10707
Deep Learning: Spring 2021

Andrej Risteski

Machine Learning Department

Lecture 5: Intro to optimization

Supervised learning

Empirical risk minimization approach:
minimize a **training** loss l over a class of **predictors** $\mathcal{F}$:

$$\hat{f} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{(x,y)\text{:training samples}} l(f(x), y)$$

Three pillars:

(1) How expressive is the class $\mathcal{F}$? (**Representational power**)

(2) How do we minimize the training loss efficiently? (**Optimization**)

(3) How does $\hat{f}$ perform on unseen samples? (**Generalization**)

The world of continuous optimization

The typical training task in ML can be cast as: $\min_{x \in \mathbb{R}^d} \quad f(x)$

Usually, it is cheap to calculate f(x), $\nabla f(x)$, but (more) expensive to calculate higher-order derivatives.

Most algorithms we will look at are iterative: they progressively pick points $x_1, x_2, \ldots$ that are supposed to bring “improvement”.

Non-exhaustive coverage: entire field of optimization, with applications vastly beyond ML. We focus on deep-learning-relevant methods.

The mother of all optimization algorithms: gradient descent

The simplest optimization algorithm: Taylor expand and find the direction of "steepest" descent. More precisely:

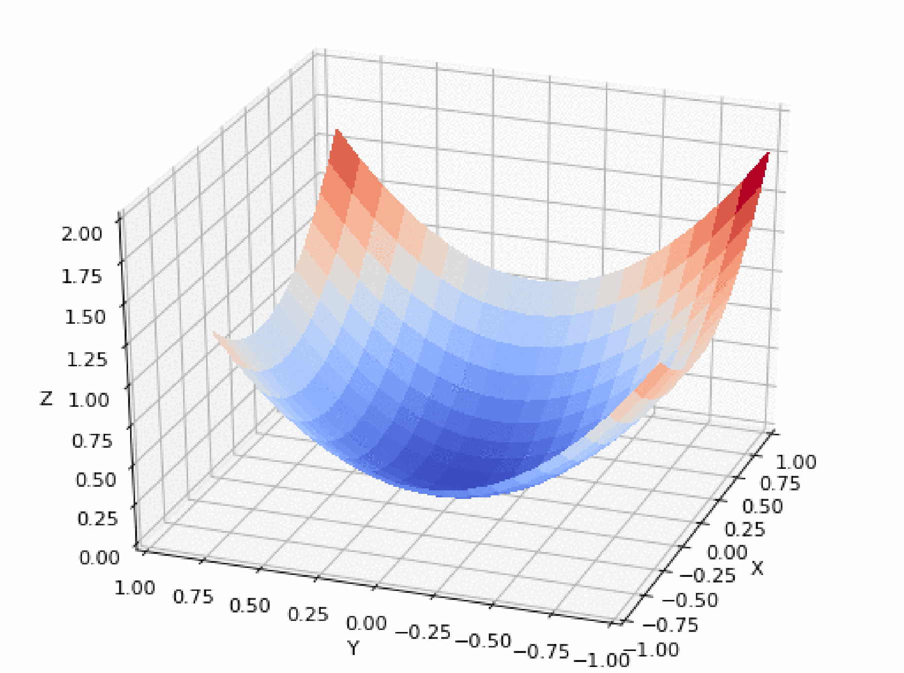
By Taylor's theorem, we have $f(x+\Delta) \approx f(x) + \Delta^T \nabla f(x) + O(||\Delta||^2)$

So, if we ignore higher-order effects, we have

$$\operatorname*{argmin}_{\Delta, ||\Delta|| \leq \epsilon} \{ f(x+\Delta) - f(x) \} = -\epsilon \frac{\nabla f(x)}{||\nabla f(x)||}$$

i.e. we should move (appropriately scaled) opposite of the gradient

Gradient descent, pictorially

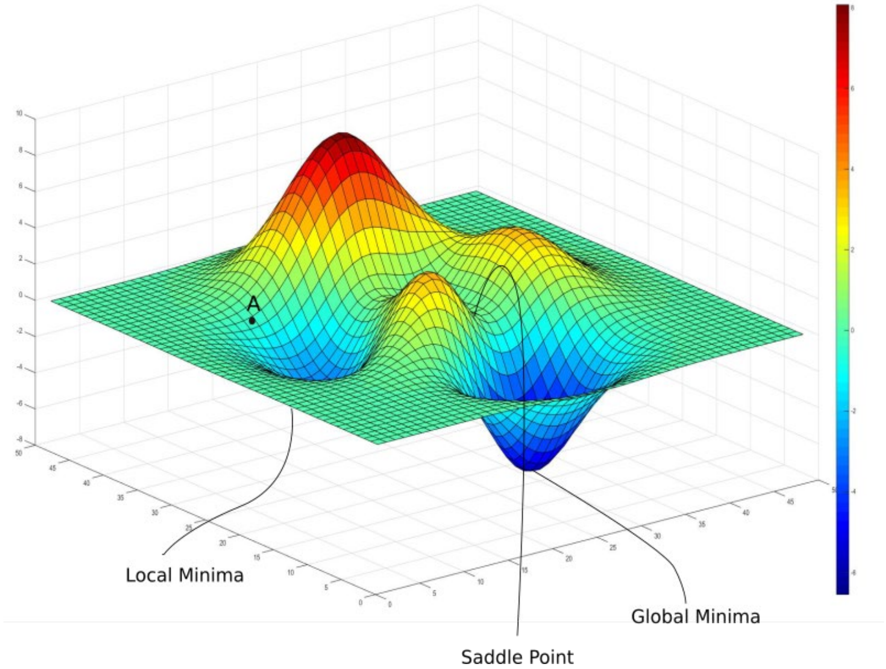


What can we hope for, in the case that $||\Delta|| \to 0$?

