Deep Learning

Fundamentals of Artificial Intelligence

Supervised Learning in Mathematical Terms

map input to output: $y = f(\boldsymbol{x})$ (estimated: $\hat{f}(\boldsymbol{x})$)

random variables Y and $\boldsymbol{X} = (X_1, X_2, \cdots, X_p)$ ← usually high-dimensional

training (curve fitting / parameter estimation):

fit data set of $(y_i, \boldsymbol{x}_i)$ pairs → minimize deviations between y and $\hat{f}(\boldsymbol{x})$

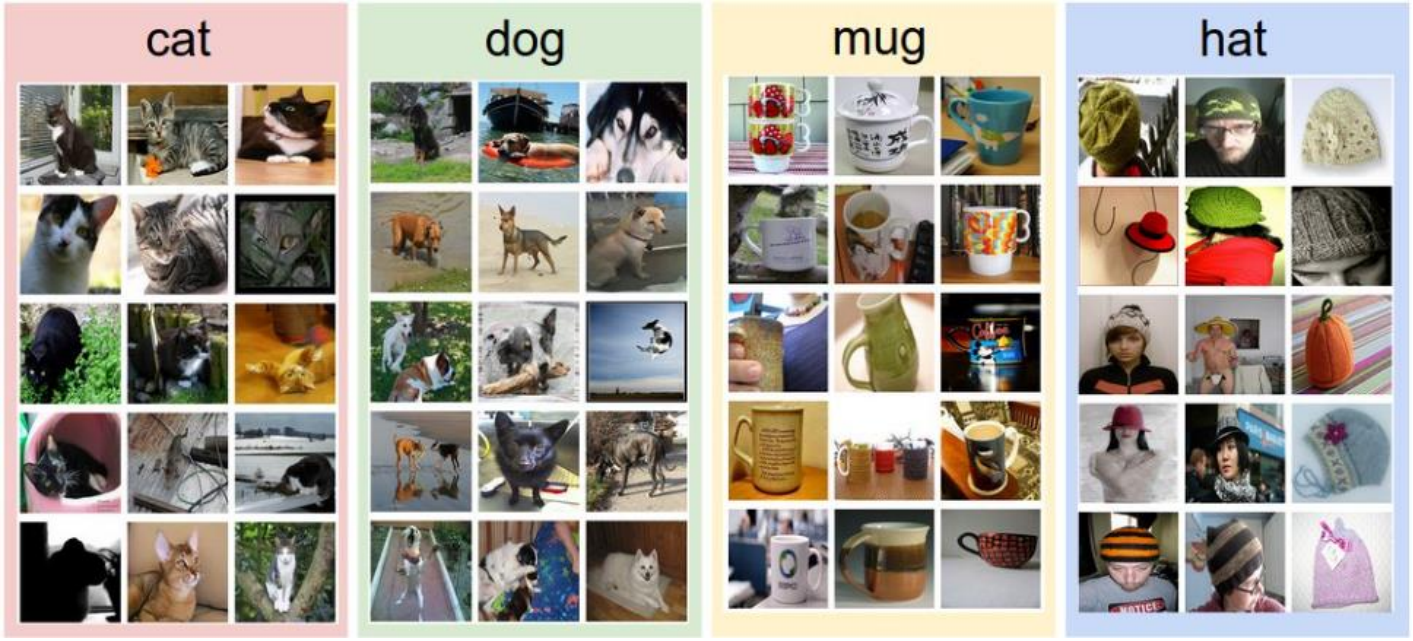
discriminative models: predict conditional density function $p(y|\boldsymbol{x})$

as opposed to generative models: predict $p(y, \boldsymbol{x})$ (or just $p(\boldsymbol{x})$) → $\boldsymbol{x}$ not given, more difficult

Example: Image Classification

training data set:

labels:



test data with learned classifier:

$f($  $) =$ “Cat”

$f($  $) =$ “Dog”

General Recipe of Statistical Learning

ML 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)

Linear Regression Revisited

fit: $\hat{f}(\boldsymbol{x}_i)$

$$y_i = \hat{b} + \sum_{j=1}^{p} \hat{w}_j \, x_{ij} + \varepsilon_i$$

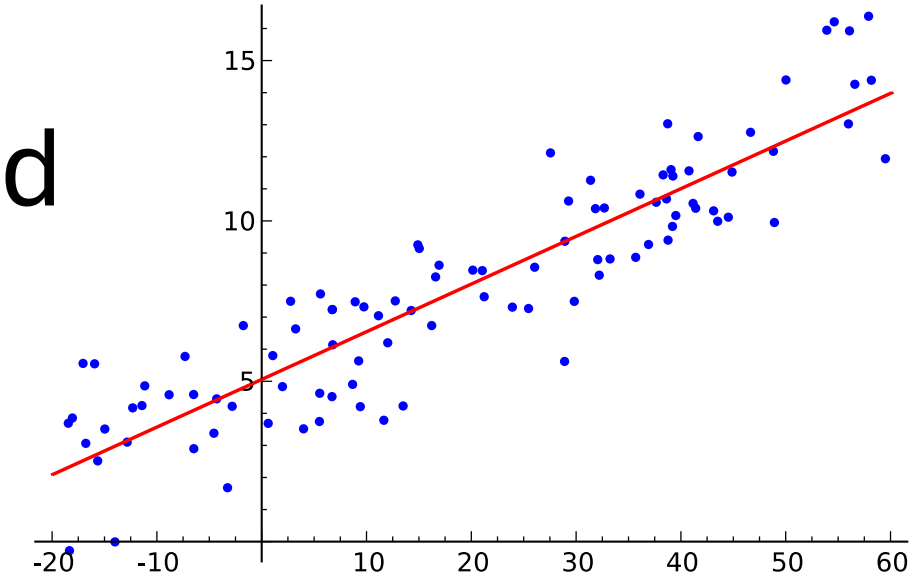
Gaussian

predict:

$$\hat{y}_i = E[Y|\boldsymbol{X} = \boldsymbol{x}_i] = \hat{f}(\boldsymbol{x}_i) = \hat{b} + \sum_{j=1}^{p} \hat{w}_j \, x_{ij}$$

$$p(y|\boldsymbol{x}_i) = \mathcal{N}(y; \hat{y}_i, \hat{\sigma}^2)$$

Gaussian mean variance



parameter estimation:
e.g., least squares

parameters to be estimated: $\hat{b}, \hat{\boldsymbol{w}}$

$\rightarrow \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(\boldsymbol{x}_i)\right)^2$

(approximating assumed true $b, \boldsymbol{w}, \sigma$)

Brief Notation

data: vector with p different feature values for this example i

$$f(\boldsymbol{x}_i) = \boldsymbol{w} \cdot \boldsymbol{x}_i + b$$

vector with slope parameters (one for each of the p features, but identical for all examples), aka weights

single intercept parameter, aka bias

for simplicity of notation, dropping the hats of estimated parameters here

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., $\hat{b}, \widehat{\boldsymbol{w}}$ in linear regression

prime example: 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 $\widehat{\boldsymbol{\theta}}$:

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

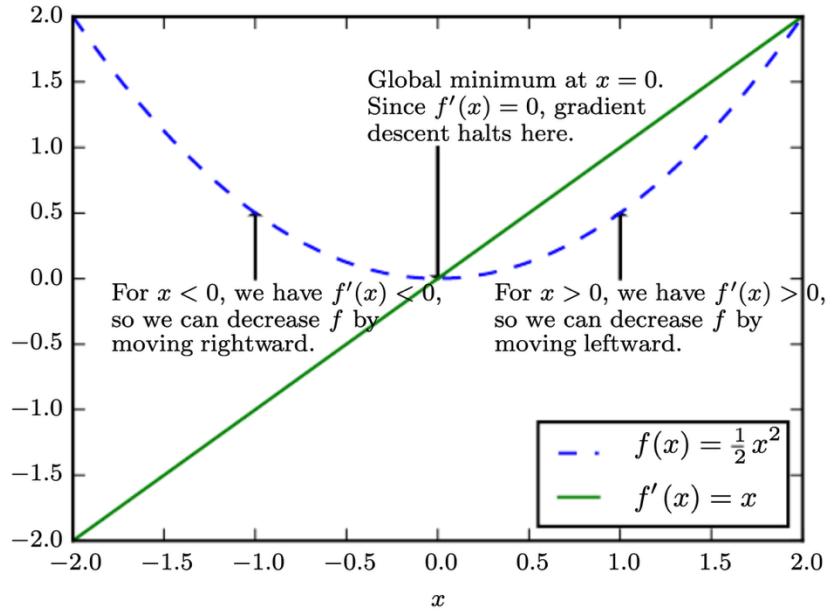
for mean squared error (aka least squares method):

