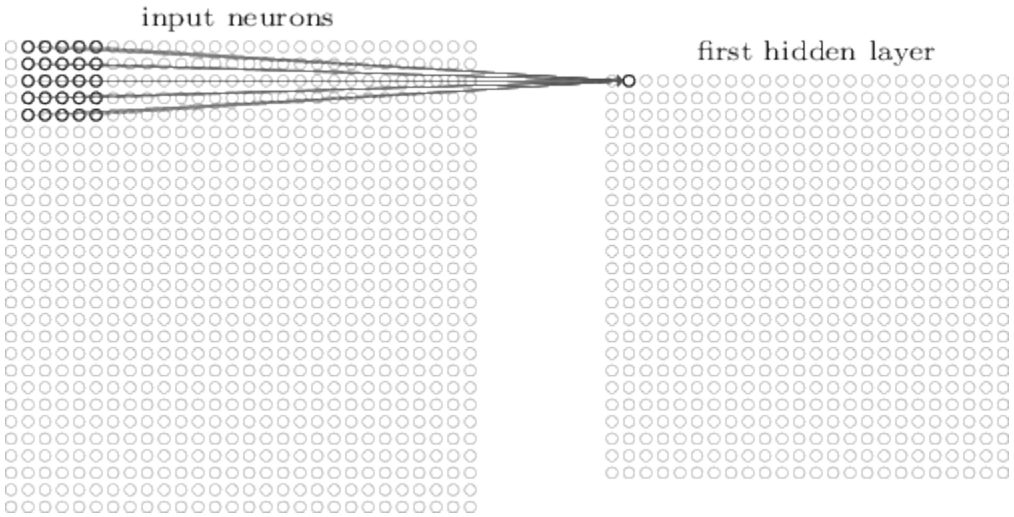
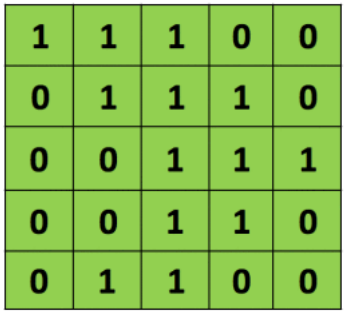
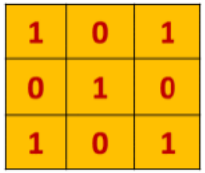


Image taken from Michael Neilsen's book "Neural Networks and Deep Learning"

