CS839: AI for Scientific Computing
ML Basics

Misha Khodak

University of Wisconsin-Madison

22 January 2026

Announcements

Enrollment:

- Finalized next week. Please keep checking your status.

Office hours:

- By appointment. Email me at khodak@wisc.edu.

Outline:

- Today: ML basics
- Tuesday: (relevant) topics in advanced ML
- Thursday: scientific computing basics

Outline

- **Supervised learning**
  - features, parametric modeling, estimation, optimization
- **Unsupervised learning**
  - dimensionality reduction
- **Neural networks**
  - MLPs, CNNs, training

Outline

- **Supervised learning**
  - features, parametric modeling, estimation, optimization
- **Unsupervised learning**
  - dimensionality reduction
- **Neural networks**
  - MLPs, CNNs, training

Supervised Learning

- Can I eat this?
- Safe or poisonous?
  - **Never seen it before**
- How to decide?

Supervised Learning: Training Instances

- I know about other mushrooms:

safe



poisonous




- Training set of **examples/instances/labeled data**

Supervised Learning: Formal Setup

Problem setting

- Set of possible instances $\mathcal{X}$
- Unknown *target function* $f : \mathcal{X} \rightarrow \mathcal{Y}$
- Set of *models* (a.k.a. *hypotheses*): $\mathcal{H} = \{h | h : \mathcal{X} \rightarrow \mathcal{Y}\}$
- Training set of instances for unknown target function,

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})$$

safe

poisonous

safe

Supervised Learning: Formal Setup

Problem setting

- Set of possible instances $\mathcal{X}$
- Unknown *target function* $f : \mathcal{X} \rightarrow \mathcal{Y}$
- Set of *models* (a.k.a. *hypotheses*): $\mathcal{H} = \{h | h : \mathcal{X} \rightarrow \mathcal{Y}\}$
- Training set of instances for unknown target function,

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})$$

Goal: model *h* that best approximates *f*

Supervised Learning: Objects

Three types of sets

- Input space, output space, hypothesis class

$$\mathcal{X}, \mathcal{Y}, \mathcal{H}$$

- **Examples**:
  - Input space: feature vectors $\mathcal{X} \subseteq \mathbb{R}^d$

  - Output space:
    - **Binary classification** $\mathcal{Y} = \{-1, +1\}$

safe poisonous

    - **Continuous** $\mathcal{Y} \subseteq \mathbb{R}$

Input Space: Feature Vectors

- Need a way to represent instance information (no need to use raw image):

cap-shape, cap-surface, cap-color, bruises, odor

$$\mathbf{x}^{(1)} = \langle \text{bell}, \quad \text{fibrous}, \text{gray}, \quad \text{false}, \text{foul}, \square \rangle$$

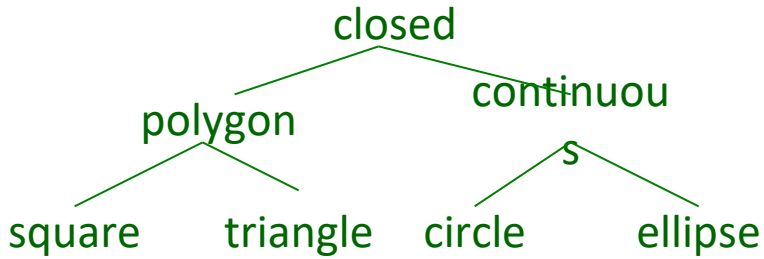
safe

- For each instance, store features as a vector.
  - **Next: What kinds of features can we have?**

Input Space: Feature Types

- *nominal* (including Boolean)
  - no ordering among values (e.g. *color* $\in$ {*red*, *blue*, *green*})
- *ordinal*
  - values of the feature are totally ordered (e.g. *size* $\in$ {*small, medium, large*})
- *numeric (*continuous*)*

  height $\in$ [0, 100] inches

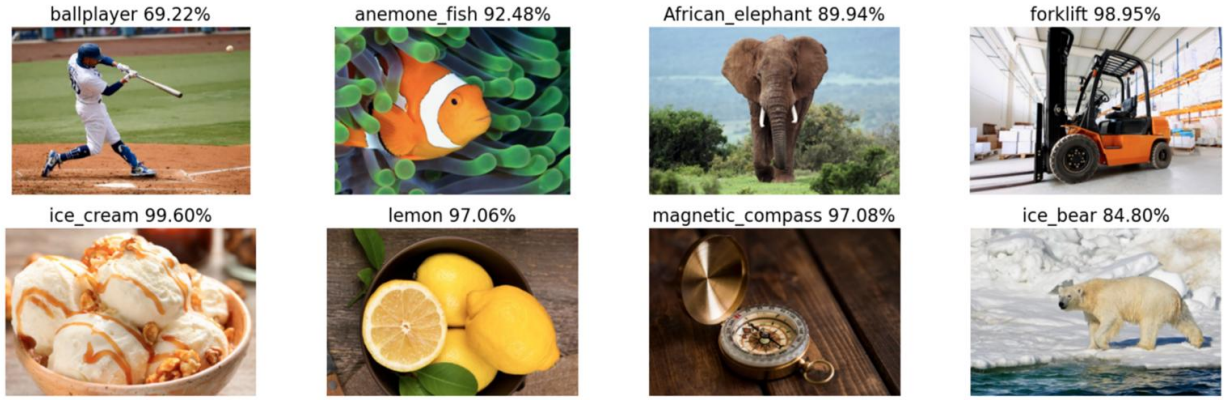


- *hierarchical*
  - possible values are partially *ordered* in a hierarchy, e.g. *shape*

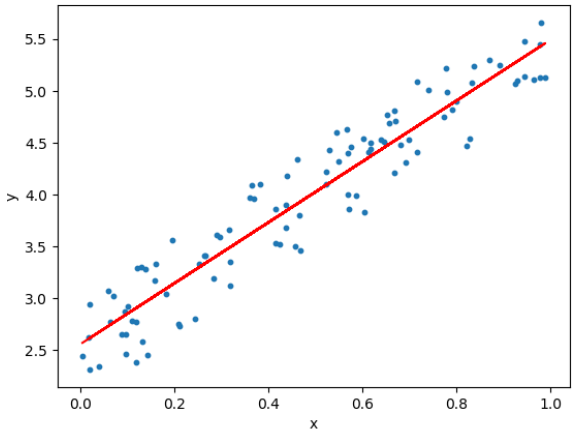
Output space: Classification vs. Regression

Choices of $\mathcal{Y}$ have special names:

- Discrete: "**classification**". The elements of $\mathcal{Y}$ are **classes**
  - Note: does not have to be binary



- Continuous: "**regression**"
  - Example: linear regression



- There are other types…

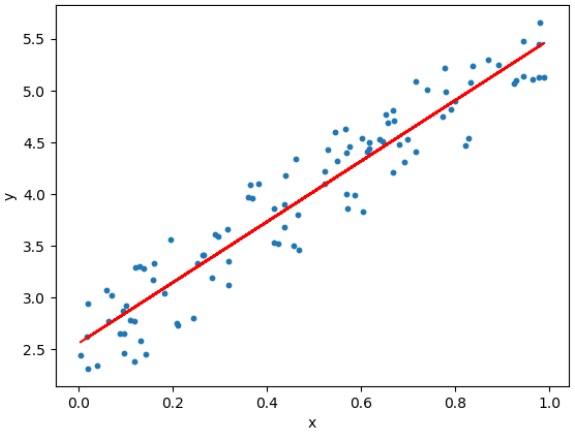
Hypothesis class

We talked about $\mathcal{X}, \mathcal{Y}$ what about $\mathcal{H}$?

- Recall: hypothesis class / model space.
  - Theoretically, could be all maps from $\mathcal{X}$ to $\mathcal{Y}$
  - But - does not work! We'll see why later.

- Pick specific class of models. E.g. linear models:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d$$



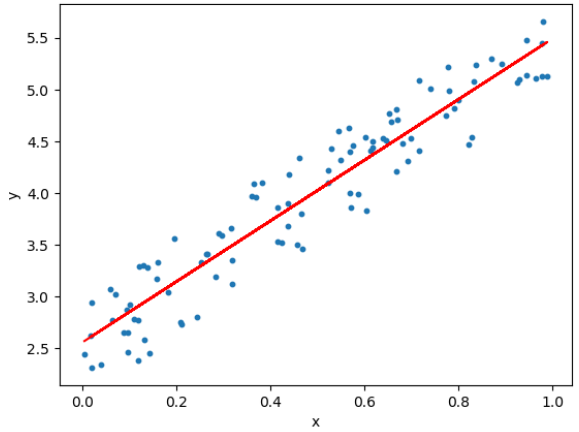
Hypothesis class: Linear Functions

- **Example** class of models: linear models

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d$$

Parameters (weights)

Features

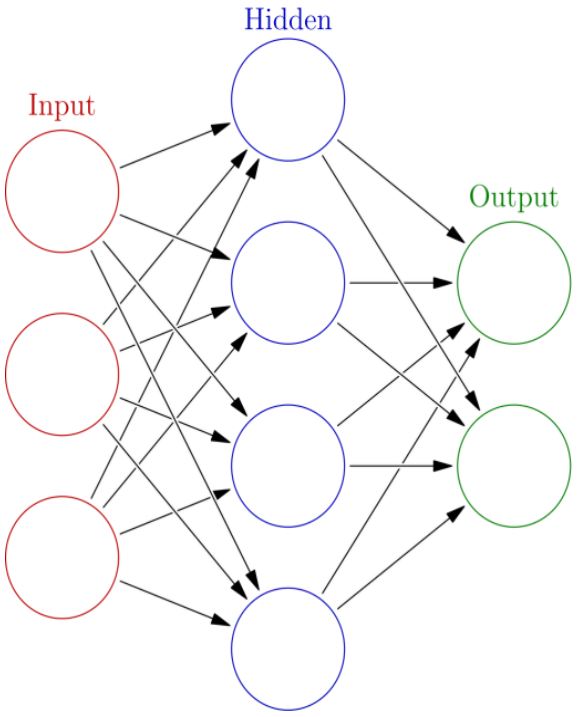


- How many linear functions are there?
  - Can any function be fit by a linear model?

Hypothesis class: Other Examples

Example classes of models: (deep) neural networks

$$f^{(k)}(x) = \sigma(W_k^T f^{(k-1)}(x)))$$



Wikipedia

Feedforward network

- Each layer:
  - linear transformation
  - Non-linearity

  - What are the parameters here?

Supervised Learning: Training

Goal: model *h* that best approximates *f*

- One way: empirical risk minimization (ERM)

$$\hat{f} = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(h(x^{(i)}), y^{(i)}))$$

Hypothesis Class

Loss function: how far is the prediction from the label)?

Model prediction

Supervised Learning: Predicting

Now that we have our learned model, we can use it for predictions.

$$\mathbf{x} = \langle \text{bell}, \text{fibrous}, \text{brown}, \text{false}, \text{foul}, \ldots \rangle$$

```
odor = a: e (400.0)
odor = c: p (192.0)
odor = f: p (2160.0)
odor = l: e (400.0)
odor = m: p (36.0)
odor = n
|   spore-print-color = b: e (48.0)
|   spore-print-color = h: e (48.0)
|   spore-print-color = k: e (1296.0)
|   spore-print-color = n: e (1344.0)
|   spore-print-color = o: e (48.0)
|   spore-print-color = r: p (72.0)
|   spore-print-color = u: e (0.0)
|   spore-print-color = w
|   |   gill-size = b: e (528.0)
|   |   gill-size = n
|   |   |   gill-spacing = c: p (32.0)
|   |   |   gill-spacing = d: e (0.0)
|   |   |   gill-spacing = w
|   |   |   |   population = a: e (0.0)
|   |   |   |   population = c: p (16.0)
|   |   |   |   population = n: e (0.0)
|   |   |   |   population = s: e (0.0)
|   |   |   |   population = v: e (48.0)
|   |   |   |   population = y: e (0.0)
|   spore-print-color = y: e (48.0)
odor = p: p (256.0)
odor = s: p (576.0)
odor = y: p (576.0)
```

safe or **poisonous**

Generalization

Fitting data isn't the only task, we want to **generalize**

- Apply learned model to unseen data:
  - For $(x, y) \sim \mathcal{D}$,

$$\mathbb{E}_{\mathcal{D}}[\ell(\hat{f}(x), y)]$$

- Can study theoretically or empirically
  - For theory: need assumptions, e.g. training instances are iid

Supervised Learning: Review

Problem setting

- Set of possible instances $\mathcal{X}$
- Unknown *target function* $f : \mathcal{X} \rightarrow \mathcal{Y}$
- Set of *models* (a.k.a. *hypotheses*) $\mathcal{H} = \{h | h : \mathcal{X} \rightarrow \mathcal{Y}\}$

Get

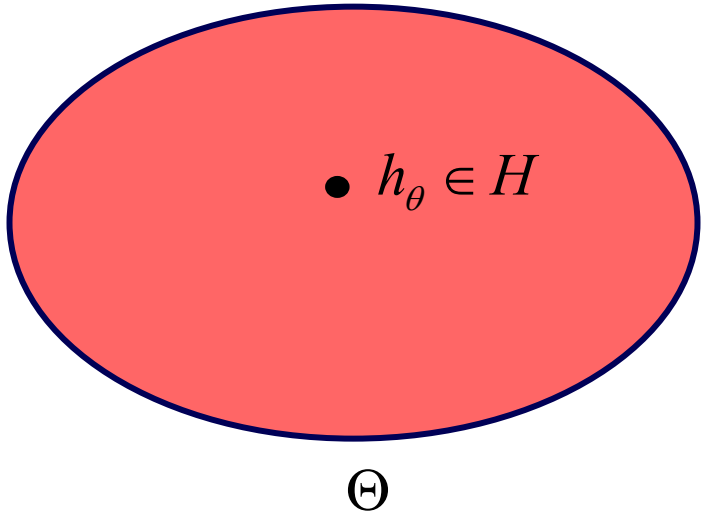
- Training set of instances for unknown target function *f*,