We stop moving when $\nabla f(\hat{x}) \approx 0$: these care called **stationary points**.

What kinds of stationary points are there?

Types of stationary points



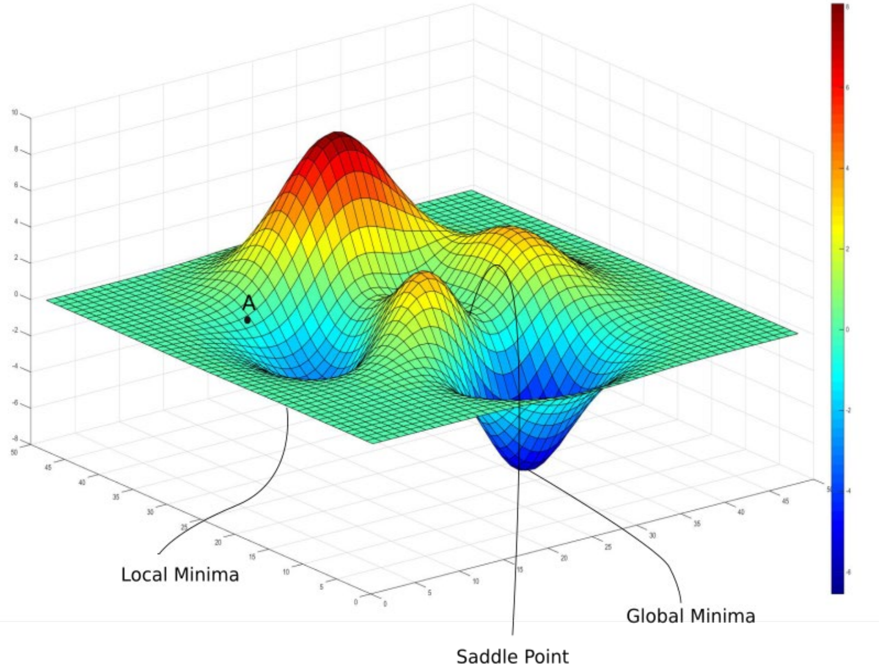
Global minimum: actual minimizer, namely $f(\hat{x}\,) \leq f(x), \forall x \in \mathbb{R}^d$

Local minimum: $f(\hat{x}\,) \leq f(x),\ \forall x \text{ s.t. } \left||x - \hat{x}|\right| \leq \epsilon$ for some $\epsilon > 0$

Local maximum: $f(\hat{x}\,) \geq f(x),\ \forall x \text{ s.t. } \left||x - \hat{x}|\right| \leq \epsilon$ for some $\epsilon > 0$

Saddle points: stationary point that is *not* a local min/max.

Types of stationary points



Global minimum: finding these in general is very hard (both in theory – NP-hard, as well as in practice)

Local minimum: seem to work quite well often. Some theoretical understanding of why in very restricted cases.

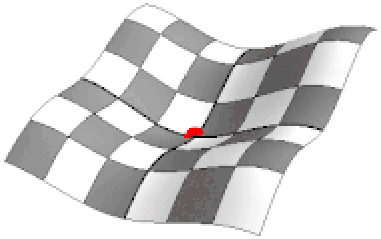
Saddle points: typically bad, arise from invariances in input. Want to avoid these. (Stay tuned.)

Checking for local minima?

Second order checks: Hessian approximates a function to second order

Taylor's thm: $f(x+\Delta) \approx f(x) + \Delta^T \nabla f(x) + \frac{1}{2}\Delta^T \nabla^2 f(x)\Delta + \ O(||\Delta||^3)$

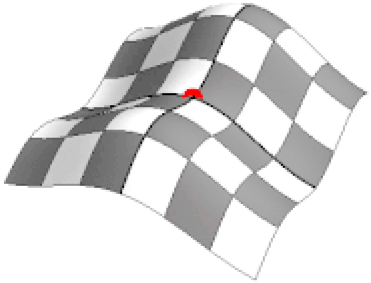
$$\approx f(x) + \frac{1}{2}\Delta^T \nabla^2 f(x)\Delta + \ O(||\Delta||^3)$$



If $\nabla^2 f(x) \ \succ 0$: for any direction Δ, and small enough $||\Delta||$

$\Delta^T \nabla^2 f(x)\Delta + \ O(||\Delta||^3) \geq 0$, so $f(x+\Delta) > f(x)$

Local minimum! (Flipped for local maximum)



If $\nabla^2 f(x)$ has both positive and negative eigenvalues:

Saddle point (not a local minimum/maximum)

If neither of these attains, test is inconclusive!

