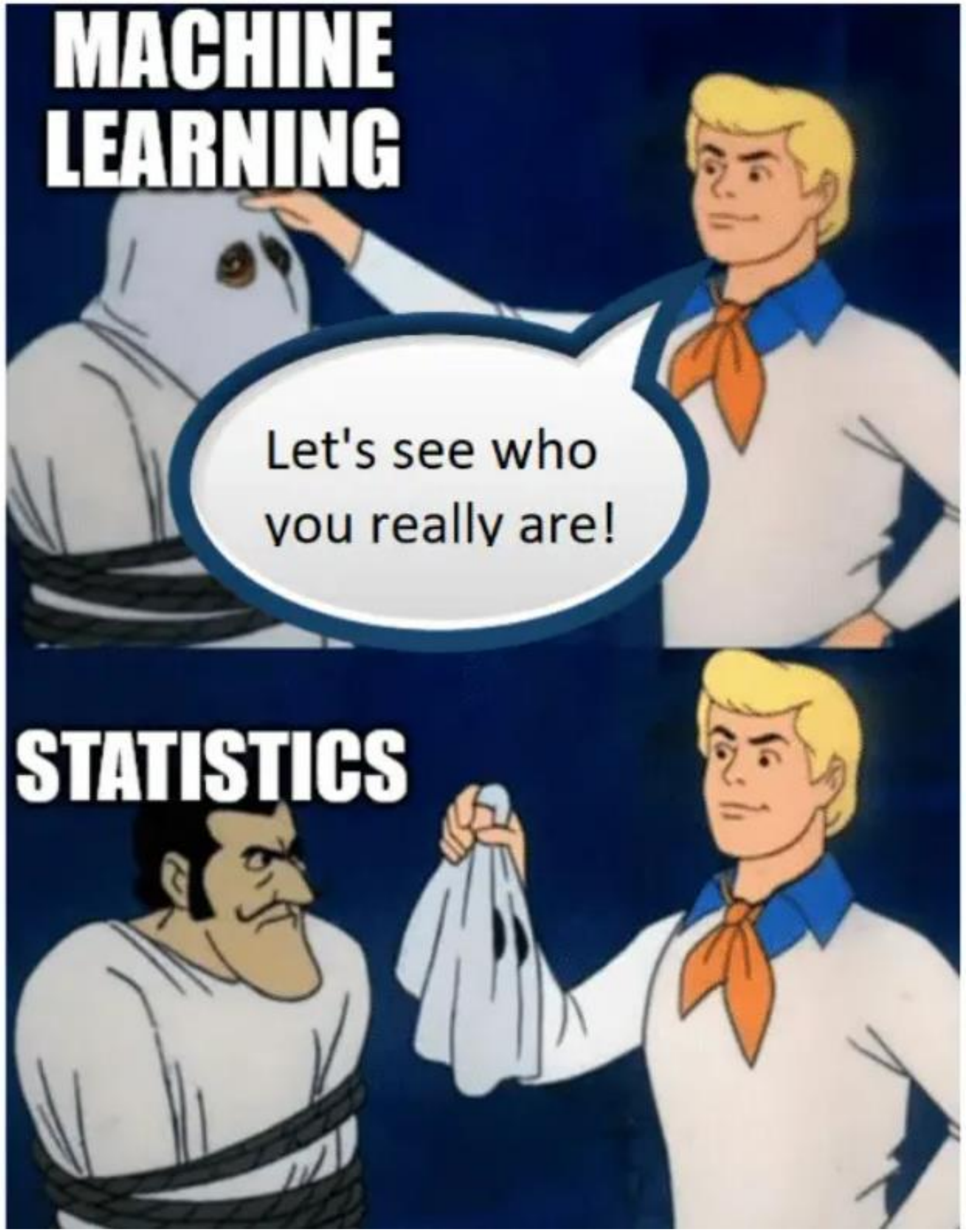
Deep Learning


MACHINE LEARNING
Let's see who you really are!
STATISTICS

General Recipe of Statistical Learning

statistical learning algorithm by combining:

- **model** (e.g., linear function & Gaussian distribution)
- **objective function** (e.g., squared residuals)
- **optimization algorithm** (e.g., gradient descent)
- **regularization** (e.g., L2, dropout)