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})$$

Goal: model *h* that best approximates *f*

Parametric Learning

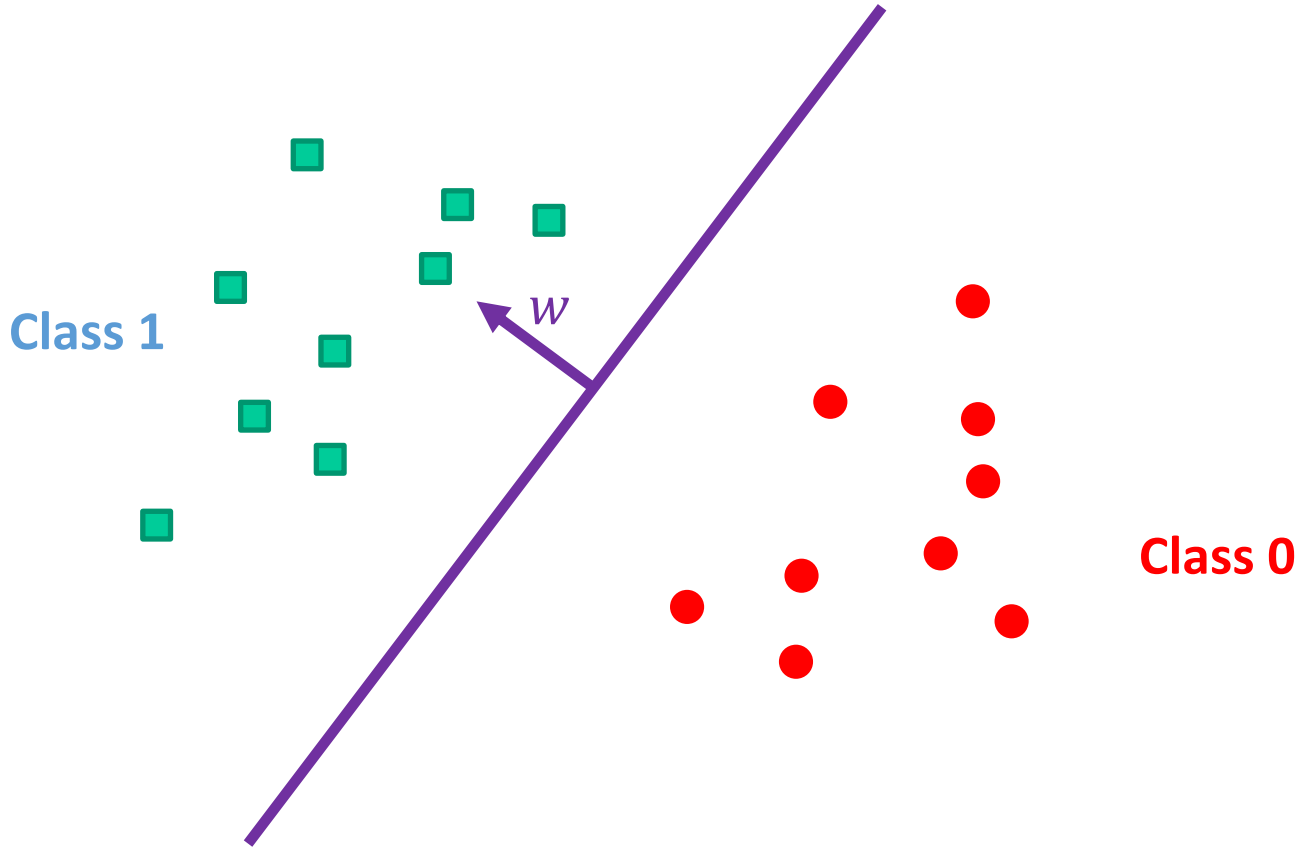
- A way to categorize learning techniques
  - Parametric: hypotheses indexed by a **parameter**
  - Learning: find parameter yielding model that best approximates the target
  - **Ex**: linear models, neural networks



- Nonparametric methods:
  - Instance-based methods (k-NN)
  - Decision trees

Classification: Linear models

- How do we learn a linear separator between two classes?



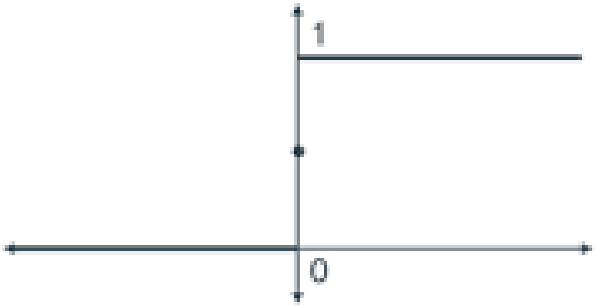
Linear Classification: Attempt 1

- Hyperplane: solutions to $\theta^T x = c$

- So… try to use such hyperplanes as separators?
  - Model: $f_\theta(x) = \theta^T x$
  - Predict: *y=1* if $\theta^T x > 0$, y=0 otherwise
  - i.e. $y = \text{step}(f_\theta(x))$
  - Training objective:

$$\ell(f_\theta) = \frac{1}{n}\sum_{i=1}^{m} 1\left[\text{step}\left(f_\theta\left(x^{(i)}\right)\right) \neq y^{(i)}\right]$$

step function



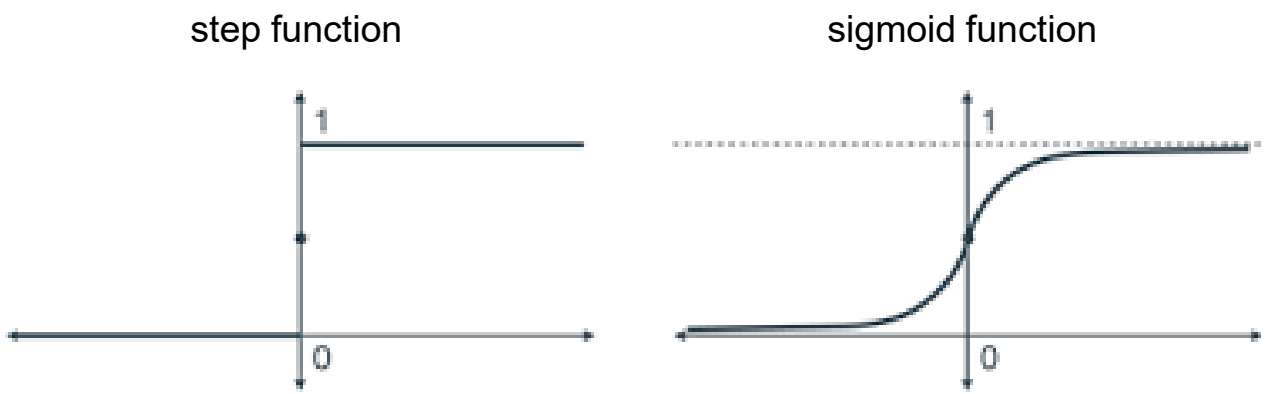
difficult loss function to optimize!!

Linear Classification: Attempt 2

Let us instead think probabilistically and learn $P_\theta(y|x)$ instead

How?

- Specify the conditional distribution $P_\theta(y|x)$
- Use maximum likelihood estimation (MLE) to get a nicer loss
- Run gradient descent (or related optimization algorithm)



Likelihood Function

- Captures the probability of seeing some data as a function of model parameters:

$$\mathcal{L}(\theta; X) = P_\theta(X)$$

  - If data is iid, we have $\mathcal{L}(\theta; X) = \prod_j p_\theta(x_j)$

- Often more convenient to work with the log likelihood
  - Both mathematically and for numerical stability
  - Log is a monotonic + strictly increasing function

ML: Conditional Likelihood

Similar idea, but now using conditional probabilities:

$$\mathcal{L}(\theta; Y, X) = p_\theta(Y|X)$$

If data is iid, we have

$$\mathcal{L}(\theta; Y, X) = \prod_j p_\theta(y_j|x_j)$$

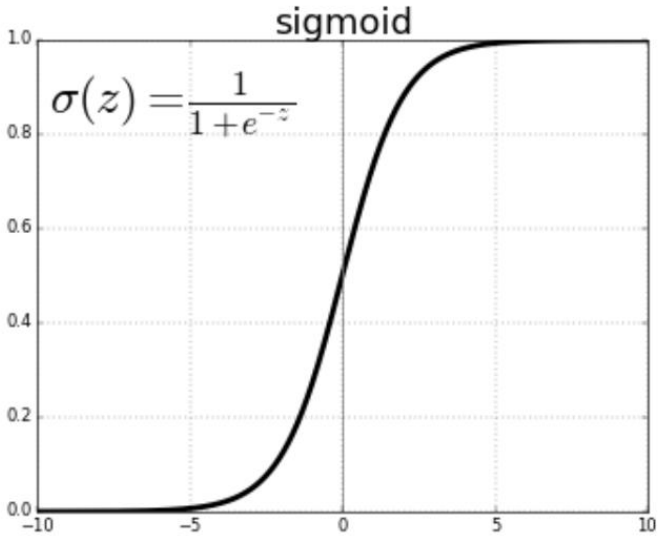
Apply this to linear classification to get **logistic regression.**

Logistic Regression: Conditional Distribution

- Notation: $\sigma(z) = \dfrac{1}{1+\exp(-z)} = \dfrac{\exp(z)}{1+\exp(z)}$ $\quad z \leftarrow \theta^T x$

sigmoid

"soft" version of step function



- **Conditional distribution model for logistic regression:**

$$P_\theta(y=1|x) = \sigma(\theta^T x) = \frac{1}{1+\exp(-\theta^T x)}$$

Logistic Regression: Loss

Conditional MLE:

$$\log \text{ likelihood}(\theta|x^{(i)}, y^{(i)}) = \log P_\theta(y^{(i)}|x^{(i)})$$

So:

$$\min_\theta \ell(f_\theta) = \min_\theta -\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(y^{(i)}|x^{(i)})$$

Equivalently:

$$\min_\theta -\frac{1}{n}\sum_{y^{(i)}=1} \log \sigma(\theta^T x^{(i)}) - \frac{1}{n}\sum_{y^{(i)}=0} \log(1-\sigma(\theta^T x^{(i)}))$$

Logistic regression: Summary

- **logistic regression = sigmoid conditional distribution + MLE**

- More precisely:
  - Give training data iid from some distribution D,
  - **Train**:

$$\min_{\theta} \ell(f_\theta) = \min_{\theta} -\frac{1}{n} \sum_{i=1}^{n} \log P_\theta(y^{(i)}|x^{(i)})$$

  - **Test**: output label probabilities

$$P_\theta(y = 1|x) = \sigma(\theta^T x) = \frac{1}{1 + \exp(-\theta^T x)}$$

Linear Regression: Setup

Training/learning: given

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$

Find $f_\theta(x) = \theta^T x = \sum_{i=0} \theta_i x_i$ that minimizes

Note: set $x_0 = 1$

Hypothesis Class

$$\ell(f_\theta) = \frac{1}{n} \sum_{j=1}^{n} (f_\theta(x^{(j)}) - y^{(j)})^2$$

Loss function (how far are we)?

Linear regression: MLE

How did we get this objective?

$$\ell(f_\theta) = \frac{1}{n}\sum_{j=1}^{n}(f_\theta(x^{(j)}) - y^{(j)})^2$$

Once again, conditional MLE:

- model: $y = f_\theta(x) + \varepsilon$, $\varepsilon \sim N(0, \sigma^2)$
- likelihood:

$$\propto \prod_{j=1}^{n} \exp\left(-\frac{\left(f_\theta\left(x^{(j)}\right) - y^{(j)}\right)^2}{2\sigma^2}\right)$$

Linear Regression: Notation

- **Matrix notation**:
  - set *X* to have *j*th row be $(x^{(j)})^T$
  - And *y* to be the vector $[y^{(1)}, \ldots, y^{(n)}]^T$
- Can re-write the loss function as

$$\ell(f_\theta) = \frac{1}{n}\sum_{j=1}^{n}(f_\theta(x^{(j)}) - y^{(j)})^2 = \frac{1}{n}\|X\theta - y\|_2^2$$

Linear Regression: Fitting

- **Set gradient to 0 w.r.t. the weight,**

$$\nabla \ell(f_\theta) = \nabla \frac{1}{n} \|X\theta - y\|_2^2 = 0$$

$$\implies \nabla[(X\theta - y)^T (X\theta - y)] = 0$$

$$\implies \nabla[\theta^T X^T X\theta - 2\theta^T X^T y + y^T y] = 0$$

$$\implies 2X^T X\theta - 2X^T y = 0$$

$$\implies \theta = (X^T X)^{-1} X^T y$$

(assume $X^T X$ is invertible)

Evaluation: Metrics

- MSE/RMSE (mean-square error + root version)
- MAE (mean average error)
- R-squared
- Usually, compute on training data… (but should do cross validation!)

High-dimensional linear regression

Data matrix X is $n \times d$

- number of data points n
- number of features d

If $n > d$ and X has full column rank then $X^\top X$ is invertible

But what if $d \gg n$?

Solution: **Regularization**

Same setup, new loss (**Ridge regression**):

$$\ell(f_\theta) = \frac{1}{n}\sum_{j=1}^{n}(f_\theta(x^{(j)}) - y^{(j)})^2 + \lambda\|\theta\|_2^2$$

regularization parameter

Conveniently, still has a closed form solution

$$\theta = (X^T X + \lambda n I)^{-1} X^T y$$

Goals:

- solves the problem of $X^\top X$ not being invertible
- results in a θ with small norm, often less likely to overfit

Alternative regularization: **LASSO**

- Another type of regularization:

$$\ell(f_\theta) = \frac{1}{n}\sum_{j=1}^{n}(f_\theta(x^{(j)}) - y^{(j)})^2 + \lambda\|\theta\|_1$$

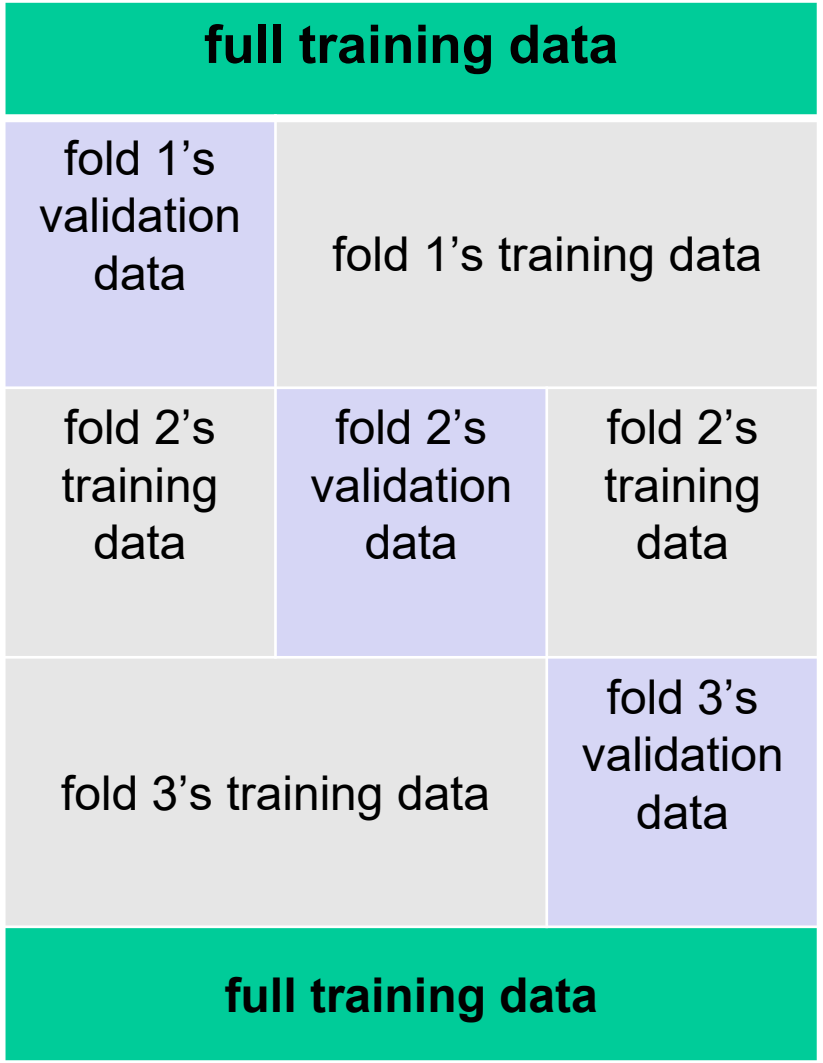
regularization parameter

  - unlike the ℓ_2-norm, regularizing by the ℓ_1-norm is known to encourage a sparse θ
    - theoretical understanding of this phenomenon exists under assumptions on X and y (**compressed sensing**)
    - useful for both regularization and **feature selection**

Choosing the regularization strength λ

For prediction: use cross-validation!