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

Gradient Descent

typically no closed-form solution to minimization of cost function: $\nabla_{\widehat{\boldsymbol{\theta}}} J(\widehat{\boldsymbol{\theta}}) = 0$

→ need for numerical methods



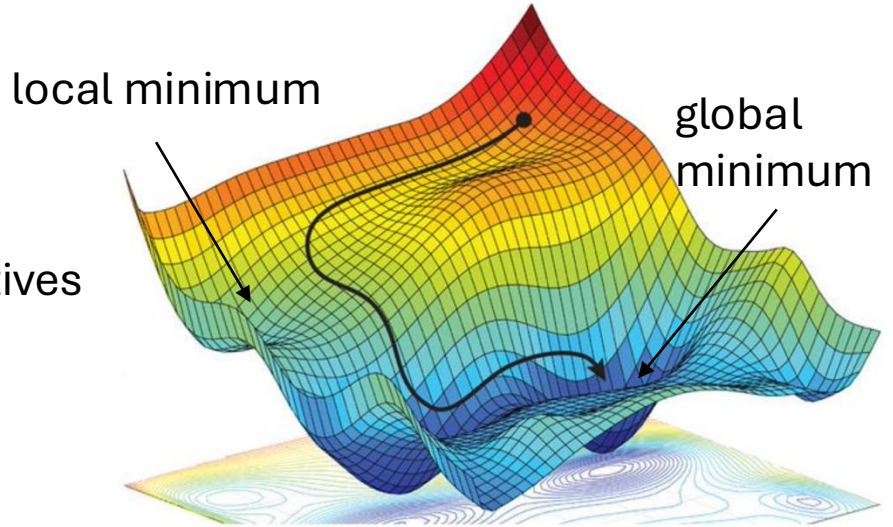
source

popular choice: gradient descent

iteratively moving in direction of steepest descent (negative gradient): $\widehat{\boldsymbol{\theta}} \leftarrow \widehat{\boldsymbol{\theta}} - \eta \nabla_{\widehat{\boldsymbol{\theta}}} J(\widehat{\boldsymbol{\theta}})$

step size (learning rate)

vector containing all partial derivatives



L^2 Regularization (Ridge Regression)

add **penalty term** on parameter L^2 norm to cost function

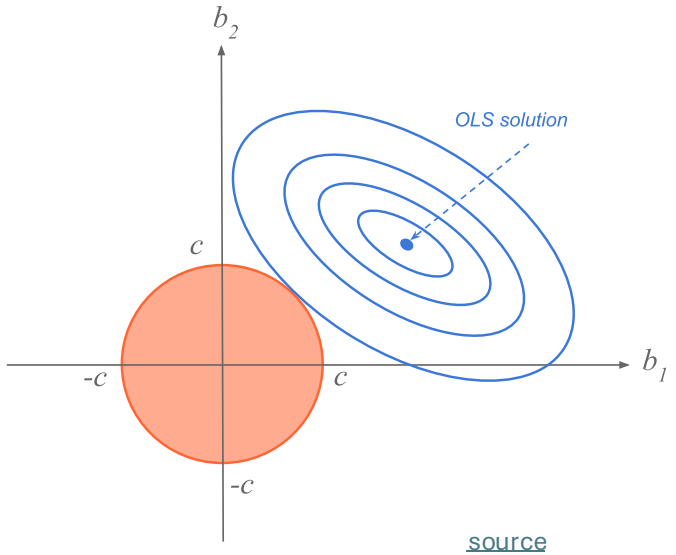
$$\tilde{J}(\hat{\boldsymbol{\theta}}) = J(\hat{\boldsymbol{\theta}}) + \lambda \cdot \hat{\boldsymbol{\theta}}^{\mathrm{T}} \hat{\boldsymbol{\theta}}$$

hyperparameter controlling strength of regularization

aka **weight decay** in neural networks

corresponds to imposing constraint on parameters to lie in L^2 region (size controlled by λ)