Loss Function

loss function L: expressing deviation between prediction and target

$L\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{\theta}}\right)$

with $\widehat{\boldsymbol{\theta}}$ corresponding to parameters of model $\hat{f}(\boldsymbol{x})$

e.g., $\widehat{\alpha}, \widehat{\boldsymbol{\beta}}$ in linear regression

e.g., squared residuals (for regression problems):

$L\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{\theta}}\right) = \left(y_i - \hat{f}\left(\boldsymbol{x}_i; \widehat{\boldsymbol{\theta}}\right)\right)^2$

Cost Function

averaging losses over (empirical) training data set:

$$J(\widehat{\boldsymbol{\theta}}) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{\theta}}\right)$$

cost function to be minimized according to model parameters $\widehat{\boldsymbol{\theta}}$

→ objective function

Cost Minimization

minimize training costs $J(\widehat{\boldsymbol{\theta}})$ according to model parameters $\widehat{\boldsymbol{\theta}}$:

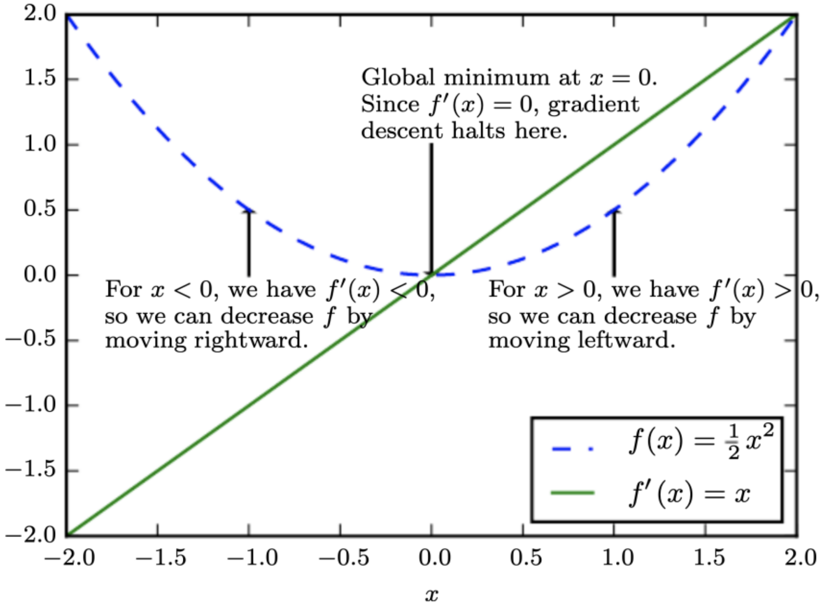
$$\nabla_{\widehat{\boldsymbol{\theta}}}\, J(\widehat{\boldsymbol{\theta}}) = 0$$

for mean squared error (aka least squares method):

$$\nabla_{\widehat{\boldsymbol{\theta}}}\, \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{f}(\boldsymbol{x}_i; \widehat{\boldsymbol{\theta}}) \right)^2 = 0$$

Gradient Descent

$$f(x+\varepsilon) \approx f(x) + \varepsilon \, \frac{df(x)}{dx}$$

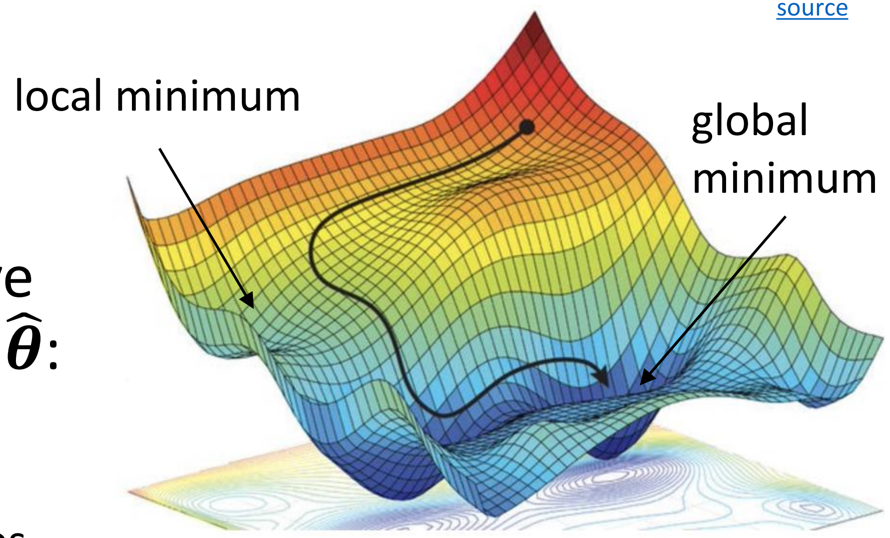


source

usually (except for special cases like ordinary least squares) no closed-form solution to ML optimization problems like minimization of a cost function or maximization of a likelihood function:

$$\nabla_{\widehat{\boldsymbol{\theta}}} J(\widehat{\boldsymbol{\theta}}) = 0$$

→ need for numerical methods



most popular choice: gradient descent

decreasing J by iteratively moving in direction of negative gradient (steepest descent) with respect to input vector $\widehat{\boldsymbol{\theta}}$:

$$\widehat{\boldsymbol{\theta}} \leftarrow \widehat{\boldsymbol{\theta}} - \eta \nabla_{\widehat{\boldsymbol{\theta}}} J(\widehat{\boldsymbol{\theta}})$$

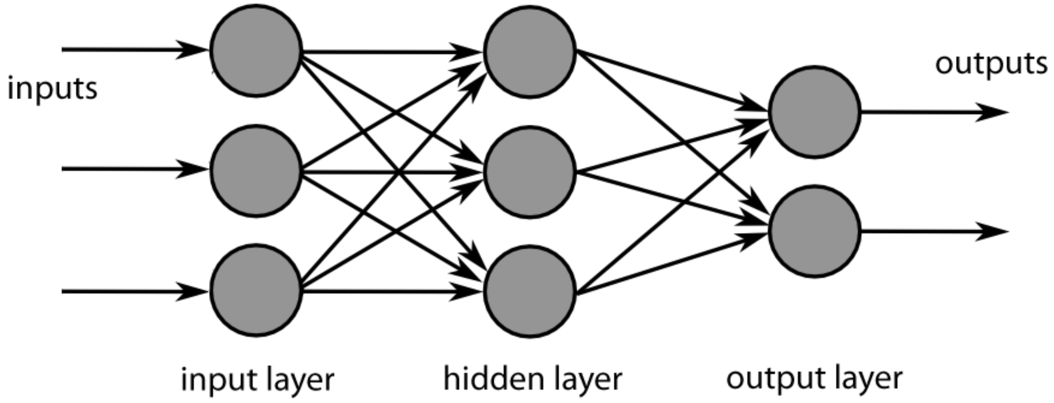
step size
(learning rate)

vector containing all partial derivatives

Neural Networks

idea: powerful ML algorithm by combining many linear building blocks

→ reductionism with complex interactions

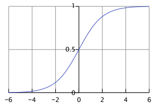
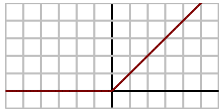


from wikipedia

Artificial Neuron

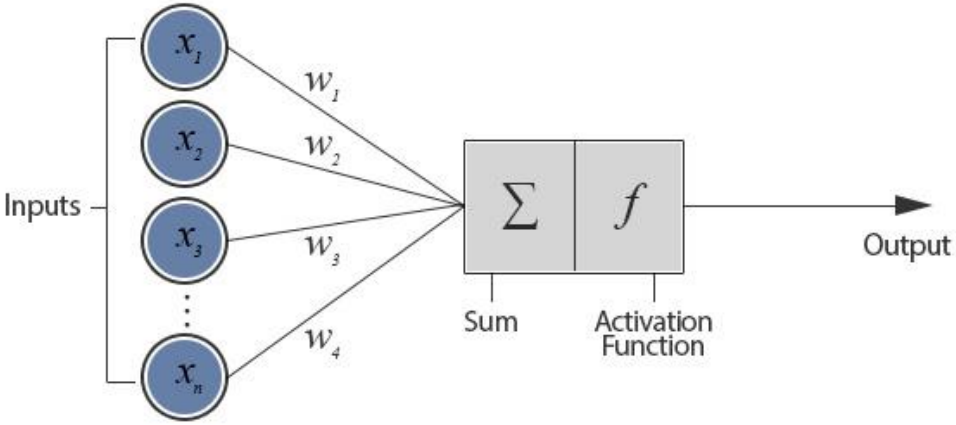
linear model with parameters called weights $\boldsymbol{w}$

non-linear via (differentiable) activation function on sum of inputs times weights

