Statistical Machine Learning

https://cvml.ist.ac.at/courses/SML_W20

Christoph Lampert (with material by Andrea Palazzi and others)

I|S|T AUSTRIA

Institute of Science and Technology

Fall Semester 2020/2021

Lecture 10

Overview (tentative)

Date		no.	Topic
Oct 05	Mon	1	A Hands-On Introduction
Oct 07	Wed	2	Bayesian Decision Theory, Generative Probabilistic Models
Oct 12	Mon	3	Discriminative Probabilistic Models
Oct 14	Wed	4	Maximum Margin Classifiers, Generalized Linear Models
Oct 19	Mon	5	Estimators; Overfitting/Underfitting, Regularization, Model Selection
Oct 21	Wed	6	Bias/Fairness, Domain Adaptation
Oct 26	Mon	-	no lecture (public holiday)
Oct 28	Wed	7	Learning Theory I, Concentration of Measure
Nov 02	Mon	8	Learning Theory II
Nov 04	Wed	9	Learning Theory III, Deep Learning I
Nov 09	Mon	10	Deep Learning II
Nov 11	Wed	11	Deep Learning III
Nov 16	Mon	12	project presentations
Nov 18	Wed	13	buffer

(Non-convex) Numeric Optimization

Reminder: Stochastic Optimization

$$F(\theta) = \frac{1}{n}\sum_{i=1}^{n} f_i(\theta), \qquad \text{for differentiable functions } f_1, \dots, f_n.$$

Minibatch SGD

```
input step sizes $\alpha_1, \alpha_2, \dots$ (=learning rate)
input number of iterations, $T$
input mini-batch size $B$
1: initialize $\theta_0$
2: for $t = 1, \dots, T$ do
3:     $i_1, \dots, i_B \leftarrow B$ random indices
4:     $v \leftarrow \frac{1}{B}\sum_{j=1}^{B} \nabla f_{i_j}(\theta_{t-1})$
5:     $\theta_t \leftarrow \theta_{t-1} - \alpha_t v$
6: end for
output $\theta_T$, or average $\frac{1}{T-T_0}\sum_{t=T_0}^{T} \theta_t$
```

- Time for each iteration is proportional to B
- Variance of gradient estimate is reduced by $\frac{1}{B}$ compared to plain SGD
- Optimal batchsize is problem dependent, often as much as fits into GPU memory
- The computation of v can be performed in a parallel/distributed way.

Coordinate-Dependent Rate Selection: AdaGrad

In vanilla GD or SGD, each entry of the parameter vector is updated with the same learning rate. In some cases, it might be better to allow parameters to change at different rates.

AdaGrad

For parameter vector $\theta = (\theta_j)_{j=1,\dots,d}$,

- base learning rate α, small constant ϵ, e.g. $\epsilon = 10^{-8}$.

AdaGrad uses the update rule: $\theta_j^{(t)} \leftarrow \theta_j^{(t-1)} - \frac{\alpha}{\sqrt{G_{jj}+\epsilon}} v_j^{(t)}$ for $j = 1, \dots, d$

where $v^{(t)} = \nabla f_i(\theta_{t-1})$ is the gradient and $G_{jj} = \sum_{\tau=1}^{t} (v_j^{(\tau)})^2$.

The learning rate it increased for dimensions that are updated rarely or only little ($v^{(\tau)}$ often 0 or small). It is decreased for parameters that are updated often by large amounts.

AdaGrad was observed to help learning, e.g., when different data dimensions have different sparsity levels, e.g. one-hot vectors in natural language processing.

Adaptive Learning Rate Selection: RMSProp

RMSProp resembles AdaGrad, but its learning rate schedule is more flexible.

RMSProp

For parameter vector $\theta = (\theta_j)_{j=1,\dots,d}$,

- base learning rate α, small constant ϵ, e.g. $\epsilon = 10^{-8}$,
- discount factor γ, e.g. $\gamma = 0.9$.

RMSProp uses the update rule: $\theta_j^{(t)} \leftarrow \theta_j^{(t-1)} - \frac{\alpha}{\sqrt{G_j^{(t)} + \epsilon}} v_j^{(t)}$ for $j = 1, \dots, d$

with

- $G_j^{(t)} = \gamma G_j^{(t-1)} + (1-\gamma) v_j^{(t)}$ (exponentially weighted running average)

Main idea: divide the learning rate for each weight by a running average of the magnitudes of recent gradients for that weight.

Adam [Kingma, Ba. "Adam: A Method for Stochastic Optimization", ICLR 2015] – 47774 citations as of 06/11/2020

Adam is similar to RMSProp, but also uses a form of momentum.

Adaptive Moment Estimation (Adam)

For parameter vector $\theta = (\theta_j)_{j=1,\dots,d}$, and

- base learning rate α, small constant ϵ, e.g. $\epsilon = 10^{-8}$, discount factors $\gamma_1, \gamma_2 < 1$.

Exponentially weighted running averages:

- $m_j^{(t)} \leftarrow \gamma_1 m_j^{(t-1)} + (1-\gamma_1)\nabla f(\theta^{(t-1)})$ $\qquad \hat{m}_j^{(t)} = \frac{m_j^{(t)}}{1-\gamma_1^t}$
- $v_j^{(t)} \leftarrow \gamma_2 v_j^{(t-1)} + (1-\gamma_2)[\nabla f(\theta^{(t-1)})]^2$ $\qquad \hat{v}_j^{(t)} = \frac{v_j^{(t)}}{1-\gamma_2^t}$

Adam update rule: $\theta_j^{(t)} \leftarrow \theta_j^{(t-1)} - \frac{\alpha}{\sqrt{\hat{v}_w^{(t)}} + \epsilon}\hat{m}_j^{(t)}$ for $j = 1, \dots, d$

- often Adam converges very fast, but sometimes not at all
- Models trained by Adams sometimes generalize worse than with vanilla/momentum SGD
  $\rightarrow$ for best performance one has to try different optimizers

Deep Learning: Regularization

Overfitting

On first sight, overfitting should be nightmare for deep networks:

- they have often tens or hundreds of millions of parameters
- they are trained on a many data points, but not hundreds of millions

Surprisingly, this is not the case:

- deep networks are often trained to very small training loss (even 0 training error), but this does not cause test errors to get outrageously high.
- in fact, increasing model size very often leads to better test error, if done properly.
- the exact reasons for this are not well understood, yet.

Nevertheless, some forms of regularization are popular in deep learning, too:

- early stopping
- dropout
- data augmentation

We'll discuss them only briefly, as we encountered them before.

Early Stopping

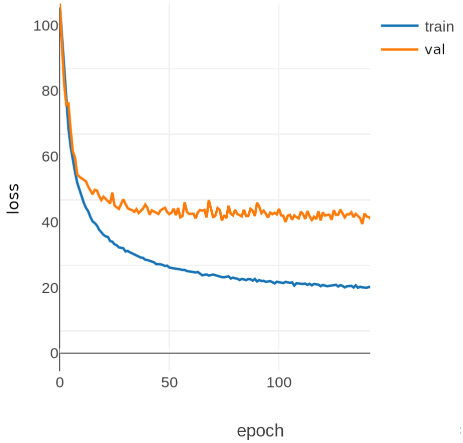
Deep learning training is organized in epochs, i.e. passes through the dataset, typically in random order.