goal: reduce overfitting



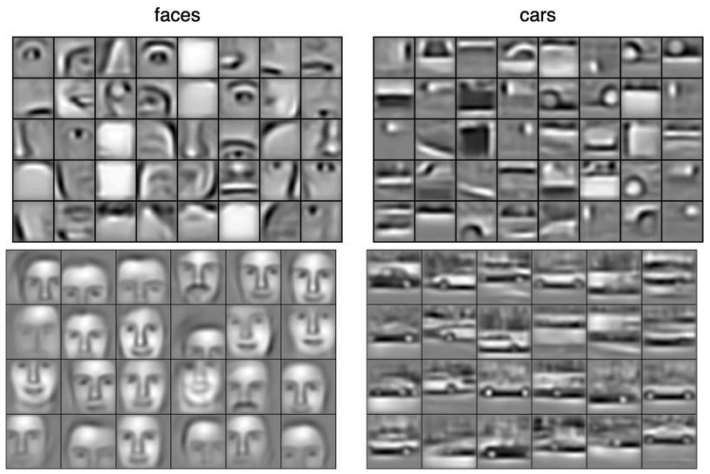
source

Ladder of Generalization

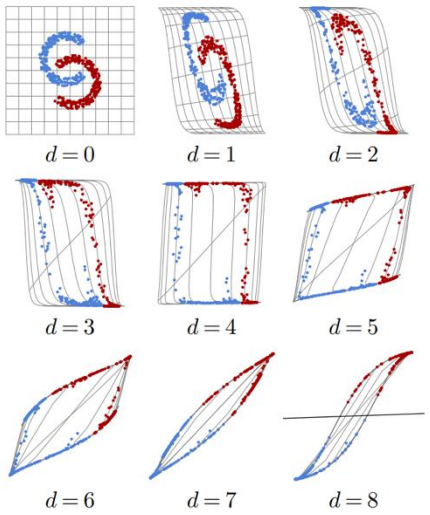
classic ML: feature engineering

deep learning: feature learning

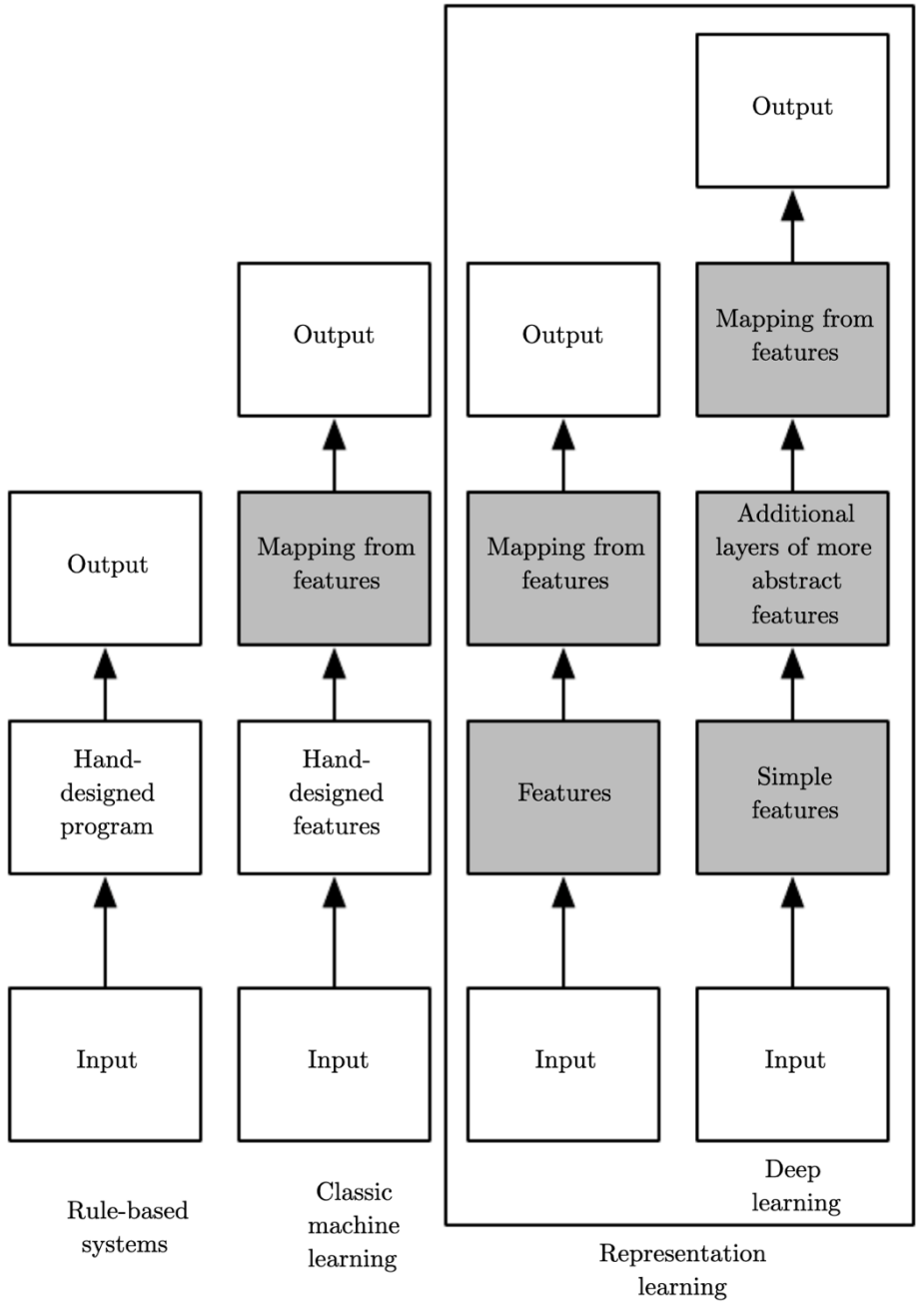
(hierarchy of concepts learned from raw data in deep graph with many layers)



source



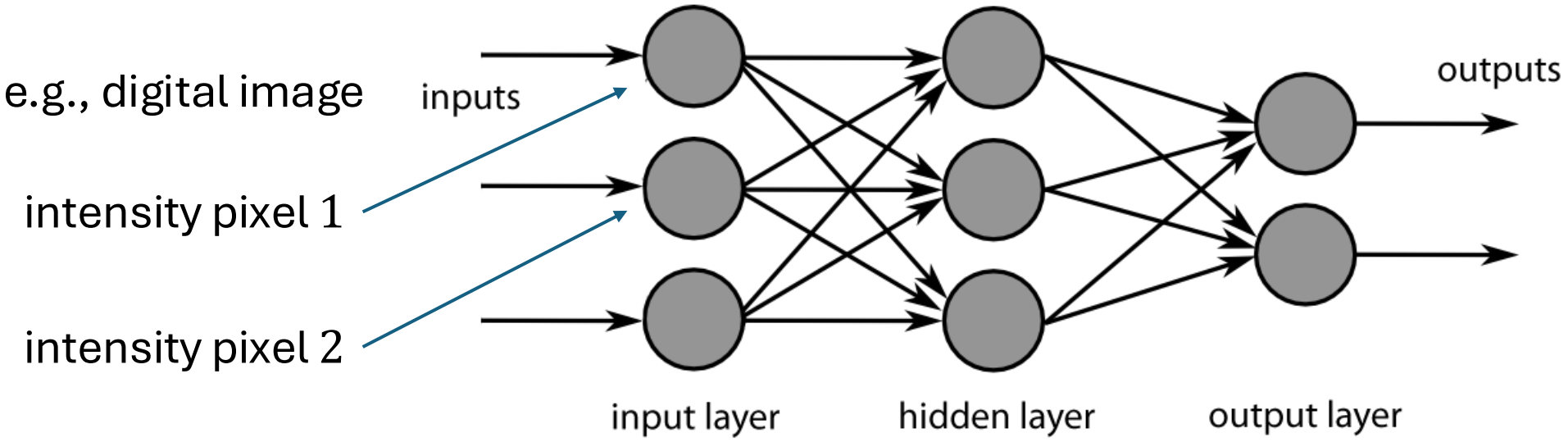
source



source

Neural Networks

idea: powerful algorithm by combining many linear building blocks



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

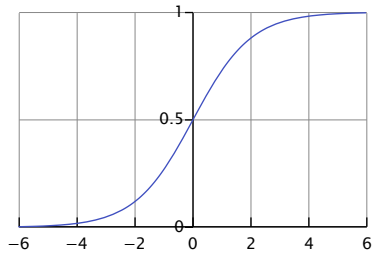
Artificial Neuron

linear model with parameters called weights $\boldsymbol{w}$ (including bias, not shown for simplicity)

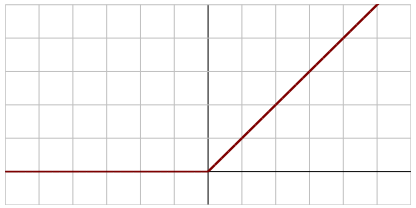
non-linear via (differentiable) activation function on top of linear model

examples:

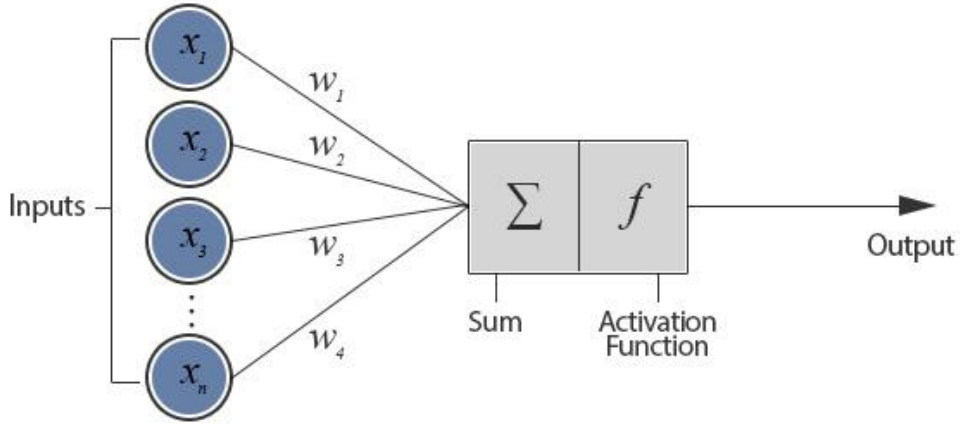
- sigmoid



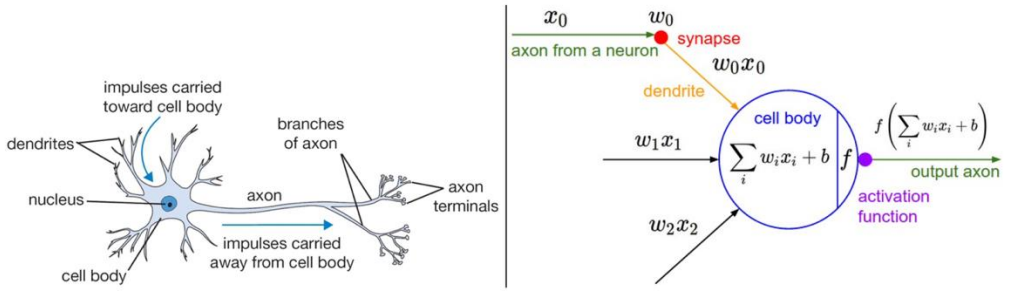
- ReLU (Rectified Linear Unit)



artificial neuron (node in neural network):

