Neural Networks

Maschinelles Lernen 1 -
Grundverfahren WS20/21

Prof. Gerhard Neumann
KIT, Institut für Anthrophomatik und Robotik

Learning Outcomes

We will learn today…

- What a neuron network is
- How do we train it?
- … which requires a calculus refresher ☺
- Why is everybody talking about it?
- Various ways to accelerate gradient descent
- How to prevent overfitting in NNs?
- Practical tips and tricks for training NNs

Today‘s Agenda!

Neural Networks:

- What is a Neuron?
- Architectures and Activation Functions
- Loss-functions
- Backpropagation and the Chain Rule
- Computation graphs

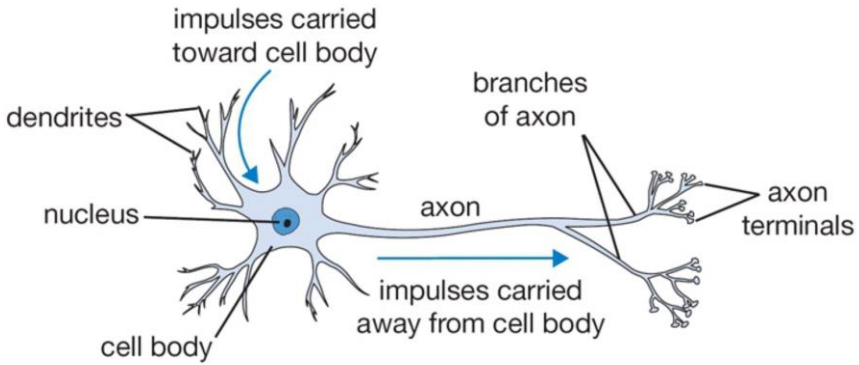
Advanced Topics:

- Accelerating gradient descent
- Regularization in Neural Networks
- Practical considerations

Credit: M. Ren and M. MacKay, University of Toronto,
Fei-Fei Li & Justin Johnson & Serena Yeung, Stanford

Biological Inspiration: The brain

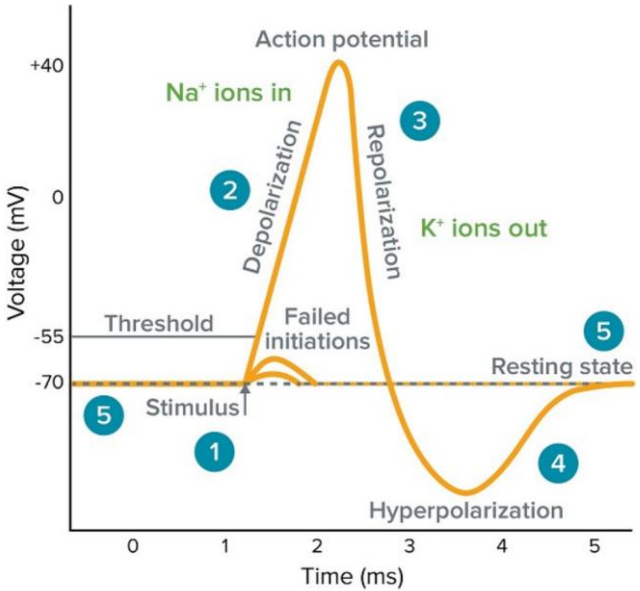
A neuron is the basic computational unit of the brain:



- Our brain has ~ 10^{11} neurons
- Each neuron is connected to ~ 10^{4} other neurons (via synapses)

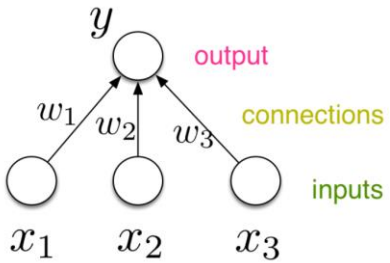
Biological Inspiration: The brain

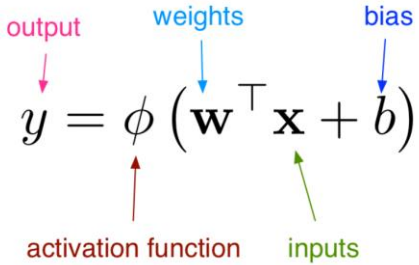
Neurons receive input signals and accumulate voltage. After some threshold they will fire spiking responses.



Artificial Neurons

For neural nets, we use a much simpler neuron, or unit:





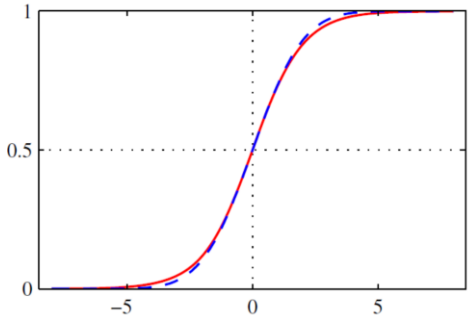
3 ingredients:

- Weighting of the input
- Summation
- Non-linear activation function

Example we already know:

- Logistic regression:

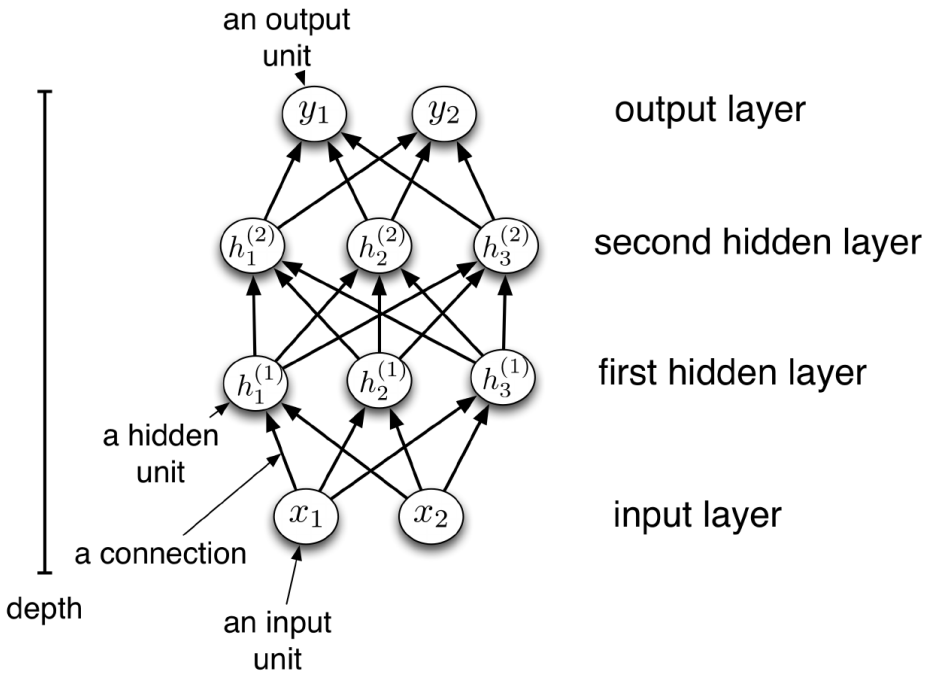
$$y = \sigma(\mathbf{w}^T \mathbf{x} + b)$$



Feedforward Neural Networks

Building a network:

- We can connect lots of units together into a directed acyclic graph.
- This gives a feed-forward neural network. That's in contrast to recurrent neural networks, which can have cycles.
- Typically, units are grouped together into layers.



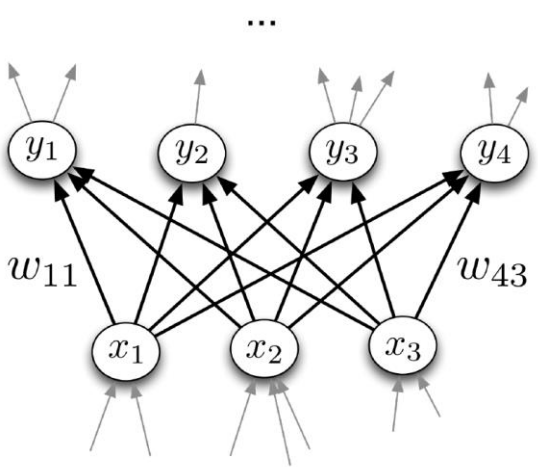
Feedforward Neural Networks

- Each layer connects *N* input units to *M* output units.
- In the simplest case, all input units are connected to all output units. We call this **a fully connected layer.**
- **Note:** the inputs and outputs for a layer are distinct from the inputs and outputs to the network.

- I.e., each layer has a M x N weight matrix **W**
- Equation in matrix form:

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

  - Output units are a function of input units
- Feedforward neural networks are also often called multi-layer perceptrons

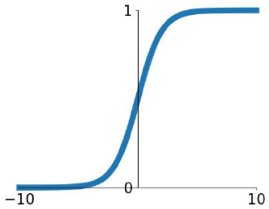


Activation funcitons

Different activation functions for introducing non-linearities:

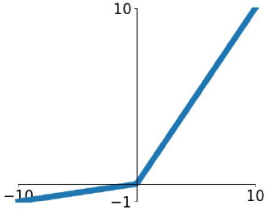
Sigmoid

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



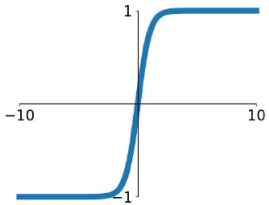
Leaky ReLU

$$\max(0.1x, x)$$



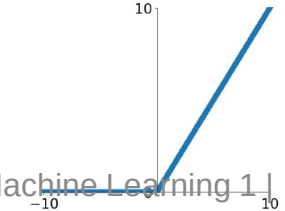
tanh

$$\tanh(x)$$



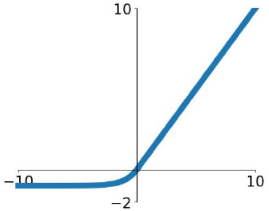
ReLU

$$\max(0, x)$$

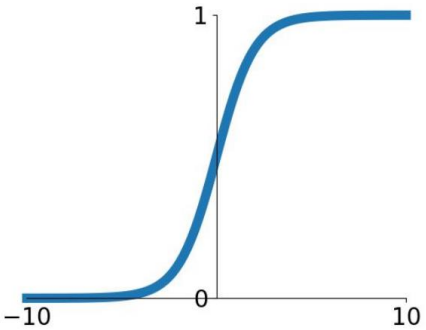


ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation functions



Sigmoid

Computes $\sigma(x) = \frac{1}{1+\exp(-x)}$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

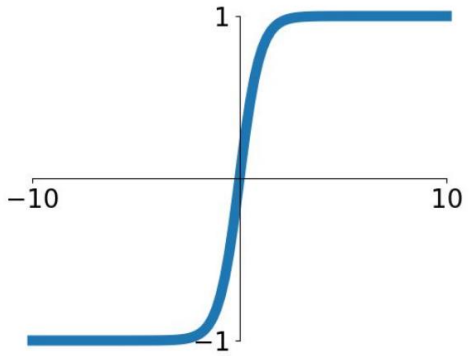
Problems:

× Saturated neurons “kill” the gradients

× Sigmoid outputs are not zero-centered (important for initialization)

× exp() is a bit compute expensive

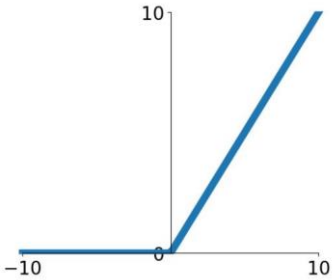
Activation functions



tanh(x)

- Squashes numbers to range [-1,1]
- ✓ zero centered (nice)
- × still kills gradients when saturated :(

Activation functions



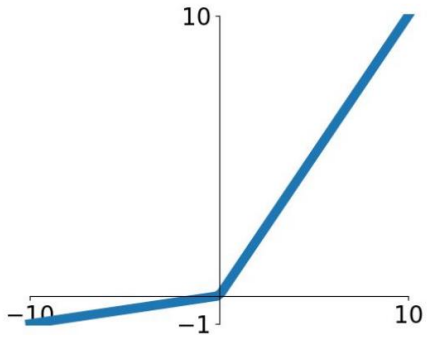
ReLU
(Rectified Linear Unit)

Computes $f(x) = \max(0, x)$

- ✓ Does not saturate (in +region)
- ✓ Very computationally efficient
- ✓ Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- × Not zero-centred output
- × No gradient for x < 0

Activation functions



Leaky ReLU

Computes $f(x) = \max(0.1x, x)$

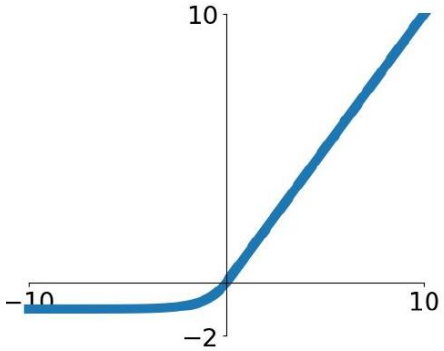
- ✓ Does not saturate
- ✓ Computationally efficient
- ✓ Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- ✓ will not “die”

Parametric Rectifier (PReLu):

$$f(x) = \max(\alpha x, x)$$

- Also learn alpha

Activation functions



Computes $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$

- ✓ All benefits of ReLU
- ✓ Closer to zero mean outputs
- ✓ Negative saturation regime compared with Leaky ReLU (adds some robustness to noise)
- × Computation requires exp()

Exponential Linear Units (ELU)

Activation functions

In practice:

- Use ReLU. Be careful with your learning rates / initialization
- Try out Leaky ReLU / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid

Feedforward Neural Networks

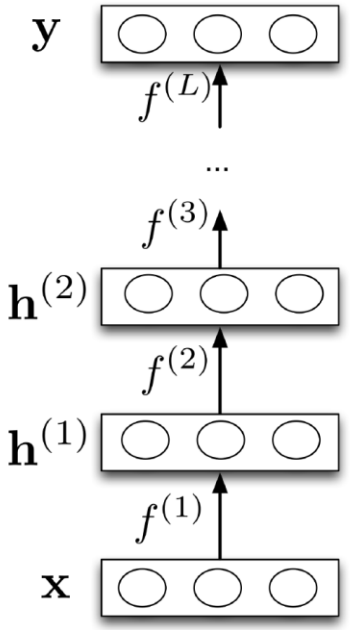
Formalisation:

- Each layer computes a function, so the network computes a composition of functions:

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x})$$
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)})$$
$$\vdots$$
$$\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)})$$

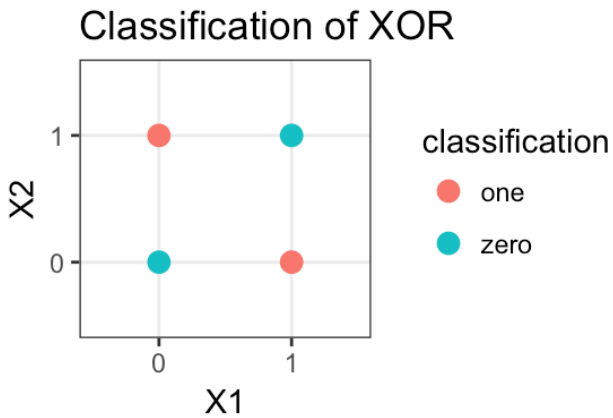
- Or more simply: $\mathbf{y} = f^{L} \circ f^{L-1} \circ \dots f^{(1)}(\mathbf{x})$

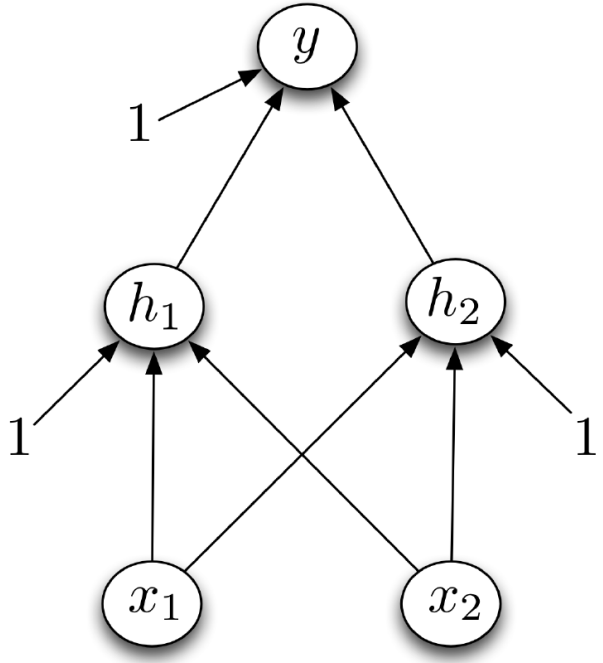
- Neural nets provide modularity: we can implement each layer's computations as a black box.



Example: XOR

Design a network that implements XOR:





- Not computable by a single unit!
- Classical example why we need multiple layers

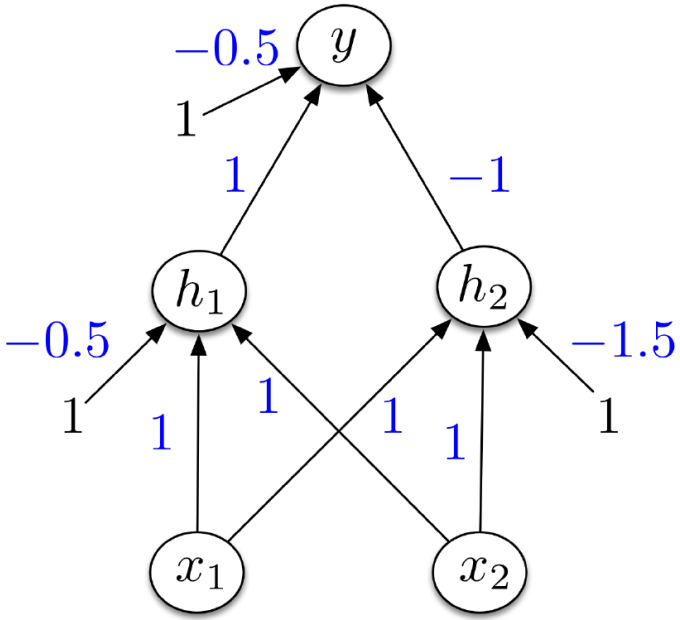
Example: XOR

XOR in terms of elemental operations:

- XOR(a,b) = (a OR b) AND NOT (a AND b)

Design a network that implements XOR:

- Hard threshold for activation function
- h_1 computes x_1 OR x_2
- h_2 computes x_1 AND x_2
- y computers h_1 AND NOT h_2



Deep Architectures

Why do we need to be deep ?

- Any sequence of linear layers can be equivalently represented with a single linear layer

$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\tilde{\mathbf{W}}}\mathbf{x}$$

  - I.e., we need non-linearities to exploit multiple layers

- FF-NNs with nonlinear activation functions are universal function approximators:
  - Given a potentially infinite amount of units, they can approximate any function arbitrarily well
  - Universal Function Approximation Theorem: Already a single layer is enough to achieve "universality"

Deep Architectures

So, is a single layer enough?

- Even though the Universal Approximation Theorem says a single layer is enough, we would need an exponential number of units to achieve this
  - If you can learn any function, you'll just overfit.
- Instead, multiple layers allow for a similar effect with less units
  - Compact representation >> "Universal representation"

Output layers and loss functions

Objective functions for training neural nets:

- General ML recipe: per sample loss + regularization penalty (see lecture 2)

$$\boldsymbol{\theta}^* = \underset{\text{parameters } \boldsymbol{\theta}}{\arg\min} \sum_{i=1}^{N} l(\boldsymbol{x}_i, \boldsymbol{\theta}) + \lambda \, \text{penalty}(\boldsymbol{\theta})$$

Which kind of loss and output activation function depends on the task

- Regression
- Binary classification
- Multi-class classification

Today‘s Agenda!

Neural Networks:

- What is a Neuron?
- Architectures and Activation Functions
- Loss-functions
- Backpropagation and the Chain Rule
- Computation graphs

Advanced Topics:

- Accelerating gradient descent
- Regularization in Neural Networks
- Practical considerations

Credit: M. Ren and M. MacKay, University of Toronto,
Fei-Fei Li & Justin Johnson & Serena Yeung, Stanford

Output layers and loss functions

Regression:

	Deterministic	Probabilistic
• **Output layer:**	linear $\mathbf{f} = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \boldsymbol{b}^{(L)}$	linear Gaussian $p(\mathbf{y}\|\mathbf{x}) = \mathcal{N}(\mathbf{y}\|\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}, \boldsymbol{\Sigma})$
• **Loss:**	squared error $l_i(\mathbf{x}_i, \boldsymbol{\theta}) = \frac{1}{2}(\mathbf{f}(\mathbf{x}_i) - \mathbf{y}_i)^2$	negative log-likelihood $l_i(\mathbf{x}_i, \boldsymbol{\theta}) = -\log \mathcal{N}(\mathbf{y}_i\|\boldsymbol{\mu}(\mathbf{x}_i), \boldsymbol{\Sigma})$

Output layers and loss functions

Binary classification:

	Deterministic	Probabilistic
• **Output layer:**	linear $f = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + b^{(L)}$	sigmoid $f = \sigma(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + b^{(L)})$
• **Loss function:**	hinge-loss $l(\mathbf{x}_i, \boldsymbol{\theta}) = \max\left(0, 1 - y_i f(\boldsymbol{x}_i)\right)$ -1/+1 labels	neg-loglike $l_i(\mathbf{x}_i, \boldsymbol{\theta}) = -c_i \log f(\mathbf{x}_i) - (1 - c_i)\log(1 - f(\mathbf{x}_i))$ 0/1 labels

Output layers and loss functions

Multi-class classification:

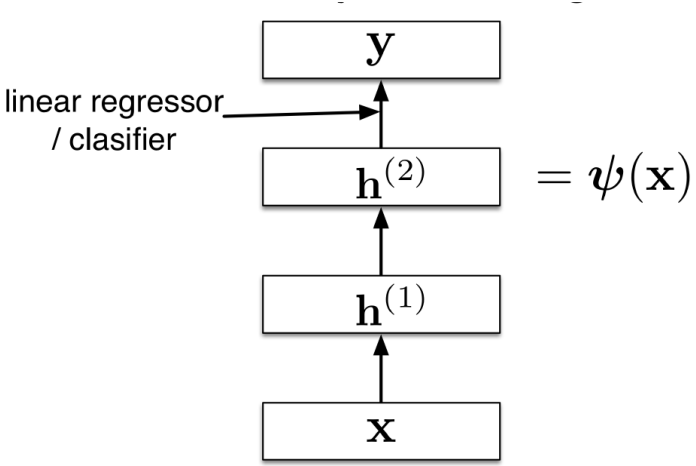
	Deterministic	Probabilistic
• **Output layer:**	linear	sigmoid
	$\mathbf{f} = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$	$\mathbf{f} = \text{softmax}(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)})$
• **Loss function:**	Multi-class SVM loss	negative loglikelihood
	Not covered	$l_i(\mathbf{x}_i, \boldsymbol{\theta}) = -\sum_{k=1}^{K} \boldsymbol{h}_{c_i,k} \log y_k(\mathbf{x}_i)$

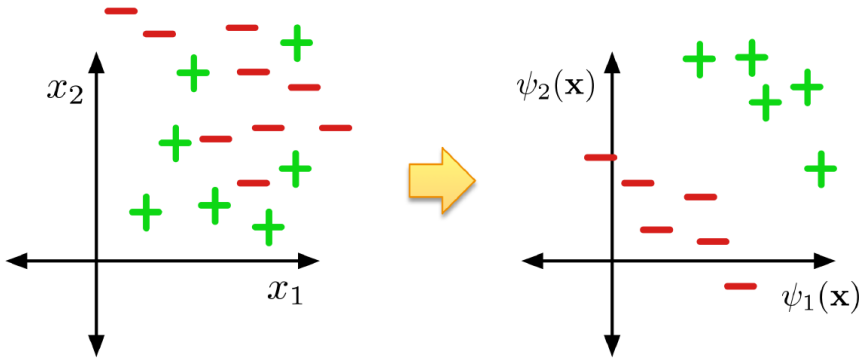
One hot coding

Feature Learning

- Neural nets can be viewed as a way of learning features
  - The last layer is a standard linear regression / classification layer

- The network learns the features $\boldsymbol{\psi}(\mathbf{x})$ such that linear regression / classification can solve it

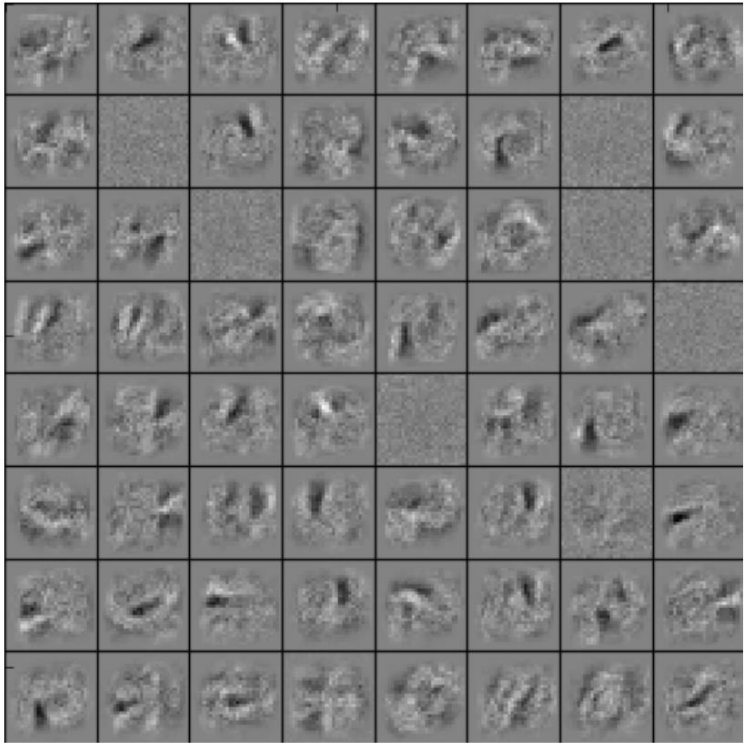




Example: Feature Learning

Classify images of handwritten digits:

- Each image is represented as a vector of 28 × 28 = 784 pixel values.
- Each first-layer hidden unit computes $\sigma(\mathbf{w}_i^T \mathbf{x})$. It acts as a feature detector.
- We can visualize **w** by reshaping it into an image.
- These weights **w** are visualized on the right for some units
- Edge-detectors at different orientations and locations



Today‘s Agenda!

Neural Networks:

- What is a Neuron?
- Architectures and Activation Functions
- Loss-functions
- Backpropagation and the Chain Rule
- Computation graphs

Advanced Topics:

- Accelerating gradient descent
- Regularization in Neural Networks
- Practical considerations

Credit: M. Ren and M. MacKay, University of Toronto,
Fei-Fei Li & Justin Johnson & Serena Yeung, Stanford

Gradient Descent

Multi-layer perceptrons are usually trained using back-propagation for computing the gradients

Same algorithms as for logistic regression can be used, however

- Much bigger parameter space
- Non-convex, many local optima
- Can get stuck in poor local optima
- The design of a working backprop algorithm is somewhat of an art

Because of that, the use of NNs was in absolute winter between ~2000 and 2012

However, in the last 5-7 years, we have seen that with:

- More compute
- More data
- And a few tricks…

they work amazingly well … (side comment: we just do not know why…)

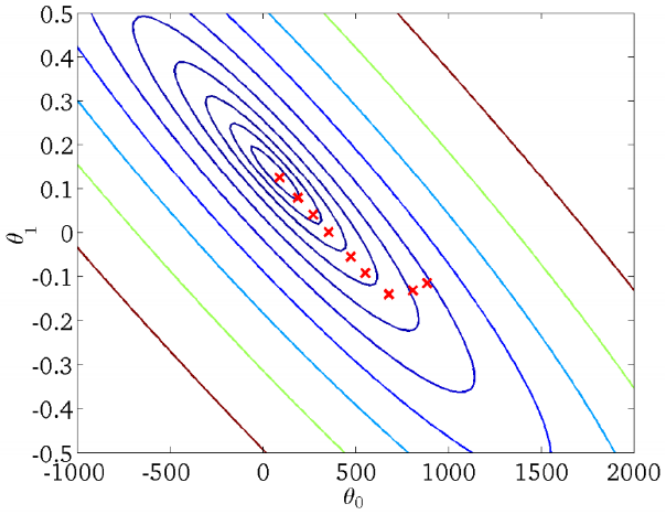
Recap: Gradient Descent

Move in the opposite direction of the gradient (steepest descent)

- Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in all the layers

$$\boldsymbol{\theta} = \{\mathbf{W}^{(L)}, \ldots, \mathbf{W}^{(1)}, \mathbf{b}^{(L)}, \ldots, \mathbf{b}^{(1)}\}$$

- Conceptually, not any different from what we've seen so far (Lecture 2) — just higher dimensional and harder to visualize!