The convolution operation

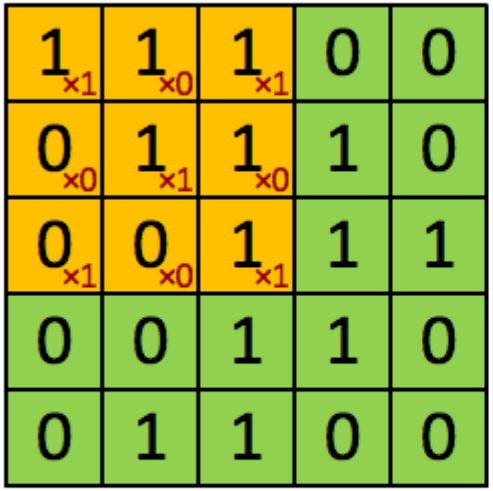


Image

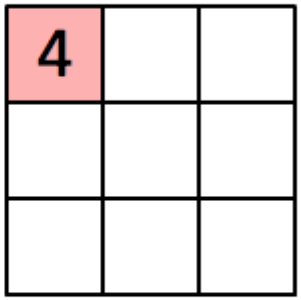


Filter/
Feature detector

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

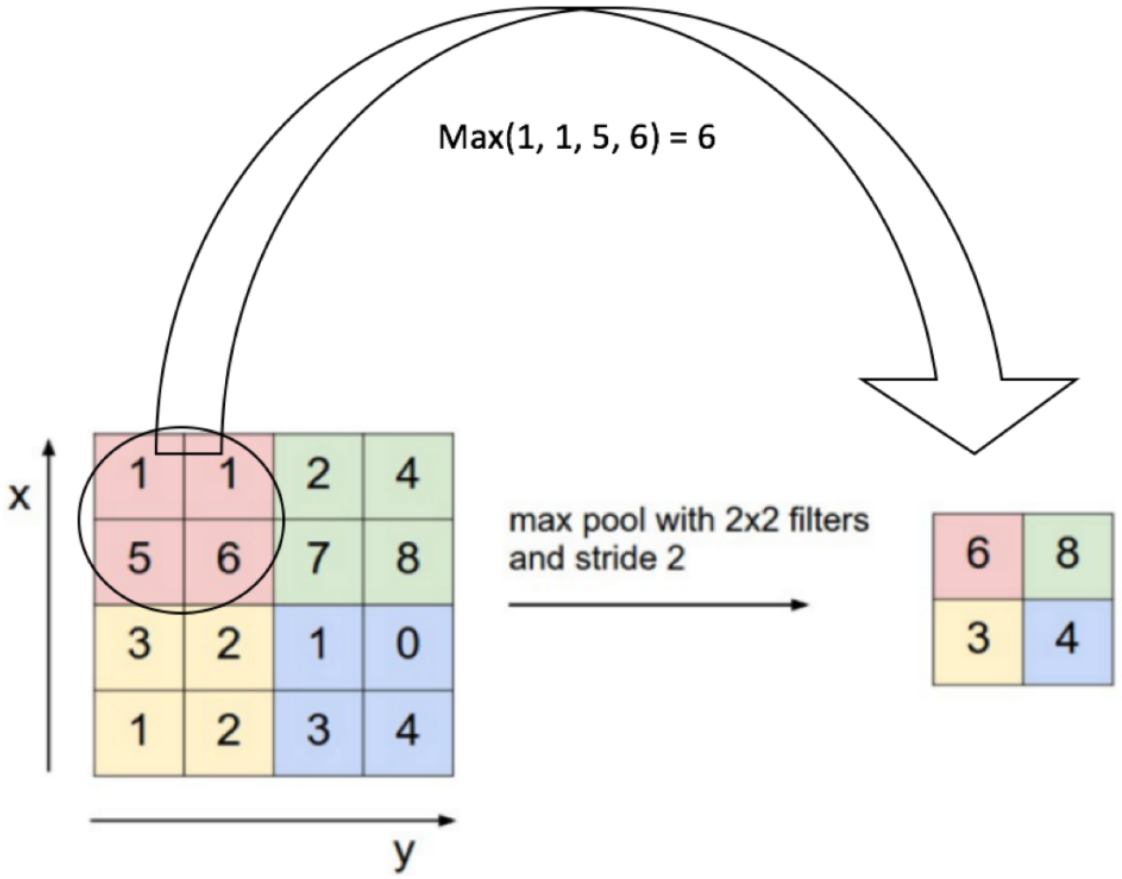


Image



Convolved