Typically, one trains for a certain number of epochs, until the loss on a validation set stops decreasing (or even increases).

Note: training and validation loss are typically not monotonic, so the exact criterion differ, e.g.

- loss did no decrease for k subsequent epochs
- average loss over k epochs did not decrease
- "the picture looks like it converged"
- "I don't want to wait any longer"

"learning curve"



Dropout

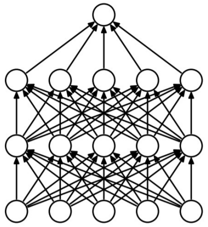
Dropout is a regularization method used almost exclusively for neural networks.

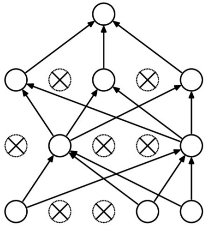
During the training process, for any forward pass, the outputs of each neuron is set to zero with a *drop probability* p. Other outputs are multiplied by $\frac{1}{1-p}$ to compensate.

This can be seen as sampling at each training step a different (sparse) sub-network and updating only the corresponding portion of parameters.

At prediction time, no neurons are dropped.

This can be interpreted as taking the average prediction of the ensemble of sub-networks.


(a) Standard Neural Net


(b) After applying dropout.

[Srivastava *et al*. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

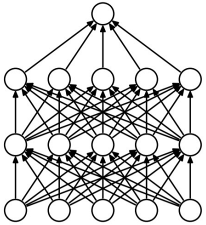
DropConnect

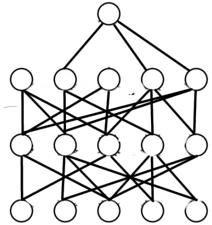
DropConnect regularization works similarly to dropout, but on the level of weights rather than neurons.

During the training process, for any example any weight is set to 0 (dropped) with a *drop probability* p. Other weights are multiplied by $\frac{1}{1-p}$ to compensate.

The result are still sparse subnetworks, but mainly number of weigths is changed compared to the original network, not (so much) the number of neurons.

At prediction time, no weights are dropped.


(a) Standard Neural Net


(a) DropConnect

[Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, Rob Fergus. "Regularization of Neural Networks using DropConnect", 2013.]

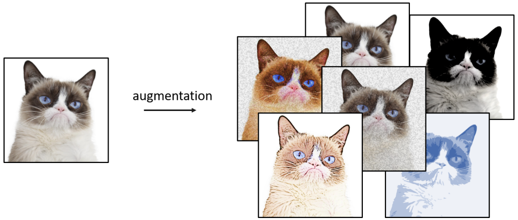
Data Augmentation

If properties of the input data are well understood, **data augmentation** is often used.

Example (Object recognition from images)

For images, we know that certain transformations do not affect which object is visible, e.g.

- small amounts translation, rotation or scaling
- minor color variations
- small amounts of noise

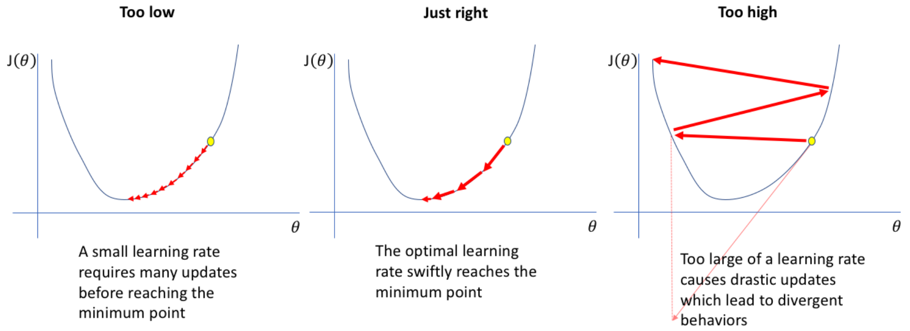


Adding images to the training set in which such changes have artificially be added can improve generalization.
Typically, the additional images are not precomputed but created on the fly.

Deep Learning: Learning Rates

What learning rate to use?

Intuition from ordinary (non-stochastic) gradient descent:

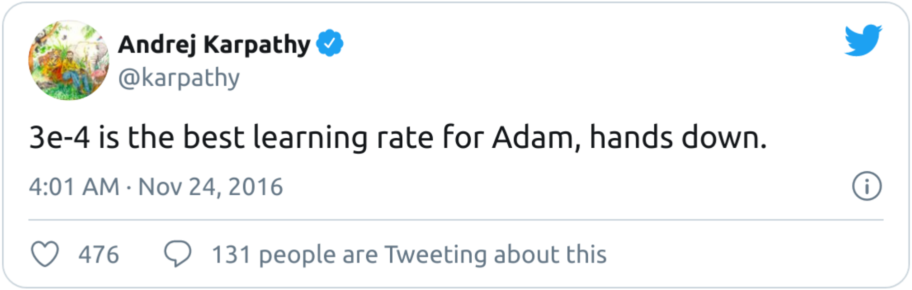


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

- same learning rate α can be used for all steps of the optimization
- if chosen small enough, convergence to optimum is guaranteed
  - close to optimum, gradient gets shorter, so updates get smaller
  - at optimum, gradient is zero, update is zero
- unfortunately, for stochastic, situation is less easy

What’s the best learning rate to use?

What's the best learning rate to use?



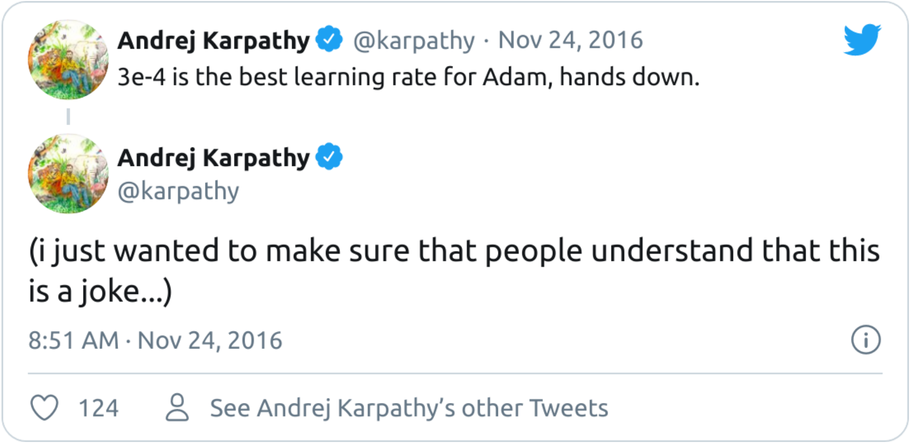
Andrej Karpathy
@karpathy

3e-4 is the best learning rate for Adam, hands down.

4:01 AM · Nov 24, 2016

476 131 people are Tweeting about this

What's the best learning rate to use?



Andrej Karpathy @karpathy · Nov 24, 2016

3e-4 is the best learning rate for Adam, hands down.

Andrej Karpathy
@karpathy

(i just wanted to make sure that people understand that this is a joke...)