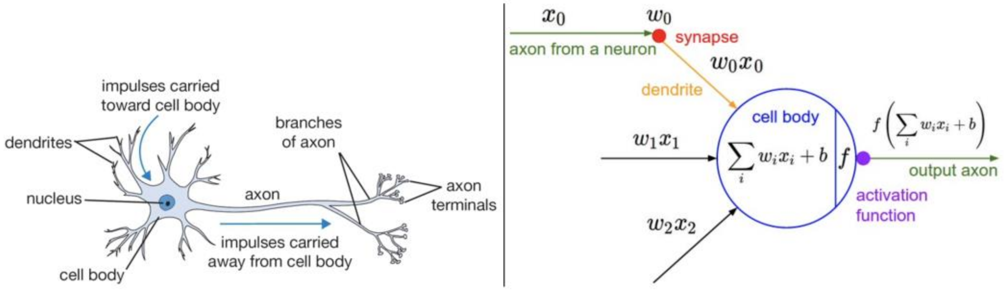
- sigmoid, tanh, … 
- nowadays, mainly ReLU (Rectified Linear Unit), more on this later … 

bias node/input to model intercept

artificial neuron (perceptron or node in neural network):

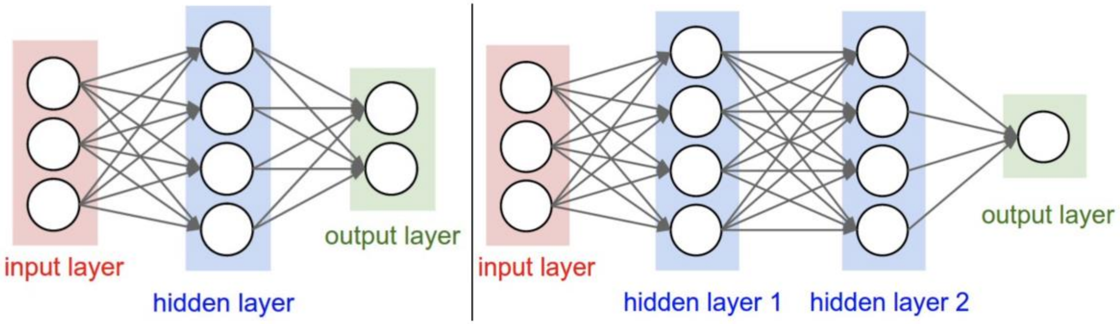


inspired from biological neurons:



Multi-Layer Perceptron (MLP)

fully-connected feed-forward network with at least one hidden layer
→ universal function approximator



toward deep learning: add hidden layers

more layers (depth) more efficient than just more nodes (width): less parameters needed for same function complexity

classification:

- logistic regression in hidden nodes
- cross-entropy loss: $L_i\big(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{w}}\big) = -\sum_{k=1}^{K} y_{ik} \log \hat{f}_k(\boldsymbol{x}_i; \widehat{\boldsymbol{w}})$
- several output nodes k for multi-classification
- softmax output function: $g_k(\boldsymbol{t}_i) = \frac{e^{t_{ik}}}{\sum_{l=1}^{K} e^{t_{il}}}$