Feature

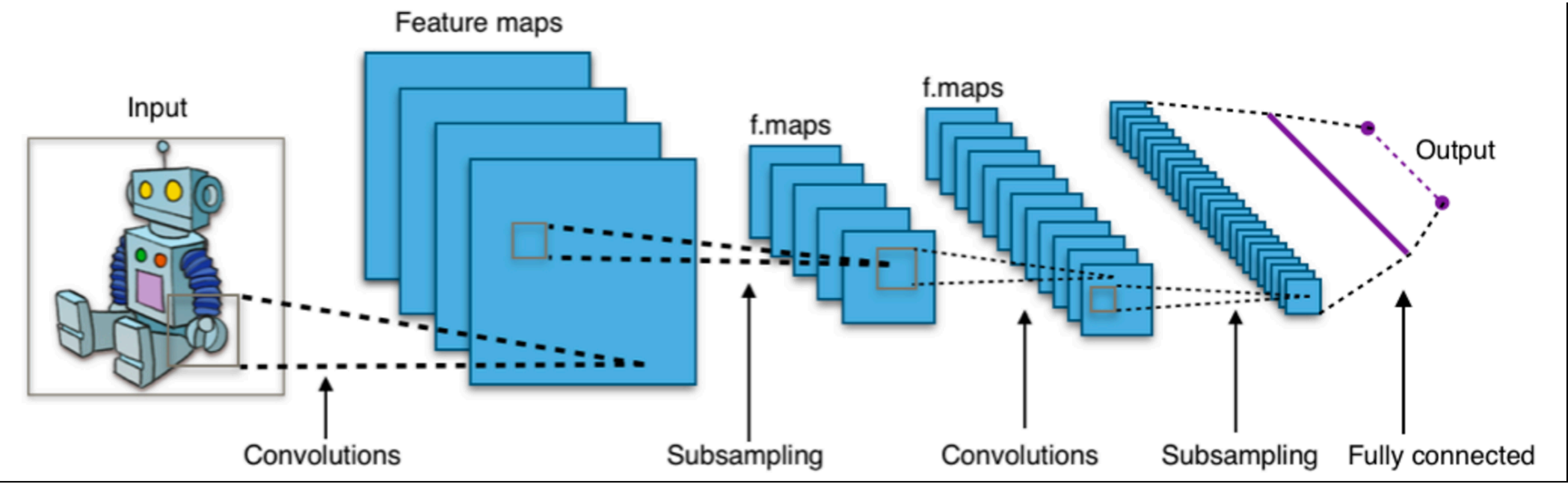
Pooling



Downsampling the results by pooling values by averaging or max

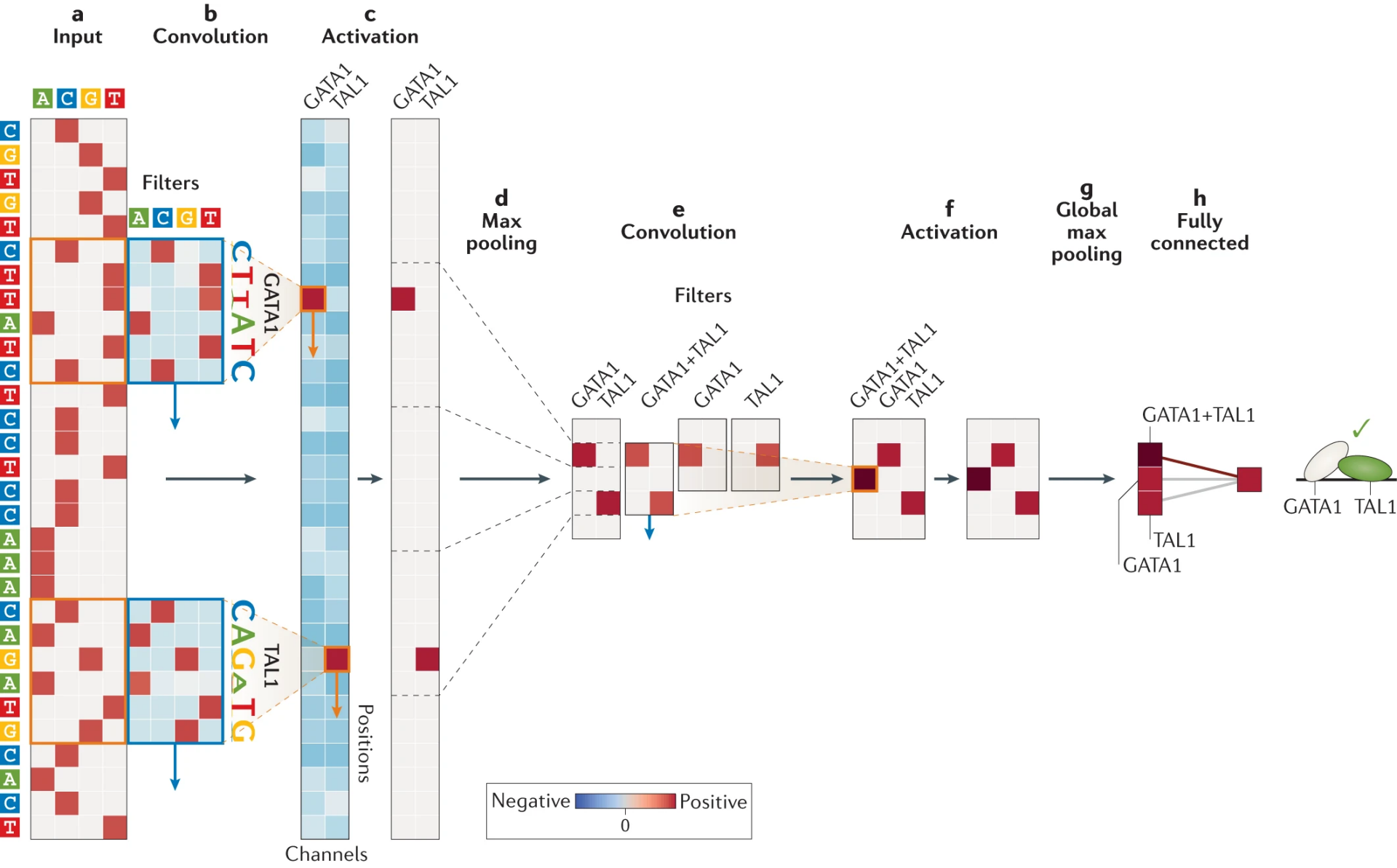
Max pool is used more commonly

CNN Architecture



Each convolutional unit has the same weight.