- split dataset into k train-validation folds
- for each candidate λ:
  - compute average across folds $i = 1, \ldots, k$ of
    - MSE (or other metric) of $\theta_{\lambda,i}$ on fold i's validation data
    - $\theta_{\lambda,i}$ minimizes Ridge/LASSO with parameter λ on fold i's training data
- retrain on the full training data with the optimal candidate λ



Another Approach: **Bayesian Inference**

- Let's consider a different approach
- Need a little bit of terminology

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

- *H* is the hypothesis
- *E* is the evidence

Bayesian Inference Definitions

Terminology:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)} \longleftarrow \textbf{Prior}$$

Prior: estimate of the probability **without** evidence

Bayesian Inference Definitions

Terminology:

Likelihood

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Likelihood: probability of evidence **given a hypothesis**.

- Compare to the way we defined the likelihood earlier

Bayesian Inference Definitions

Terminology:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

↑

Posterior

Posterior: probability of hypothesis **given evidence**.

MAP Definition

- Suppose we think of the parameters as random variables
  - There is a prior $P(\theta)$

- Then, can do learning as Bayesian inference
  - "Evidence" is the data

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}$$

  - **Maximum a posteriori probability** (**MAP**) estimation

$$\theta^{\mathrm{MAP}} = \arg\max_{\theta} \prod_{i=1}^{n} p(x^{(i)}|\theta)p(\theta)$$

MAP vs ML

What's the difference between ML and MAP?

$$\theta^{\mathrm{MLE}} = \arg\max_{\theta} \prod_{i=1}^{n} p(x^{(i)}|\theta)$$

$$\theta^{\mathrm{MAP}} = \arg\max_{\theta} \prod_{i=1}^{n} p(x^{(i)}|\theta)p(\theta)$$

the prior!

Probabilistic interpretation

the ordinary least squares (OLS) estimator $\theta = (X^T X)^{-1} X^T y$ estimator is the MLE of a Gaussian probabilistic model:

- $y^{(i)} \sim N(\theta^\top x^{(i)}, \sigma^2)$
- assume variance σ^2 is known

Ridge regression and LASSO are **MAP estimators of the same probabilistic model** with different priors for θ

- Ridge regression: $\theta \sim N(0_d, \tau^2 I_d)$
- LASSO: $\theta \sim \text{Laplace}(0_d, \tau)$
- in both cases τ depends on σ^2 and λ

Iterative Methods: Gradient Descent

What if there's no closed-form solution to the objective?

Use an iterative approach to gradually get closer to the solution.

Gradient descent:

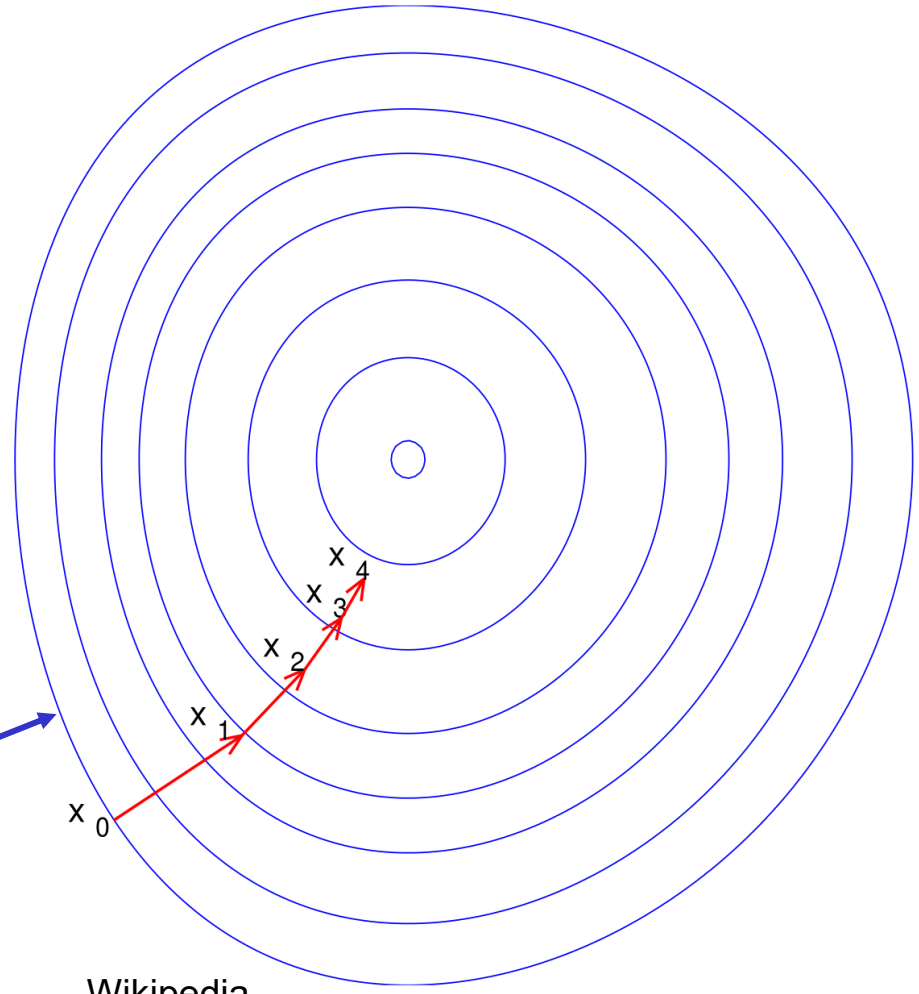
- Suppose we're computing $\min_{\theta} g(\theta)$
- Start at some θ_0
- Iteratively compute $\theta_{t+1} = \theta_t - \alpha \nabla g(\theta_t)$
- Stop after some # of steps

learning rate / step size

Gradient Descent: Illustration

- **Goal**: steps get closer to minimizer

- Some notes:
  - Step size can be fixed or varying
  - Under certain conditions, will converge to global minimum



level sets

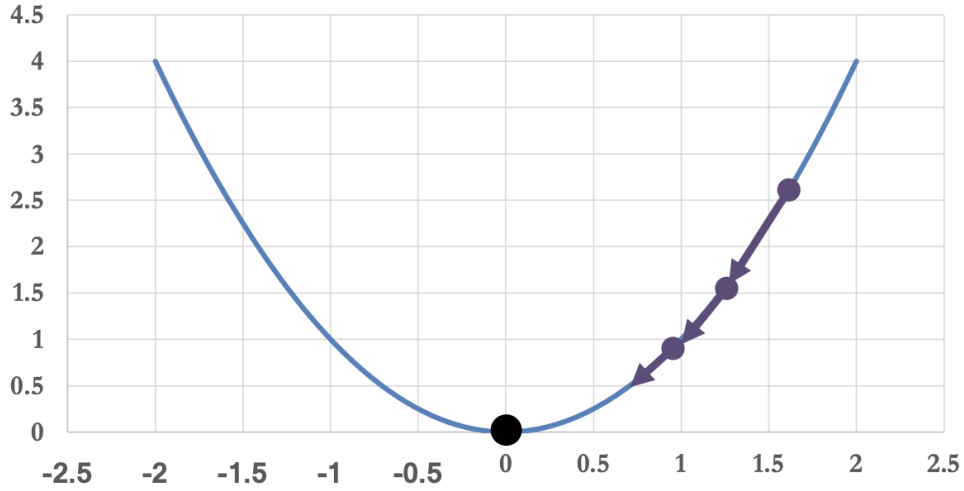
Wikipedia

Gradient Descent: Convergence

Gradient descent is guaranteed to converge under a variety of assumptions on the objective (e.g. smoothness, convexity)

These hold for the regression models we've seen so far

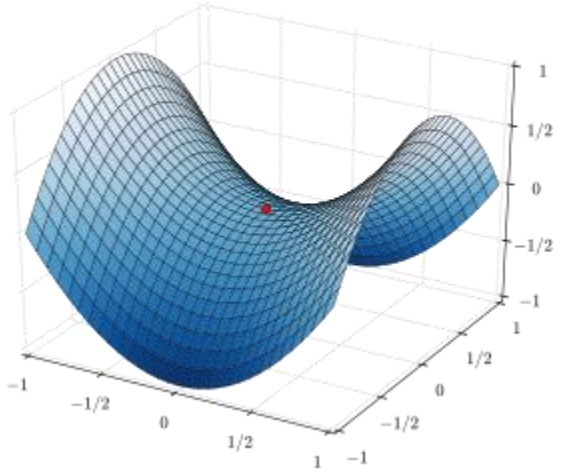
They do NOT hold for deep nets



Gradient Descent as a heuristic

If a function is non-convex, gradient descent

- can still be applied so long as it is differentiable
- is only guaranteed to reach a **stationary point**, not necessarily a global minimum



Wikipedia

Nevertheless, neural networks are commonly fit using (extensions of) gradient descent:

- objectives are non-convex AND non-smooth
- often get parameters that **are optimal** (low training loss) AND **generalize well** (high test accuracy)
- required decades hacking and experimentation
- should be viewed as a poorly understood heuristic

Gradient Descent: Drawbacks

- Why would we use anything but GD?
- Let's go back to ERM. $\arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(h(x^{(i)}, y^{(i)})$
- For GD, need to compute $\nabla \ell(h(x^{(i)}, y^{(i)})$
  - Each step: n gradient computations
  - ImageNet: 10^6 samples… so for 100 iterations, **10^8 gradients**

Solution: **Stochastic** Gradient Descent

Simple modification to GD:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{n} \sum_{i=1}^{n} \nabla \ell(f(\theta_t; x^{(i)}), y^{(i)})$$

SGD:

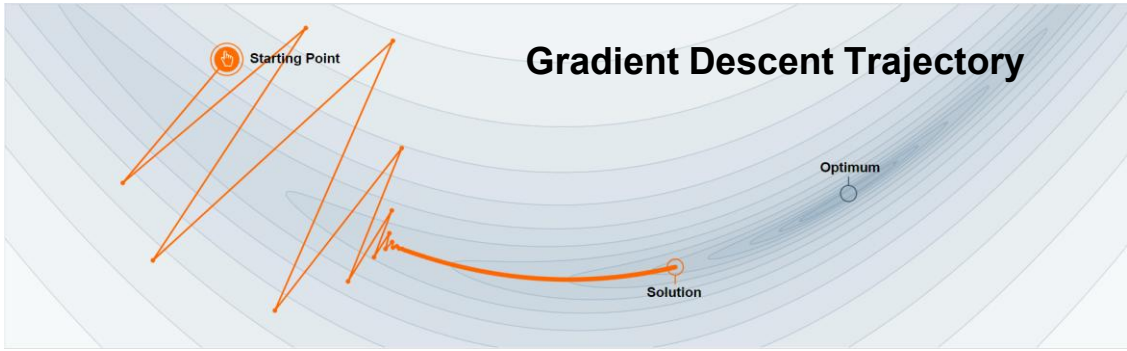
$$\theta_{t+1} = \theta_t - \alpha \nabla \ell(f(\theta_t; x^{(a)}), y^{(a)})$$

- Here a is selected uniformly from 1,…,n (“**stochastic**” bit)
- Note: **no sum**!
- In expectation, same gradient as GD.
- In practice we often update using **minibatches** of data to take advantage of (GPU) parallelism

Other drawbacks of gradient descent

Behaves poorly on many important functions:

- e.g. LASSO is convex but non-differentiable, so we use coordinate descent or proximal methods
- on poorly conditioned problems, GD struggles to make progress down narrow valleys.
- alternatives:
  - momentum methods
  - second-order methods (Newton)
  - approximate second-order methods (includes widely used deep net optimizers such as Adam)



Source: Goh. *Why Momentum Really Works*. Distill 2017.