8:51 AM · Nov 24, 2016

124 See Andrej Karpathy's other Tweets

Choosing a Learning Rate

The best learning rate is problem dependent:

Procedure to find a good learning rate

```
input  $A$: range of possible values for $\alpha$, e.g. $A = \{2^{-15}, \dots, 2^{-1}\}$
  for $\alpha \in A$ do
    $F_\alpha \leftarrow$ run optimizer with learning rate $\alpha$ starting at $\theta^{(0)}$
  end for
  $\alpha^* \leftarrow \mathbf{argmin}_\alpha F_\alpha$
output  $\alpha^*$
```

Procedure makes no sense if we train until convergence every time.

Instead:

- train only for a small number of steps, e.g. 1 epoch
- if necessary, use only a subset of the data

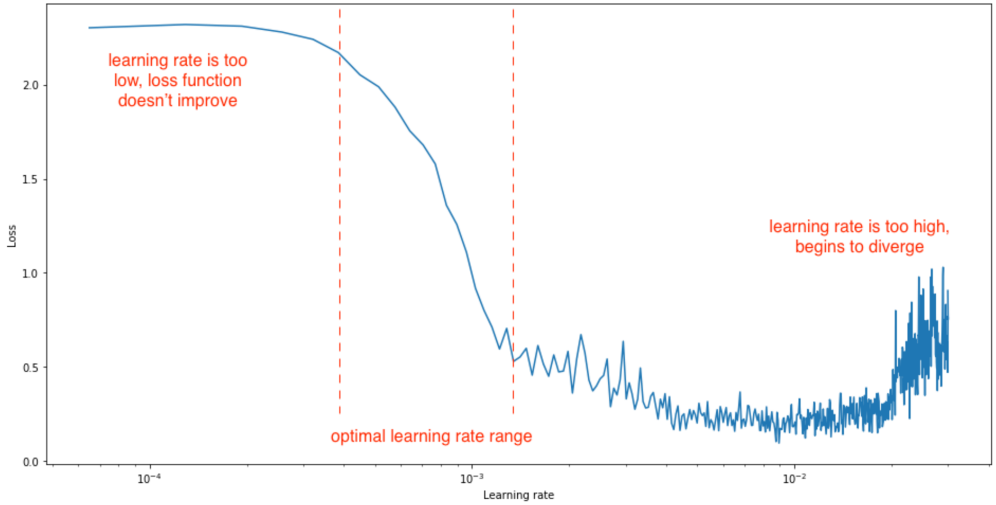
Choosing a Learning Rate

Single-Pass Procedure

```
input $\alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$ target range for $\alpha$
input $\gamma$ decay factor, e.g. $\gamma = 0.999$
  $\theta \leftarrow \theta^{(0)}$
  $\alpha \leftarrow \alpha_{\mathsf{min}}$
  while $\alpha < \alpha_{\mathsf{max}}$ do
    $\theta \leftarrow$ one more step of (mini-batch) SGD with learning rate $\alpha$
    $F_\alpha \leftarrow F(\theta)$
  end while
```

Choose α by looking at resulting curve.

What's the learning rate to use?



- make sure all three regions are visible
- pick learning rate where curve has strong downwards slope

Optimal Learning Rate?

Formally, it is known how the learning rate influences SGD's converge speed:

Theorem (Convergence Rate of SGD)

Let $f : \mathbb{R}^d \to \mathbb{R}$ be bounded from below by 0, let ∇f be L-Lipschitz continuous. Assume there exists a $\sigma^2 > 0$ such that $\mathbb{E}[\|\nabla f_i(\theta)\|^2] \leq \sigma^2$ for all θ (stochastic variance is bounded). Then SGD with small enough step sizes $\alpha_1, \alpha_2, \dots$ fulfills

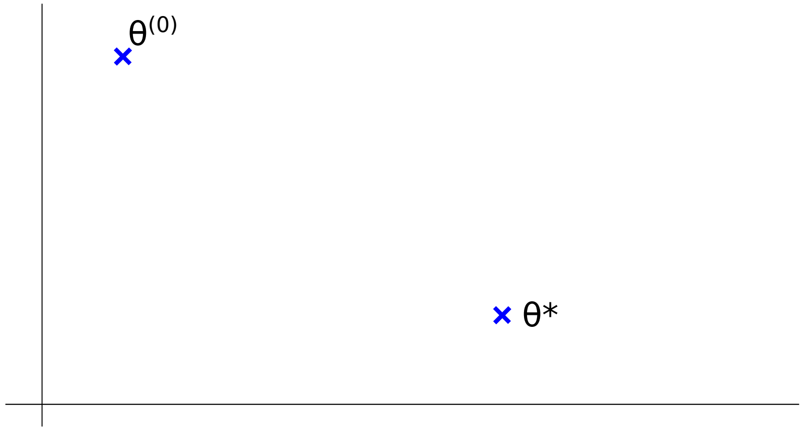
$$\min_{t=1,\dots,T} \{\mathbb{E}\,\|\nabla f(\theta^{(t)})\|^2\} \leq \frac{f(\theta^{(0)}) - f(\theta^*)}{\sum_{t=1}^{T} \alpha_t} + \frac{L\sigma^2}{2} \frac{\sum_{t=1}^{T} (\alpha_t)^2}{\sum_{t=1}^{T} \alpha_t}$$

(conditions are not fulfilled for ReLU networks, but similar results exist)

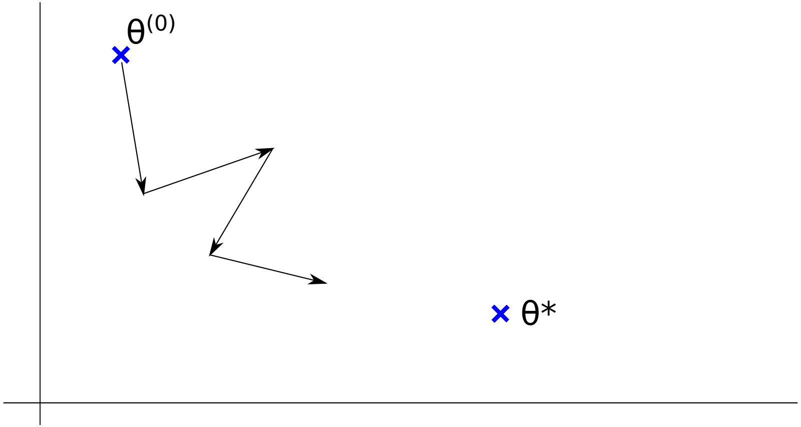
Observation: for constant learning rate, $\alpha_t = \alpha$ for $t = 1, 2, \dots$:

- if $\sigma = 0$ (no stochasticity), right hand side converges to 0 with rate $O(\frac{1}{T})$
- with $\sigma > 0$, right hand side does not converge to 0, (because $\frac{\sum_{t=1}^{T} \alpha^2}{\sum_{t=1}^{T} \alpha} = \alpha$)