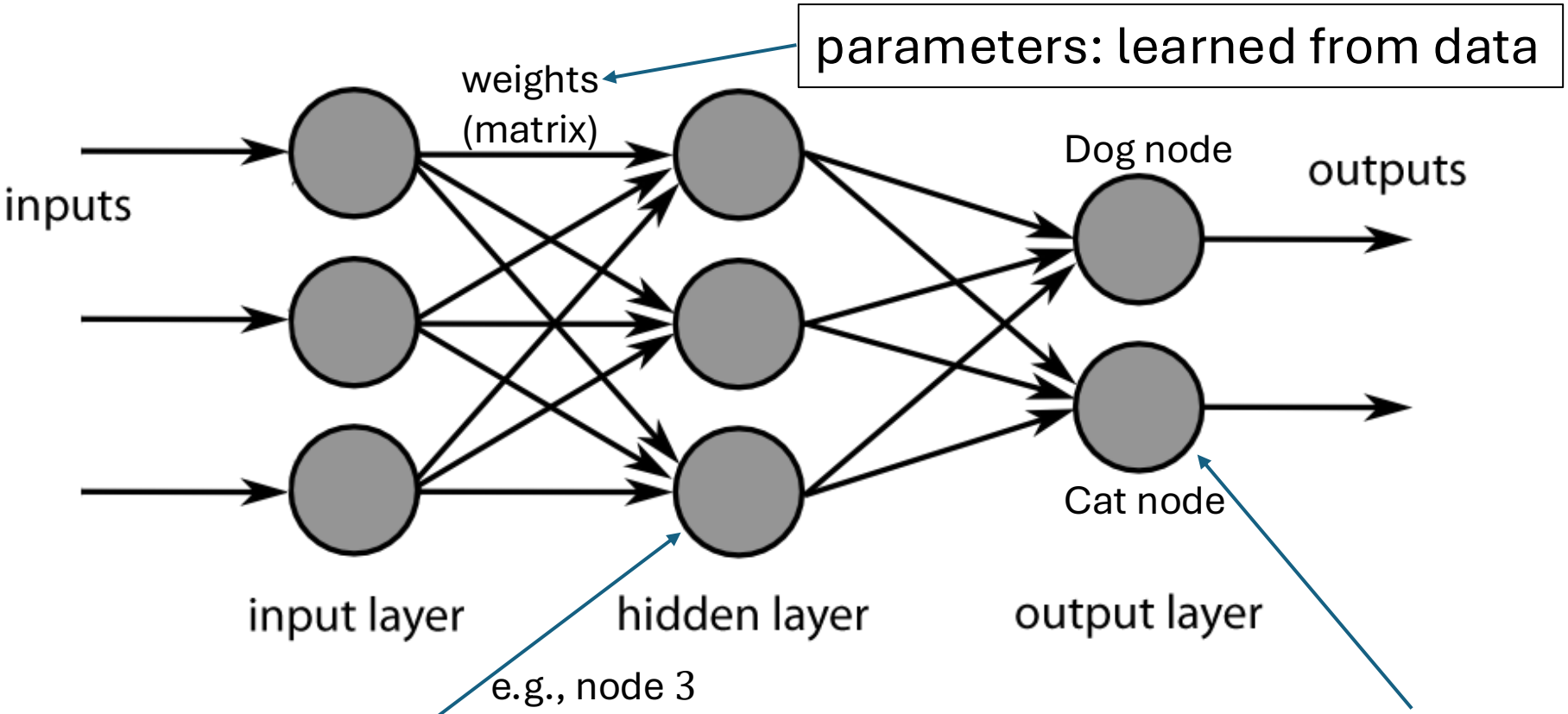


inspired from biological neurons:

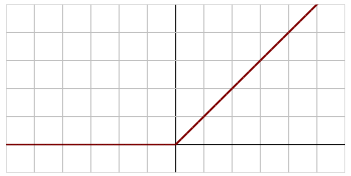


source

Neural Network as Nonlinear Function



linear transformation with nonlinear activation functions

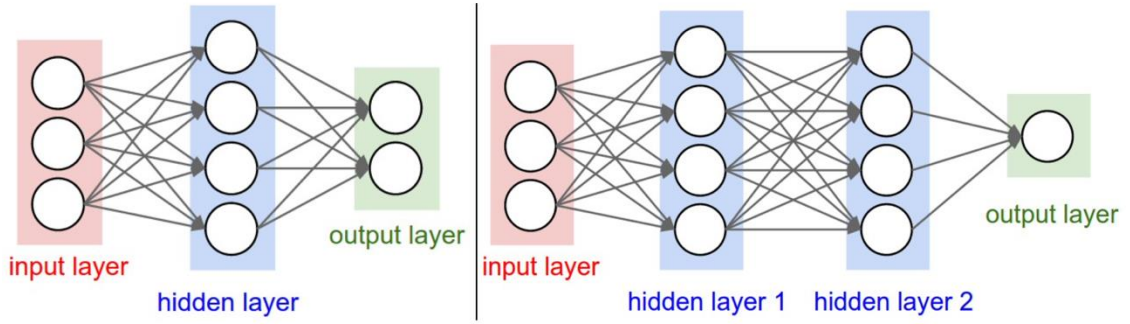


$$\left(\overrightarrow{w_3} \cdot \vec{x}\right)$$

Multi-Layer Perceptron (MLP)