Outline

- **Supervised learning**
  - features, parametric modeling, estimation, optimization
- **Unsupervised learning**
  - dimensionality reduction
- **Neural networks**
  - MLPs, CNNs, training

Unsupervised Learning: Setup

- Given instances $\{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$

- **Goal**: discover interesting regularities/structures/patterns that characterize the instances. For example:
  - clustering
  - **dimensionality reduction**
  - generative models
  - …

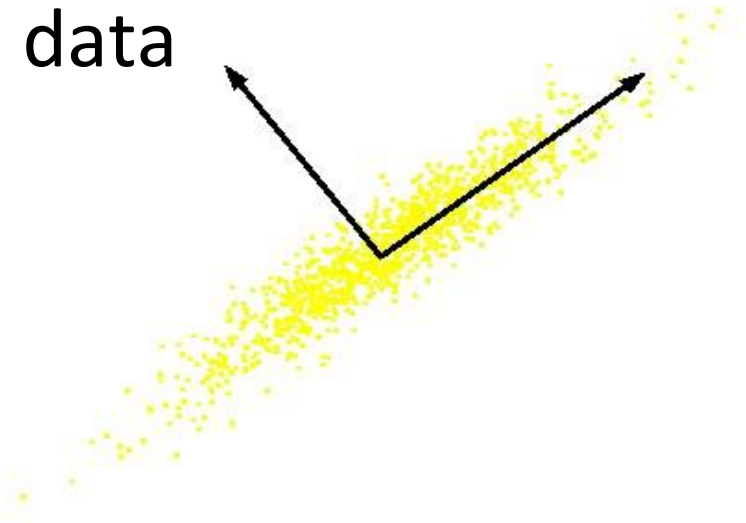
Principal Components Analysis

Unsupervised technique for extracting variance structure from high dimensional datasets

- also reduces dimensionality

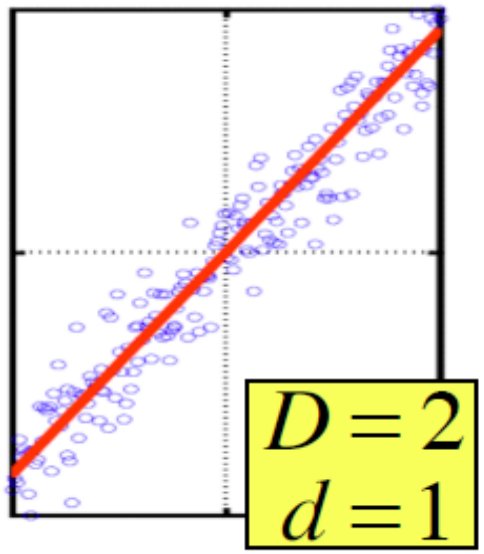
PCA: orthogonal projection / transformation of the data

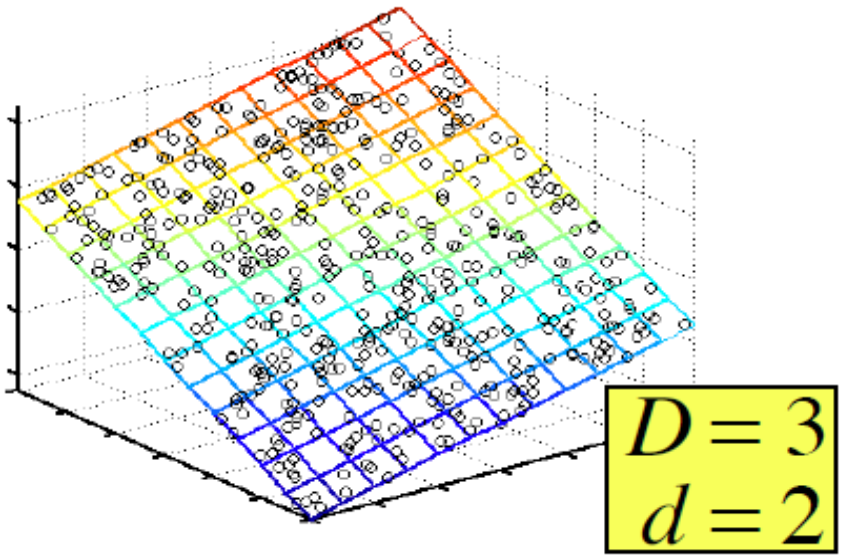
- Into a (possibly lower dimensional) subspace
- Goal: maximize variance of the projected data



PCA Intuition

The dimension of the ambient space (ie, R^d) might be much higher than the **intrinsic** data dimension



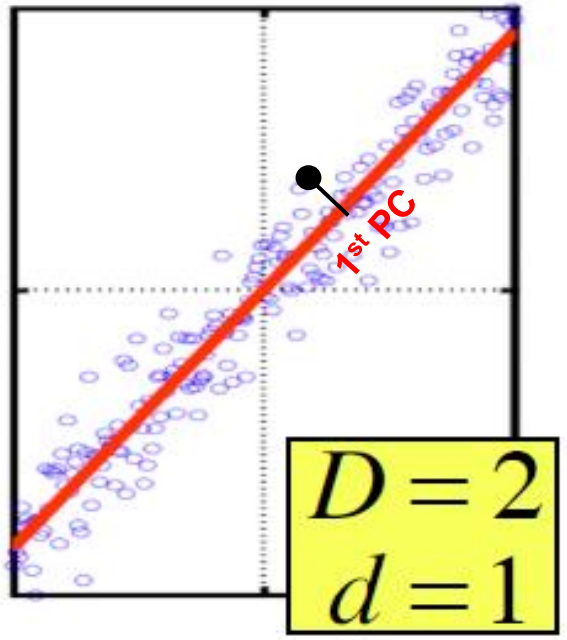


In case where data lies on or near a low d-dimensional linear subspace, axes of this subspace are an effective representation of the data.

PCA: Principal Components

Principal Components (PCs) are orthogonal directions that capture most of the variance in the data.

- First PC – direction of greatest variability in data.
- Projection of data points along first PC discriminates data most along any one direction

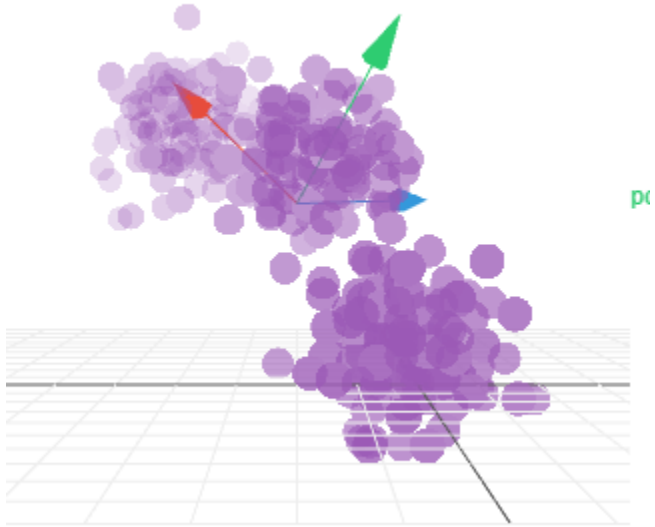


PCA: Principal Components and Projection

How does dimensionality reduction work?
From *d* dimensions to *r* dimensions:

- get orthogonal vectors (PCs) $v_1, \ldots, v_r \in \mathbb{R}^d$ that maximize variability (equivalently minimize reconstruction error)

- then project data onto PCs



Victor Powell

PCA First Step

First component,

$$v_1 = \arg\max_{\|v\|=1} \sum_{i=1}^{n} \langle v, x_i \rangle^2$$

Same as getting

$$v_1 = \arg\max_{\|v\|=1} \|Xv\|^2$$

PCA Recursion

Once we have *k-1* components, next?

$$\hat{X}_k = X - \sum_{i=1}^{k-1} X v_i v_i^T$$

deflation

Then do the same thing

$$v_k = \arg\max_{\|v\|=1} \|\hat{X}_k w\|^2$$

PCA Interpretations

The v's are eigenvectors of XX^T **(Gram matrix)**

XX^T (proportional to) sample covariance matrix

- when data is 0 mean!
- i.e. PCA is the eigendecomposition of sample covariance
- nested subspaces *span(v1), span(v1,v2),…,*

PCA Interpretations: First Component

Two specific ways to think about the first component

Maximum variance direction

- What we saw so far

$$\sum_{i=1}^{n} (\mathbf{v}^T \mathbf{x}_i)^2 = \mathbf{v}^T \mathbf{X}\mathbf{X}^T \mathbf{v}$$

Minimum reconstruction error

- A direction so that projection yields minimum MSE in reconstruction

$$\sum_{i=1}^{n} \|\mathbf{x}_i - (\mathbf{v}^T \mathbf{x}_i)\mathbf{v}\|^2$$

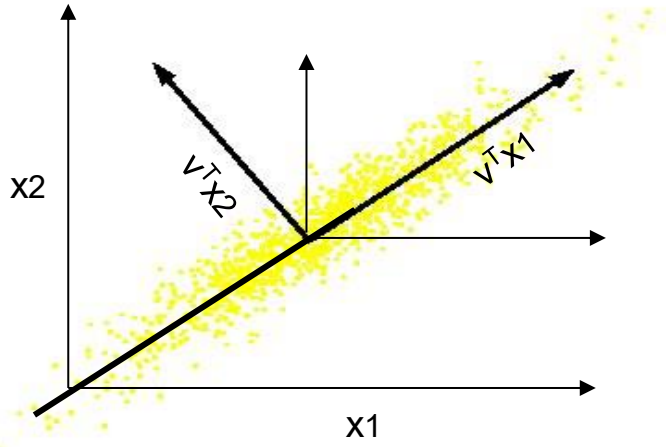
PCA Covariance Matrix Interpretation

So $\Rightarrow \boxed{(\mathbf{X}\mathbf{X}^T)\mathbf{v} = \lambda\mathbf{v}}$

- means that v (the first PC) is an eigenvector of XX^T
- eigenvalue λ denotes the amount of variability captured along that dimension

- PCs are just the eigenvectors…
  - How to find them? Eigendecomposition

- Don't need to keep all eigenvectors
  - Just the ones for largest eigenvalues



Outline

- **Supervised learning**
  - features, parametric modeling, estimation, optimization
- **Unsupervised learning**
  - dimensionality reduction
- **Neural networks**
  - MLPs, CNNs

Neural Networks: Basics

So far we've seen simple parametric models: $f_\theta(x) = \theta^\top x$

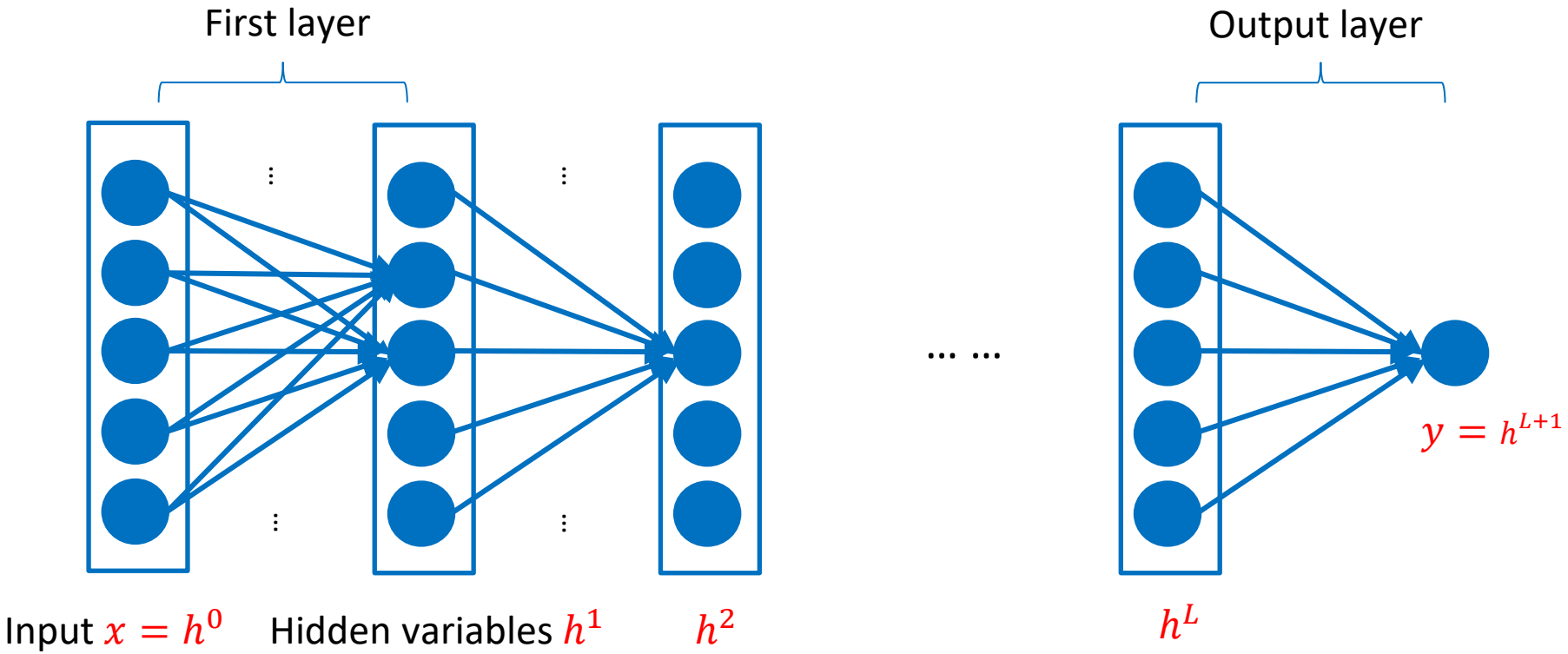
A neural network is just a more complicated one:

$$f_{W_1,\ldots,W_L,\theta}(x) = \theta^\top \sigma\left(W_L \sigma\left(W_{L-1}\sigma\left(\cdots W_2 \sigma(W_1 x)\right)\right)\right)$$

- σ is an elementwise nonlinearity (e.g. $\sigma(x)_i = \max\{x_i, 0\}$)
- this is a feedforward network or multi-layer perceptron (MLP)
- each matrix multiply followed by nonlinearity is called a **layer**
- $h^l = \sigma\left(W_l \sigma\left(W_{l-1}\sigma\left(\cdots W_2 \sigma(W_1 x)\right)\right)\right)$ is called an **activation** or **hidden representation**

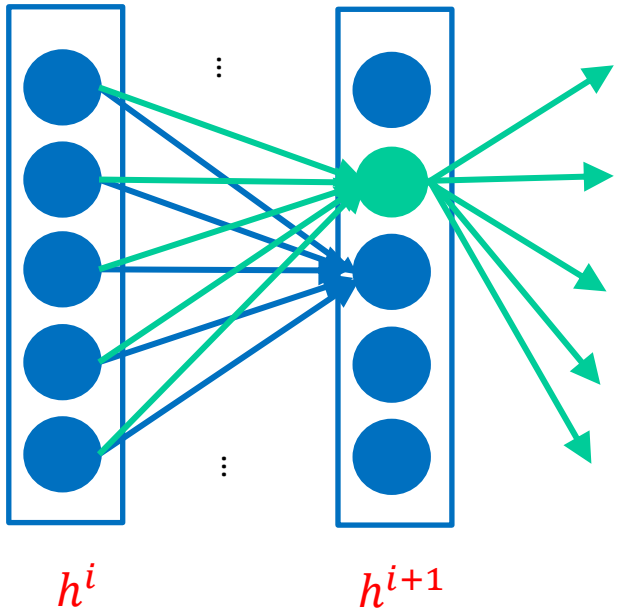
Neural Network Components

An $(L+1)$-layer network



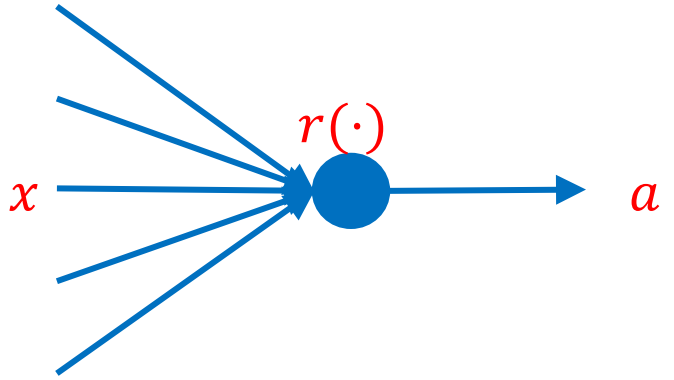
Hidden Layers

- Neuron takes weighted linear combination of the previous representation layer
  - Outputs one value for the next layer