Basics: Chain rule gradients of composite functions

Objective functions for training neural nets:

- per sample loss + regularization penalty (see lecture 2)

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) = \sum_{i=1}^{N} l(\boldsymbol{x}_i, \boldsymbol{\theta}) + \lambda \text{ penalty}(\boldsymbol{\theta})$$

- We need to compute the following partial derivatives:
  - Layer weight matrices: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$
  - Layer bias vectors: $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$
- Can be done by a recursive use of the chain rule!

Univariate case

Lets start simple…

Univariate chain rule:

- **Recall:** if *f(x)* and *x(t)* are univariate functions, then

$$\frac{d}{dt} f(x(t)) = \frac{df}{dx}\frac{dx}{dt}$$

Example

Univariate logistic least squares model

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Lets compute the loss derivatives…

101 in calculus:

$$\mathcal{L} = \frac{1}{2}(\sigma(wx + b) - t)^2$$
$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\frac{1}{2}(\sigma(wx + b) - t)^2$$
$$=$$
$$=$$
$$=$$

- Similar for $\frac{\partial \mathcal{L}}{\partial b}$

Example

Univariate logistic least squares model

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Lets compute the loss derivatives…

101 in calculus:

$$\mathcal{L} = \frac{1}{2}(\sigma(wx + b) - t)^2$$
$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\frac{1}{2}(\sigma(wx + b) - t)^2$$
$$= (\sigma(wx + b) - t)\frac{\partial}{\partial w}\sigma(wx + b)$$
$$= (\sigma(wx + b) - t)\sigma'(wx + b)\frac{\partial}{\partial w}(wx + b)$$
$$= (\sigma(wx + b) - t)\sigma'(wx + b)x$$

- Similar for $\frac{\partial \mathcal{L}}{\partial b}$

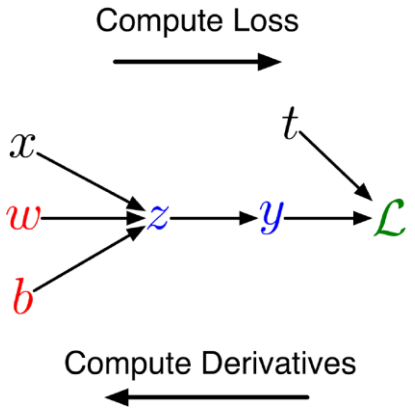
Computation graph (example)

Can we find an algorithm to do it more systematically?

- The goal isn't to obtain closed-form solutions…
- but to be able to write a program that efficiently computes the derivatives.

We can diagram out the computations using a computation graph:

- The nodes represent all the inputs and computed quantities
- The edges represent which nodes are computed directly as a function of which other nodes.

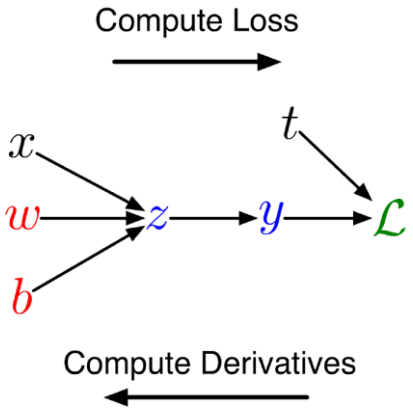


Computation graph (example)

Computing the loss:

- forward pass

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Computing the derivative

- backward pass

$$\frac{\partial L}{\partial y} = y - t$$
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial y}\sigma'(z) \quad \frac{\partial y}{\partial z}$$
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z}x \quad \frac{\partial z}{\partial w}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot 1 \quad \frac{\partial z}{\partial b}$$

Computation graph (example)

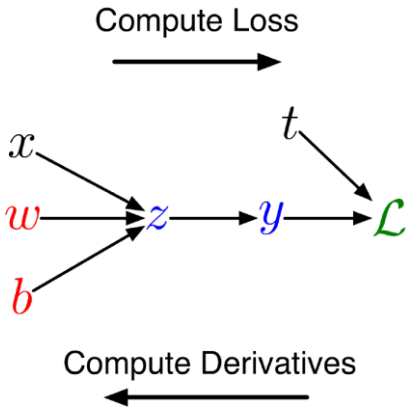
Lets simplify notation:

- Use $\bar{y}$ to denote the derivative $\frac{\partial \mathcal{L}}{\partial y}$ (also called error signals)
- Emphasizes that error signals are just values (rather than mathematical operations)

Computing the loss:

- forward pass

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Computing the derivative

- backward pass

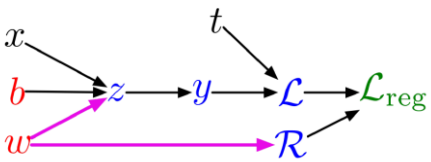
$$\bar{y} = y - t$$
$$\bar{z} = \bar{y}\sigma'(z)$$
$$\bar{w} = \bar{z}x$$
$$\bar{b} = \bar{z}$$

General computation graphs

Problem: what if the computation graph has **fan-out > 1**?

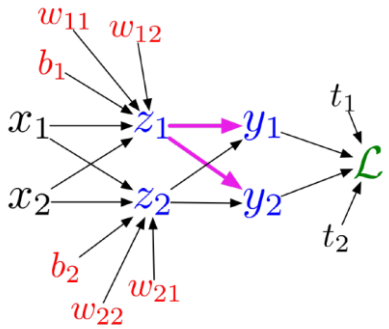
- This requires the multivariate Chain Rule!

Regularized Regression:



$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Softmax classification:

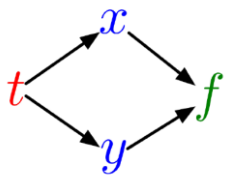


$$z_l = \sum_j w_{lj} x_j + b_j$$
$$y_k = \frac{\exp(z_k)}{\sum_l \exp(z_l)}$$
$$\mathcal{L} = -\sum_k t_k \log y_k$$

Multivariable chain rule

Suppose we have a function *f(x,y)* and functions *x(t)* and *y(t) (*All the variables here are scalar-valued.) Then

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



Example:

$$f(x, y) = y + \exp(xy)$$
$$x(t) = \cos t$$
$$y(t) = t^2$$

Plug in Chain Rule:

$$\begin{aligned}\frac{d}{dt} f(x(t), y(t)) &= \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} \\ &= \big(y \exp(xy)\big)\big(-\sin t\big) \\ &\quad + \big(1 + x \exp(xy)\big)\big(2t\big)\end{aligned}$$

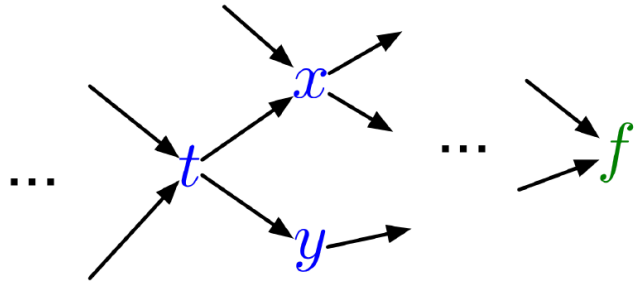
Multivariable chain rule

In the context of back-propagation:

Values already computed by the algorithm

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

Mathematical expressions to be evaluated



Using our notation:

$$\bar{t} = \bar{x}\frac{dx}{dt} + \bar{y}\frac{dy}{dt}$$

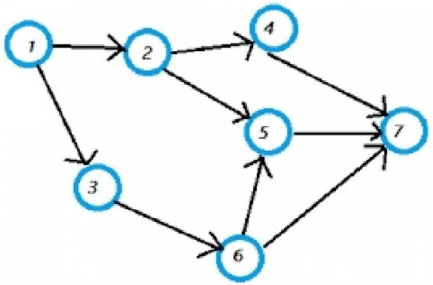
In vector notation:

$$\frac{d}{dt} f(\boldsymbol{x}(t)) = \sum_i \frac{\partial f}{\partial x_i}\frac{\partial x_i}{\partial t}$$

Backpropagation

Full backpropagation algorithm:

- Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)
- v_N denotes the variable we're trying to compute derivatives of (e.g. loss).



forward pass

$$\text{For } i = 1, \ldots, N$$

$$\text{Compute } v_i \text{ as a function of } \mathrm{Pa}(v_i)$$

backward pass

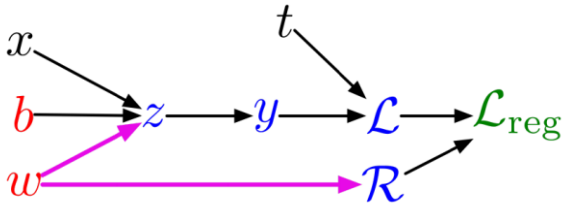
$$\overline{v_N} = 1$$

$$\text{For } i = N-1, \ldots, 1$$

$$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

Backpropagation

Example: univariate logistic least squares regression



Forward pass:

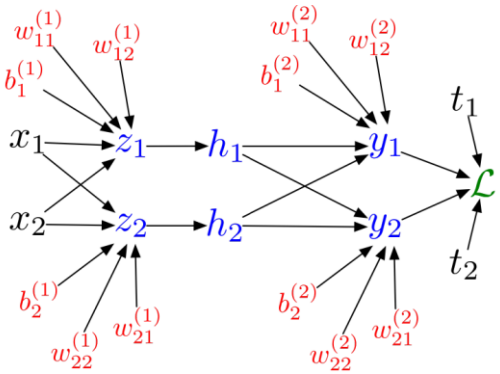
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Backward pass:

$$\overline{\mathcal{L}_{\text{reg}}} = 1$$
$$\overline{\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}}\frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}}\lambda$$
$$\overline{\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}}\frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}}$$
$$\overline{y} = \overline{\mathcal{L}}\frac{d\mathcal{L}}{dy} = \overline{\mathcal{L}}(y - t)$$
$$\overline{z} = \overline{y}\frac{dy}{dz} = \overline{y}\sigma'(z)$$
$$\overline{w} = \overline{z}\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{d\mathcal{R}}{dw} = \overline{z}x + \overline{\mathcal{R}}w$$
$$\overline{b} = \overline{z}\frac{\partial z}{\partial b} = \overline{z}$$

Backpropagation

Example: Multi-layer Perceptron (multiple outputs)



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}}(y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k} h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

$$\overline{z_i} = \overline{h_i} \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i} x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

Can we also do this in matrix form?

Recap: Matrix Calculus

Derivatives of a scalar function w.r.t a vector…

- Yields the gradient vector: $\nabla_{\boldsymbol{x}} f = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_d}\right]^T$
- Example: Quadratic form $\nabla_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{x} = 2\boldsymbol{x}$ $\nabla_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x} = 2\boldsymbol{A}\boldsymbol{x}$

Derivatives of a vector-valued function w.r.t a vector…

- Yields a matrix (the Jacobian) $\nabla_{\boldsymbol{x}} \boldsymbol{f} = \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial f_k(\boldsymbol{x})}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1(\boldsymbol{x})}{\partial x_d} & \cdots & \frac{\partial f_k(\boldsymbol{x})}{\partial x_d} \end{bmatrix}$
- Example: Linear form $\nabla_{\boldsymbol{x}} \boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T$

Recap: Matrix Calculus

Derivatives of a scalar function w.r.t. a matrix…

- … is again a matrix $$\nabla_{\boldsymbol{W}} f = \frac{\partial f(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{W})}{\partial W_{11}} & \cdots & \frac{\partial f(\boldsymbol{W})}{\partial W_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(\boldsymbol{W})}{\partial W_{k1}} & \cdots & \frac{\partial f(\boldsymbol{W})}{\partial W_{kd}} \end{bmatrix}$$

Derivatives of a vector-valued function w.r.t. a matrix…

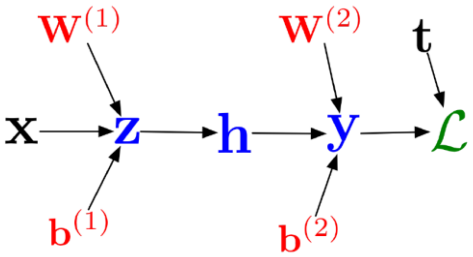
- … is a 3D tensor !
- However, we only have matrix-vector products: $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$
- In this case, the chain-rule does not require to evaluate the tensor, i.e. (proof not shown)

$$\nabla_{\boldsymbol{W}} f = \frac{\partial f(\boldsymbol{z})}{\partial \boldsymbol{W}} = \frac{\partial f(\boldsymbol{z})}{\partial \boldsymbol{z}} \frac{\partial}{\partial \boldsymbol{W}} (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \frac{\partial f(\boldsymbol{z})}{\partial \boldsymbol{z}} \boldsymbol{x}^T$$

- This is the outer product (i.e., again a matrix)

Example in matrix form

Example: Multi-layer Perceptron (vector form)



Forward pass:

$$\boldsymbol{z} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$$

$$\boldsymbol{h} = \sigma(\boldsymbol{z})$$

$$\boldsymbol{y} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$$

$$\mathcal{L} = \frac{1}{2}(\boldsymbol{y} - \boldsymbol{t})^T(\boldsymbol{y} - \boldsymbol{t})$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{\boldsymbol{y}} = \overline{\mathcal{L}}(\boldsymbol{y} - \boldsymbol{t})$$

$$\overline{\boldsymbol{W}^{(2)}} = \underbrace{\overline{\boldsymbol{y}}\boldsymbol{h}^T}_{\text{outer product}}$$

$$\overline{\boldsymbol{b}^{(2)}} = \overline{\boldsymbol{y}}$$

$$\overline{\boldsymbol{h}} = \boldsymbol{W}^{(2)T}\overline{\boldsymbol{y}}$$

$$\overline{\boldsymbol{z}} = \overline{\boldsymbol{h}} \circ \sigma'(\boldsymbol{z})$$

element-wise product

$$\overline{\boldsymbol{W}^{(1)}} = \underbrace{\overline{\boldsymbol{z}}\boldsymbol{x}^T}_{\text{outer product}}$$

$$\overline{\boldsymbol{b}^{(1)}} = \overline{\boldsymbol{z}}$$

Computational costs

- **Computational cost of forward pass:**

$$z = Wx + b$$

  - Matrix-vector product
  - Roughly one add-multiply operation per weight
- **Computational cost of backward pass:**

$$\overline{W} = \overline{h} z^T, \quad \overline{h} = W^T \overline{y}$$

  - Matrix-vector product + outer product
  - Roughly two add-multiply operation per weight (twice the forward pass)

- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.