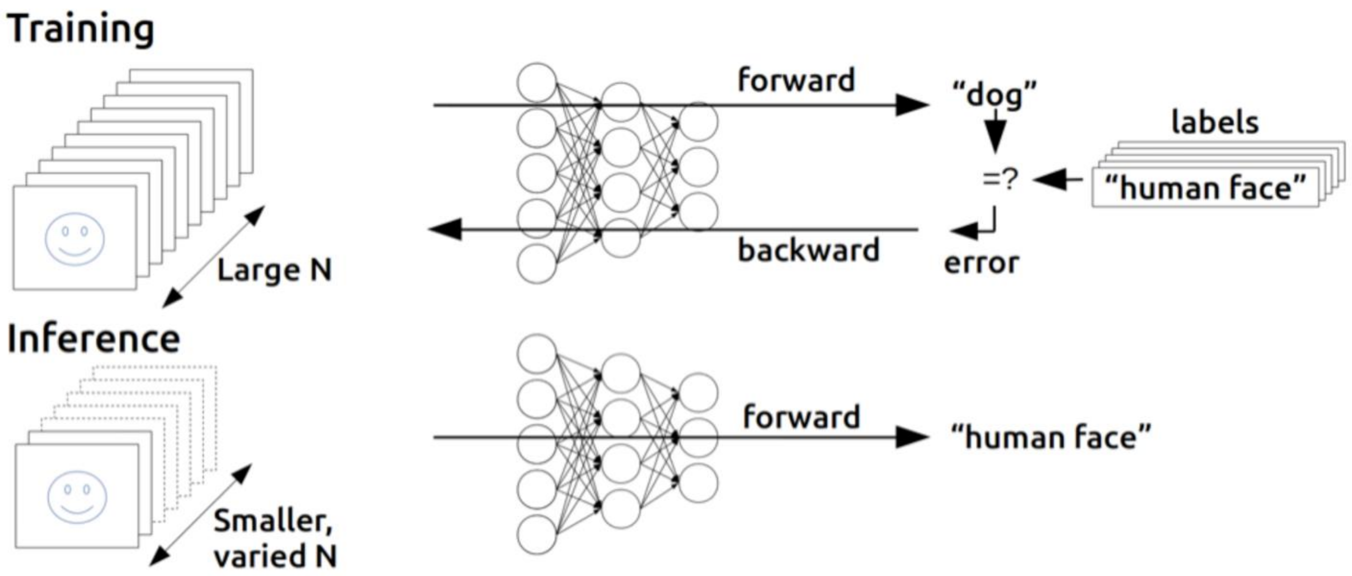
regression:

- squared error loss
- identity output function
- usually just one output node

Find Gradients for (Deep) Neural Networks

back-propagation of errors through layers via chain rule of calculus (avoiding redundant calculations of intermediate terms) → generalization of single-layer delta rule

each node exchanges information only with directly connected nodes → enables efficient, parallel computation



source

- forward pass: current weights fixed, predictions computed
- backward pass: errors computed from predictions and back-propagated, weights updated accordingly, e.g., via gradient descent