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

binary classification



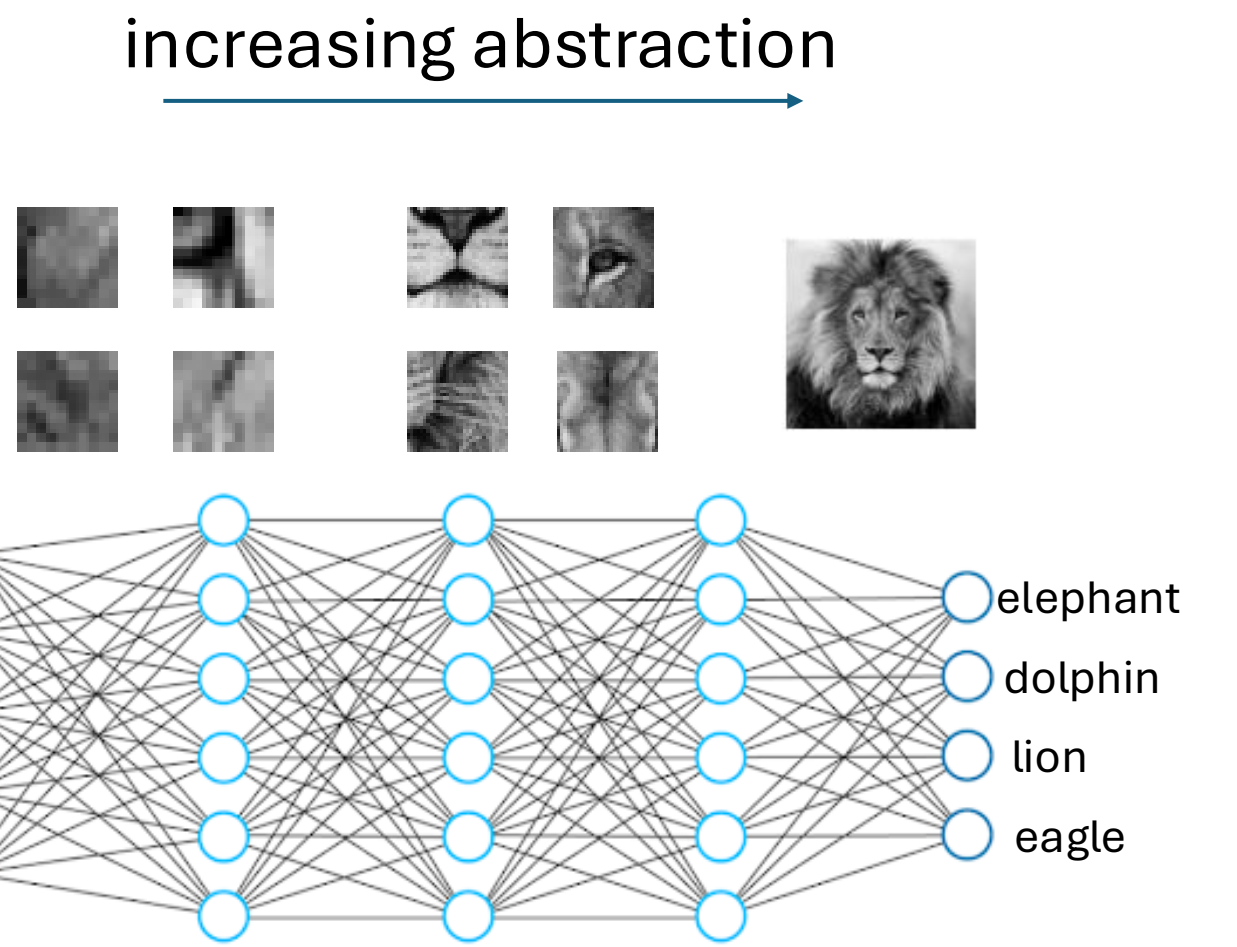
regression

toward deep learning: add hidden layers

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

Hierarchical Representation

deep learning:
several hidden layers



Classification: Logistic Regression

binary classification: $y = 1$ or $y = 0$

→ predict probabilities p_i for $y = 1$

- logit (log-odds) as link function
- Y following Bernoulli distribution

$$\text{logit}(E[Y|\boldsymbol{X} = \boldsymbol{x}_i]) = \ln\left(\frac{p_i}{1 - p_i}\right) = \boldsymbol{w} \cdot \boldsymbol{x}_i + b$$

Multi-Classification: Softmax

for classification in K categories:

- use “score” function with K outputs

$$f_k(\boldsymbol{x}_i) = \boldsymbol{w}_k \cdot \boldsymbol{x}_i + b_k$$

- and convert into probabilities by means of softmax function

$$\frac{\exp\left(f_k(\boldsymbol{x}_i)\right)}{\sum_{l=1}^{K} \exp\left(f_l(\boldsymbol{x}_i)\right)}$$

Classification Networks

cross-entropy loss (one output node for each class k):

$$L(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

using softmax on output nodes:

$$\hat{y}_k = \frac{e^{o_k}}{\sum_{l=1}^{K} e^{o_l}}$$

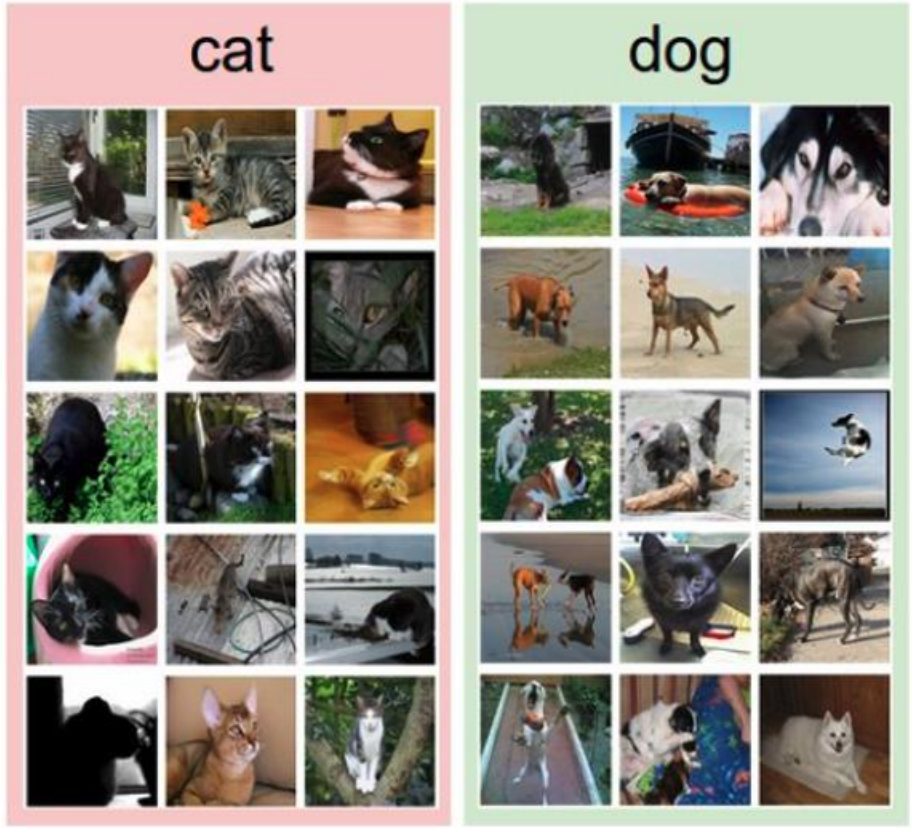
→ prediction of probabilities

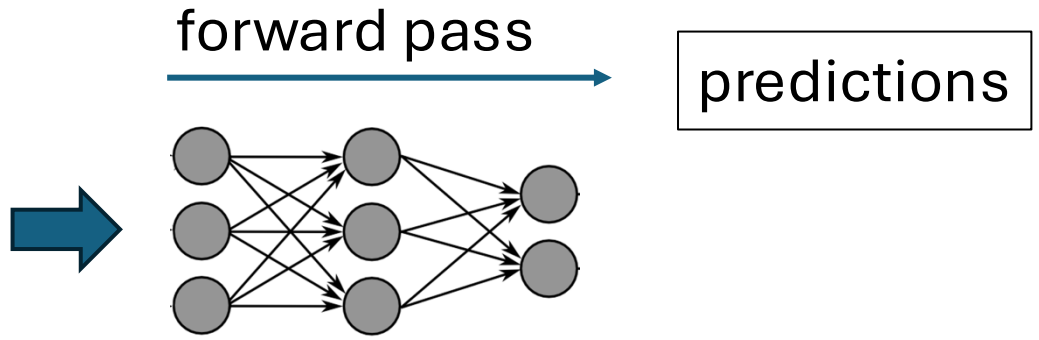
regression networks:

- squared error loss: $L(y, \hat{y}) = (y - \hat{y})^2$
- just one output node
- no function on output → prediction of real numbers