Wrap-up for backpropagation

- **Backprop is used to train the overwhelming majority of neural nets today.**
  - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.
- **Despite its practical success, backprop is believed to be neurally implausible.**
  - No evidence for biological signals analogous to error derivatives.
  - Forward & backward weights are tied in backprop.
  - Backprop requires synchronous update (1 forward followed by 1 backward).
- All the biologically plausible alternatives we know about learn much more slowly (on computers).

Today‘s Agenda!

Neural Networks:

- What is a Neuron?
- Architectures and Activation Functions
- Loss-functions
- Backpropagation and the Chain Rule
- Computation graphs

Advanced Topics:

- Accelerating gradient descent
- Regularization in Neural Networks
- Practical considerations

Credit: M. Ren and M. MacKay, University of Toronto, Fei-Fei Li & Justin Johnson & Serena Yeung, Stanford

Gradient descent for Neural Networks

We know now how to compute the gradient using backpropagation…

We still have to decide on…

- When to update **W**?
- How to choose the learning rate?
- How to initialize **W**?

When do update W?

Mini-Batches: Take **subset of samples** $I_t \subset \{1, \ldots, n\},\ |I_t| = b,\ b \ll n$ to approximate real gradient:

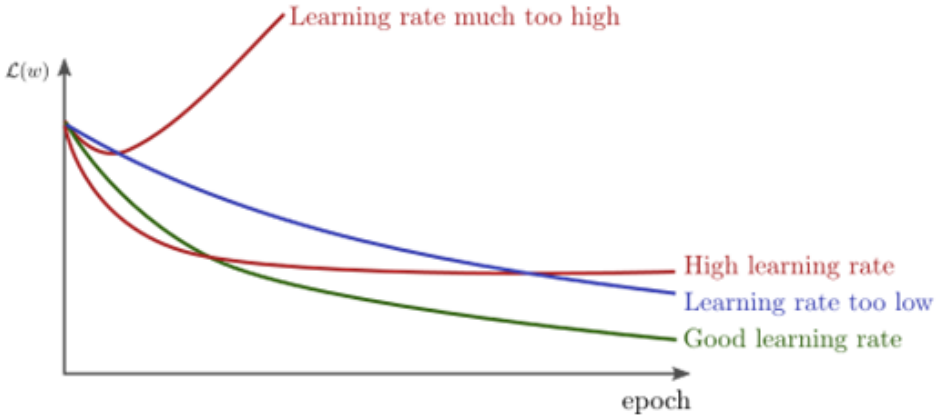
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{b} \sum_{i \in I_t} \nabla_{\boldsymbol{\theta}} l(\boldsymbol{x}_i; \boldsymbol{\theta}_t)$$

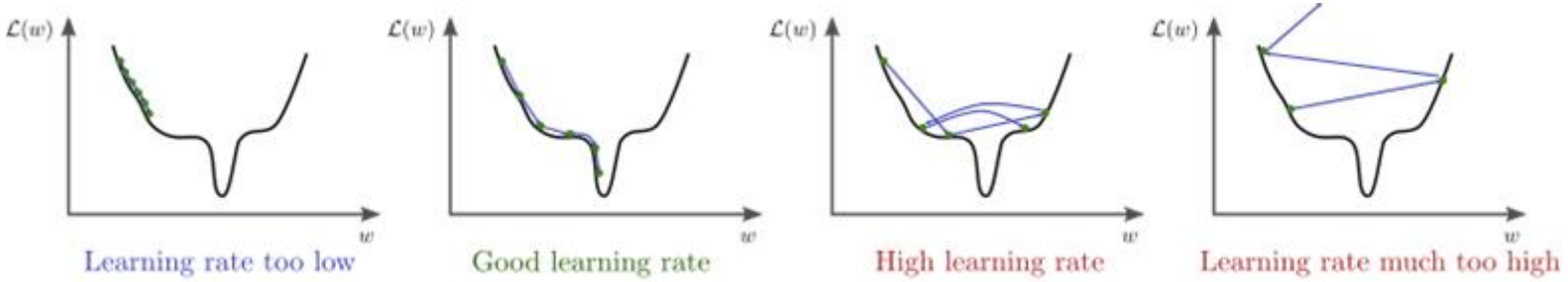
- Intermediate version of stochastic and batch gradient descent
- Less noisy estimates than stochastic gradient descent
- More efficient than batch gradient descent
- Preferable for GPU implementations

How do choose the learning rate?

If the learning rate is chosen:

- Too low: slow convergence
- Too high: oscillations and slow convergence
- Much too high: divergence





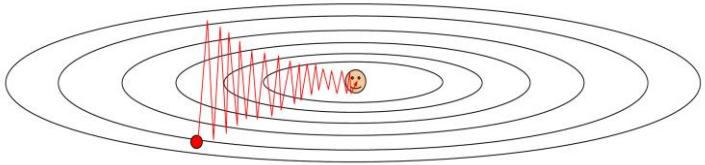
Speeding up gradients descent

- Momentum terms
- Adaptive learning rates
- 2^{nd} order methods (only for smaller networks)

Problems with standard SGD

(1) Loss changes quickly in one direction and slowly in another

- Very slow progress along shallow dimension, jitter along steep direction

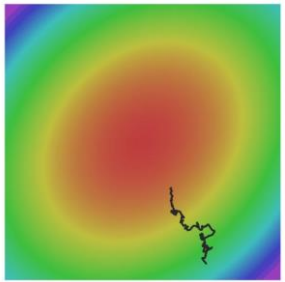


(2) Loss function has local minima and plateaus

- Zero gradient, gradient descent gets stuck

(3) Loss function is noisy

- Due to minibatches…



Momentum term

Insight: Compute running average for gradient (or other quantities)

- $\boldsymbol{m}_0 = \mathbf{0}, \quad \boldsymbol{m}_{k+1} = \gamma_k \boldsymbol{m}_k + (1-\gamma_k)\boldsymbol{g}_k, \quad$ where $\boldsymbol{g}_k$ is the gradient.
- Geometric Average (constant γ) : $\boldsymbol{m}_k = (1-\gamma)\sum_{i=1}^{k} \gamma^{k-i}\boldsymbol{g}_i$
- Arithmetic Average ($\gamma_k = (k-1)/k$): $\boldsymbol{m}_k = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{g}_i$

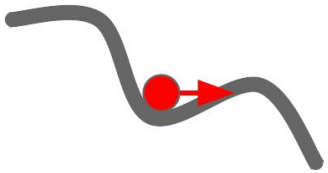
Applied to the gradient update:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta\, \boldsymbol{m}_{k+1}$$

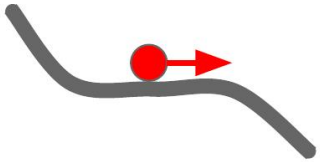
Intuitively: You can think of it as a “velocity term” for the update. The gradient is in this view the acceleration.

SGD + Momentum

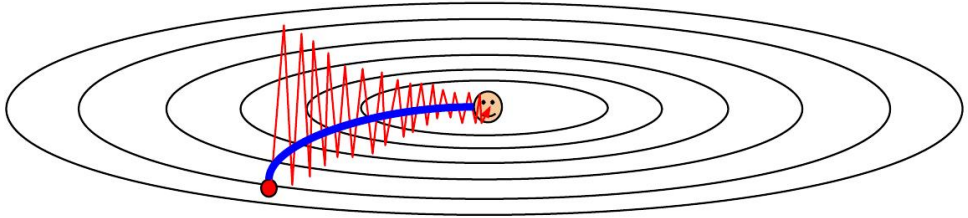
Local Minima



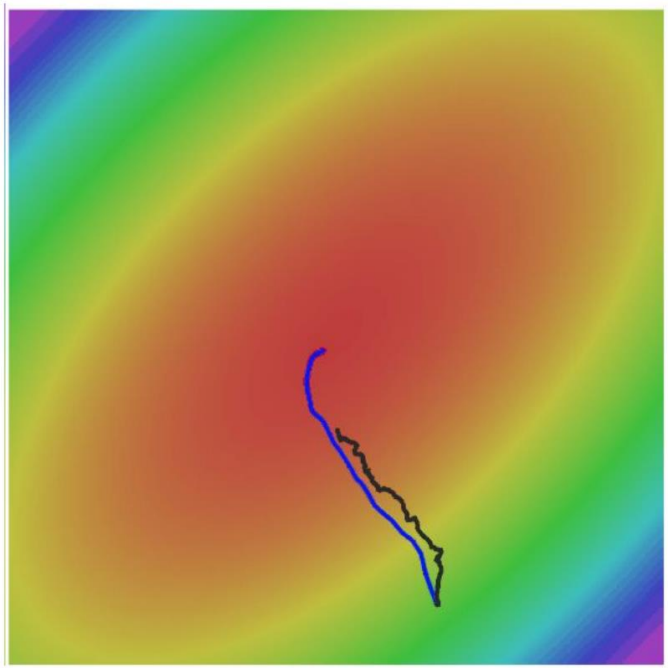
Saddle points



Poor Conditioning



Gradient Noise



Gradient Normalisation (RMSProp)

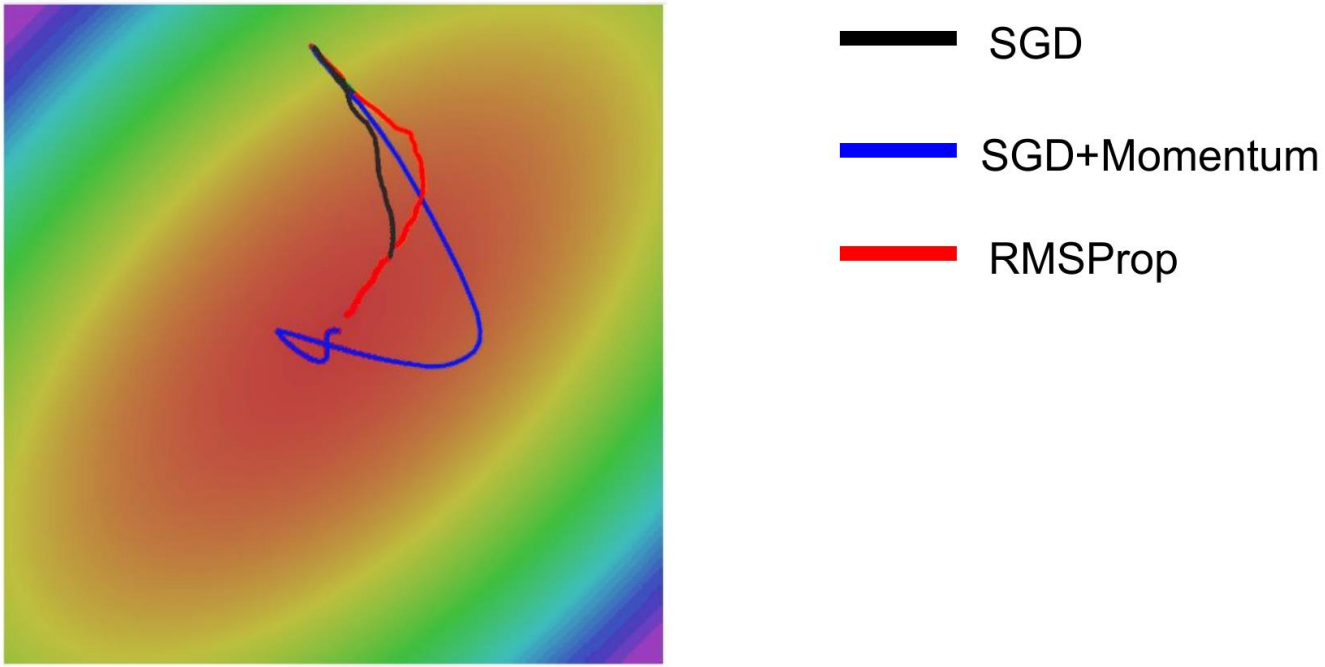
- In plateaus, take large steps as they do not have much risk. In steep areas take smaller steps
- Normalize gradient by running average of gradient norm

$$\boldsymbol{g}_k = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_k)$$
$$\boldsymbol{v}_{k+1,i} = \gamma \boldsymbol{v}_{k,i} + (1-\gamma)\boldsymbol{g}_{k,i}^2$$
$$\boldsymbol{\theta}_{k+1,i} = \boldsymbol{\theta}_{k,i} - \frac{\eta}{\sqrt{\boldsymbol{v}_{k+1,i} + \epsilon}} \boldsymbol{g}_{k,i}$$

  - $\boldsymbol{v}_{k,i}$ computes running average of the squared gradients (root mean square, RMS)
  - with a small ϵ to prevent division by zero
  - This algorithm is called ADADELTA or RMSProp