Binding Motif Prediction



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

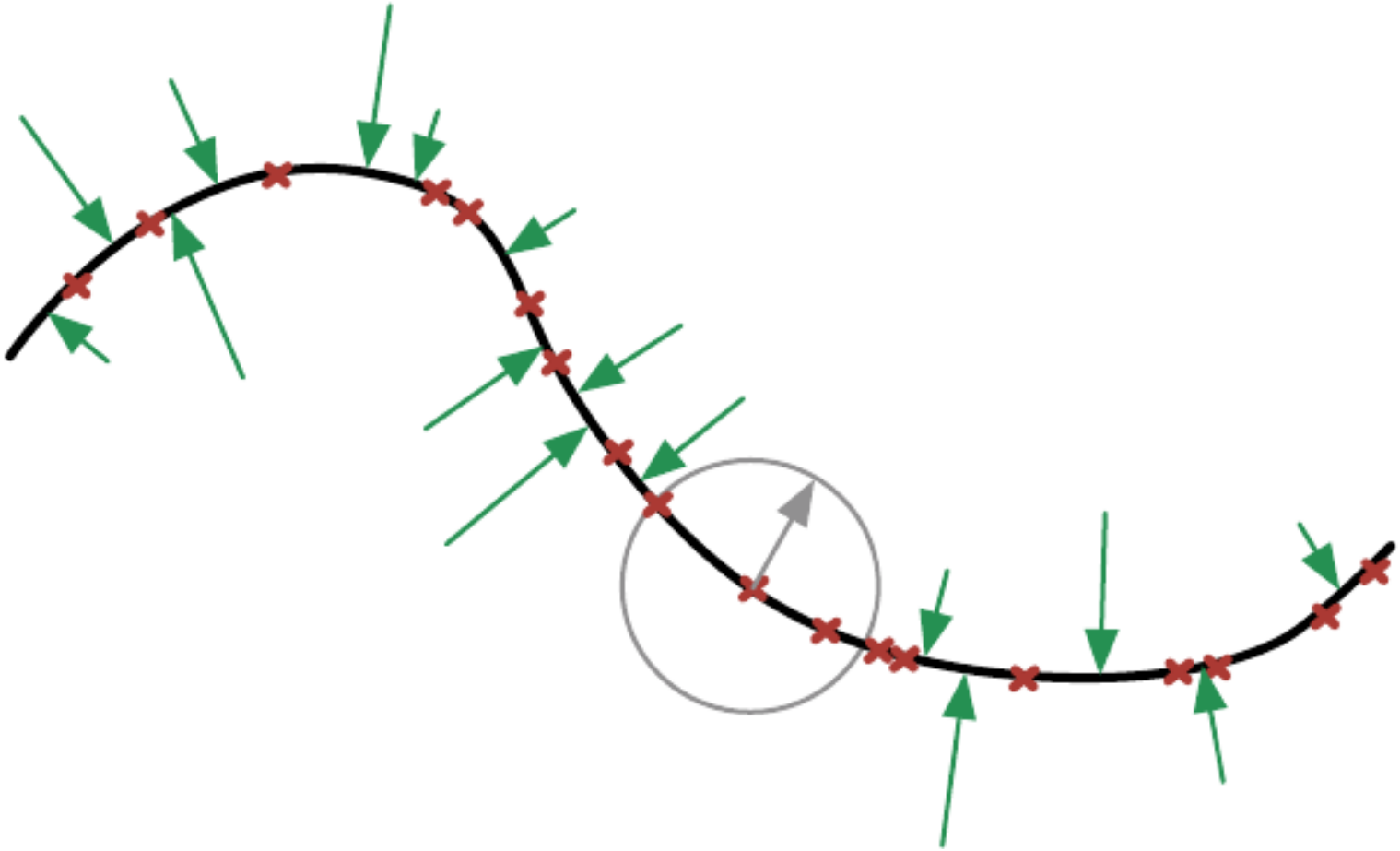
Representation learning

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

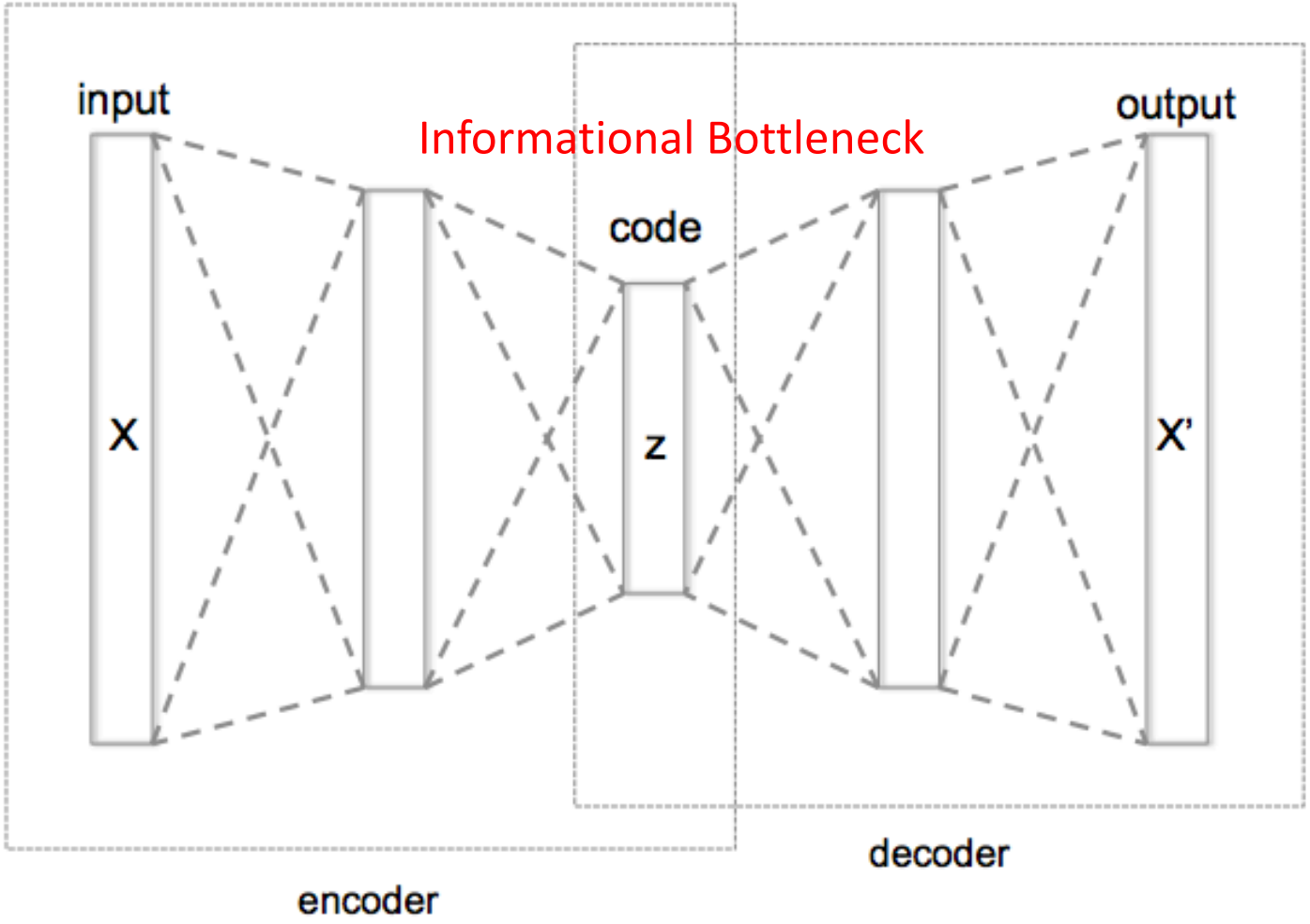


Example

- What is a good lower dimensional representation of this data?