Hidden Layers

- Outputs $a = r(w^T x + b)$

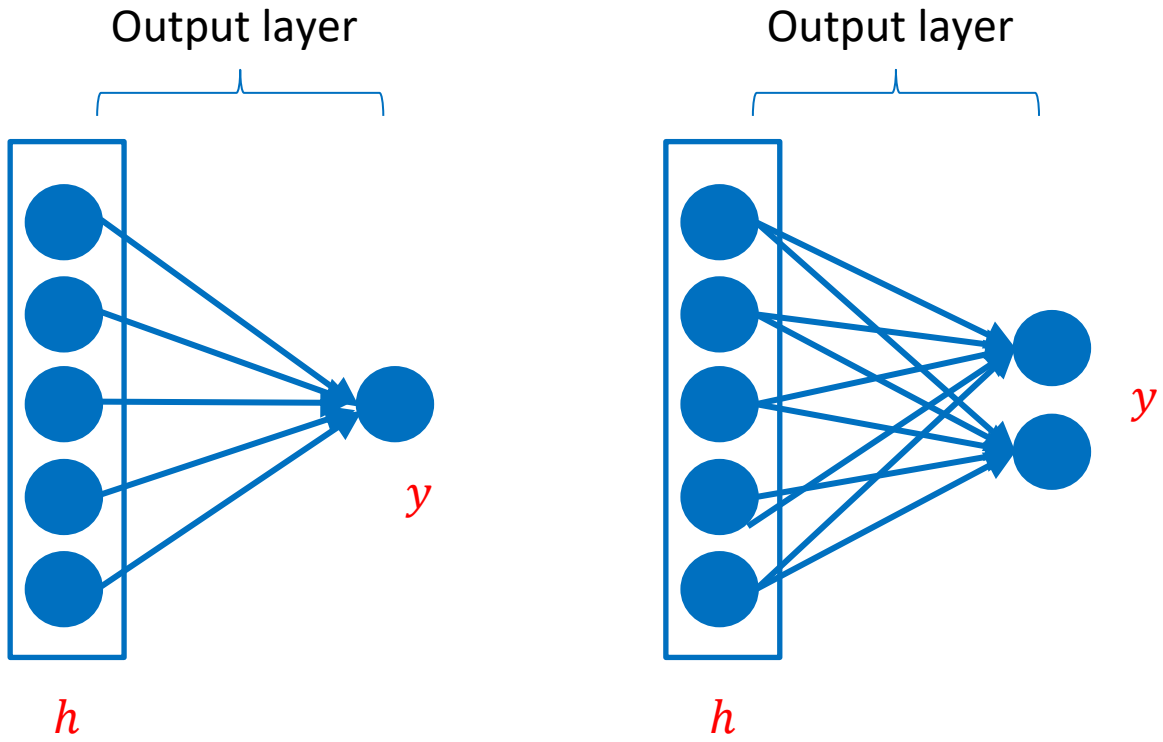


- Typical activation function r
  - threshold $\mathrm{h}(z) = 1_{\{z \geq 0\}}$
  - ReLU $\mathrm{ReLU}(z) = z \cdot t(z) = \max\{0, z\}$
  - sigmoid $\sigma(z) = 1/(1 + \exp(-z))$
  - hyperbolic tangent $\tanh(z) = 2\sigma(2z) - 1$

- Why not **linear activation** functions?
  - Model would be linear.

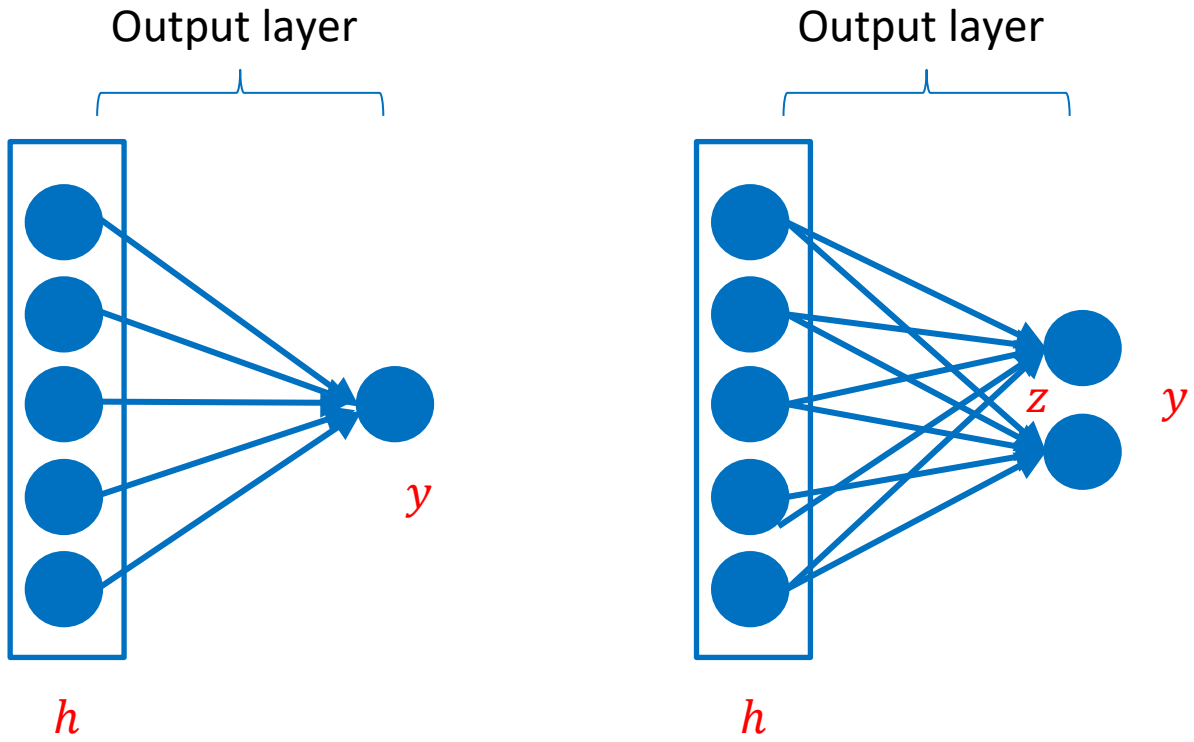
Output Layer: Examples

- Regression: $y = w^T h + b$
  - Linear units: no nonlinearity
- Multi-dimensional regression: $y = W^T h + b$
  - Linear units: no nonlinearity



Output Layer: Examples

- Binary classification: $y = \sigma(w^T h + b)$
  - Corresponds to using logistic regression on h
- Multiclass classification:
  - $y = \text{softmax}(z)$ where $z = W^T h + b$



Training Neural Networks

Training is done in the usual way: pick a loss and optimize it

- **Example**: 2 scalar weights

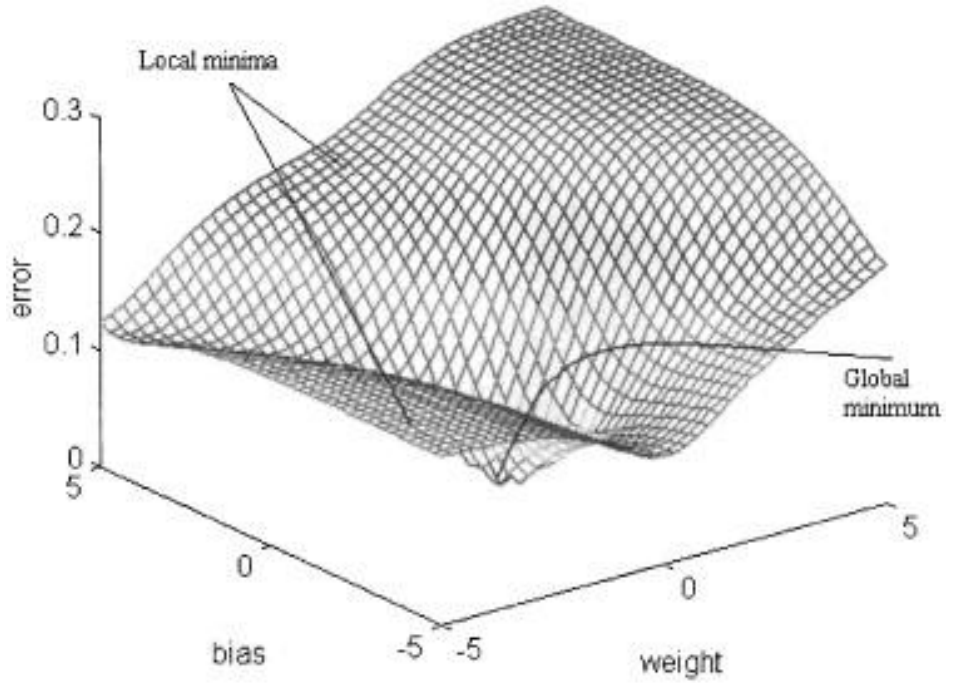


figure from Cho & Chow, *Neurocomputing* 1999

Training Neural Networks with SGD

Algorithm:

- Input dataset $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$
- Initialize weights
- Until stopping criterion is met:
  - For each training point $(x^{(i)}, y^{(i)})$ do
    - Compute prediction: $\hat{y}^{(i)} = f_w(x^{(i)})$ ⟵ **forward pass**
    - Compute loss: $L^{(i)} = L(\hat{y}^{(i)}, y^{(i)})$ ⟵ e.g. negative log-likelihood (NLL) loss $L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$
    - Compute gradient: $\nabla_w L^{(i)} = (\partial_{w_1} L^{(i)}, \partial_{w_2} L^{(i)}, \ldots, \partial_{w_m} L^{(i)})^\top$ ⟵ **backward pass**
    - Update weights: $w \leftarrow w - \alpha \nabla_w L^{(i)}$ ⟵ SGD step

Training Neural Networks with **minibatch** SGD

Algorithm:

- Input dataset $D = \{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$
- Initialize weights
- Until stopping criterion is met:
  - Sample a **batch of b training points** $i_1, \dots, i_b$
  - Compute predictions: $\{\hat{y}^{(i_1)}, \dots, \hat{y}^{(i_b)}\} = \{f_w(x^{(i_1)}), \dots, f_w(x^{(i_b)})\}$
  - Compute avg. loss: $L^{(i_1,\dots,i_b)} = \frac{1}{b}\sum_{j=1}^{b} L(\hat{y}^{(i_j)}, y^{(i_j)})$
  - Compute gradient: $\nabla_w L^{(i_1,\dots,i_b)} = (\partial_{w_1} L^{(i_1,\dots,i_b)}, \dots, \partial_{w_m} L^{(i_1,\dots,i_b)})^\top$
  - Update weights: $w \leftarrow w - \alpha \nabla_w L^{(i_1,\dots,i_b)}$

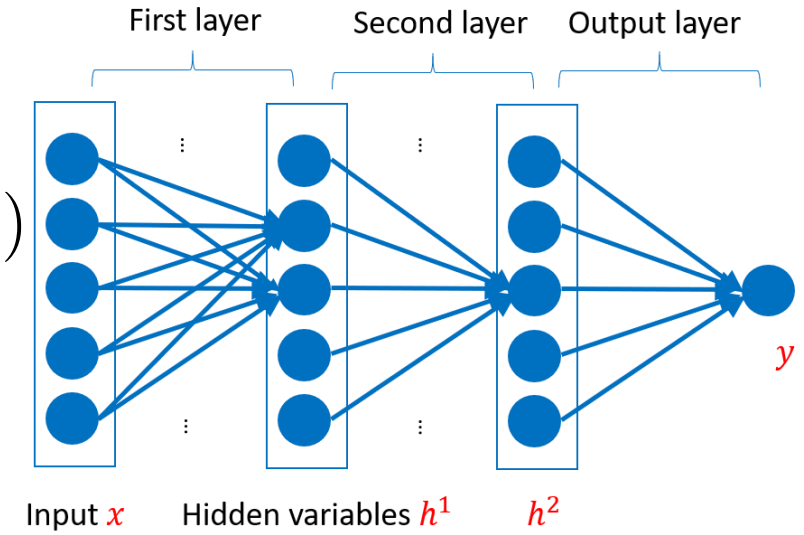
Training Neural Networks: Chain Rule

Will need to compute terms like: $\frac{\partial L}{\partial w_1}$

- But, L is a composition of:
  - Loss with output y
  - Output itself a composition of softmax with outer layer
  - Outer layer a combination of outputs from previous layer
  - Outputs from prev. layer a composition of activations and linear functions…

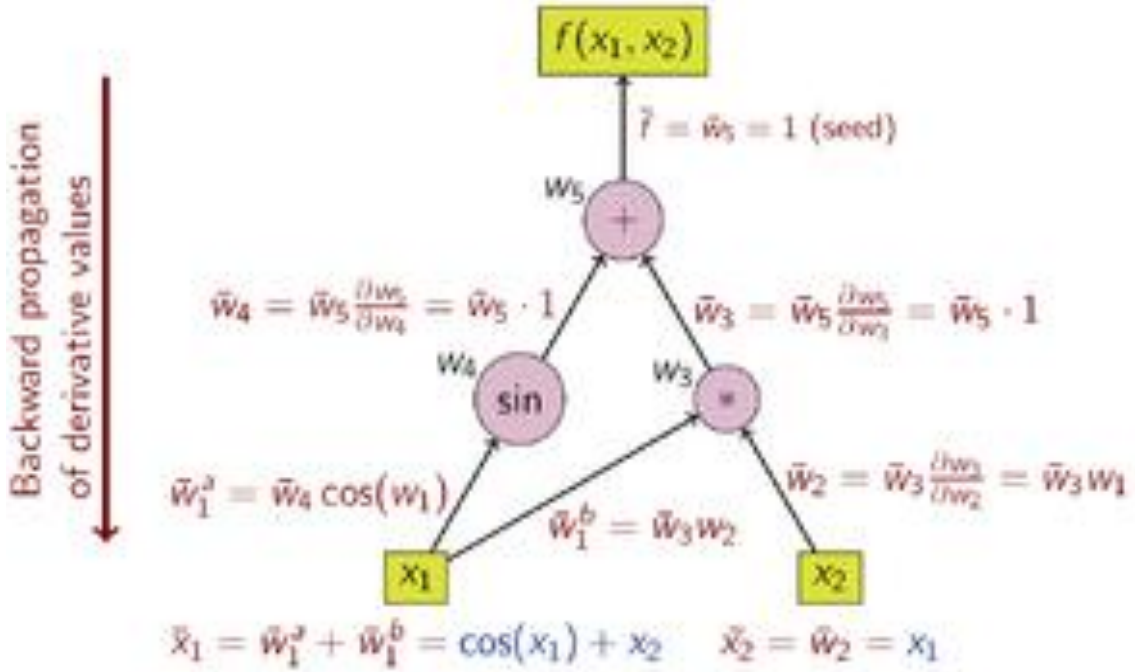
Need the **chain rule**!