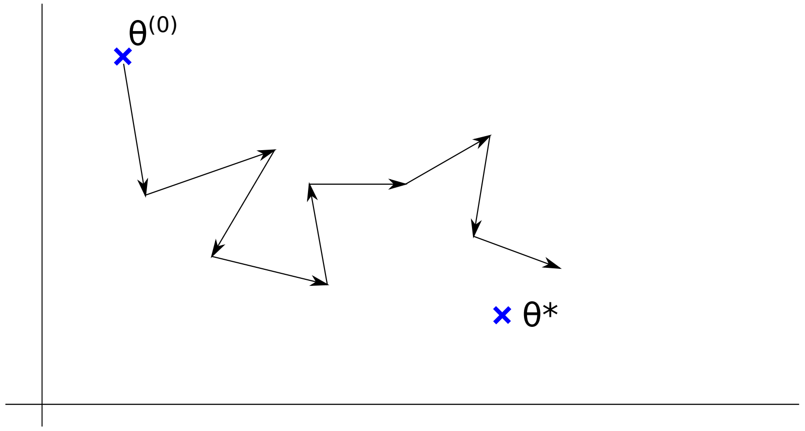
SGD with Constant Learning Rate



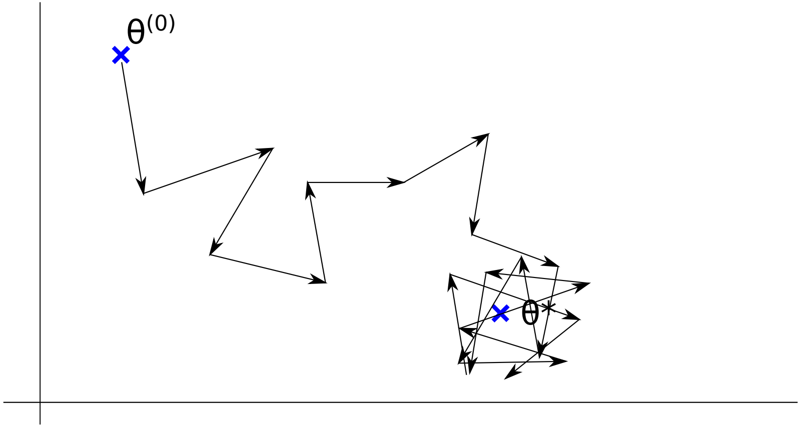
SGD with Constant Learning Rate



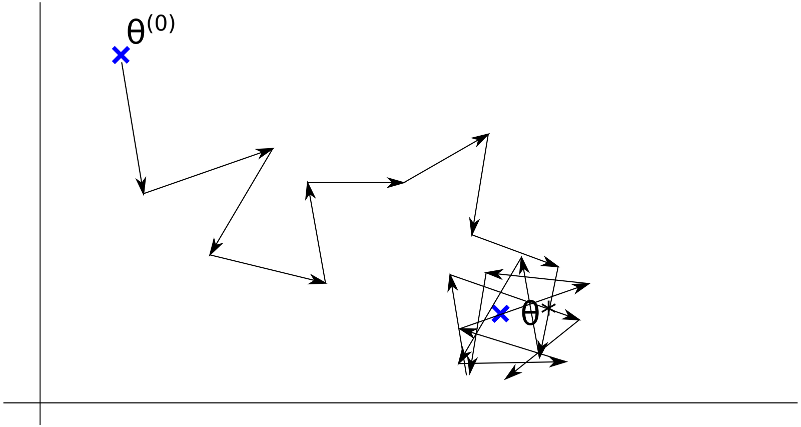
SGD with Constant Learning Rate



SGD with Constant Learning Rate



SGD with Constant Learning Rate



Two regimes:

- (vanilla) SGD reaches general vicinity of a good point rather quickly
- but then it fails to truly convergence because of stochasticity

Optimal Learning Rate?

Formally, it is known how fast stochastic gradient descent converges:

Theorem (Convergence Rate of SGD)

Let $f : \mathbb{R}^d \to \mathbb{R}$ *be bounded from below by* 0, *let* ∇f *be* L*-Lipschitz continuous. Assume there exists a* $\sigma^2 > 0$ *such that* $\mathbb{E}[\|\nabla f_i(\theta)\|^2] \leq \sigma^2$ *for all* θ *(stochastic variance is bounded). Then SGD with small enough step sizes* $\alpha_1, \alpha_2, \dots$ *fulfills*

$$\min_{t=1,\dots,T} \left\{ \mathbb{E}\, \|\nabla f(\theta^{(t)})\|^2 \right\} \leq \frac{f(\theta^{(0)}) - f(\theta^*)}{\sum_{t=1}^T \alpha_t} + \frac{L\sigma^2}{2} \frac{\sum_{t=1}^T (\alpha_t)^2}{\sum_{t=1}^T \alpha_t}$$

(conditions are not fulfilled for ReLU networks, but similar theorem exist)

Consequence: learning rate should decrease over time

- convergence, e.g., if
  - $\sum_{t=1}^{\infty} (\alpha_t)^2 < \infty$ "square summable"
  - $\sum_{t=1}^{T} \alpha_t \overset{T\to\infty}{\to} \infty$ but "not summable"
- prototypical example: $\alpha_t = \frac{\alpha}{t}$ or $\alpha_t = \frac{\alpha}{t+t_0}$

Time-Varying Learning Rates

Theory-suggested learning rate schedule

$$\alpha_t = O(\frac{1}{t+t_0})$$

can be quite slow in practice. E.g., distance from initialization $\sum_{t=1}^{T} \alpha_t = O(\log T)$

Many other proposed schedules:

- reduce learning rate by multiplicative factor, e.g. $\gamma = 0.1$, at predetermined epochs
- reduce learning rate by multiplicative factor, e.g. $\gamma = 0.95$, after every epoch
- reduce learning rate quickly, raise it again, reduce it again, etc.

Time-Varying Learning Rates

Theory-suggested learning rate schedule

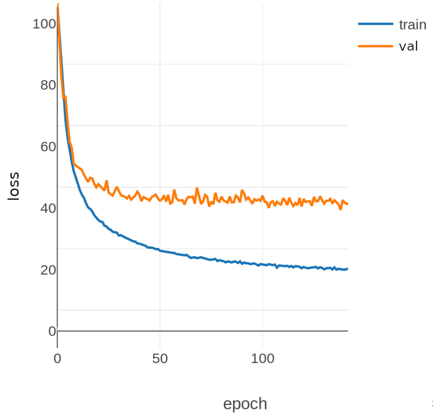
$$\alpha_t = O(\frac{1}{t+t_0})$$

can be quite slow in practice. E.g., distance from initialization $\sum_{t=1}^{T} \alpha_t = O(\log T)$

Many other proposed schedules:

- reduce learning rate by multiplicative factor, e.g. $\gamma = 0.1$, at predetermined epochs
- reduce learning rate by multiplicative factor, e.g. $\gamma = 0.95$, after every epoch
- reduce learning rate quickly, raise it again, reduce it again, etc.

A good schedule can have tremendous effects!



Time-Varying Learning Rates

Theory-suggested learning rate schedule

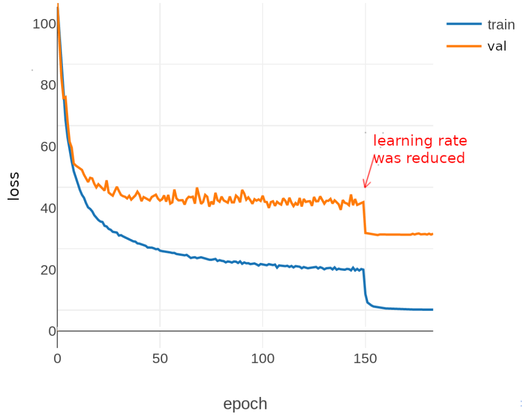
$$\alpha_t = O(\frac{1}{t+t_0})$$

can be quite slow in practice. E.g., distance from initialization $\sum_{t=1}^{T} \alpha_t = O(\log T)$

Many other proposed schedules:

- reduce learning rate by multiplicative factor, e.g. $\gamma = 0.1$, at predetermined epochs
- reduce learning rate by multiplicative factor, e.g. $\gamma = 0.95$, after every epoch
- reduce learning rate quickly, raise it again, reduce it again, etc.