- How can we find it?

Autoencoder



[Hinton, Salakhutdinov, Science 2006]

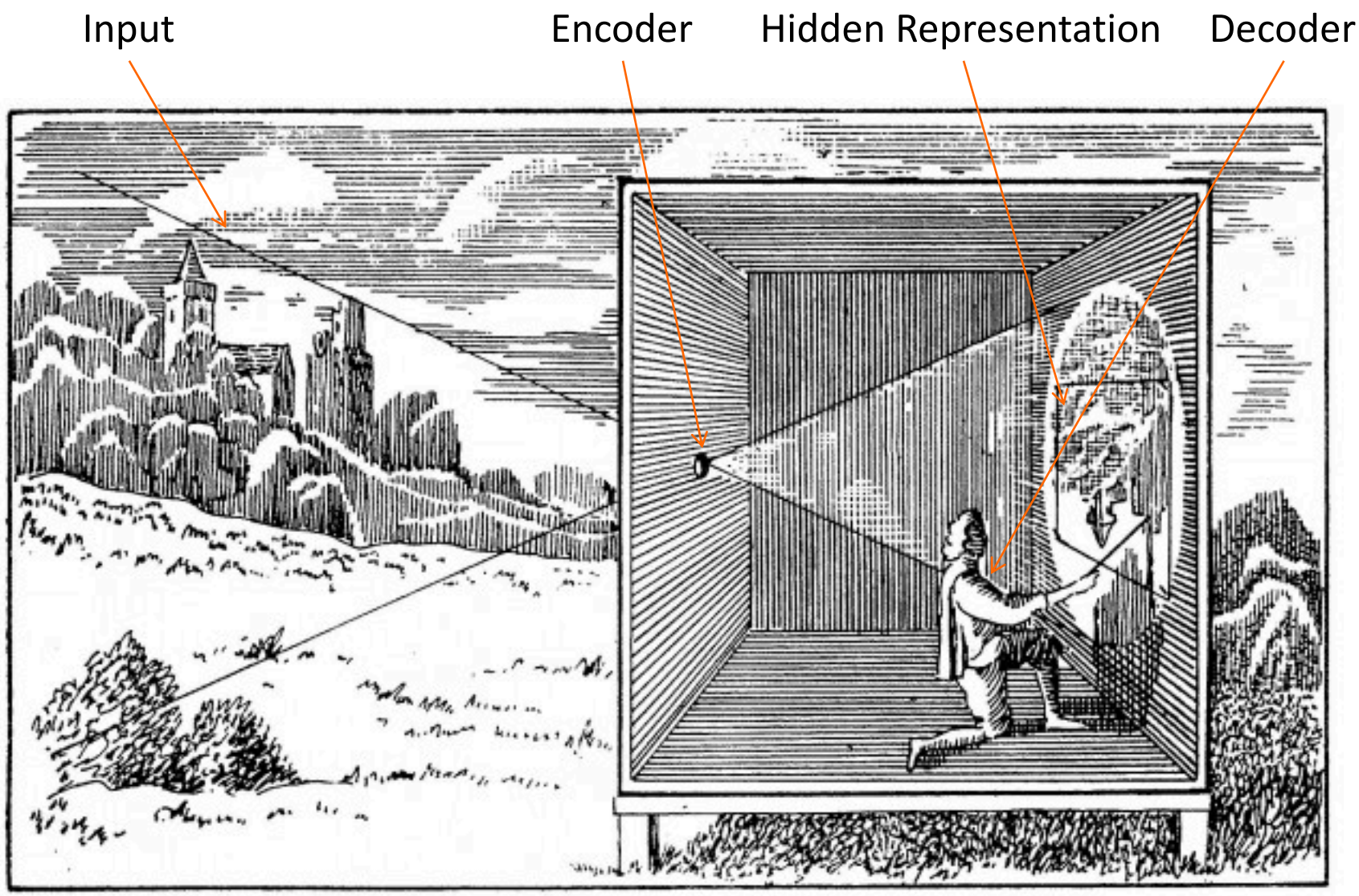
Autoencoder (AE)