Training with Labeled Images

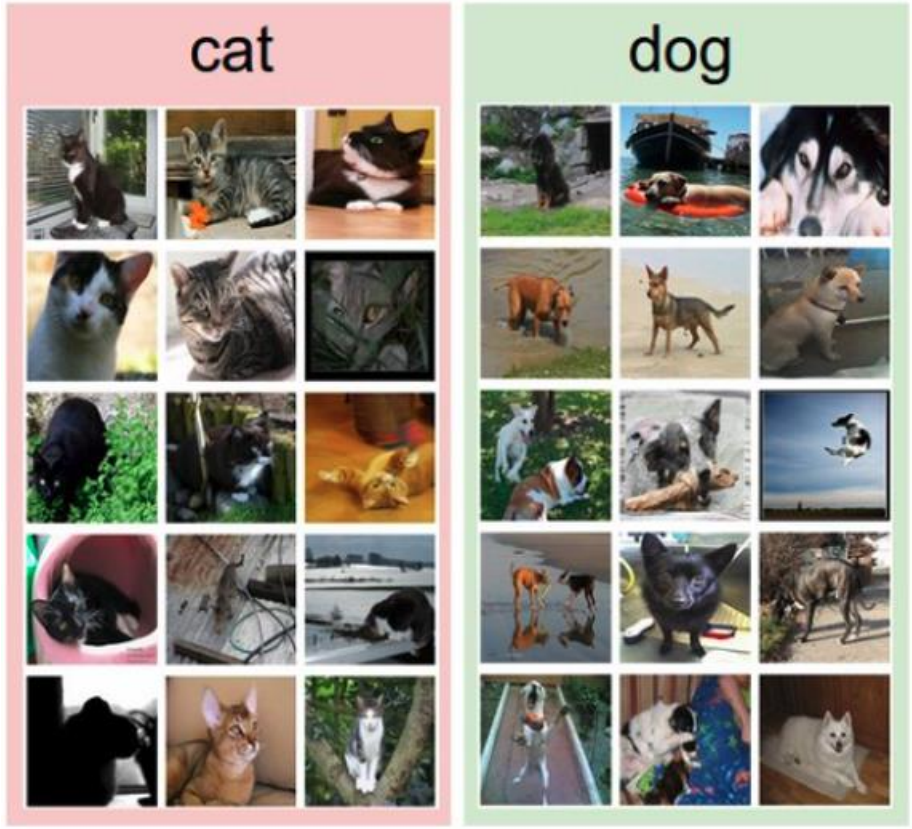
label:

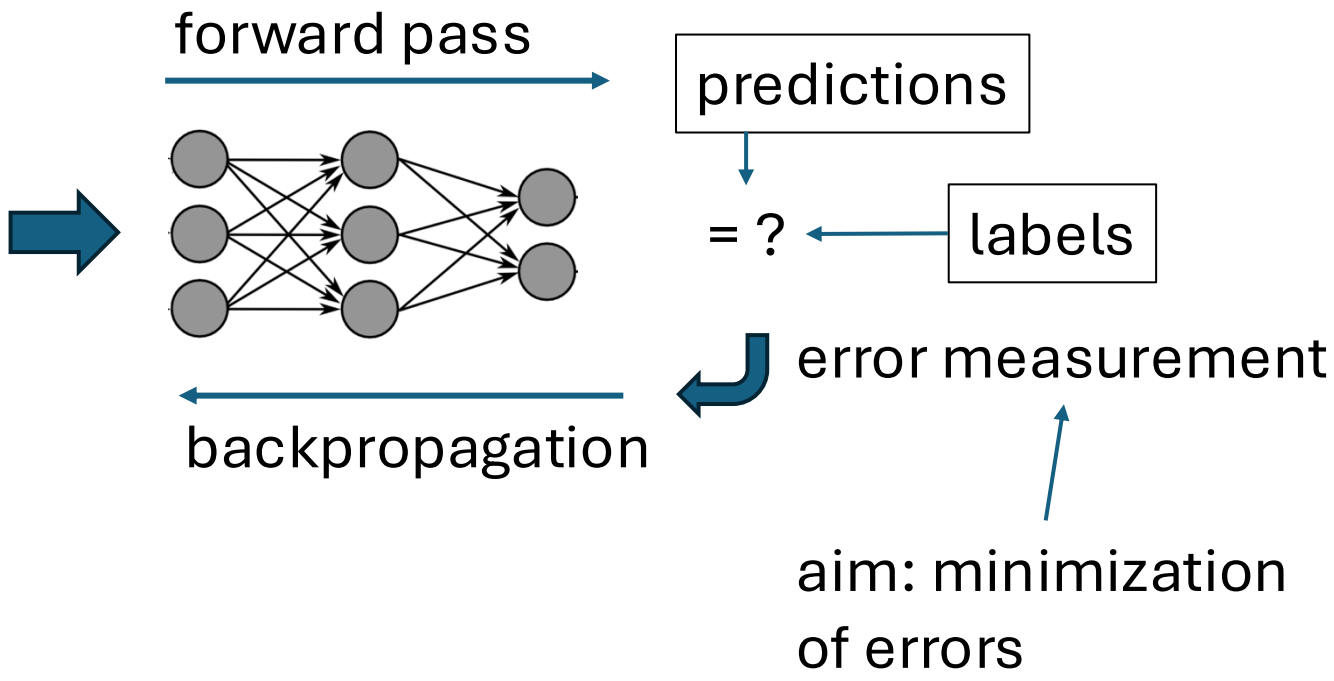




Training with Labeled Images

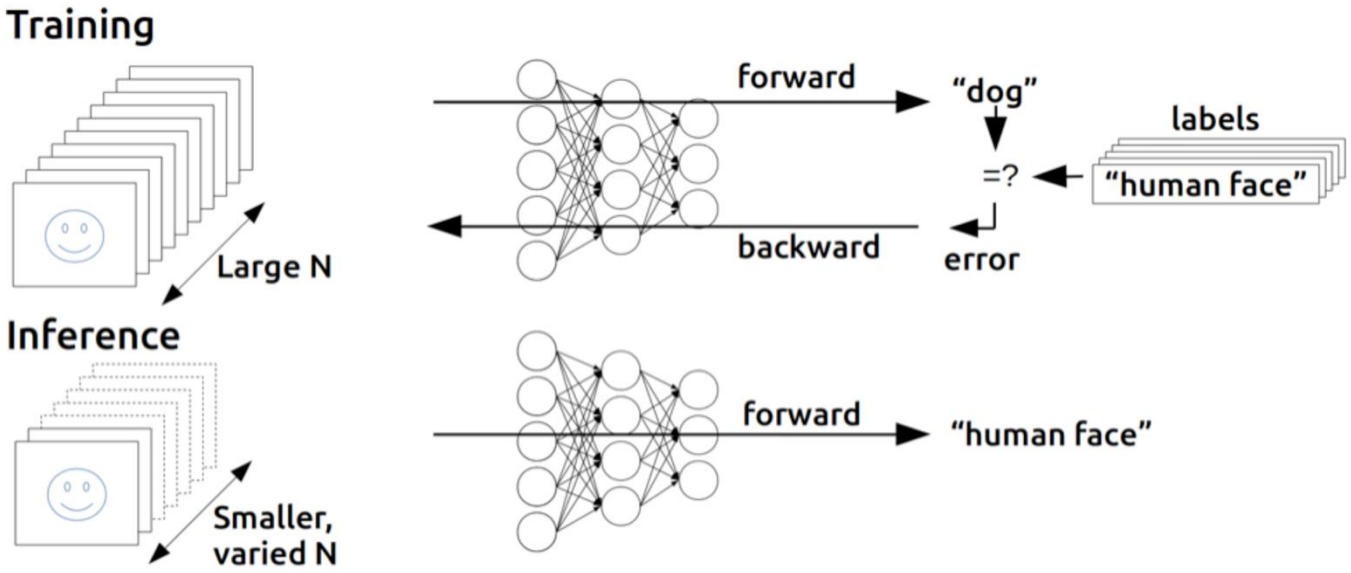
label:





Find Gradients for (Deep) Neural Networks

backpropagation of errors through network layers via chain rule of calculus (enables learning of deep neural networks)



source

forward pass:

- fix current weights
- compute predictions

backward pass:

- compute errors
- calculate gradients (backpropagation of errors)
- update weights accordingly (gradient descent)