A good schedule can have tremendous effects!



Deep Learning: Backpropagation

Computing Gradient

Reminder: we want to solve

$$\min_{\theta} F(\theta) \quad \text{with} \quad F(\theta) = \sum_{i=1}^{n} f_i(\theta) \quad \text{for} \quad f_i(\theta) = \mathcal{L}(y_i, h(x_i; \theta))$$

by gradient-based optimization. To compute the gradient, we'll need to use the chain rule.

To keep the notation simple, we pretend that all quantities are scalar and we write any f_i simply as $f : \mathbb{R} \to \mathbb{R}$.

Chain Rule

Let $g_2 \circ g_1 : \mathbb{R} \to \mathbb{R}$ with $g_1 : \mathbb{R} \to \mathbb{R}$ and $g_2 : \mathbb{R} \to \mathbb{R}$. We write the argument to g_2 as $w \in \mathbb{R}$ and the argument to g_1 as $v \in \mathbb{R}$. Then

$$\frac{d}{dv} g_2(g_1(v)) = \left(\frac{d}{dv} g_1 \Big|_v \right) \left(\frac{d}{dw} g_2 \Big|_{w = g_1(v)} \right) = \frac{dg_2}{dg_1} \frac{dg_1}{dv}$$

For a neural network, the chain is very long, because the NN is a concatenation of the many per-layer functions:

$$h(\theta,) = h^{(L)}(\, h^{(L-1)}(\cdots h^2(\, h^1(x)) \cdots)$$

In fact, it's even longer, because each layer $h^{(l)}(\cdot) = \sigma_l(\, g^{(l)}(\cdot)\,)$ is a concatenation itself, of a linear $g^{(l)}(a_{l-1}) = b_l + W_3 a_{l-1}$, followed by the non-linear activation function σ_l.

Let a_l denote the output of the l-th layer. Then,

$$a_l = \sigma_l(b_l + W_l a_{l-1}) \qquad \text{for } l = 1, \ldots, L\text{, with } a_0 := x$$

Denote by y_l the value computed by the layer before the activation function. Then,

$$y_l = b_l + W_l a_{l-1} \qquad a_l = \sigma_l(y_l) \qquad \text{for } l = 1, \ldots, L.$$

$$\boxed{y_l = b_l + W_l a_{l-1} \qquad a_l = \sigma_l(y_l) \qquad \text{for } l = 1, \dots, L.}$$

First note that the derivatives of functions with respect to their immediate arguments are easy to compute:

$$\begin{aligned}
\frac{dy_l}{db_l} &= \frac{d}{db_l}\Big(b_l + W_l a_{l-1}\Big) = 1 \\
\frac{dy_l}{dW_l} &= \frac{d}{dW_l}\Big(b_l + W_l a_{l-1}\Big) = a_{l-1} \\
\frac{dy_l}{da_{l-1}} &= \frac{d}{da_{l-1}}\Big(b_l + W_l a_{l-1}\Big) = W_l \\
\frac{da_l}{dy_l} &= \frac{d}{dy_l}\sigma_l(y_l) = \sigma_l' \\
\frac{d\mathcal{L}}{da_L} &= \mathcal{L}' \quad \text{(something explicit)}
\end{aligned}$$

If a function depends indirectly on some parameters, we need to apply the chain rule (potentially many times).

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \frac{d\,\mathcal{L}}{d\,a_L} \frac{da_L}{dy_L} \qquad \frac{d\,y_L}{d\,W_L}$$

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

$$\frac{d}{dW_{L-1}} f(\theta, x) = \frac{d\,\mathcal{L}}{d\,a_L} \frac{da_L}{dy_L} \frac{dy_L}{dW_{L-1}}$$

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

$$\frac{d}{dW_{L-1}} f(\theta, x) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{dW_{L-1}}}^{?}$$

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

$$\frac{d}{dW_{L-1}} f(\theta, x) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{dW_{L-1}}}^{?}$$

$$= \frac{d\,\mathcal{L}}{d\,a_L} \frac{da_L}{dy_L} \frac{dy_L}{da_{L-1}} \frac{da_{L-1}}{d\,W_{L-1}}$$

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

$$\begin{aligned}\frac{d}{dW_{L-1}} f(\theta, x) &= \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{dW_{L-1}}}^{?} \\ &= \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{da_{L-1}}}^{=W_L} \overbrace{\frac{da_{L-1}}{d\,W_{L-1}}}^{?}\end{aligned}$$

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

$$\frac{d}{dW_{L-1}} f(\theta, x) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{dW_{L-1}}}^{?}$$

$$= \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{da_{L-1}}}^{=W_L} \overbrace{\frac{da_{L-1}}{d\,W_{L-1}}}^{?}$$

$$= \frac{d\,\mathcal{L}}{d\,a_L} \frac{da_L}{dy_L} \frac{dy_L}{da_{L-1}} \frac{da_{L-1}}{dy_{L-1}} \frac{dy_{L-1}}{dW_{L-1}}$$

Example:

$$\frac{d}{dW_L} f(\theta, x) = \frac{d}{dW_L} \mathcal{L}(a_L(\theta, x)) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{d\,y_L}{d\,W_L}}^{=a_{L-1}(\theta,x)}$$

$\rightarrow$ good, now all components are known.

$$\begin{aligned}
\frac{d}{dW_{L-1}} f(\theta, x) &= \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{dW_{L-1}}}^{?} \\
&= \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{da_{L-1}}}^{=W_L} \overbrace{\frac{da_{L-1}}{d\,W_{L-1}}}^{?} \\
&= \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{dy_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{da_{L-1}}}^{=W_L} \overbrace{\frac{da_{L-1}}{dy_{L-1}}}^{=\sigma'_{L-1}} \overbrace{\frac{dy_{L-1}}{dW_{L-1}}}^{=a_{L-1}}
\end{aligned}$$

In general:

$$\frac{d}{dW_l} f(\theta, x) = \frac{d\,\mathcal{L}}{d\,a_L} \frac{da_L}{d\,y_L} \frac{dy_L}{d\,a_{L-1}} \frac{da_{L-1}}{d\,y_{L-1}} \frac{dy_{L-1}}{d\,a_{L-1}} \frac{da_{L-2}}{d\,y_{L-2}} \cdots \frac{dy_{l+1}}{d\,a_l} \frac{da_l}{d\,y_l} \frac{dy_l}{d\,W_l} \tag{1}$$

Reminder what a_l is:

$$a_l = \sigma_l(b_l + W_l \sigma(b_{l-1} + W_{l-1}\sigma(\cdots b_1 + W_1 x)\cdots) \tag{2}$$

To obtain the gradient of any individual layer l, we need:

(1) compute a matrix product of $2(L-l)$ many factors

(2) the value of a_l, which is a concatenation of $2l$ many operations (matrix multiplications and non-linearities)