- 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



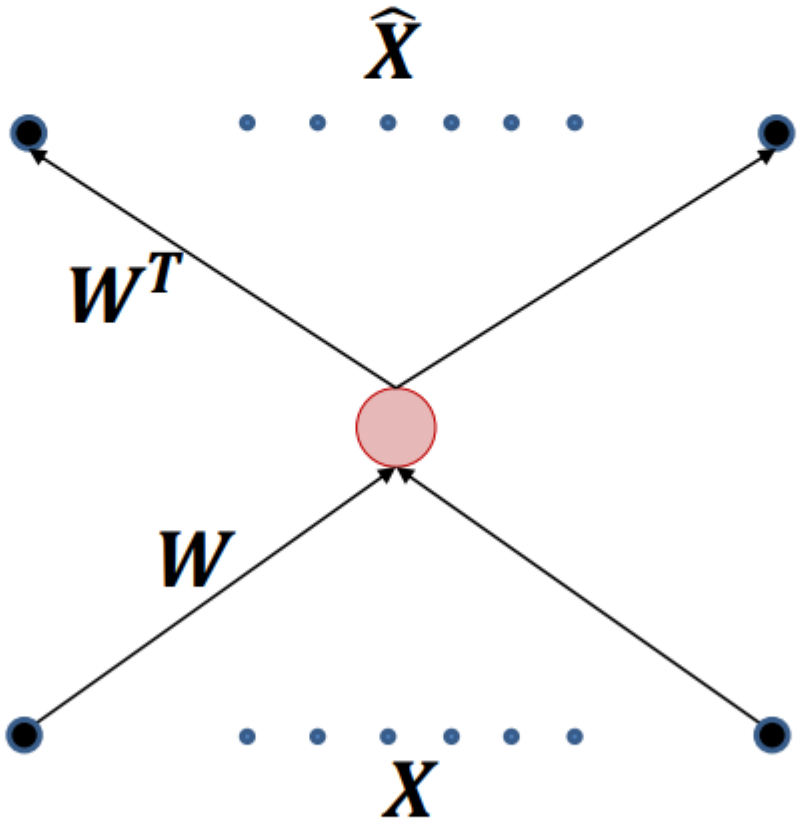
Camera Obscura, the renaissance autoencoder

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?



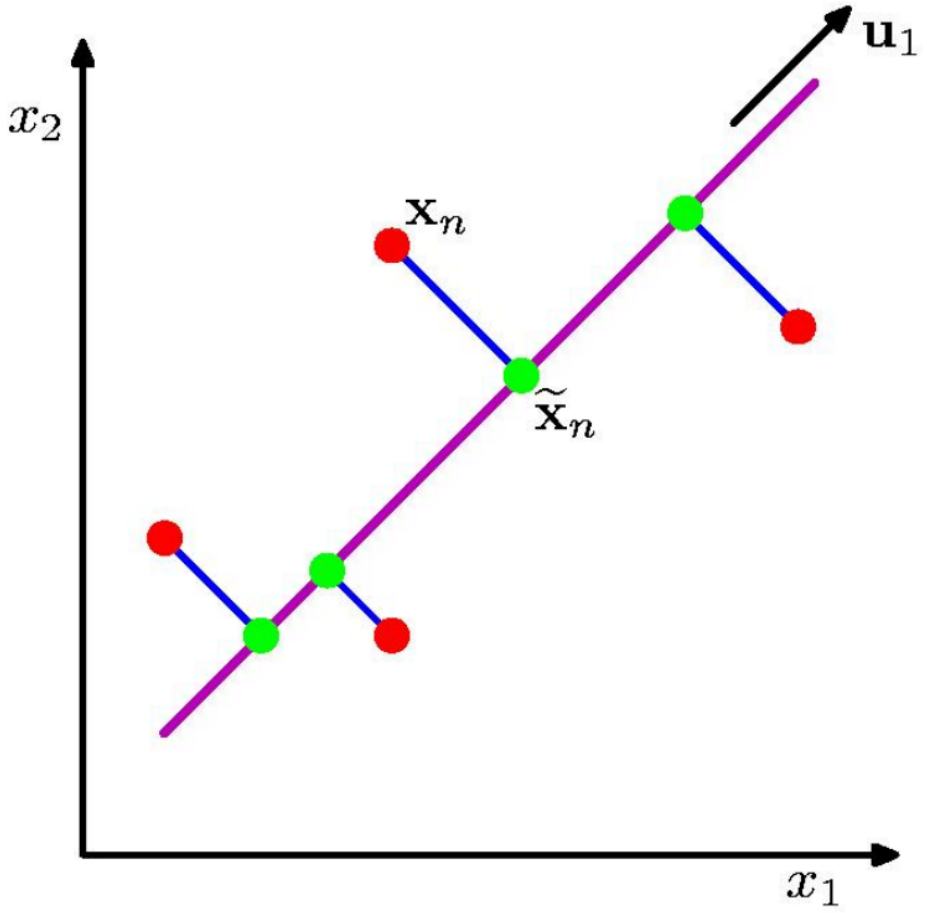
- Minimize reconstruction error:

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

- Equivalent to PCA!

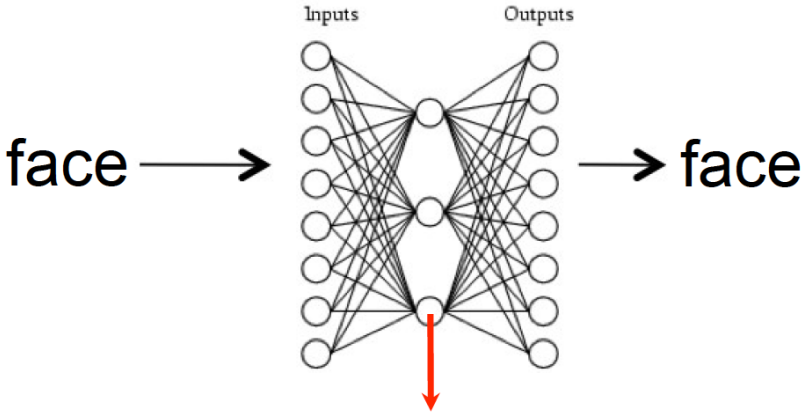
PCA: Two Views

1. Maximize variance (scatter of green points)
2. Minimize error (red-green distance per data point)



PCA as Reconstruction

$$\text{face}_i = \sum_k c_{ik}\ \text{eigenface}_k$$



"eigenfaces"