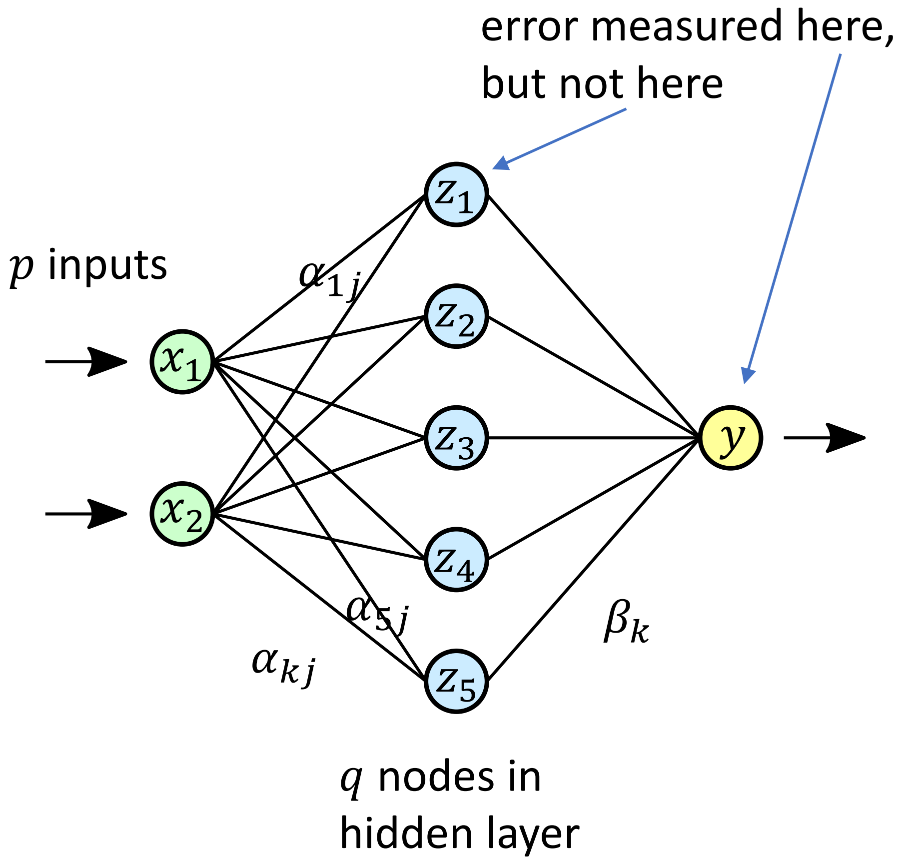
Example WOLOG

- regression (squared error loss, identity output function g)
- with one hidden layer ($\widehat{\boldsymbol{w}}$: $\widehat{\boldsymbol{\alpha}}$, $\widehat{\boldsymbol{\beta}}$)

$$\hat{y}_i = \hat{f}(\boldsymbol{x}_i; \widehat{\boldsymbol{w}}) = g\left(\boldsymbol{z}_i; \widehat{\boldsymbol{\beta}}\right) = \sum_{k=0}^{q} \hat{\beta}_k z_{ik}$$

$$z_{ik} = h(\boldsymbol{x}_i; \widehat{\boldsymbol{\alpha}}_k) = h\left(\sum_{j=0}^{p} \hat{\alpha}_{kj} x_{ij}\right)$$

activation function



cost function:

$$J(\widehat{\boldsymbol{w}}) = \sum_{i=1}^{n} L_i\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{w}}\right) = \sum_{i=1}^{n} \left(y_i - \hat{f}\left(\boldsymbol{x}_i; \widehat{\boldsymbol{\alpha}}, \widehat{\boldsymbol{\beta}}\right)\right)^2$$

Example WOLOG

gradients:

$$\frac{\partial L_i}{\partial \hat{\beta}_k} = -2\left(y_i - \hat{f}(\boldsymbol{x}_i; \widehat{\boldsymbol{w}})\right) z_{ik} = \delta_i z_{ik}$$

$$\frac{\partial L_i}{\partial \hat{\alpha}_{kj}} = -2\left(y_i - \hat{f}(\boldsymbol{x}_i; \widehat{\boldsymbol{w}})\right) \hat{\beta}_k h_k' \left(\sum_{j=0}^{p} \hat{\alpha}_{kj} x_{ij}\right) x_{ij} = s_{ik} x_{ij}$$

→ back-propagation equations (use errors of later layers to calculate errors of earlier ones):

$$s_{ik} = h_k' \left(\sum_{j=0}^{p} \hat{\alpha}_{kj} x_{ij}\right) \hat{\beta}_k \delta_i$$

computed gradients then used, e.g., in gradient descent (r denoting iteration), to update weights:

$$\hat{\beta}_k^{(r+1)} = \hat{\beta}_k^{(r)} - \eta_r \sum_{i=1}^{n} \frac{\partial L_i}{\partial \hat{\beta}_k^{(r)}} \qquad \hat{\alpha}_{kj}^{(r+1)} = \hat{\alpha}_{kj}^{(r)} - \eta_r \sum_{i=1}^{n} \frac{\partial L_i}{\partial \hat{\alpha}_{kj}^{(r)}}$$

learning rate

Using Gradients for Iterative Learning

use gradients found via back-propagation for iterative optimization

usually, by means of (stochastic)gradient descent

- learning rate η_r potentially per iteration adjusted (e.g., via heuristic)
- choose small random weights as starting values to break symmetry

→ back-propagation enables learning of deep neural networks which can encode complex data representations in its hidden layers

→ feature learning on its own

(Stochastic) Gradient Descent

using gradient of cost (objective) function with respect to weights: $\nabla_{\hat{\boldsymbol{w}}} J(\hat{\boldsymbol{w}})$

updates $\hat{\boldsymbol{w}} \leftarrow \hat{\boldsymbol{w}} - \eta \nabla_{\hat{\boldsymbol{w}}} J(\hat{\boldsymbol{w}})$ can be done with whole training data set (n observations) or small random sample:

- $J(\hat{\boldsymbol{w}}) = \frac{1}{n} \sum_{i=1}^{n} J_i(\hat{\boldsymbol{w}})$ batch (or deterministic) gradient descent
- $J(\hat{\boldsymbol{w}}) = J_i(\hat{\boldsymbol{w}})$ stochastic gradient descent (single example)
- $J(\hat{\boldsymbol{w}}) = \frac{1}{m} \sum_{i=1}^{m} J_i(\hat{\boldsymbol{w}})$ mini-batch stochastic gradient descent (size m)

implicit regularization: (mini-batch) SGD follows gradient of true generalization error, if no examples are repeated (but usually many epochs in training)