In total: $O(L)$ operations

In general:

$$\frac{d}{dW_l} f(\theta, x) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{d\,y_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{d\,a_{L-1}}}^{=W_L} \overbrace{\frac{da_{L-1}}{d\,y_{L-1}}}^{=\sigma'_{L-1}} \overbrace{\frac{dy_{L-1}}{d\,a_{L-1}}}^{=W_{L-1}} \overbrace{\frac{da_{L-2}}{d\,y_{L-2}}}^{=\sigma'_{L-2}} \cdots \overbrace{\frac{dy_{l+1}}{d\,a_l}}^{=W_{l+1}} \overbrace{\frac{da_l}{d\,y_l}}^{=\sigma'_l} \overbrace{\frac{dy_l}{d\,W_l}}^{=a_l} \tag{1}$$

Reminder what a_l is:

$$a_l = \sigma_l(b_l + W_l \sigma(b_{l-1} + W_{l-1} \sigma(\cdots b_1 + W_1 x) \cdots) \tag{2}$$

To obtain the gradient of any individual layer l, we need:

(1) compute a matrix product of $2(L-l)$ many factors

(2) the value of a_l, which is a concatenation of $2l$ many operations (matrix multiplications and non-linearities)

In total: $O(L)$ operations

In general:

$$\frac{d}{dW_l} f(\theta, x) = \overbrace{\frac{d\,\mathcal{L}}{d\,a_L}}^{=\mathcal{L}'} \overbrace{\frac{da_L}{d\,y_L}}^{=\sigma'_L} \overbrace{\frac{dy_L}{d\,a_{L-1}}}^{=W_L} \overbrace{\frac{da_{L-1}}{d\,y_{L-1}}}^{=\sigma'_{L-1}} \overbrace{\frac{dy_{L-1}}{d\,a_{L-1}}}^{=W_{L-1}} \overbrace{\frac{da_{L-2}}{d\,y_{L-2}}}^{=\sigma'_{L-2}} \cdots \overbrace{\frac{dy_{l+1}}{d\,a_l}}^{=W_{l+1}} \overbrace{\frac{da_l}{d\,y_l}}^{=\sigma'_l} \overbrace{\frac{dy_l}{d\,W_l}}^{=a_l} \tag{1}$$

Reminder what a_l is:

$$a_l = \sigma_l(b_l + W_l \sigma(b_{l-1} + W_{l-1}\sigma(\cdots b_1 + W_1 x) \cdots) \tag{2}$$

To obtain the gradient of any individual layer l, we need:

(1) compute a matrix product of $2(L-l)$ many factors

(2) the value of a_l, which is a concatenation of $2l$ many operations (matrix multiplications and non-linearities)

In total: $O(L)$ operations

Does computing the gradients of all layer, $l = 1, \ldots, L$ take $O(L^2)$?

Luckily, no! We can re-use terms to increase efficiency!

We already know: computing all activations $a_1, \ldots, a_L$ is just $O(L)$, not $O(L^2)$

- $a_l = \sigma_l(b_l + W_l a_{l-1})$ fixed number of operations, independent of L
- computation operated layer-by-layer, $l = 1, \ldots, L \rightarrow$ "forward pass"

We already know: computing all activations $a_1, \dots, a_L$ is just $O(L)$, not $O(L^2)$

- $a_l = \sigma_l(b_l + W_l a_{l-1})$ fixed number of operations, independent of L
- computation operated layer-by-layer, $l = 1, \dots, L \to$ "forward pass"

Similiar trick works for gradients:

$$\frac{d}{dW_L} f(\theta, x) = \overbrace{\frac{d\mathcal{L}}{da_L}\frac{da_L}{dy_L}}^{=:\delta_L} a_{L-1} = \delta_L a_{L-1}$$

$$\frac{d}{dW_{L-1}} f(\theta, x) = \frac{d\mathcal{L}}{da_L}\frac{da_L}{dy_L}\frac{dy_L}{da_{L-1}}\frac{da_{L-1}}{dy_{L-1}} a_{L-2} = \overbrace{\delta_L \frac{dy_L}{da_{L-1}}\frac{da_{L-1}}{dy_{L-1}}}^{=:\delta_{L-1}} a_{L-2} = \delta_{L-1} a_{L-2}$$

$$\frac{d}{dW_l} f(\theta, x) = \delta_{l+1}\frac{dy_{l+1}}{da_l}\frac{da_l}{dy_l} a_{l-1} = \delta_l a_{l-1}$$

With "backwards" layer-by-layer computation of $\delta_L, \delta_{L-1}, \dots, \delta_1$ by $\delta_l = \delta_{l+1}\frac{dy_{l+1}}{da_l}\frac{da_l}{dy_l}$

Backpropagation

This trick of arranging computation is called **backpropagation** (short: **backprop**)

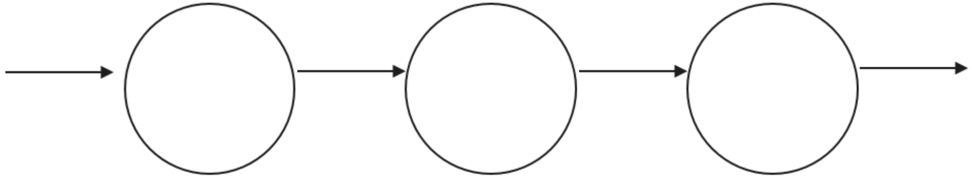
Forward pass: compute all model activations

```
input $x$
    $a_0 \leftarrow x$
    for $l = 1, \dots, L$ do
        $a_l \leftarrow \sigma_l(b_l + W_l a_{l-1})$
    end for
output $a_1, \dots, a_L$
```

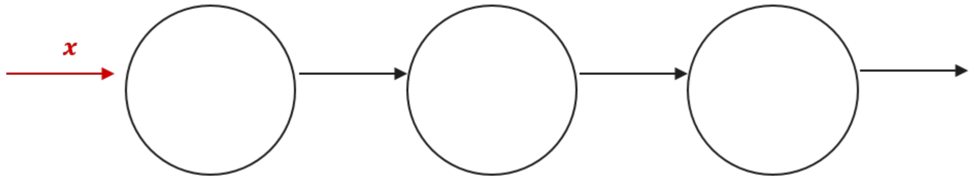
Backwards pass: compute gradients w.r.t. to all model parameters

```
input $x$, $a_1, \dots, a_L$
    $\delta_L \leftarrow \nabla \mathcal{L}(a_L)\sigma'_L$
    for $l = L-1, L-2, \dots, 1$ do
        $\delta_l \leftarrow \delta_{l+1} W_{l+1} \sigma'_l$
    end for
output $\delta_1 a_0, \dots, \delta_L a_{L-1}$
```