[Zeiler, 2012, ADADELTA - An Adaptive Learning Rate Method]

RMSProp



Adaptive Momentum (Adam)

Combine momentum term with gradient normalization:

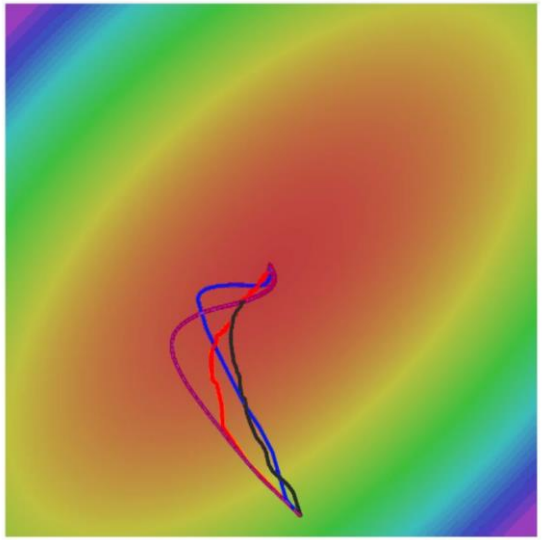
$$
\begin{aligned}
\boldsymbol{g}_k &= \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_k) \\
\boldsymbol{v}_{k+1,i} &= \gamma_1 \boldsymbol{v}_{k,i} + (1-\gamma_1)\boldsymbol{g}_{k,i}^2 \quad \dots \text{ gradient norm} \\
\boldsymbol{m}_{k+1} &= \gamma_2 \boldsymbol{m}_k + (1-\gamma_2)\boldsymbol{g}_k \quad \dots \text{ momentum} \\
\boldsymbol{\theta}_{k+1,i} &= \boldsymbol{\theta}_{k,i} - \underbrace{\frac{\eta c_2(k)}{\sqrt{c_1(k)\boldsymbol{v}_{k+1,i} + \epsilon}}}_{\text{norm-based scaling}} \boldsymbol{m}_{k+1,i}
\end{aligned}
$$

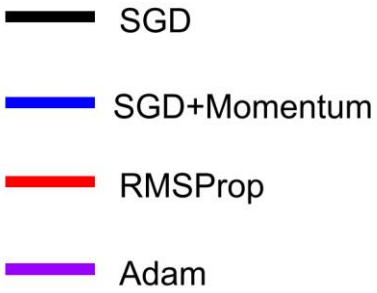
- Initialization $\boldsymbol{m}_0 = \mathbf{0},\ \boldsymbol{v}_0 = \mathbf{0}$ leads to underestimation fixed by $c_i(k) = \frac{1}{1-\gamma_i^k}$
- Choose $\gamma_1 = 0.9,\ \gamma_2 = 0.999$ and $\epsilon = 10^{-8}$. Not too sensitive to parameter changes
- **Note:** Violates convergence guarantees...

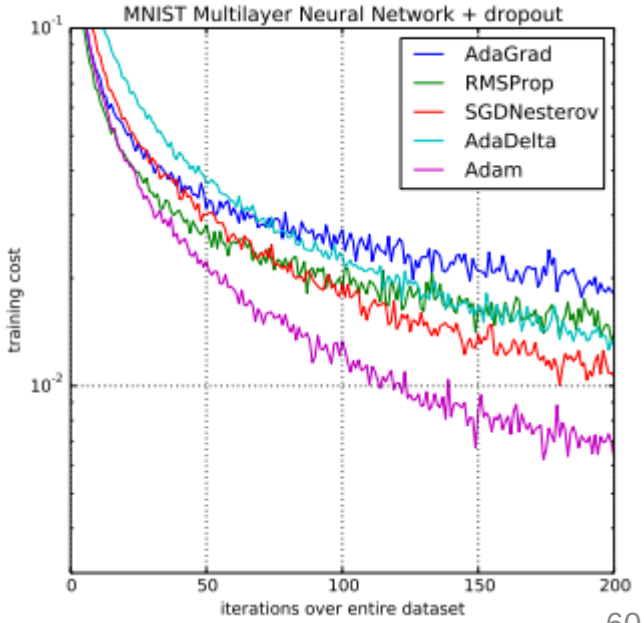
[Kingma et. al, 2015, Adam: A Method for Stochastic Optimization]

Comparison of different algorithms

Training on MNIST (hand written digits) dataset

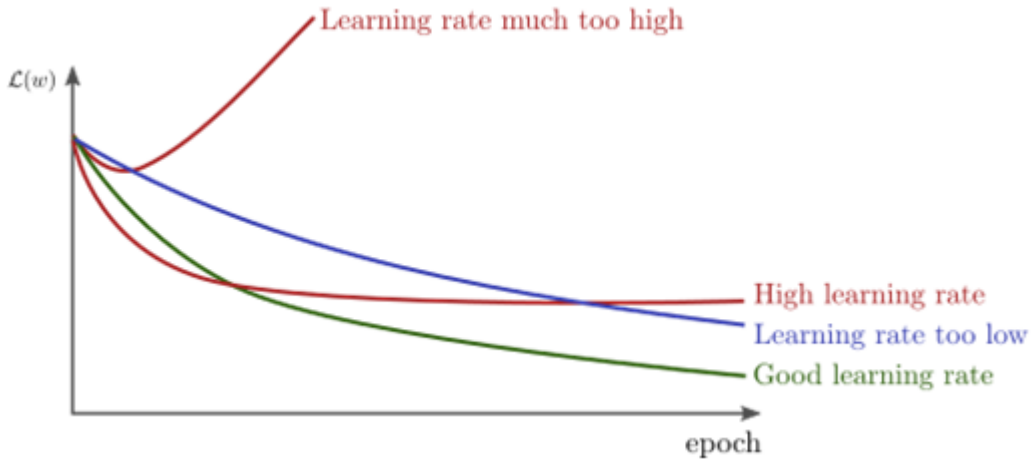






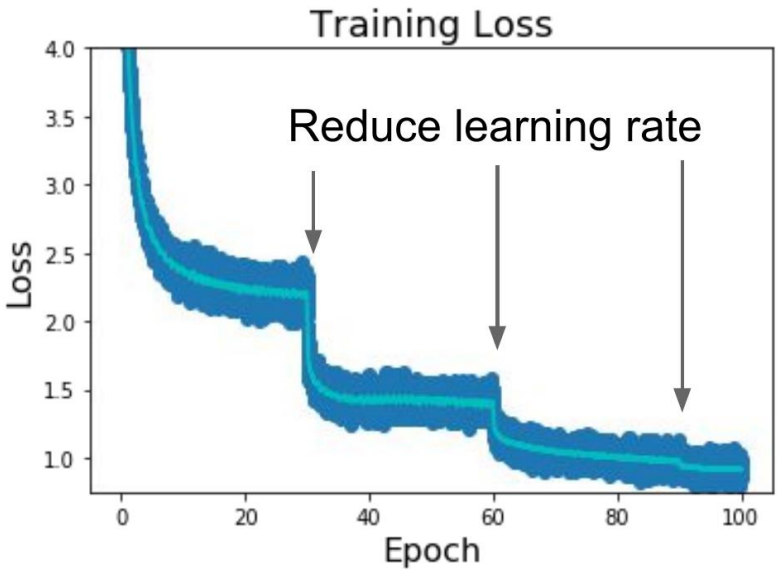
Learning rates

SGD, SGD+Momentum, RMSProp, Adam all have “base-learning rate” as a hyperparameter



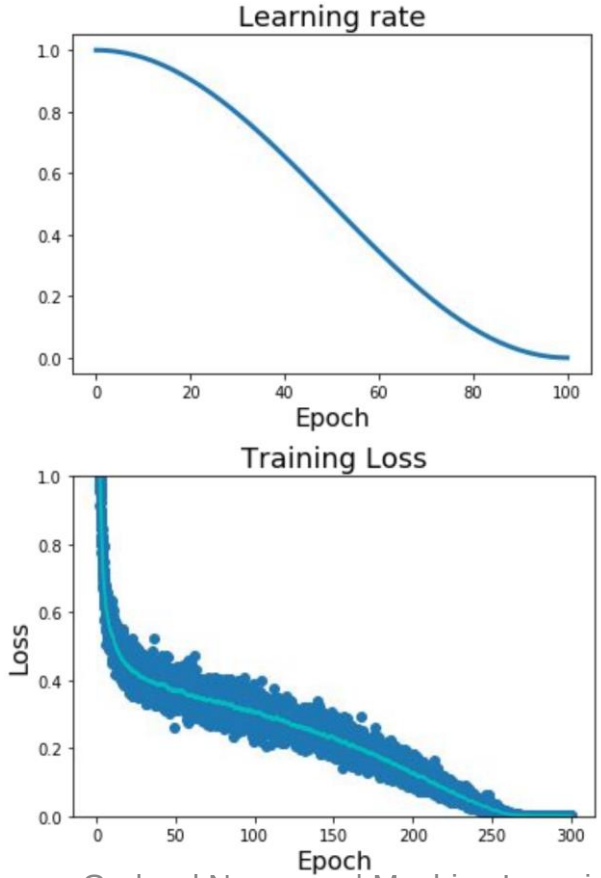
- Can we also choose the learning rate adaptively?

Learning rate decay



- **Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Learning rate decay

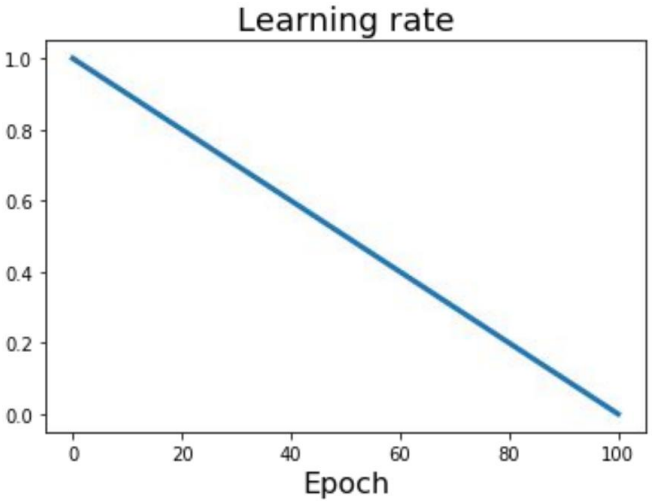


- **Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
- **Cosine:**

$$\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

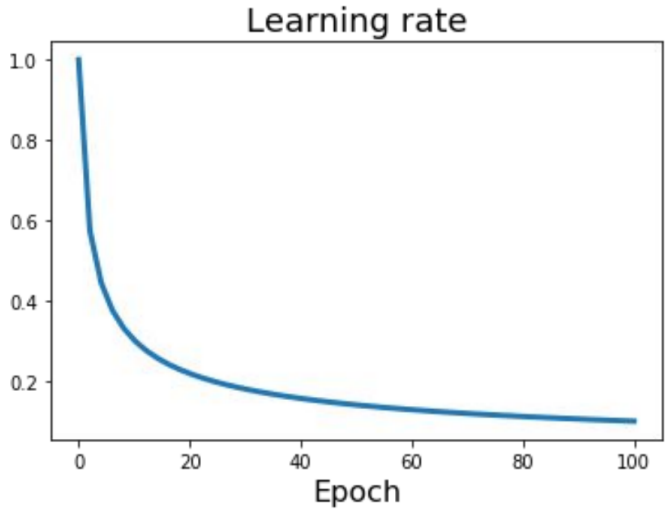
Learning rate decay



- **Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
- **Cosine:**
$$\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$$
- **Linear:** $\alpha_t = \alpha_0(1 - t/T)$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Learning rate decay



- **Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
- **Cosine:** $$\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$$
- **Linear:** $\alpha_t = \alpha_0(1 - t/T)$
- **Inverse sqrt:** $\alpha_t = \alpha_0/\sqrt{t}$

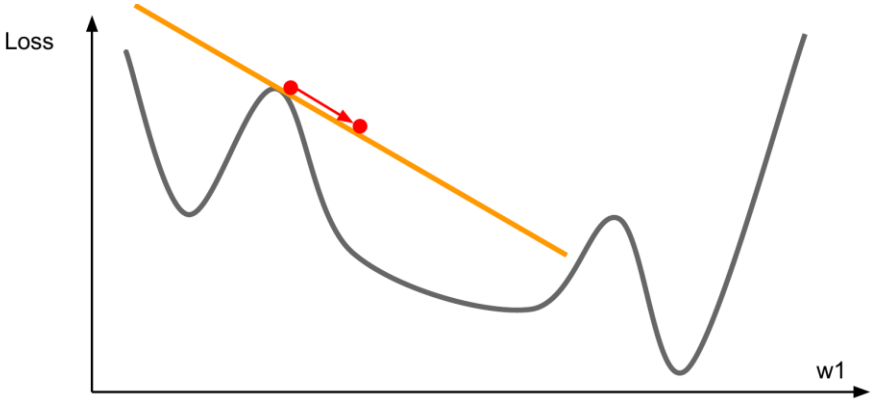
Not clear which one works best…

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

First vs. second order optimization

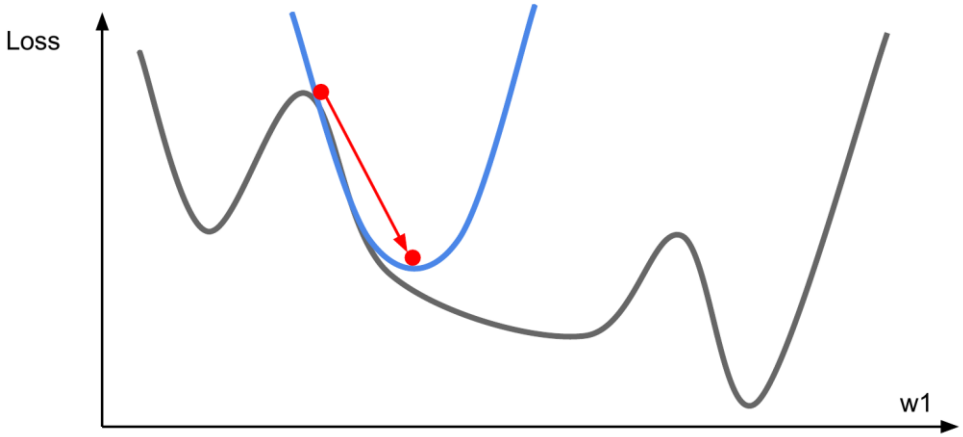
First order optimization:

1. Use gradient to form linear approximation
2. Step in the direction of the minimum of the approximation



Second order optimization:

1. Use gradient and Hessian to form quadratic approximation
2. Step to the minimum of the approximation



2nd order methods

2nd order Taylor approximation:

$$\mathcal{L}(\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \boldsymbol{g} + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

- With $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ is the gradient and $\boldsymbol{H} = \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ is the Hessian matrix

Solving for $\boldsymbol{\theta}$ yields a Newton update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \boldsymbol{g}$$

Properties:

- ✓ No hyperparameters
- ✓ No learning rate
- ✓ Less iterations required

- × Hessian has O(N^2) parameters
- × Inverse is O(N^3)
- × N is huge (several millions)!