Reconstruction

- Assume data come from d-dimensional vectors where nth vector is $x=[x_1,x_2\ldots x_d]$
- We can represent a point in d-dimensional Euclidean space in terms of any d orthogonal vectors , call them $U=\{u_1,u_2\ldots u_d\}$
- The Goal is:
- For m<d find the vectors U such that $E=\sum_{i=1}^{n} ||x - \tilde{x}||^2$ **is minimized**

- Error all due to missing $u_{m+1}\ldots u_d$ terms

Reconstruction

- $E=\sum_{i=M+1}^{d}\sum_{j=1}^{N} u_i'(x-\bar{x})^2$
- $E=\sum_{i=M+1}^{d}\sum_{j=1}^{N}(u_i'(x-\bar{x}))('(x-\bar{x})'\ u_i)$
- $E=\sum_{i=M+1}^{d} u_i'\sum u_i$

Covariance matrix

- How do you minimize this value? $\sum u_i$
- Pick the the smallest eigenvectors you can here (i.e. get rid of the low PCA components).

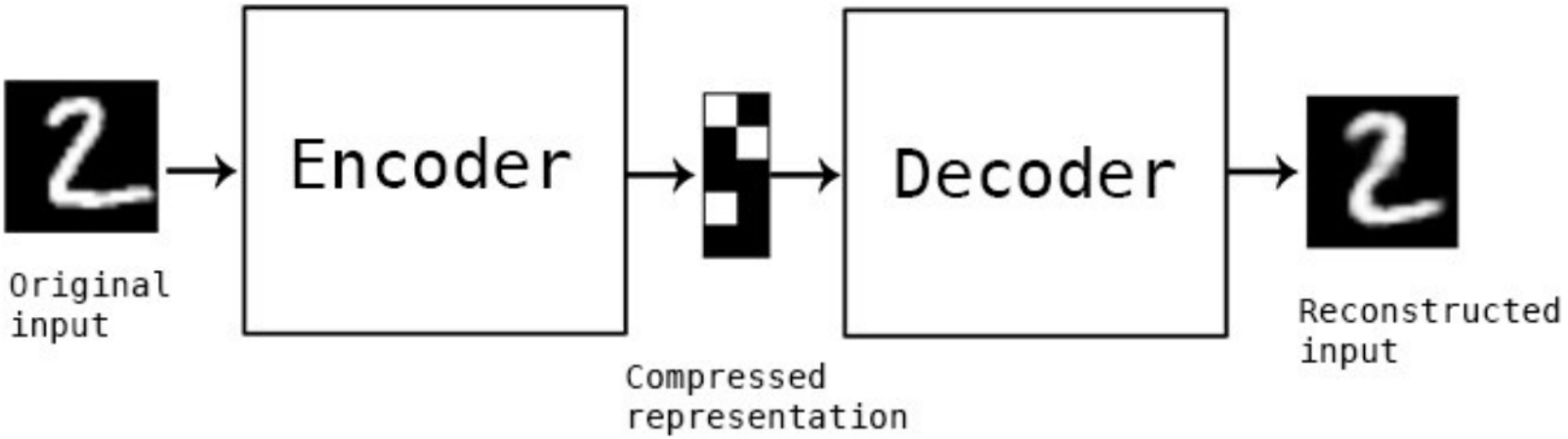
PCA on facial images

- PCA applied to 2429 19×19 images from CBCL dataset
- Reconstruction with only 3 components:

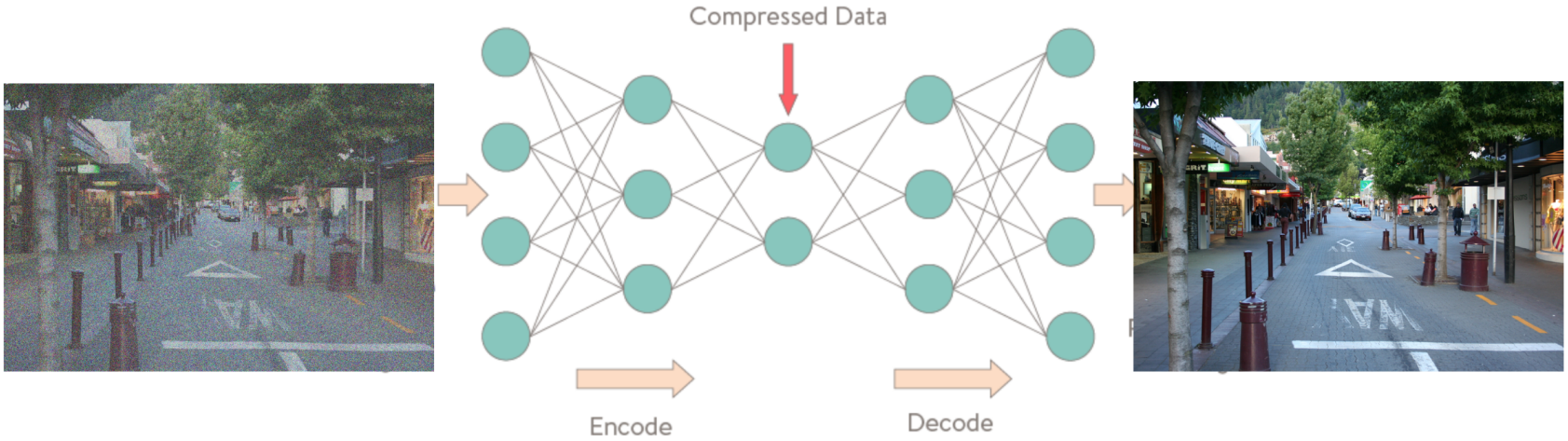
Undercomplete Autoencoders

- Special case: Linear decoder, mean squared error cost
  - Undercomplete AE learns to span the same subspace as PCA
  - The AE has learned the principal subspace of the training data
- Nonlinear encoder and decoder functions can give powerful nonlinear generalization of PCA
  - Be careful with capacity
  - Too much capacity ⇒ AE learns to copy the input w/o extracting useful information

Autoencoding MNIST



Application to Image Denoising

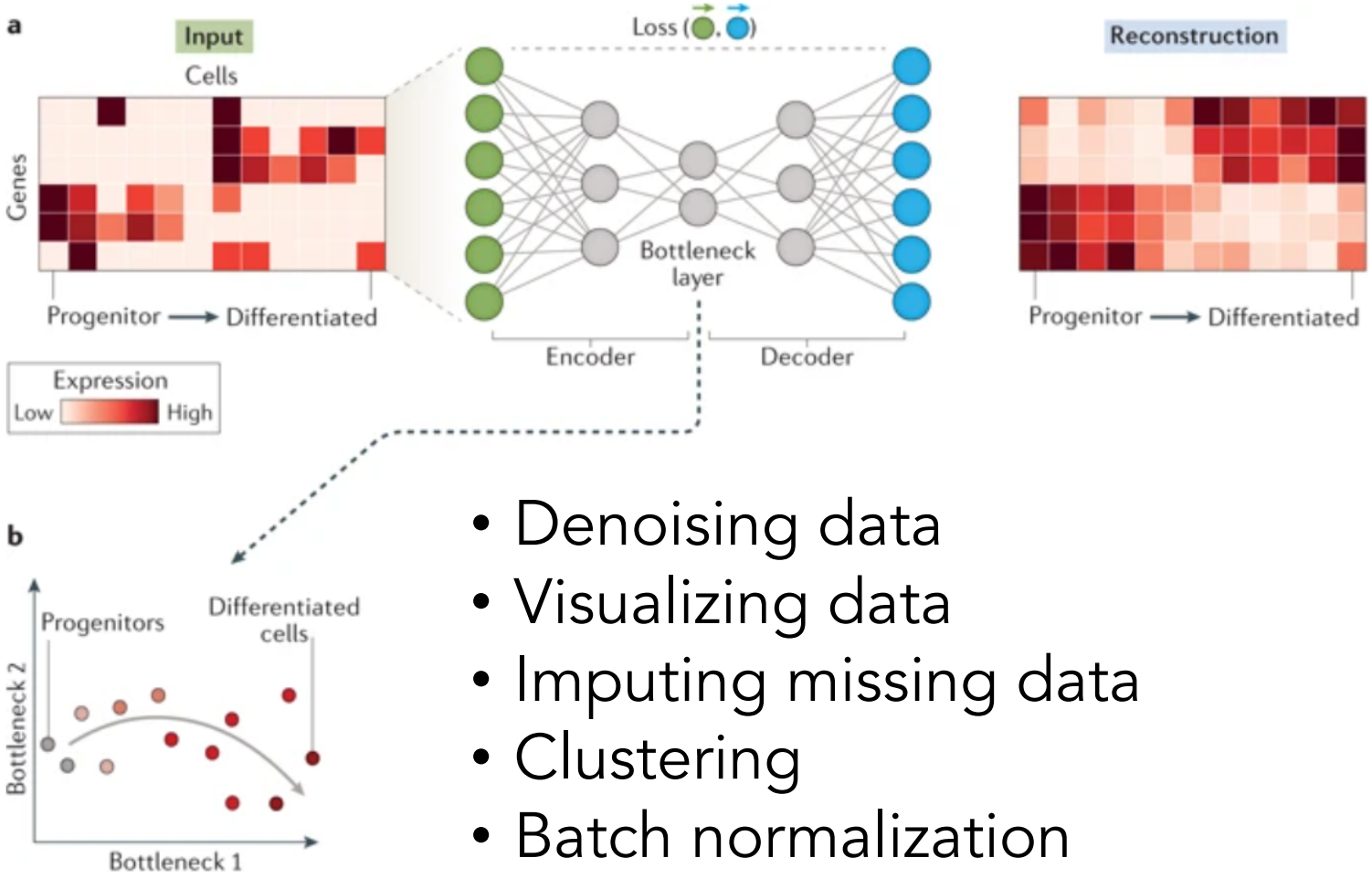


Compression layer automatically denoises

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