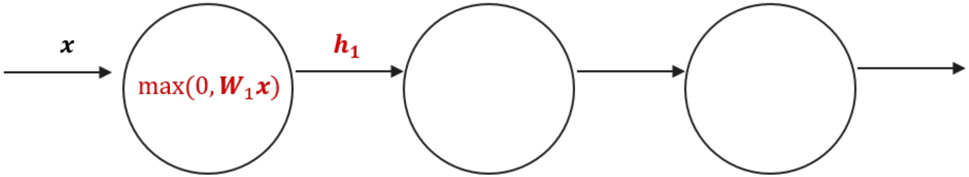
Backpropagation: animation



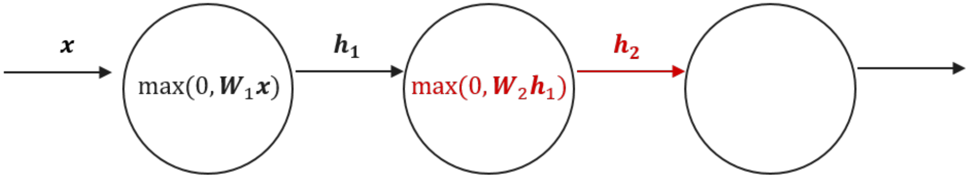
Backpropagation: animation



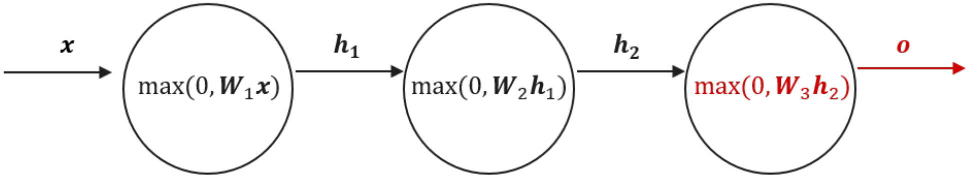
Backpropagation: animation



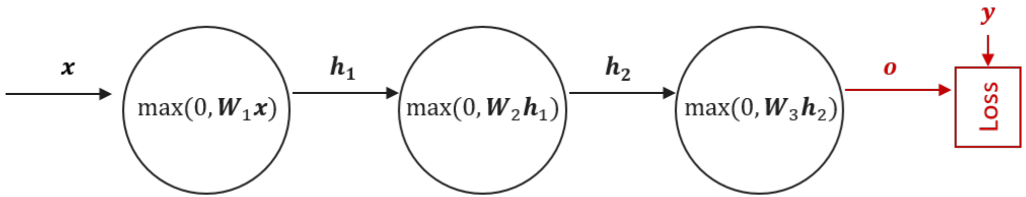
Backpropagation: animation


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$

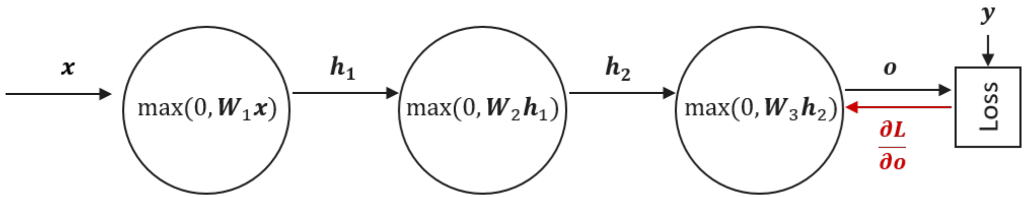
Backpropagation: animation


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1 \boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2 \boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\max(0, \boldsymbol{W}_3 \boldsymbol{h}_2)$
$\boldsymbol{o}$

Backpropagation: animation


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1 \boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2 \boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\max(0, \boldsymbol{W}_3 \boldsymbol{h}_2)$
$\boldsymbol{o}$
$\boldsymbol{y}$
Loss

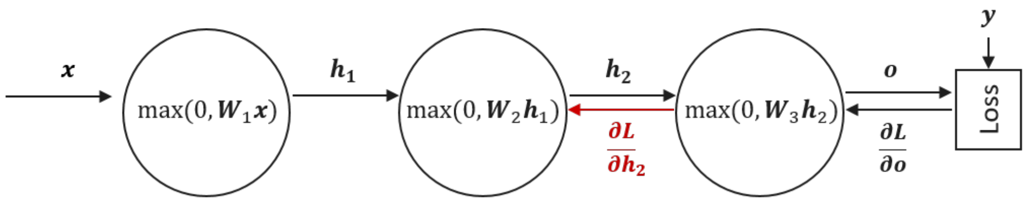
Backpropagation: animation



$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\max(0, \boldsymbol{W}_3\boldsymbol{h}_2)$
$\boldsymbol{o}$
$\boldsymbol{y}$
Loss
$\frac{\partial L}{\partial o}$

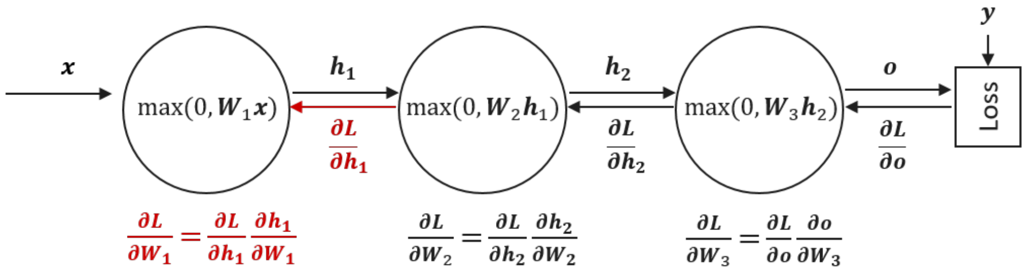
$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial W_3}$$

Backpropagation: animation



$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial W_2} \qquad \frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial W_3}$$

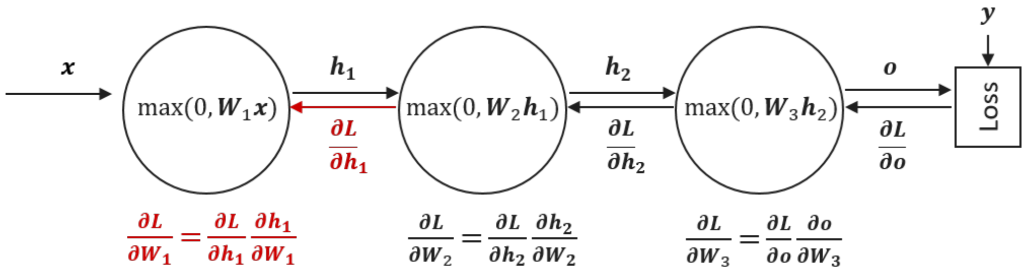
Backpropagation: animation



$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\frac{\partial L}{\partial \boldsymbol{h}_1}$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\frac{\partial L}{\partial \boldsymbol{h}_2}$
$\max(0, \boldsymbol{W}_3\boldsymbol{h}_2)$
$\boldsymbol{o}$
$\frac{\partial L}{\partial \boldsymbol{o}}$
$\boldsymbol{y}$
Loss

$$\frac{\partial L}{\partial \boldsymbol{W}_1} = \frac{\partial L}{\partial \boldsymbol{h}_1}\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}_1} \qquad \frac{\partial L}{\partial \boldsymbol{W}_2} = \frac{\partial L}{\partial \boldsymbol{h}_2}\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{W}_2} \qquad \frac{\partial L}{\partial \boldsymbol{W}_3} = \frac{\partial L}{\partial \boldsymbol{o}}\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W}_3}$$

Backpropagation: animation



x
$\max(0, W_1 x)$
h_1
$\frac{\partial L}{\partial h_1}$
$\max(0, W_2 h_1)$
h_2
$\frac{\partial L}{\partial h_2}$
$\max(0, W_3 h_2)$
o
$\frac{\partial L}{\partial o}$
y
Loss
$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial W_1}$
$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial W_2}$
$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o} \frac{\partial o}{\partial W_3}$

Disadvantage of BP: high memory usage, because all intermediate activations must be stored
$\rightarrow$ if that's a problem for you, look into "gradient checkpointing"

Automatic Differentiation

How to implement backpropagation and other gradient computations?