Training: Iterative Adjustment of Weights

```
for epoch in range(epochs):
    for X, y in mini_batches:
        pred = model(X)
        cost = loss(pred, y)
        cost.backward()
        optimizer.step()
```

quantification of error (here: cross entropy)

chain rule to calculate gradient (direction) of cost function with respect to weights

one step in direction of steepest descent: (stochastic) gradient descent

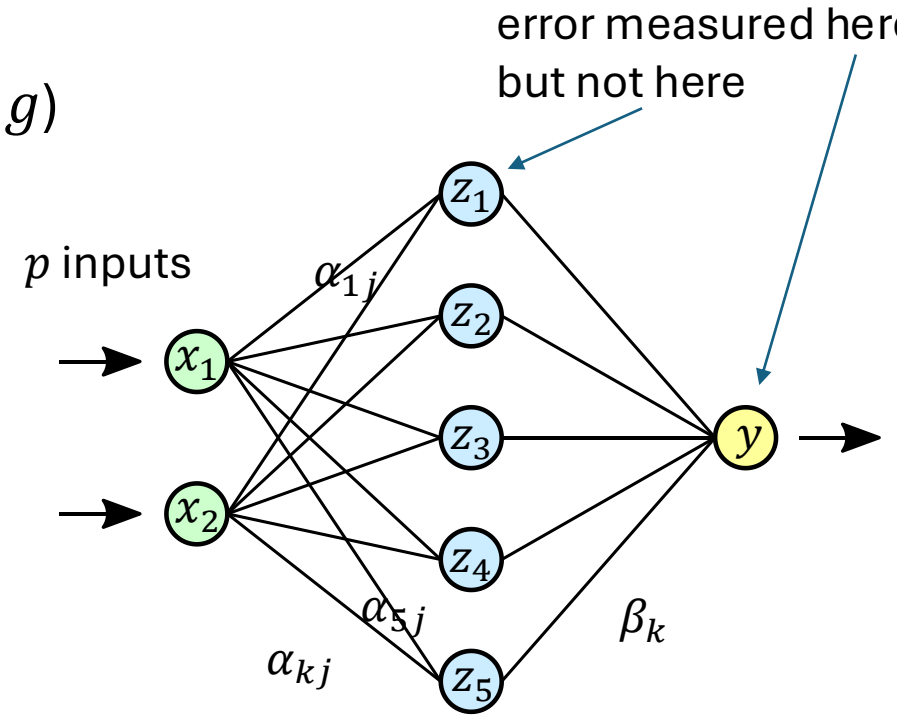
Example WOLOG

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

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

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

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:

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

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

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

use errors of later layers to calculate errors of earlier ones (avoiding redundant calculations of intermediate terms)

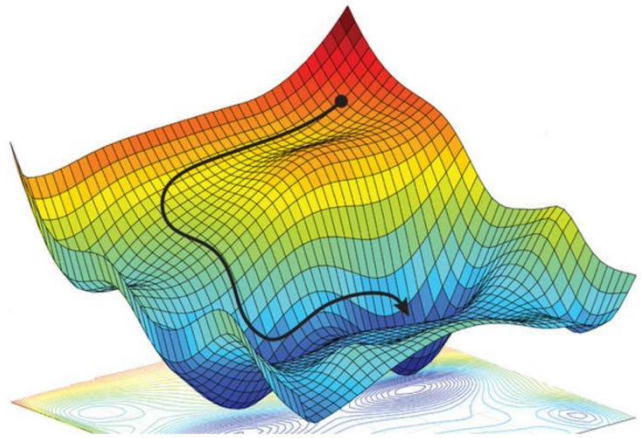
Using Gradients for Iterative Learning

use gradients found via backpropagation to iteratively update weights (gradient descent):

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

- adaptive learning rate η_r: learning rate often adjusted per iteration
- weight initialization: choose small random weights as starting values to break symmetry (typically using some heuristics)

Stochastic Gradient Descent (SGD)



step size

cost function

weight updates: $\hat{\boldsymbol{w}} \leftarrow \hat{\boldsymbol{w}} - \eta \nabla_{\hat{\boldsymbol{w}}} J(\hat{\boldsymbol{w}})$

options:

all training examples

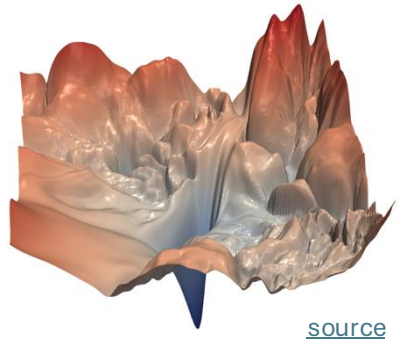
- $J(\hat{\boldsymbol{w}}) = \frac{1}{n} \sum_{i=1}^{n} J_i(\hat{\boldsymbol{w}})$ batch gradient descent (whole training dataset)
- $J(\hat{\boldsymbol{w}}) = J_i(\hat{\boldsymbol{w}})$ stochastic gradient descent (single example)
- $J(\hat{\boldsymbol{w}}) = \frac{1}{m} \sum_{i=1}^{m} J_i(\hat{\boldsymbol{w}})$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

mini-batch size → hyperparamter: tradeoffs between runtime & memory, convergence & generalization

random fluctuations of SGD help to escape saddle points

How to Train Deep Neural Networks Effectively?

typical loss surface:



source

optimization and regularization difficult

- local vs global minima, saddle points, easily overfitting
- many hyperparameters to tune

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

- optimization: activation and loss functions, weight initialization, adaptive learning rate (e.g., Adam), learning rate schedulers
- explicit regularization: weight decay, dropout, data augmentation
- implicit regularization: early stopping, batch/layer normalization, SGD

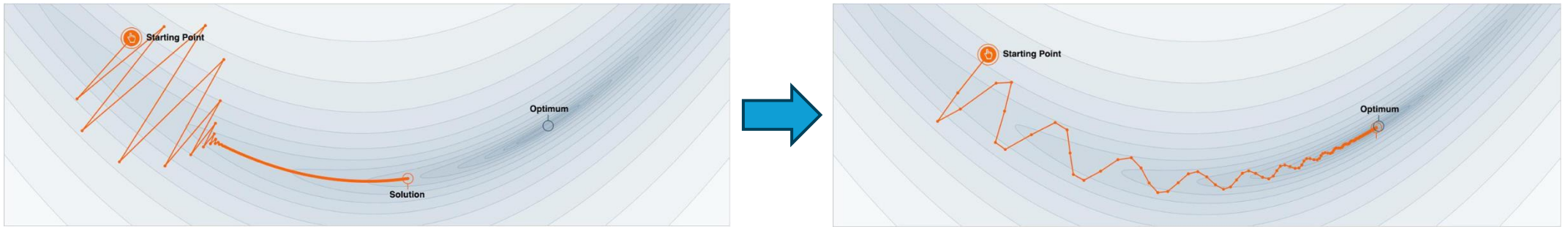
Gradient Descent with Momentum

algorithm:

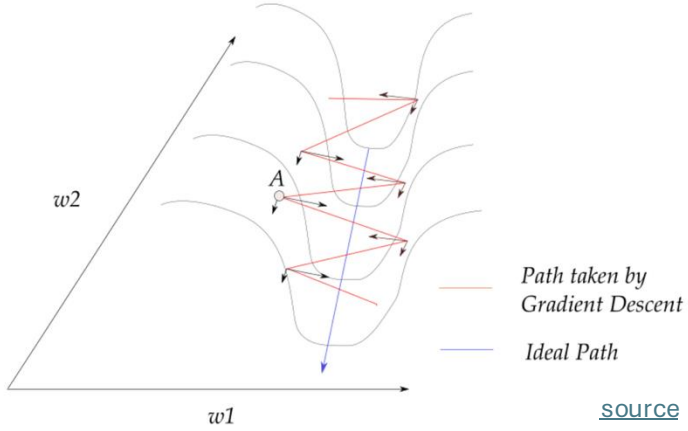
hyperparameter ($0 < \alpha < 1$) specifying exponential decay

- estimate gradient: $\boldsymbol{g} = \nabla_{\widehat{\boldsymbol{w}}} J(\widehat{\boldsymbol{w}})$
- compute velocity update: $\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta\, \boldsymbol{g}$
- apply parameter update: $\widehat{\boldsymbol{\theta}} \leftarrow \widehat{\boldsymbol{\theta}} + \boldsymbol{v}$