2nd order methods

- **Quasi-Newton methods (BGFS most popular):** instead of inverting the Hessian ($O(N^3)$), approximate inverse Hessian with rank 1 updates over time ($O(N^2)$ each).
- **L-BFGS (Limited memory BFGS):** Does not form/store the full inverse Hessian.
  - Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely
  - Does not transfer very well to mini-batch setting. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

In practise:

- Adam is a good default choice in most cases
- If you can afford to do full batch updates then try out L-BFGS (and don't forget to disable all sources of noise)

Today‘s Agenda!

Neural Networks:

- What is a Neuron?
- Architectures and Activation Functions
- Loss-functions
- Backpropagation and the Chain Rule
- Computation graphs

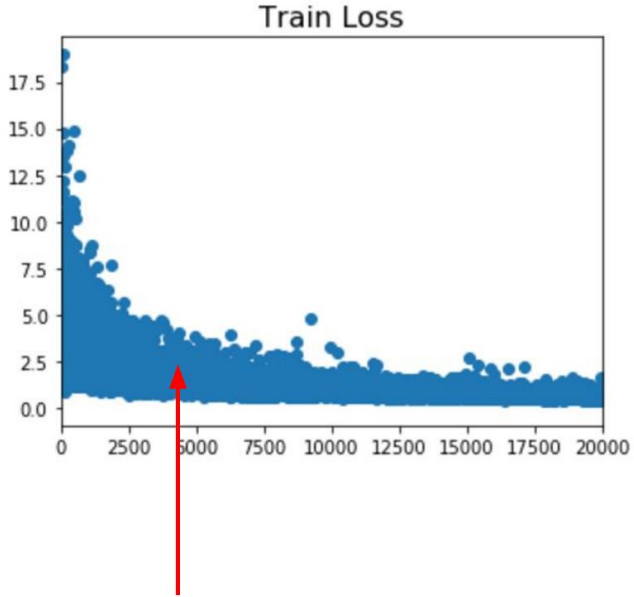
Advanced Topics:

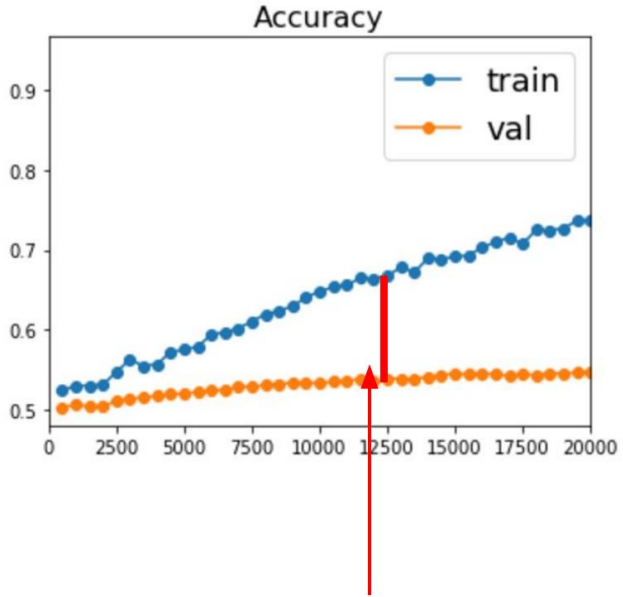
- Accelerating gradient descent
- Regularization in Neural Networks
- Practical considerations

Credit: M. Ren and M. MacKay, University of Toronto,
Fei-Fei Li & Justin Johnson & Serena Yeung, Stanford

Regularization with Neural Networks

The old story about overfitting…



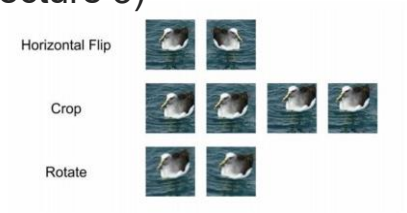


Better optimization algorithms help reduce training loss

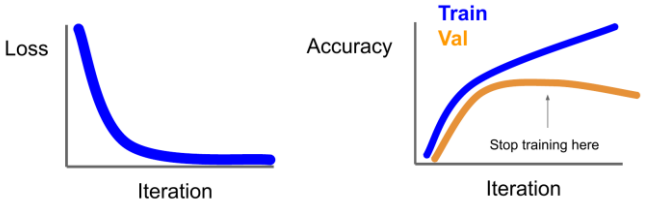
But we really care about error on new data - how to reduce the gap?

Regularization with neural networks

- Model selection (#layers, #neurons, etc…) (see lecture 3)
- Data augmentation (see lecture 3)



- Early stopping (see lecture 3)



- Regularization loss (see lecture 3)

$$\underset{\text{parameters } \boldsymbol{\theta}}{\arg\min} \sum_{i=1}^{N} l(\boldsymbol{x}_i, \boldsymbol{\theta}) + \lambda \, \text{penalty}(\boldsymbol{\theta})$$

- **Model ensembles**
- **Dropout**

Model ensembles

1. Train multiple independent models
2. At test time average their results (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra accuracy!

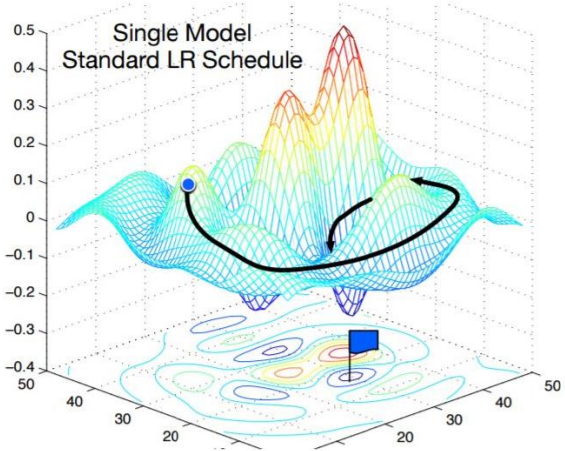
Why does this work?

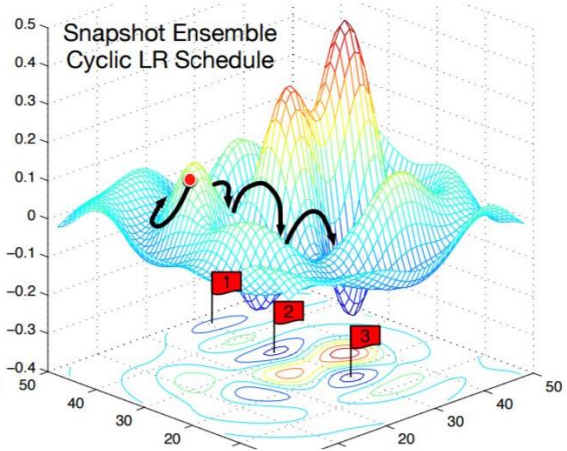
- We average over “unspecified behaviour” between the training data points
- Related to Bayesian Learning (see lecture 9)
- See recent NeurIPS tutorial on deep ensembes – they are currently the most accurate known models!

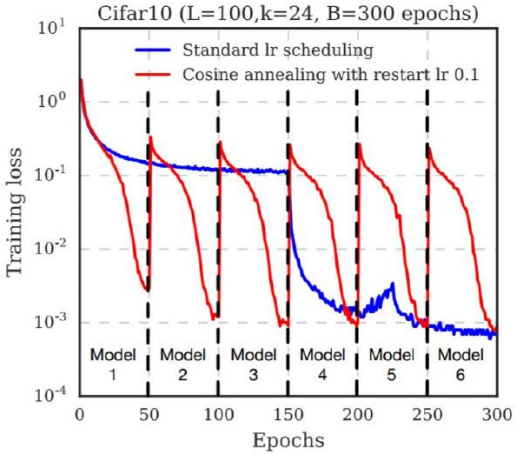
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017

Model ensembles

Instead of training independent models, use multiple snapshots of a single model during training!





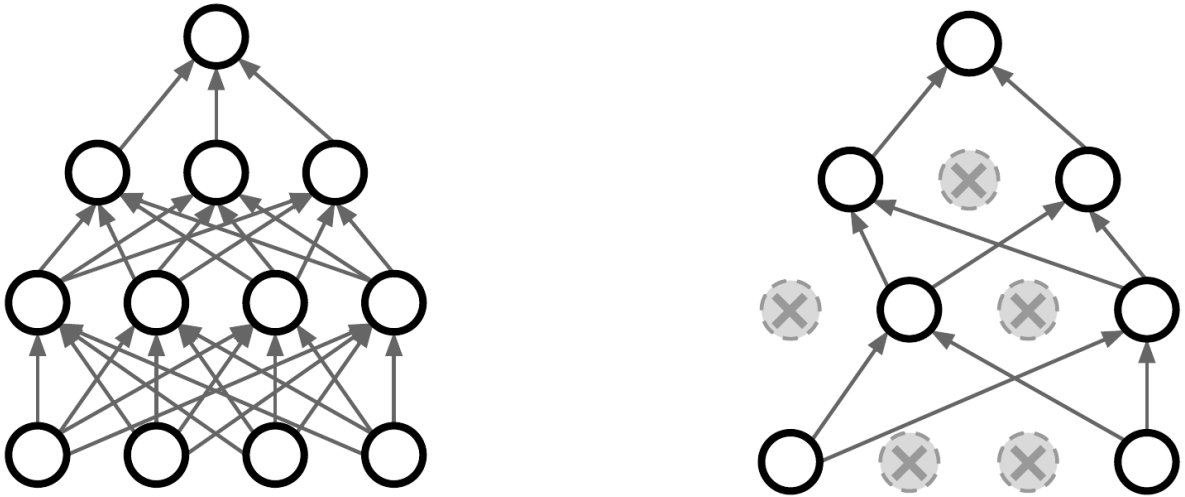


Use cyclic learning rates to make it work even better

Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017

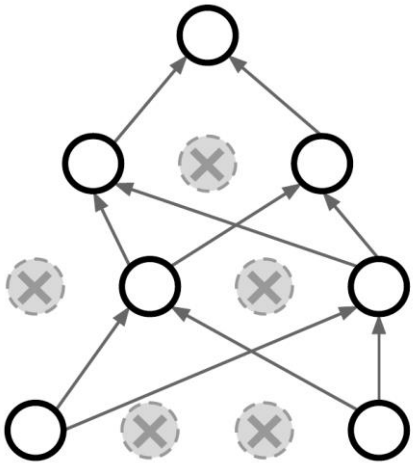
Dropout

- In each forward pass, randomly set some neurons to zero
- Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Dropout



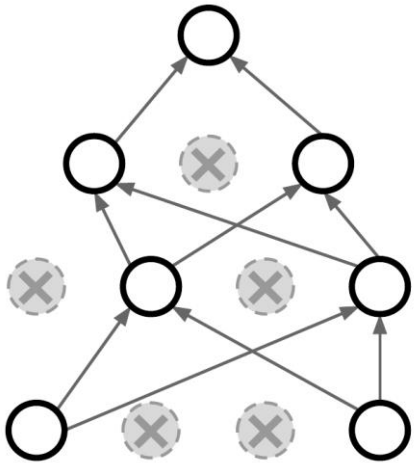
Why is this a good idea?

- Forces the network to have a redundant representation;
- Prevents co-adaptation of features

Interpretation as ensembles:

- Dropout is training a large ensemble of models (that share parameters).
- Each binary mask is one model
- An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Dropout: Testing



The output of the network is now random:

- For testing, we want to evaluate the expectation!

Ensemble view:

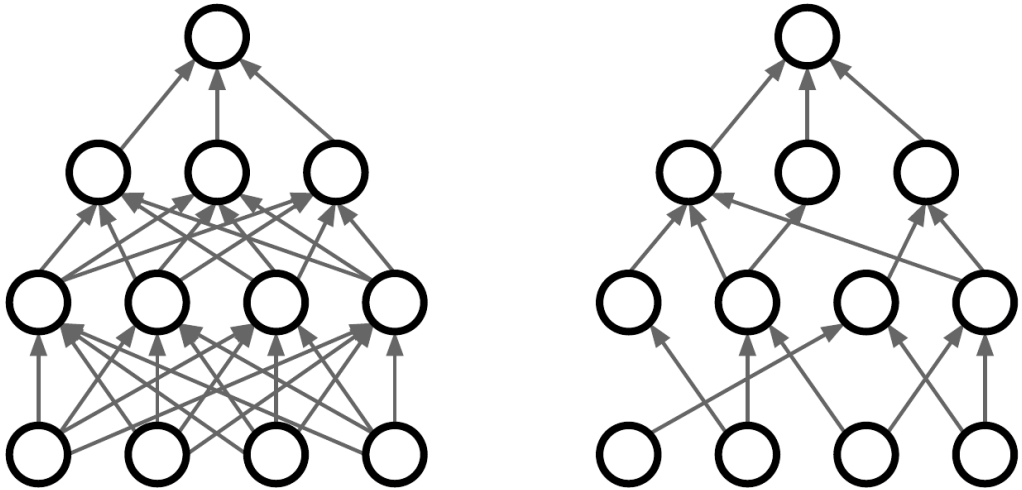
- Average over multiple dropout masks (computationally expensive but quite robust)
- Also allows to get uncertainty estimates (not covered)

Expectation view:

- Compute the expected input to the activation functions
- Multiply each weight by the dropout rate

Drop Connect

- **Training:** Drop connections between neurons (set weights to 0)
- **Testing:** Use all the connections



Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

Today‘s Agenda!