Automatic Differentiation

How to implement backpropagation and other gradient computations? **You don't!**

Automatic Differentiation

How to implement backpropagation and other gradient computations? **You don't!**

Many packages for numerical computation and all deep learning frameworks support

Automatic Differentiation (autodiff, AD)

Automatic differentiation is a set of techniques for automatically computing the numeric value of the derivatives of any function implemented on a computer.

- it's not symbolic differentiation (you don't get the functional expression)
- it's not approximate numeric differentiation e.g. by finite differences
- it's essentially a smart way of automatically
  - decomposing the implemented function into elementary operations
  - looking up the derivatives of each elementary component
  - combining the numeric values of each component via the chain rule

Note: AD is convenient, but not always the most efficient. For DL, it's state-of-the-art, though.

more information: [Baydin, Pearlmutter, Radul, Siskind; "Automatic Differentiation in Machine Learning: a Survey", JMLR 2018]

Deep Learning: Parameter Initialization

Initialization

It's lecture 10, and we haven't mentioned parameter initialization before. Why not?

For convex problems, e.g. logistic regression training, initialization matters little.

- GD/SGD convert to same solution from every starting point
- initializing all parameters at 0 typically works well

For neural network training, initialization can matter a lot.

- GD/SGD find different solutions based on the starting point.
- initializing all parameters at 0 does not work

Problem 1) symmetry

- All neurons of a layer receive the same inputs. What if they also have the same weights?
  $\rightarrow$ they compute identical functions
  $\rightarrow$ they get identical gradient updates
  $\rightarrow$ the weights will always stay the same $\rightarrow$ wasted capacity and computation

Problem 2) zero gradients

- for the gradient we have to multiply many weights matrices together
  if any of them is 0, the gradient will be 0, so no update

Initialization

Neural network parameters are typically initialized with small random values around 0.

How small?

- reminder: $a_l = \sigma_l(W_l a_{l-1})$
- if weights are too small, activations get smaller and smaller through the network
  $\rightarrow$ "vanishing signal" problem

Won't gradient descent fix that, by increasing the weights during training?

- reminder: l-th layer gradient is $\delta_l a_{l-1}$ for $\delta_l \leftarrow \delta_{l+1} W_{l+1} \sigma'_l$
- if weights are too small, gradients for earlier layers get smaller and smaller
  $\rightarrow$ "vanishing gradient" problem
- parameters of early layers take very long to converge

Initialization

What, if we initialize with large weights? Same argument in reverse!

- activations: $a_l = \sigma_l(W_l a_{l-1})$
- if weights are too large, activations get bigger and bigger through the network
- reminder: l-th layer gradient is $\delta_l a_{l-1}$ for $\delta_l \leftarrow \delta_{l+1} W_{l+1} \sigma'_l$
- if weights are too large, gradients for earlier layers get bigger and bigger
  $\rightarrow$ "exploding gradient" problem
- very small learning rate required to avoid optimization diverging $\rightarrow$ no or slow learning

Problem 3) saturation

- some activation functions, e.g. $\tanh$, are saturating $\rightarrow$ activations stay bounded
- but: $\sigma' \approx 0$ for input in region of saturation
  $\rightarrow$ weights and activations don't change much $\rightarrow$ slow or no convergence

Ideally, activations should neither grow nor shrink too much: $|a_{l-1}| \approx |a_l|$

Ideally, gradients should neither grow nor shrink too much: $|\delta_l| \approx |\delta_{l-1}|$

Initialization

Let's derive a proper initialization scheme.

Simplifying assumptions (for the time point of initialization, before training):

- activations will be $i.i.d.$ random RVs from some distribution
- weights will be sampled $i.i.d.$ from some distribution
- weights and activations will be independent

Let's look at a single neuron with n_{in} inputs $x_1, \dots, x_{n_{in}}$ and weights $w_1, \dots, w_{n_{in}}$.

$$a = \sigma\Big(\sum_{j=1}^{n_{in}} w_j x_j\Big)$$

Assume that σ behaves roughly linear around 0 with $\sigma'(0) = 1$, e.g. $\tanh$.

$$\text{Var}[a] \approx \sum_{j=1}^{n_{in}} \text{Var}[w_j x_j] \overset{\text{indep.}}{=} \sum_{j=1}^{n_{in}} \text{Var}[w_j]\text{Var}[x_j] \overset{\text{i.i.d.}}{=} n_{in}\text{Var}[w]\text{Var}[x]$$

Our goal here is to make $\text{Var}[a] \approx \text{Var}[x]$, so we should choose $\text{Var}[w] \approx \frac{1}{n_{in}}$

This will ensure that activations will neither shrink nor explode (much).

Weight Initialization

What about gradients?

$$\delta^l = \delta^{l+1} W_l \sigma'_l \qquad \text{backprop}$$

Analogous analysis for a neuron that is used by n_{out} neurons in the next layer,

$$\mathsf{Var}[\delta^l] \approx \sum_{j=1}^{n_{out}} \mathsf{Var}[\delta^{l+1}_j w_j] = \sum_{j=1}^{n_{out}} \mathsf{Var}[\delta^{l+1}_j]\mathsf{Var}[w_j] = n_{out}\mathsf{Var}[w]\mathsf{Var}[\delta^{l+1}]$$

Our goal is to make $\mathsf{Var}[\delta^l] \approx \mathsf{Var}[\delta^{l+1}]$, so we should choose $\mathsf{Var}[w] \approx \frac{1}{n_{out}}$.

Xavier initialization [Glorot, Bengio. "Understanding the difficulty of training deep feedforward neural networks", 2010]

For a network with $\tanh$ or similar activation function, initialize weights from a either a truncated Gaussian or a uniform distribution with mean 0 and variance $\frac{1}{n_{avg}}$ for $n_{avg} = \frac{1}{2}(n_{in} + n_{out})$.

$\rightarrow$ compromise between balancing activations and balancing gradients

Weight Initialization

For ReLU activation, there's two differences compared to (almost) linear σ:

- activations a_l are always non-negative and cannot have mean 0
  - not much of an issue, as $w_l a_l$ will restore mean zero, if $\mathbb{E}[w_l] = 0$
- σ sets all negative values to 0, so on average, half of the inputs to a neuron will be 0
  - only half of the terms in the summation are non-zero, so the variance will be half as big

Consequently, for ReLU networks the weights should be initialized larger by a factor of 2.

He initialization **[He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", 2015]**

For a network with ReLU activation function, initialize weights from a either a truncated Gaussian or a uniform distribution with mean 0 and variance $\frac{2}{n_{avg}}$ for $n_{avg} = \frac{1}{2}(n_{in} + n_{out})$.

He *et al*. demonstrated that they could train very deep networks (> 20 layers) with this initialization, but not with others.