Mini-Batch Sizes

trade-off:

- larger batches give more accurate gradient estimates → allowing for higher learning rate
- smaller batches have (implicit) regularization effect and better convergence

in practice, also need to consider memory limitations and run times

Embedding Layers

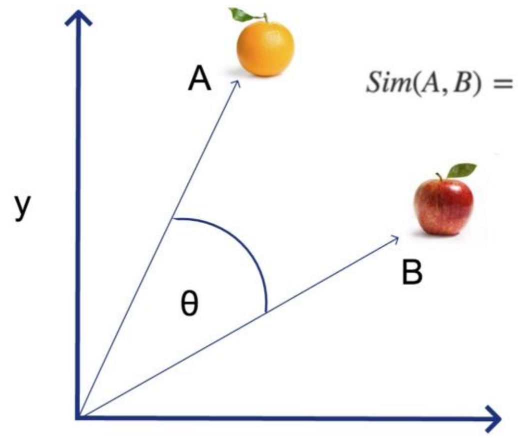
representation of entities by vectors

similarity between embeddings by, e.g., cosine similarity → semantic similarity

most famous application: word embeddings → associations (natural language processing)

but general concept: embeddings of (categorical) features (e.g., products in recommendation engines)

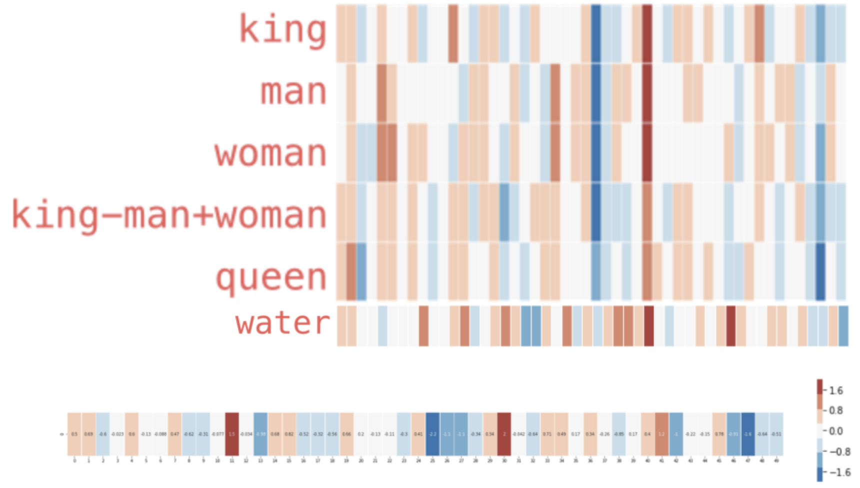
learned via co-occurrence (e.g., word2vec)



$$Sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

but also direction of difference vectors interesting (analogies):

`king – man + woman ~= queen`



source

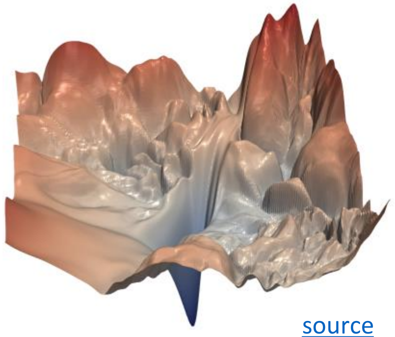
But … How to Train Deep Neural Networks?

optimization and regularization difficult

- non-convex optimization problem (e.g., local vs global minima, saddle points), easily overfitting
- many hyperparameters to tune

many methods to get it working in practice (despite partly patchy theoretical understanding)

typical loss surface:



source

optimization

- activation and loss functions
- weight initialization
- stochastic gradient descent
- adaptive learning rate
- batch normalization

explicit regularization

- weight decay
- dropout
- data augmentation
- weight sharing

implicit regularization

- early stopping
- batch normalization
- stochastic gradient descent

Deep Learning Frameworks

PyTorch, TensorFlow

GPU usage: CUDA

coding example: Kaggle Store Sales with MLP including embeddings