Neural Networks:

- What is a Neuron?
- Architectures and Activation Functions
- Loss-functions
- Backpropagation and the Chain Rule
- Computation graphs

Advanced Topics:

- Accelerating gradient descent
- Regularization in Neural Networks
- Practical considerations

Credit: M. Ren and M. MacKay, University of Toronto, Fei-Fei Li & Justin Johnson & Serena Yeung, Stanford

Practical considerations

… or the black-arts of training neural networks

Crucial for getting good performance with Neural Networks:

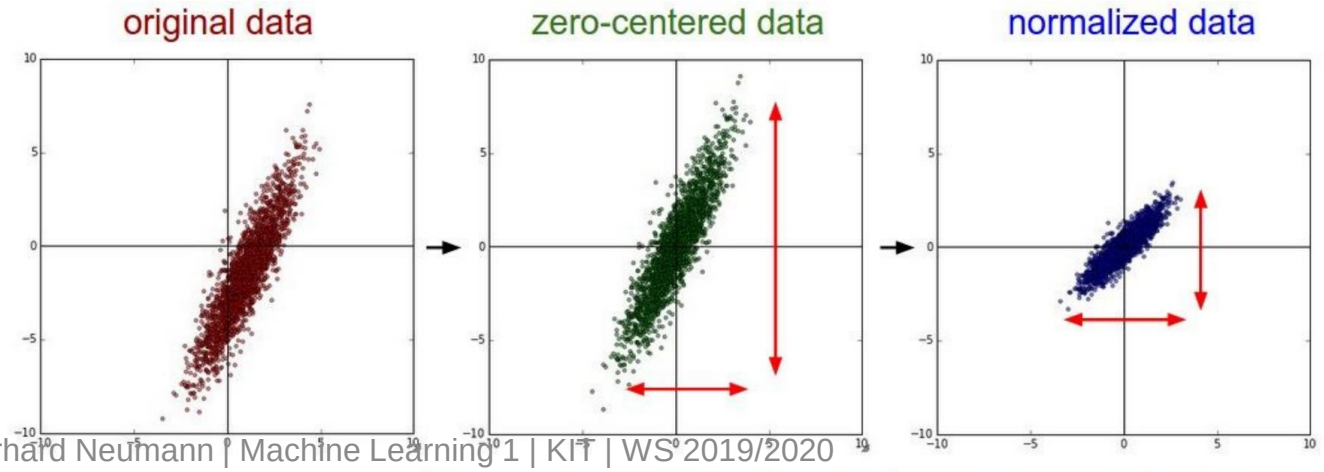
- Data preprocessing
- Weight initialization
- Hyperparameter optimization

Data preprocessing

NNs work best with zero-mean unit variance data

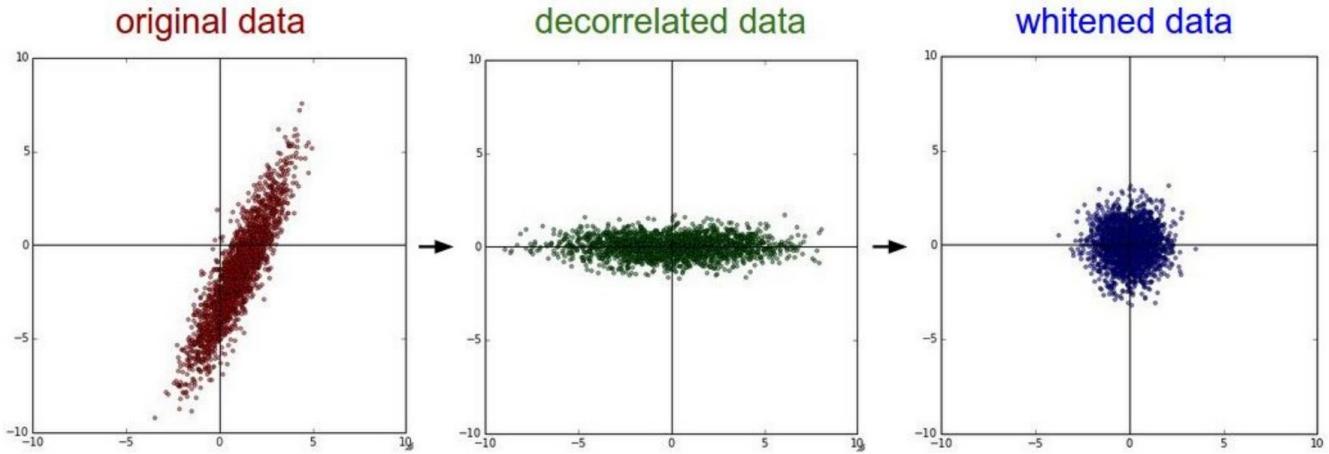
$$\tilde{\mathbf{x}}_i = (\mathbf{x}_i - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma}$$

- Where $\boldsymbol{\mu}$ is the mean, $\boldsymbol{\sigma}$ is the standard deviation and $\oslash$ the element-wise division operator
- Why? network initialization strategies are optimized for zero-mean unit variance!



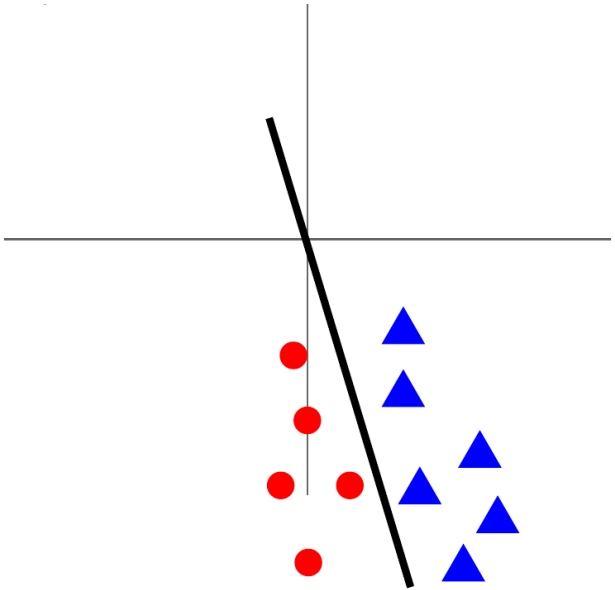
Data pre-processing

In practice, you might also see PCA (lecture 5) and whitening (of low-d data)

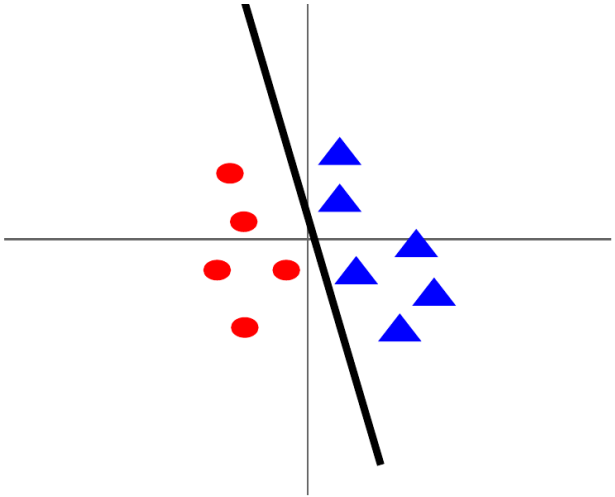


Data preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

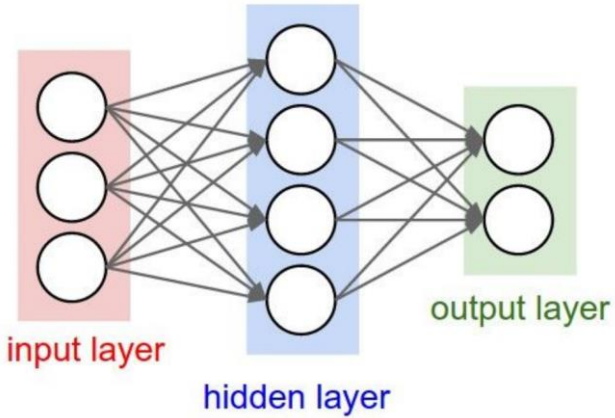


After normalization: less sensitive to small changes in weights; easier to optimize



Weight initialization

Q: What happens if we initialize all weights constantly for such a network?



- All the gradients are the same!
- Network can never learn “distinct features”

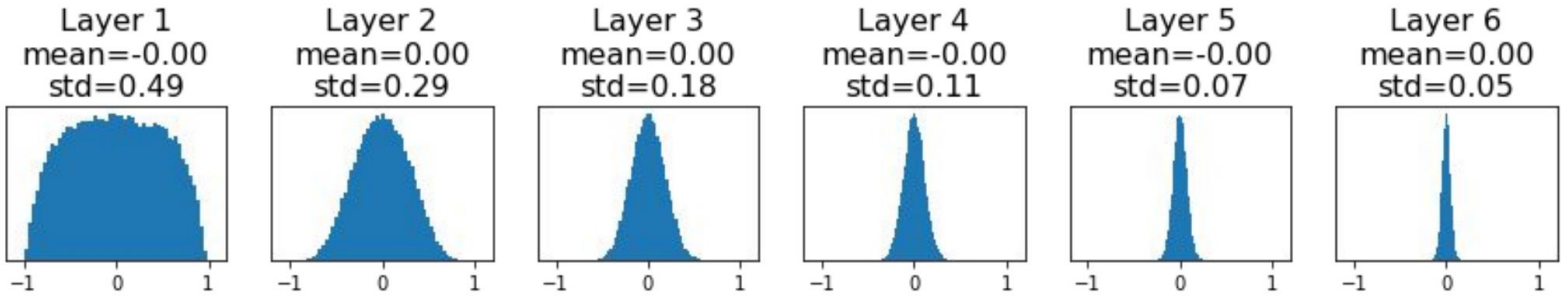
We need random initialization!

- **First idea:** Small random numbers (e.g. gaussian with zero mean and 1e-2 standard deviation)
- Works ok for small networks… how about deep networks?

Weight initialization

Activation Statistics:

- 6 layer, tanh activation, 4096 units per layer
- Zero mean, unit variance inputs
- 0.01 standard deviation for weights



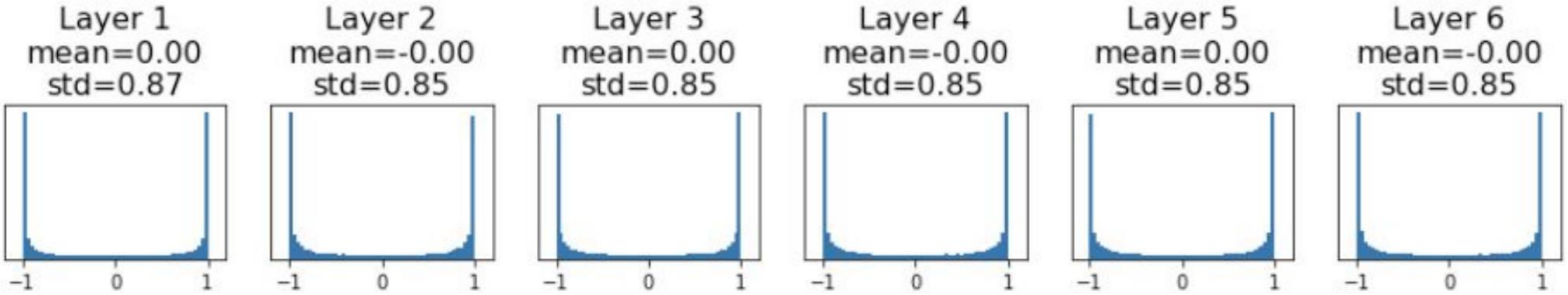
All activations tend to zero for deeper networks!

- No gradients ☹ !

Weight initialization

Activation Statistics:

- 6 layer, tanh activation, 4096 units per layer
- Zero mean, unit variance inputs
- 0.05 standard deviation for weights



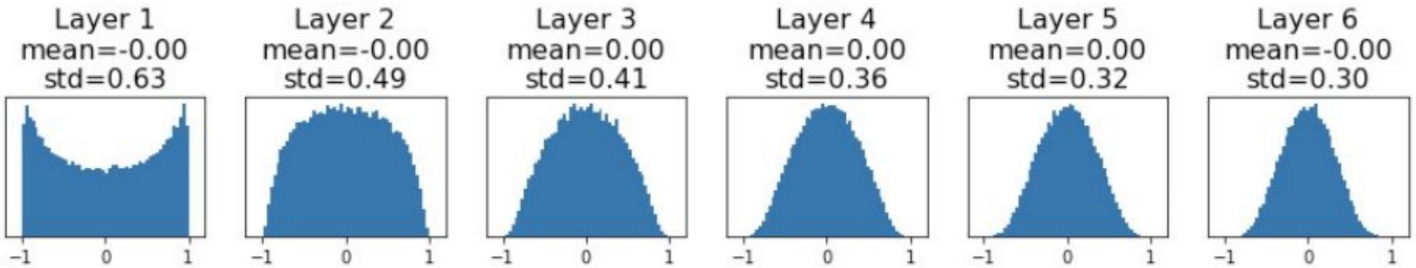
Most activations are saturated!

- Vanishing gradients !