- Suppose $L = L(g_1, \ldots, g_k)\, g_j = g_j(w_1, \ldots, w_p)$
- Then, $\frac{\partial L}{\partial w_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial g_j} \frac{\partial g_j}{\partial w_i}$



Backpropagation

- To compute gradient w.r.t specific weights we **propagate** loss information **back** through the network
- Today we do this by automatic differentiation (**autodiff**) for arbitrarily complex computation graphs
- Go backwards from top to bottom, recursively computing gradients

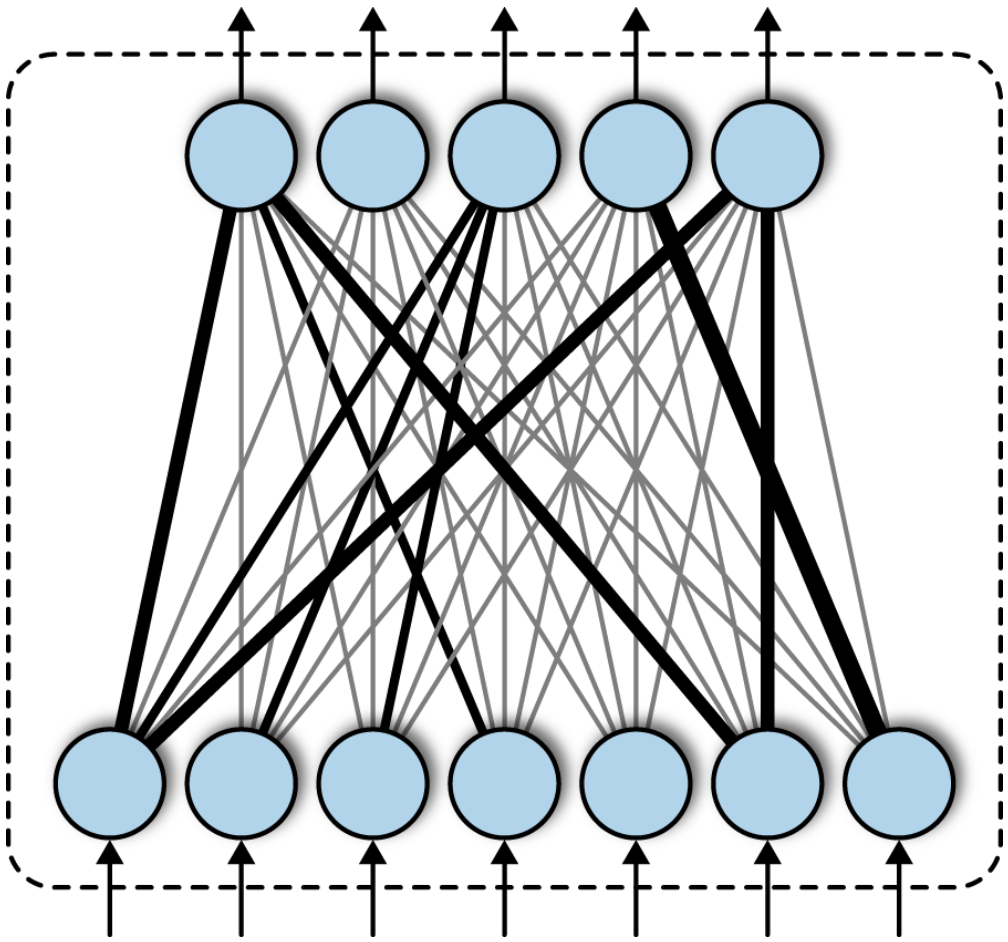


Wiki

Review: Multi-layer perceptrons (MLPs)

So far we've been using MLP networks, which consist of compositions of **fully-connected layers**, so named because every input unit is connected to every output unit

$$h^{l+1} = \sigma(Wh^l + b)$$



What if we have images as our inputs?

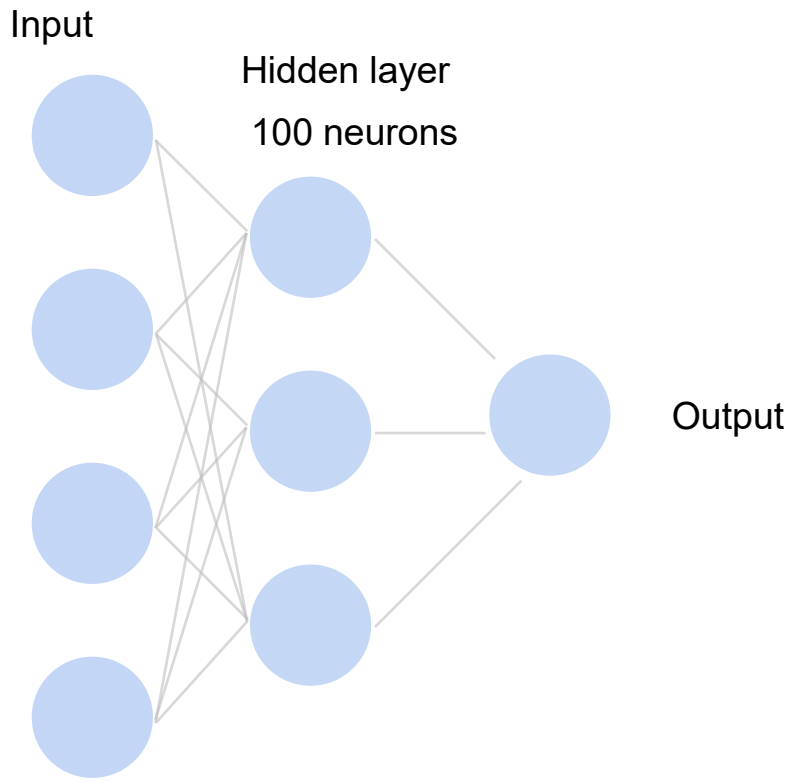
Dual

12MP

wide-angle and telephoto cameras

36M floats in a RGB image!

What if we have images as our inputs?



~ 36M input elements x 100 = **~3.6B** parameters!

Convolutions to the rescue

Convolution layers

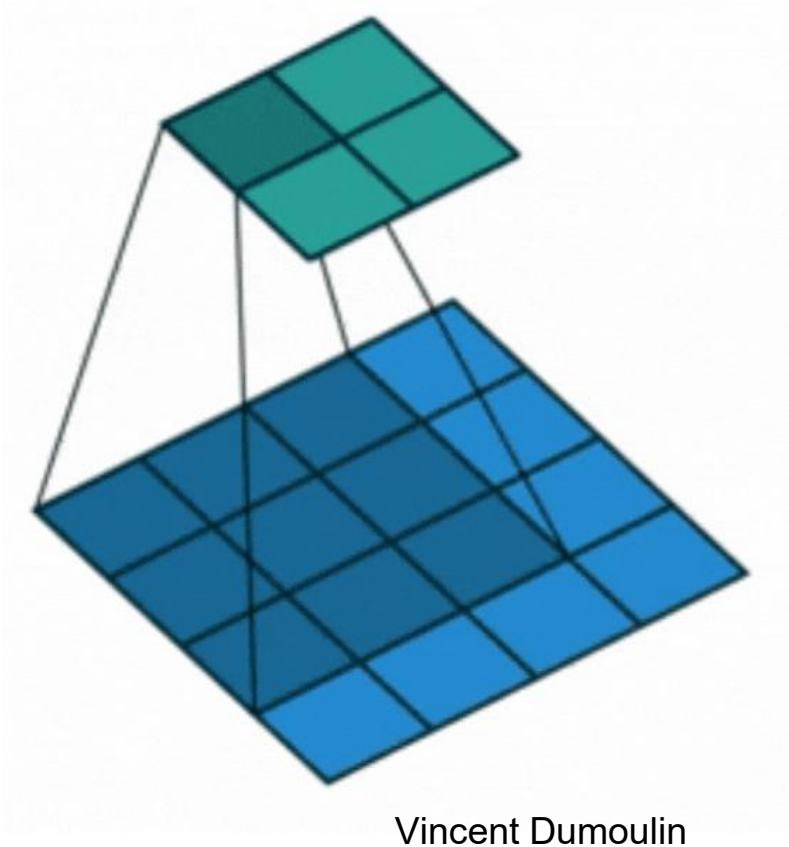
- can process images with varying numbers of pixels
- have a parameter count that doesn't increase with image resolution, unlike $O(wh)$ or more for fully connected layers
- have computational complexity $\tilde{O}(w + h)$ rather than $O(wh)$ or worse for fully connected layers
- are **translation equivariant**, i.e. extract the same feature from a translation of the image

2-D Convolutions

Example:

Input				Kernel			Output	
0	1	2	*	0	1	=	19	25
3	4	5		2	3		37	43
6	7	8						

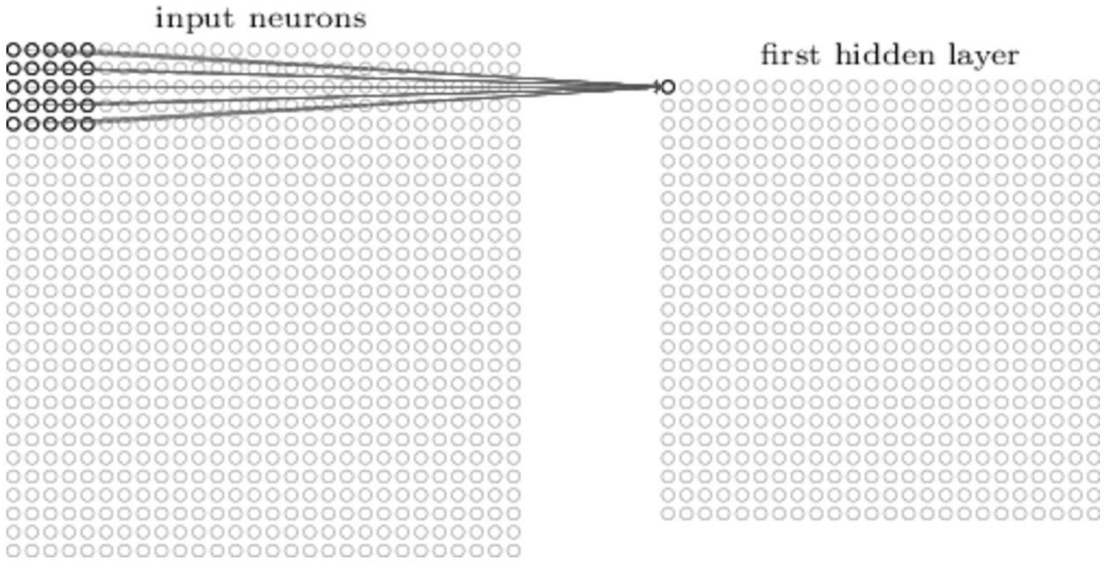
$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

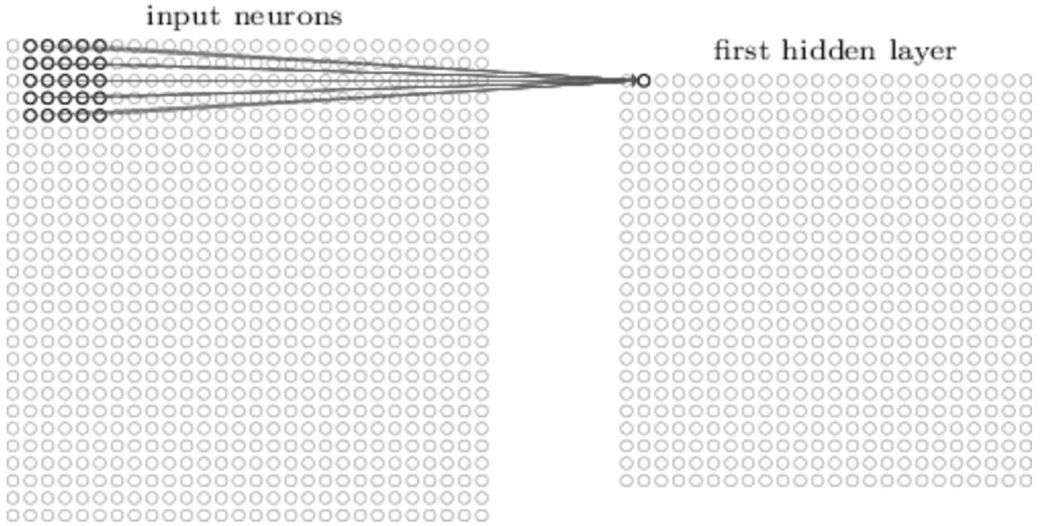


Vincent Dumoulin

Convolution Operation

- All the units used the same set of weights (kernel)
- The units detect the same “feature” but at different locations





neuralnetworksanddeeplearning.com

Kernels: Examples

(Wikipedia)

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Sharpen

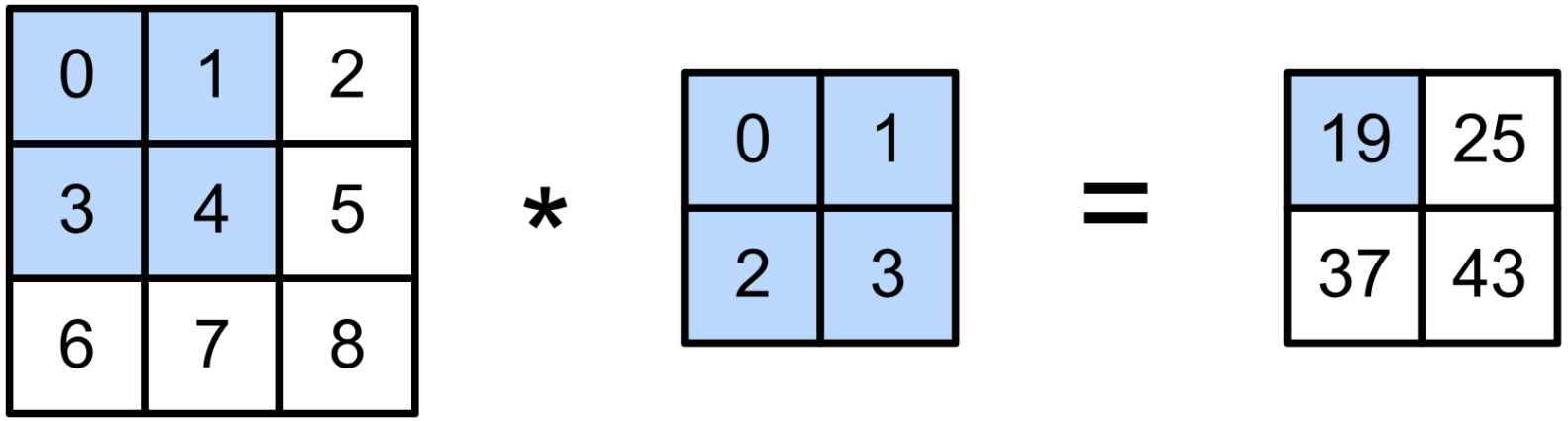
$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Gaussian Blur