→ no bouncing around of parameters, escape from local minima and saddle points



Adaptive Learning Rate



source

better convergence by adapting learning rate for each weight per iteration: lower/higher learning rates for weights with large/small updates

popular methods ($g_{\hat{w}}$ denotes component of gradient for individual weight $\hat{w}$):

- Adagrad: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_{\hat{w},\tau}^2}} g_{\hat{w}}$ with t, τ denoting current and past iterations

  (issue: sum in denominator grows with more iterations → can get stuck)
- RMSProp: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{v(\hat{w})}} g_{\hat{w}}$ with $v(\hat{w}) \leftarrow \gamma\, v(\hat{w}) + (1-\gamma) g_{\hat{w}}^2$
- Adam (Adaptive Moment Optimization): combines RMSProp with momentum

Learning Rate Schedulers

learning rate η in gradient descent typically set to small value (e.g., 0.001)

use momentum and adaptive learning rates to improve optimization

learning rate scheduling as further alternative (can be used on top):

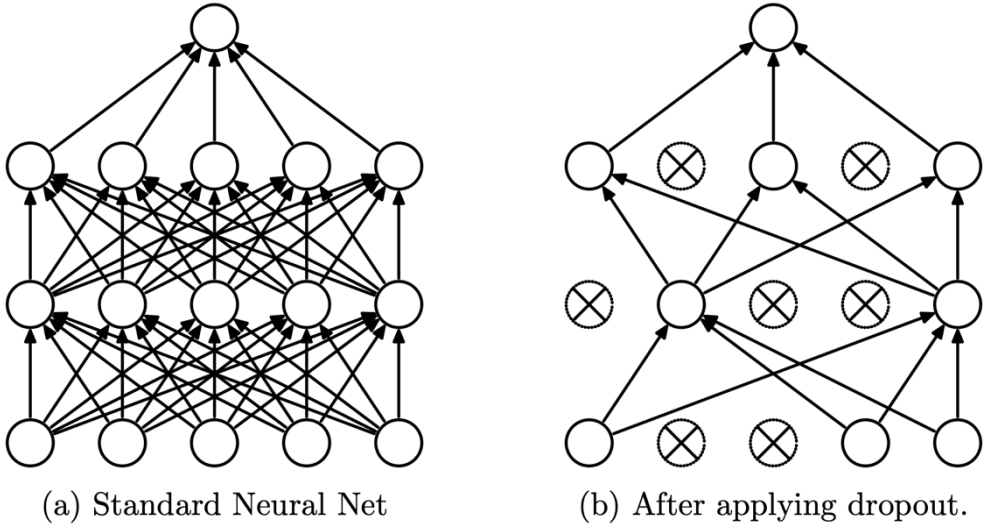
- decay by a certain factor every given number of epochs
- reduce when a metric has stopped improving
- smooth decay by cosine annealing (optionally cyclic with restarts)
- …

Dropout in Neural Networks

goal: prevent overfitting of large neural networks

randomly drop non-output nodes (along with their connections) during training (not prediction)

→ adaptability: regularizing each hidden node to perform well regardless of which other hidden nodes are in the model



(a) Standard Neural Net (b) After applying dropout.

source

- for each mini-batch, randomly sample independent binary masks for the nodes
- destroying extracted features rather than input values

dropout for 2D structures (such as images) usually drops entire channels instead of individual nodes (because locality nullifies the effect of standard dropout)

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

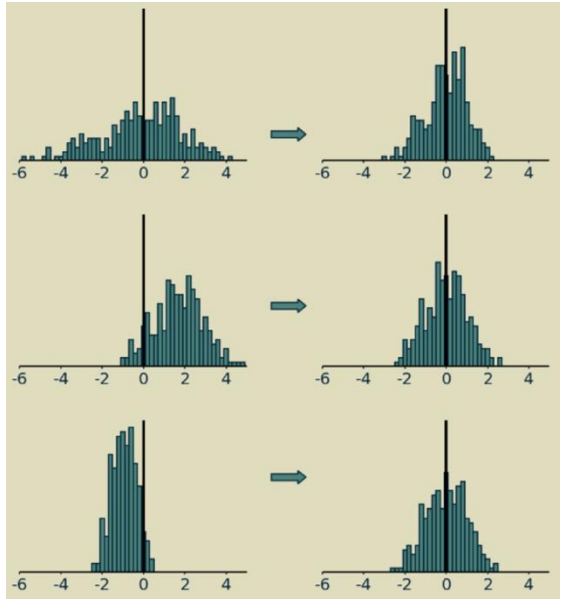
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

source



source

adaptive reparameterization of inputs to network layer (before or after activation)

independently for each input/feature

(not to confuse with weight normalization)

to maintain expressive power (optional): introduce parameters γ, β (learned together with weights via backpropagation)

Benefits from Batch Normalization

- allows higher learning rates
- reduces importance of weight initialization
- alleviates vanishing/exploding gradients
- (implicit) regularization effect: introducing noise

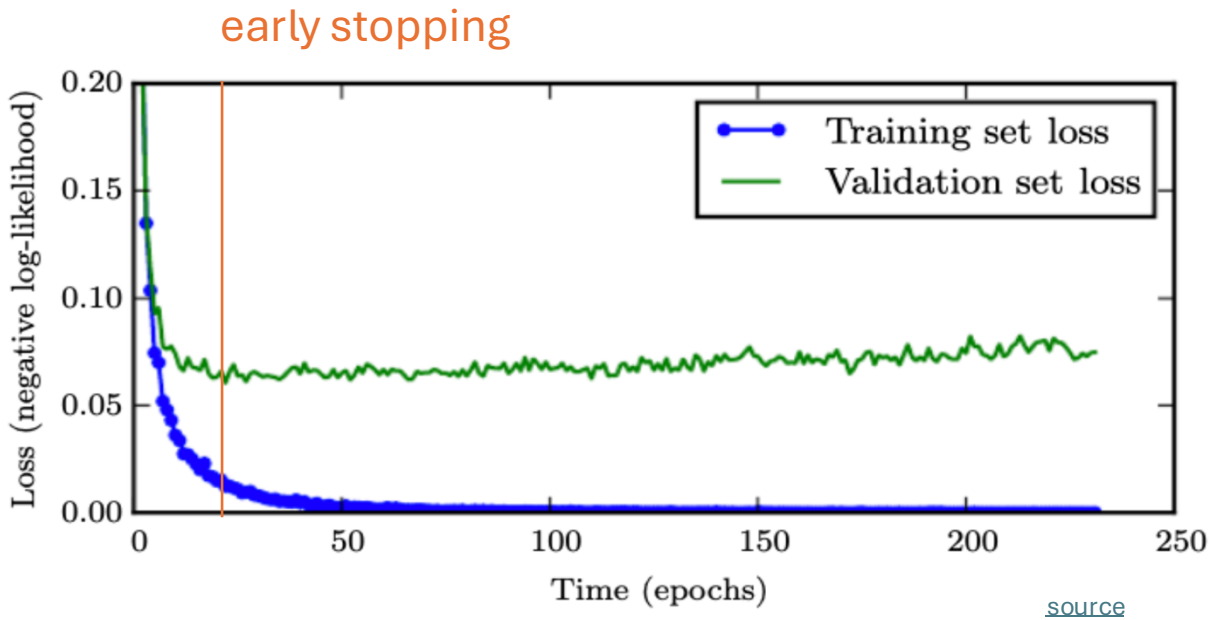
reason why batch normalization improves optimization still controversial

most plausible explanation: smoothening of loss landscape

Early Stopping

loss independently measured on validation set

halting training when overfitting begins to occur



source

Tabular vs Unstructured Data

deep learning methods dominate applications on unstructured data (like text or images), but not necessarily on tabular data

typical characteristics of tabular data difficult to handle for deep learning:

- irregular patterns in target function (neural networks require piecewise continuous targets)
- uninformative features
- non-rotationally invariant data (linear combinations of features misrepresent the information)

tree-based models (e.g., gradient boosting) can naturally deal with these situations