Weight initialization

Xavier initialization:

- Use the following standard deviation: $\sigma_{\mathbf{W}} = \frac{1}{\sqrt{D_{\text{in}}}}$



- "Just right": Activations are nicely scaled for all layers!
  - Can be derived by computing the variances of each layer (assuming linear units)

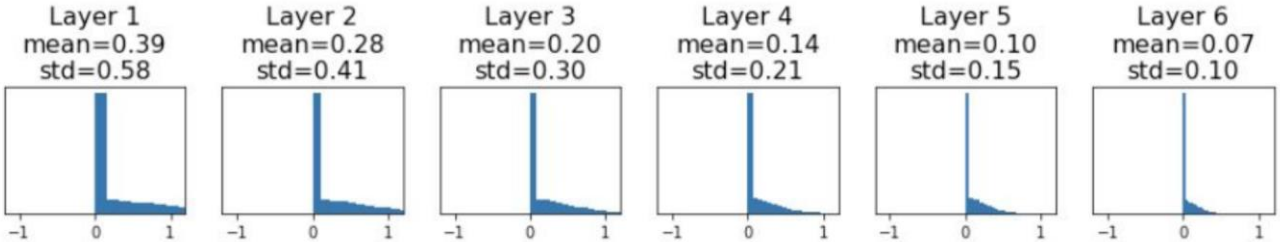
Weight Initialization

What about ReLUs?

- Xavier initialization:

$$\sigma_{\mathbf{W}} = \frac{1}{\sqrt{D_{\text{in}}}}$$

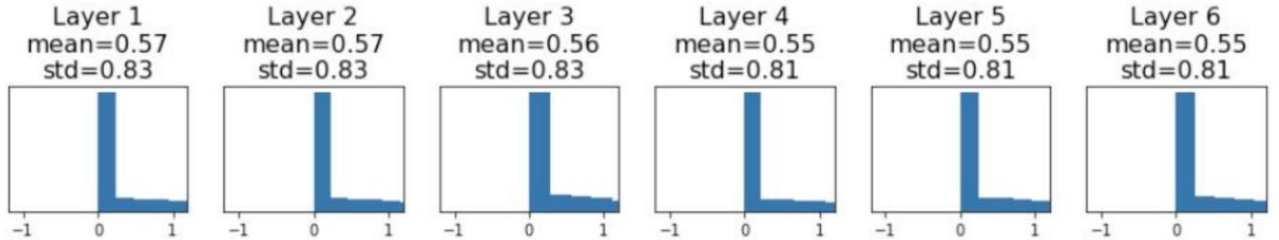
  - Activations again go to 0 ☹



- ReLU correction

$$\sigma_{\mathbf{W}} = \frac{2}{\sqrt{D_{\text{in}}}}$$

  - Activations nicely balanced



Learning with NNs in practise

Step 1: Check initial loss

Turn off weight decay (L2 regularization), sanity check loss at initialization
e.g. log(C) for softmax with C classes

Learning with NNs in practise

Step 1: Check initial loss

Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches)

- Fiddle with architecture, learning rate, weight initialization
- Loss not going down? LR too low, bad initialization
- Loss explodes to Inf or NaN? LR too high, bad initialization

Learning with NNs in practise

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations.

- Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Learning with NNs in practise

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

- Good weight decay to try: 1e-4, 1e-5, 0

Learning with NNs in practice

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

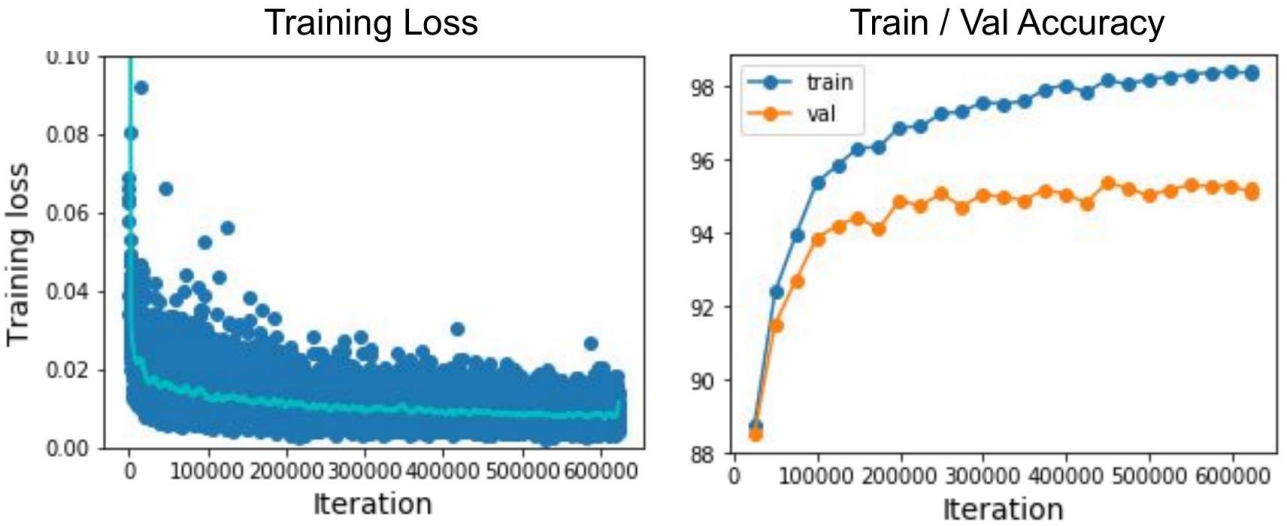
Step 5: Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Learning with NNs in practice

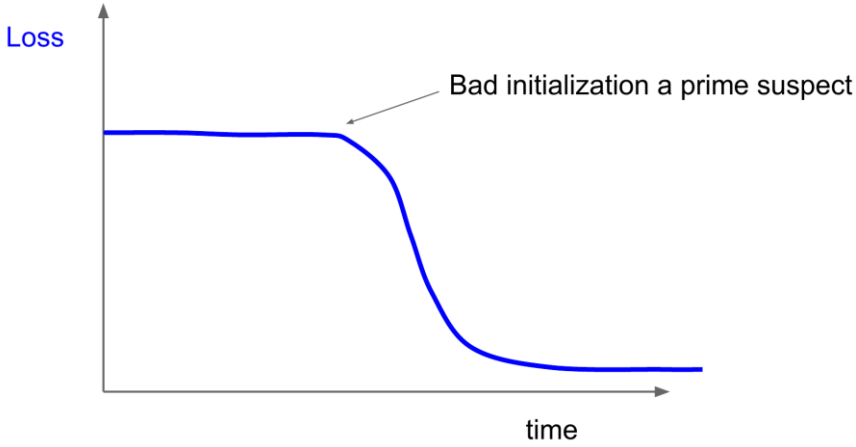
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves

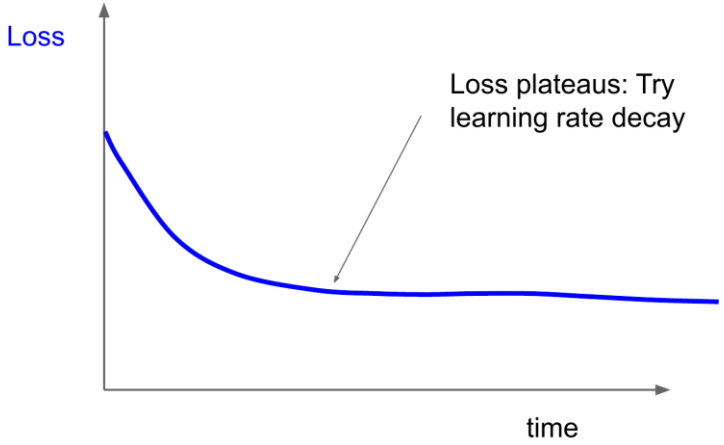
Look at learning curves!

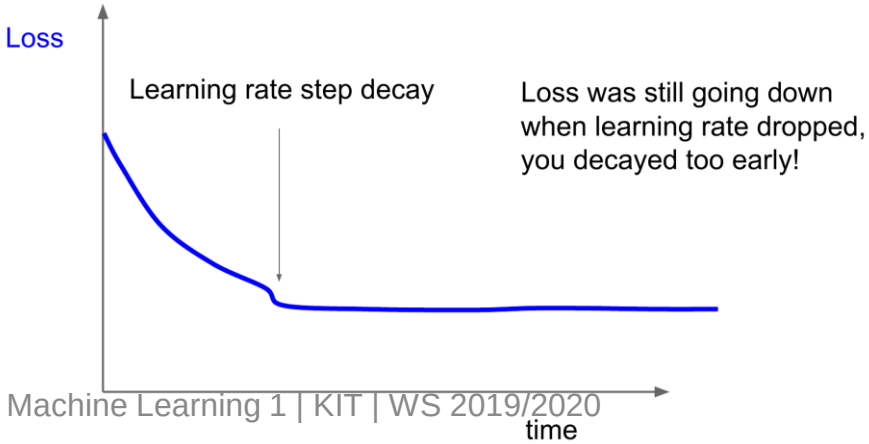


Losses may be noisy, use a scatter plot and also plot moving average to see trends better

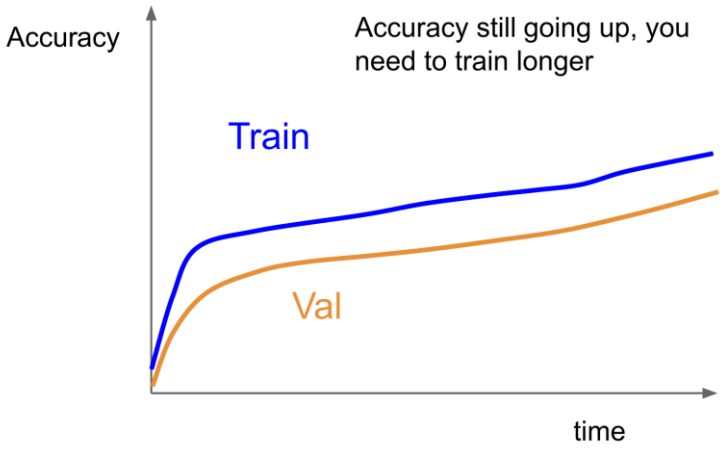
Different error sources

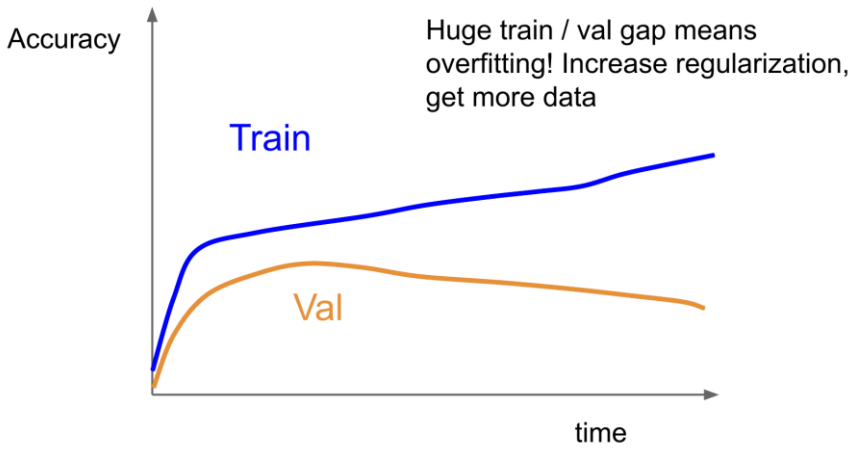


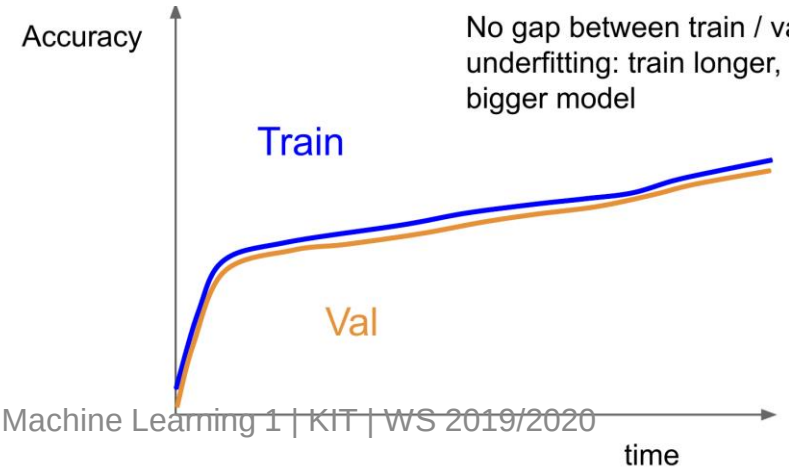




Different error sources







Learning with NNs in practice

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves
Step 7: GOTO step 5

Neural Nets summary

Neural Nets are extremely powerful and complex non-linear representations

- Can be seen as feature extraction for regression and classification
- Yet, need a lot of samples
- Can easily overfit (Criticism: they often just learn the data by heart)

The last 5-10 years they have "taken over" ML

- Deep Neural Networks (depth > 2) set the gold standard in many fields today
- Computer Vision, Natural Language Processing, Robotics and Reinforcement Learning, Time-Series Prediction, etc…
- **Why now?** More data, more computation… but almost same algorithms than 40 years ago

Yet, we do not fully understand them:

- Training them needs experience and a lot of computation
- Actually… we have no idea why they work so well
  - Theory says, it shouldn't (as we often have more parameters than training examples)

Takeaway messages

What have we learned today?

- What neural networks are and how they relate to the brain
- How neural networks build stacks of feature representations
- A network of one layer is enough, but in practice not a good idea
- How to do forward and backpropagation on computation graphs
  - How to use matrix calculus to obtain simpler gradient computations
- Different ways of doing fast gradient descent
  - Speedup training via momentum, gradient normalization and learning rate adaptation
  - How to initialize the parameters
- Why neural networks overfit and what you can do to about it

Self-test questions

- How does logistic regression relate to neural networks?
- What kind of functions can single layer neural networks learn?
- Why do we need non-linear activation functions?
- What activation functions can we use and what are the advantages/disadvantages of those?
- What output layer and loss function to use given the task (regression,
- classification)?
- Why not use a sigmoid activation function?
- Derive the equations for forward and backpropagation for a simple network
- What is mini-batch gradient descent? Why use it instead of SGD or full gradient descent?
- Why neural networks can overfit and what are the options to prevent it?
- Why is the initialization of the network important?
- What can you read from the loss-curves during training (validation and training loss)?
- How can we accelerate gradient descent? How does Adam work?