Convolution Layers

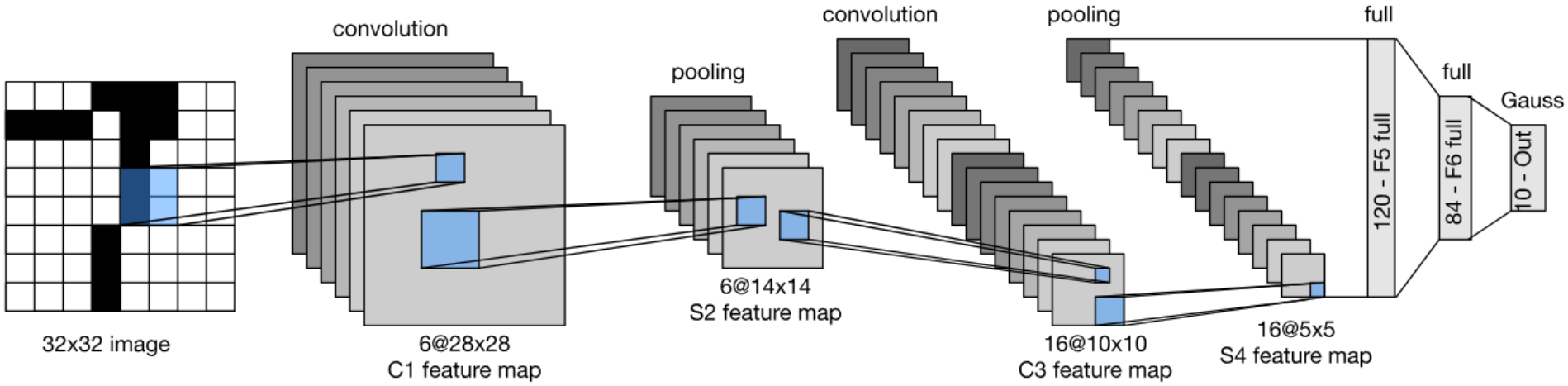
- Notation:
  - *X:* n_h x n_w input matrix
  - *W:* k_h x k_w kernel matrix
  - *b* : bias (a scalar)

- As usual *W*, *b* are learnable parameters



Convolutional Neural Networks

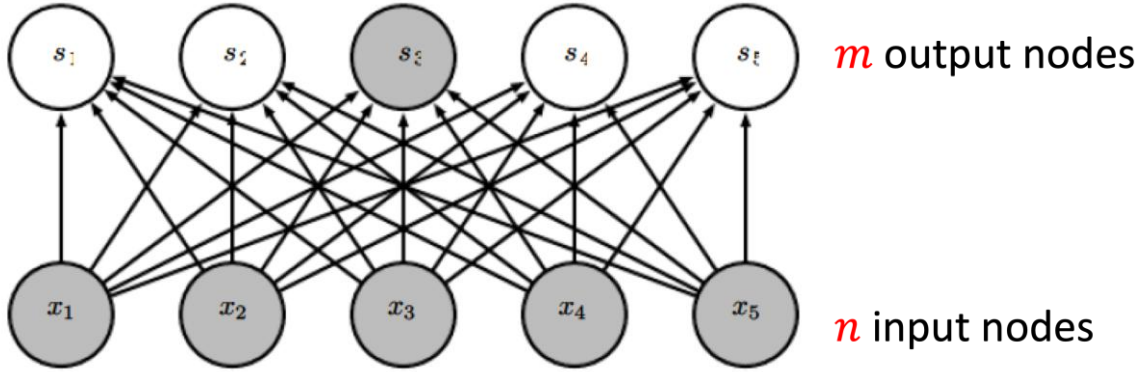
Convolutional networks: neural networks that use *convolution* in place of general matrix multiplication in at least one layer



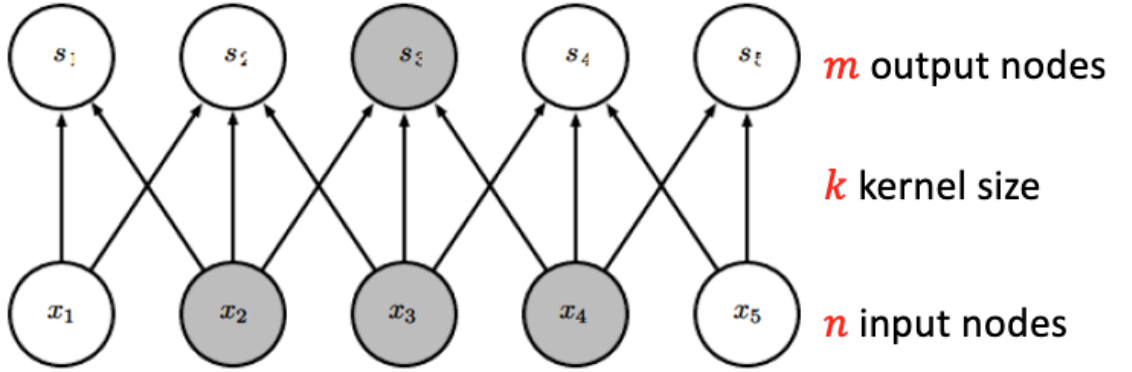
- default approach for image tasks
- still used even in modern Transformer alternatives

CNNs: Advantages

- Fully connected layer: *m* x *n* edges and parameters



- Convolutional layer: ≤ *m* x *k* edges, *k* parameters



Convolutional Layers: Channels

Color images are multi-channel, e.g. RGB:

Convolutional Layers: Channels

How to integrate multiple channels?

- Have a kernel for each channel i, then sum results over c_i channels

$$\mathbf{X} : c_i \times n_h \times n_w$$

$$\mathbf{W} : c_i \times k_h \times k_w$$

$$\mathbf{Y} : m_h \times m_w$$

$$\mathbf{Y} = \sum_{i=0}^{c_i} \mathbf{X}_{i,:,:} \star \mathbf{W}_{i,:,:}$$

Convolutional Layers: Channels

We can also have multiple **output** channels c_o

- have a kernel for each of $c_i \times c_o$ pairs (i, o) of input channel i and output channel o
- output channel o gets the sum over $i = 1, \ldots, c_i$ over the applications of the kernels (i, o)

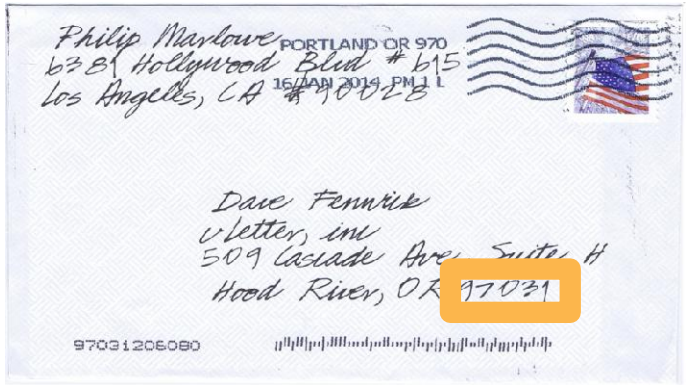
$$\mathbf{X} : c_i \times n_h \times n_w$$

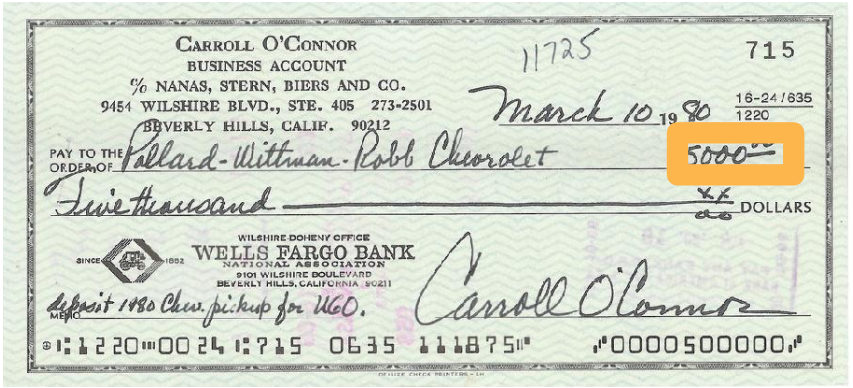
$$\mathbf{W} : c_o \times c_i \times k_h \times k_w \qquad \mathbf{Y}_{i,:,:} = \mathbf{X} \star \mathbf{W}_{i,:,:,:}$$

$$\mathbf{Y} : c_o \times m_h \times m_w$$

CNN Tasks

- Traditional tasks: handwritten digit recognition
- Dates back to the ‘70s and ‘80s
  - Low-resolution images, 10 classes

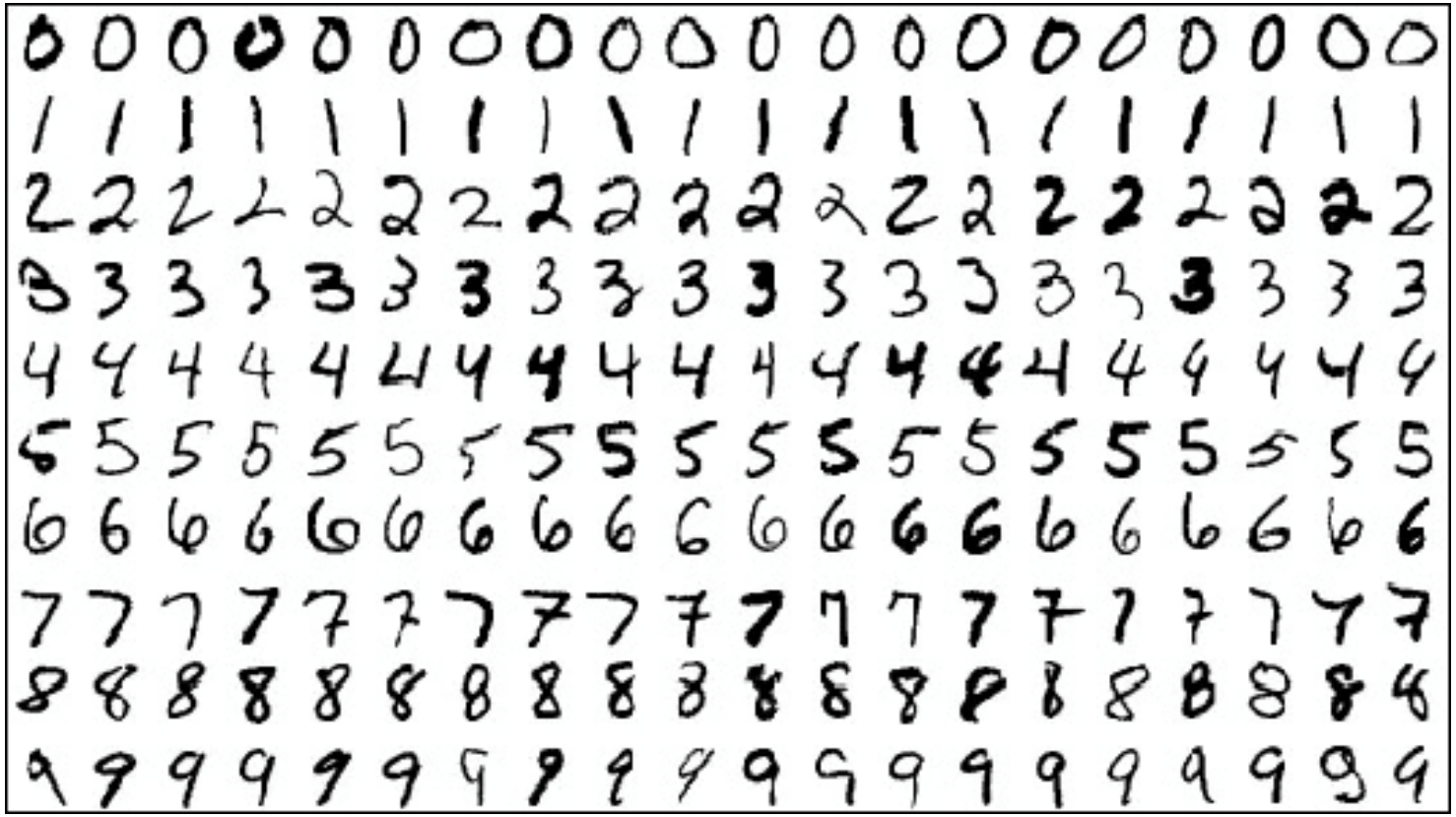




CNN Tasks

- Traditional tasks: handwritten digit recognition
- Classic dataset: MNIST

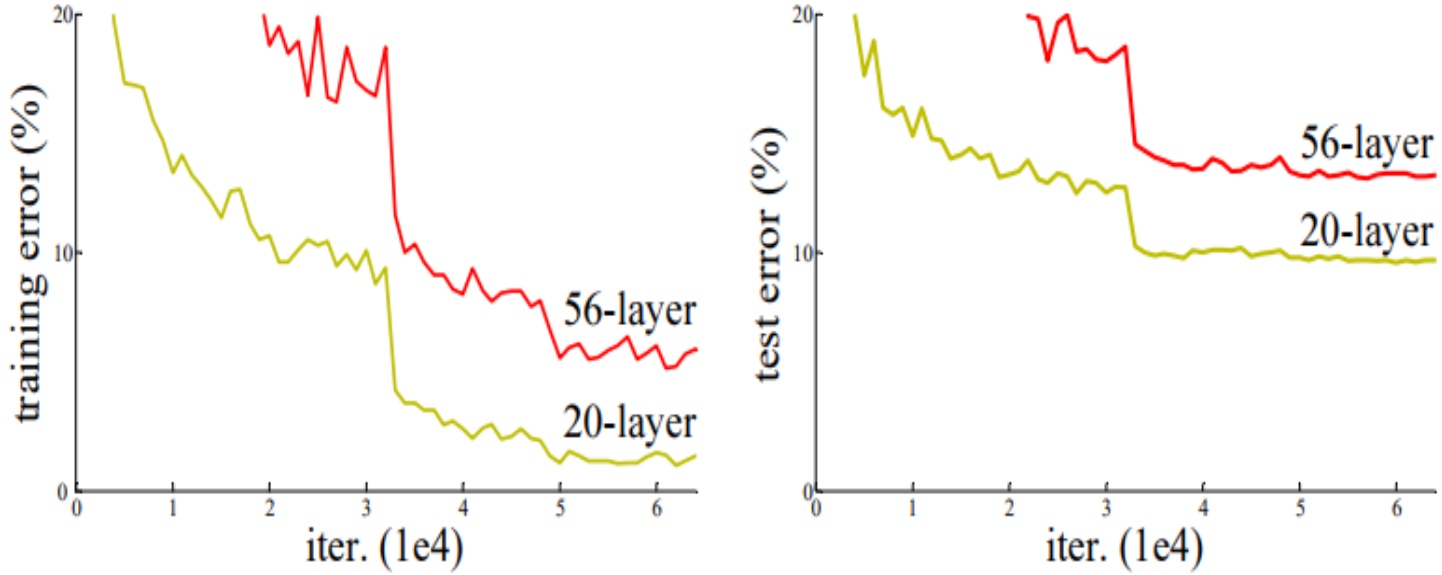
- Properties:
  - 10 classes
  - 28 x 28 images
  - Centered and scaled
  - 50,000 training data
  - 10,000 test data



How to make neural networks deep

Adding too many layers leads to optimization issues:

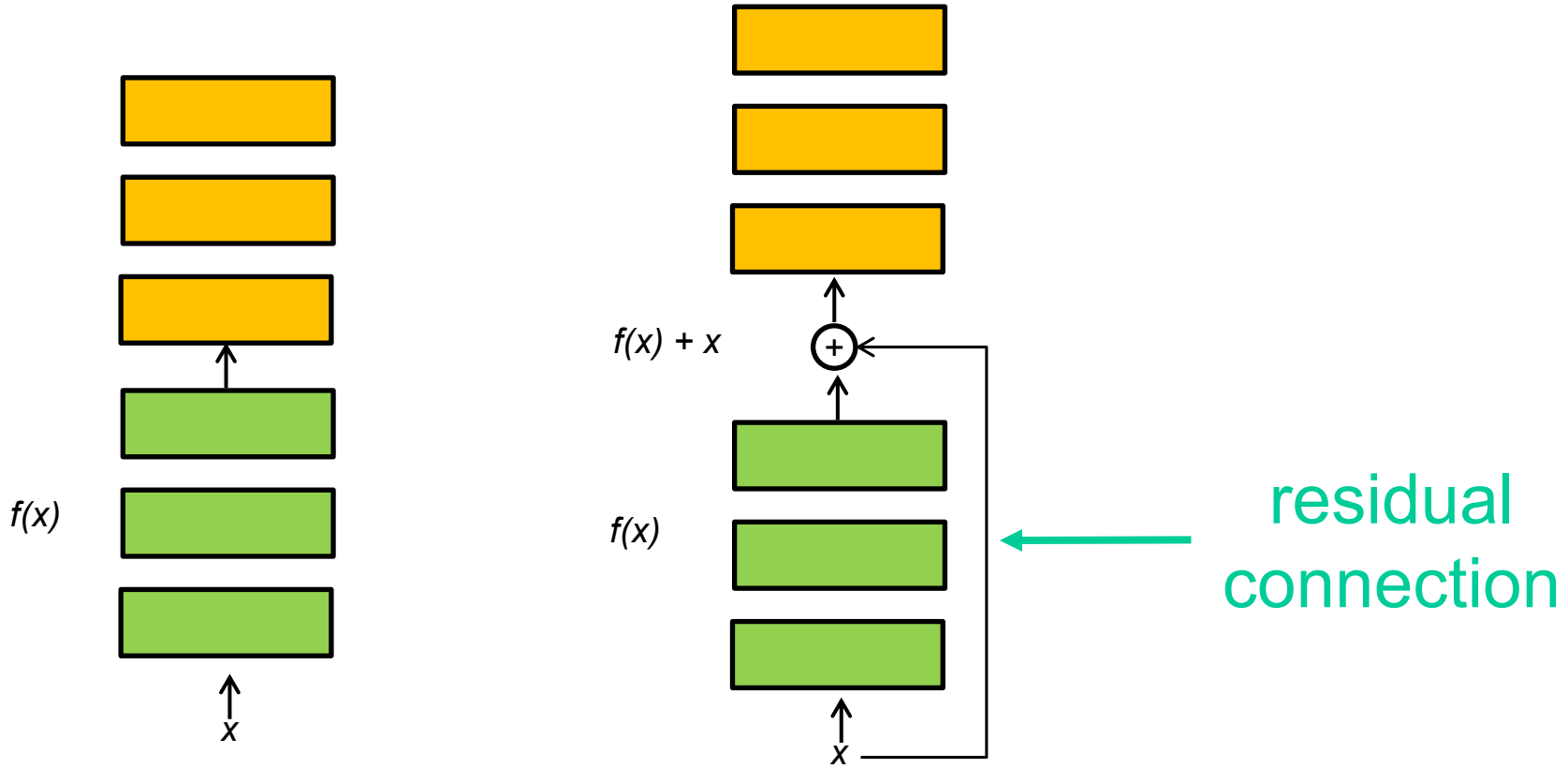
- Vanishing gradients
- Unstable training



He et al: “Deep Residual Learning for Image Recognition”

Residual Connections

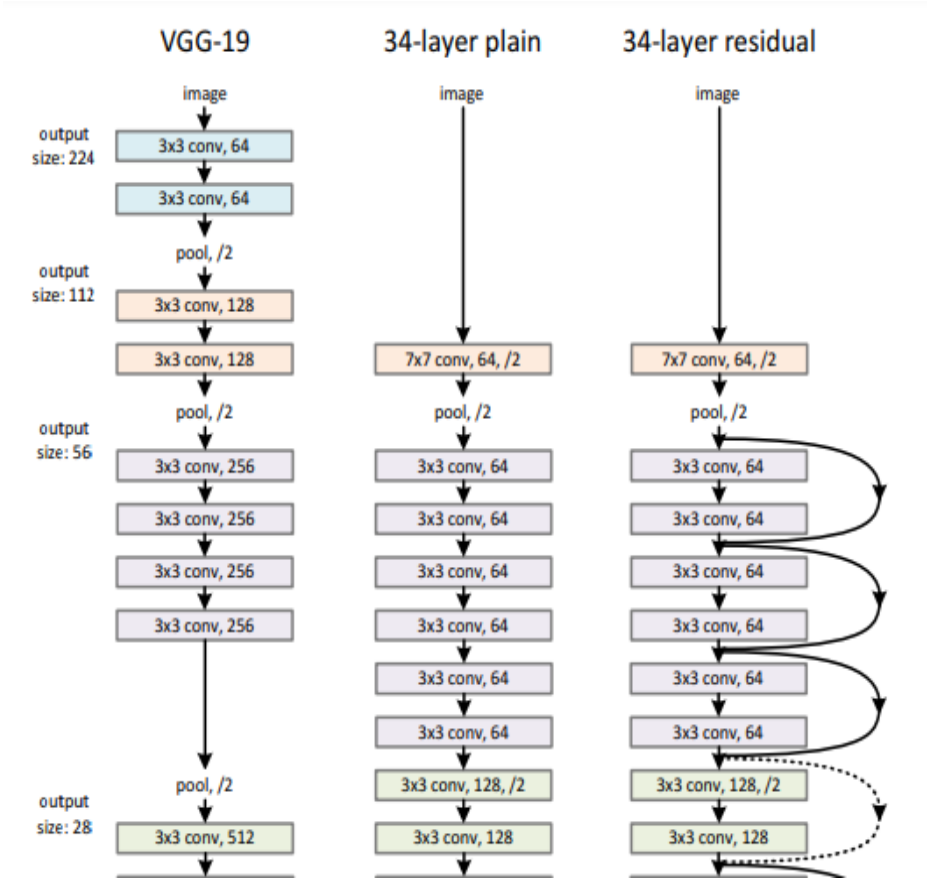
Idea: instead of transforming the input,
learn a correction of the identity



ResNet Architecture

Residual or skip connections help make learning easier

- Vastly better performance
  - No additional parameters!
  - Records on many benchmarks
- Have been used in many other models, including Transformers



He et al: “Deep Residual Learning for Image Recognition”

Tips & Tricks: Initial Pipeline

First step: building a simple pipeline

- Set up data, model training, evaluation loop
- Use a fixed seed
  - Don’t want to get different values each time
- Overfit on one batch
  - Goal: see that we can get zero loss, catch any bugs
- Check training loss: goes down?

Tips & Tricks: Data

Shuffle the training data

- In training ,usually don’t select random examples, but rather go through the dataset for each epoch
- Shuffle to avoid relationships between consecutive points

Pay attention to your data

Tips & Tricks: Learning Rate Schedule

Simple ways:

- Constant
- Divide by a factor ever certain number of epochs (annealing)
- Look at validation loss and reduce on plateau

Also simple: use an optimizer like Adam that internally tracks learning rates

- In fact, per parameter step-size

Lots of variations available

Tips & Tricks: Regularizing

Best thing to do: get more data!

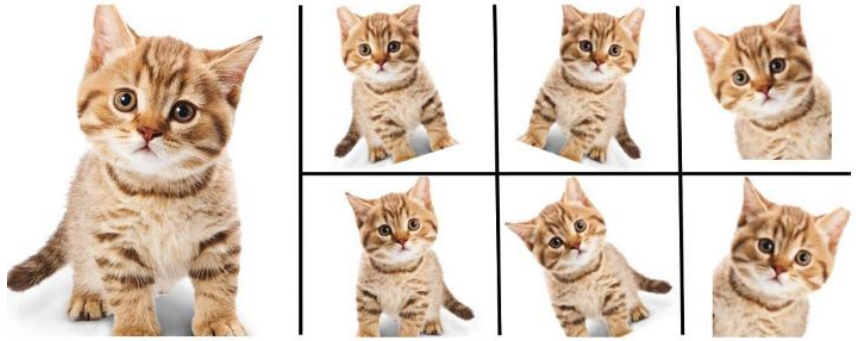
- Not always possible or cheap, but start here.

Augmentation

- But make sure you understand the transformations

Use other strategies: dropout, weight decay, early stopping

- Check each strategy one-at-a-time



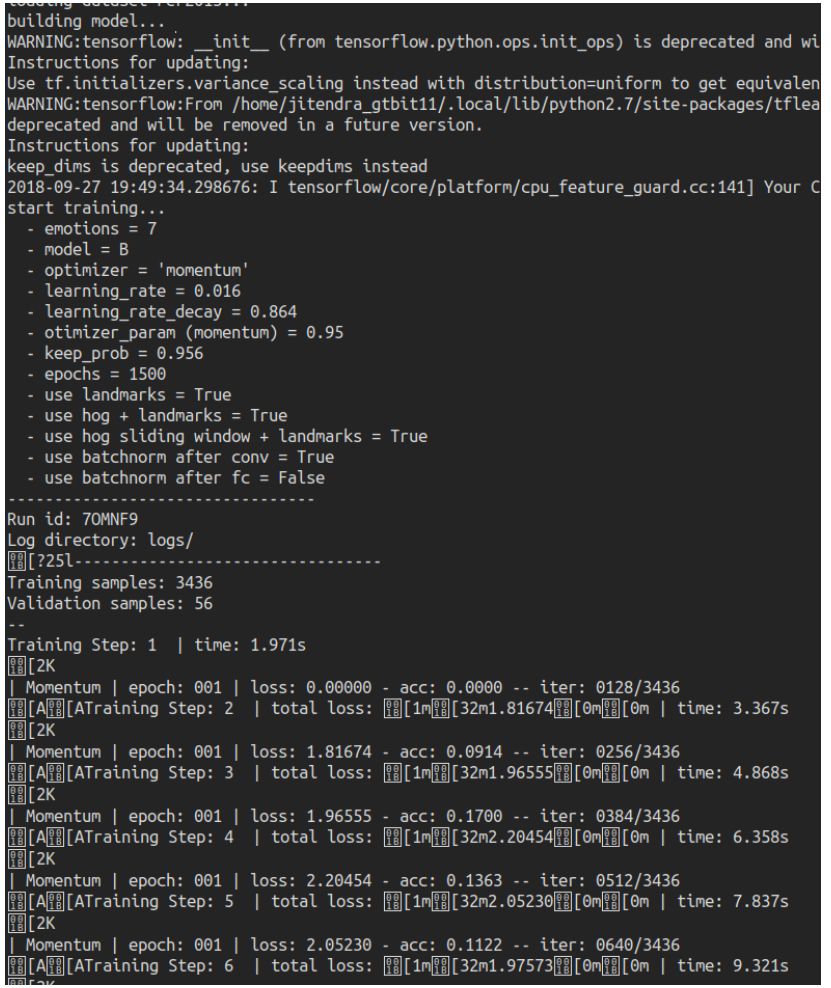
Tips & Tricks: Monitoring & Logging

Checkpoint your models

- Save weights regularly

Log information from training process

- At least keep track of train / test losses, time elapsed, current training settings. Log regularly

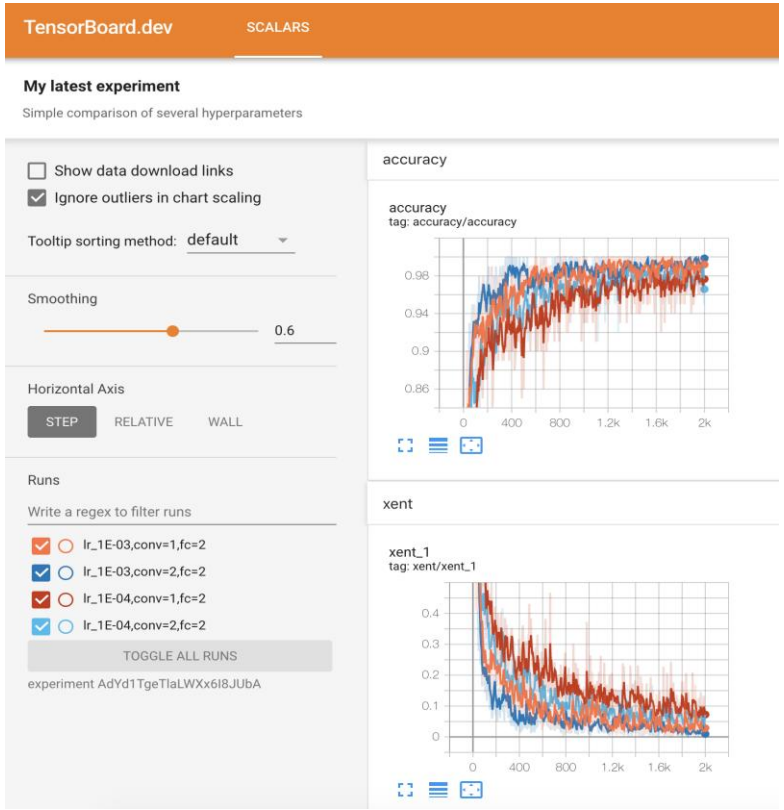


NeptuneAI

Tips & Tricks: Monitoring & Logging

Log information from training process

- Use software packages
- Also have built-in visualization
- Example: TensorBoard, WandB



pytorch.org

Thanks Everyone!

Some of the slides in these lectures have been adapted/borrowed from materials developed by Mark Craven, David Page, Jude Shavlik, Tom Mitchell, Nina Balcan, Elad Hazan, Tom Dietterich, Pedro Domingos, Jerry Zhu, Yingyu Liang, Volodymyr Kuleshov, Fred Sala, Kirthi Kandasamy, Josiah Hanna, Tengyang Xie