The descent lemma: analyzing gradient descent

So far, we've only considered the limit $||\Delta|| \to 0$.

If$||\Delta||$ is too large, the Taylor expansion will be invalid (and gradient descent can "jump over" local minima).

If$||\Delta||$ is too small, the runtime of the algorithm will suffer.
The descent lemma characterizes the "sweet spot":

Theorem (descent lemma): Let f be twice differentiable, and $||\nabla^2 f(x)||_2 \leq \beta$. Then, setting $\eta = 1/\beta$, and calling x_t the iterates of gradient descent, namely $x_{t+1} = x_t - \eta \nabla f(x_t)$, we have:

$$f(x_t) - f(x_{t+1}) \geq \frac{1}{2\beta} ||\nabla f(x_t)||_2^2$$

Using the descent lemma: Lyapunov functions

Theorem (descent lemma): Let f be twice differentiable, and $||\nabla^2 f(x)||_2 \leq \beta$. Then, setting $\eta = 1/\beta$, and calling x_t the iterates of gradient descent, namely $x_{t+1} = x_t - \eta \nabla f(x_t)$, we have:

$$f(x_t) - f(x_{t+1}) \geq \frac{1}{2\beta} ||\nabla f(x_t)|\Big|_2^2$$

Suppose f is lower bounded (e.g. $f \geq 0$), and $f(x_0) \leq M$

Suppose we want point x_t, s.t. $\Big||\nabla f(x_t)|\Big| \leq \epsilon$.

Lyapunov (potential) fn argument: suppose $\forall t \in [0, T], \Big||\nabla f(x_t)|\Big| \geq \epsilon$

Then, $f(x_T) \leq f(x_0) - T\frac{1}{2\beta}\epsilon^2 \leq M - T\frac{1}{2\beta}\epsilon^2$. Also, $f(x_T) \geq 0$.

Putting these together, we get $T \leq 2\, M\beta/\epsilon^2$

Proving the Descent Lemma

Theorem (descent lemma): Let f be twice differentiable, and $||\nabla^2 f(x)||_2 \leq \beta$. Then, setting $\eta = 1/\beta$, and calling x_t the iterates of gradient descent, we have:

$$f(x_t) - f(x_{t+1}) \geq \frac{1}{2\beta} ||\nabla f(x_t)|\Big|_2^2$$

Proof: By Taylor expansion and the mean value theorem, we have

$$f(x + \Delta) = f(x) + \Delta^T \nabla f(x) + \frac{1}{2} \Delta^T \nabla^2 f(y) \Delta$$

Moreover, $\Delta^T \nabla^2 f(y) \Delta \leq ||\nabla^2 f(y)||_2 ||\Delta||_2^2 \leq \beta ||\Delta||_2^2$. Plugging in $\Delta = -\eta \nabla f(x_t)$:

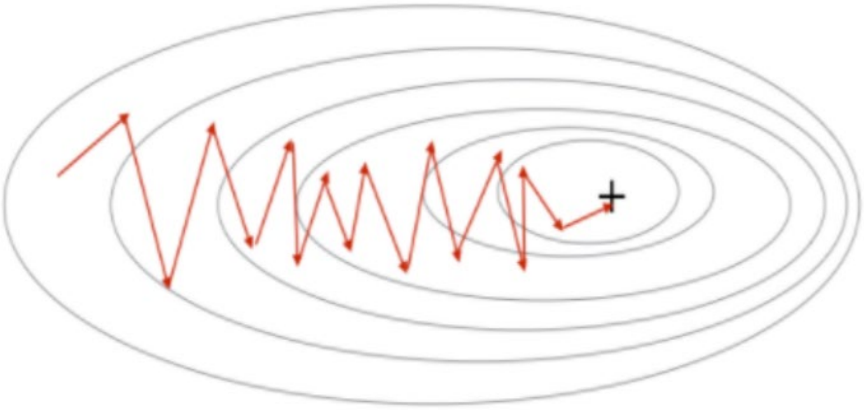
$$f(x_{t+1}) \leq f(x_t) - \eta \Big||\nabla f(x_t)|\Big|^2 + \frac{1}{2} \beta \eta^2 \Big||\nabla f(x_t)|\Big|^2$$

$$= f(x_t) - 1/\beta \Big||\nabla f(x_t)|\Big|^2 + \frac{1}{2} 1/\beta \Big||\nabla f(x_t)|\Big|^2$$

$$= f(x_t) - 1/2\beta \Big||\nabla f(x_t)|\Big|^2$$

Understanding gradient descent locally

Let's consider f's that are quadratic. (Close to local minima, this will be "true" due to Taylor). What quadratics are bad/good for gradient descent?



Bad behavior: gradients don't point towards minimizer – a lot of zig-zaging until we reach minimizer.

Intuitively: ellipsoidal contours (level sets) should be worse than spherical level sets.

Question: Let $f(x) = \frac{1}{2} x^T A\, x$, can we characterize convergence time of gradient descent more precisely? What does it depend on?

Understanding gradient descent locally

Question: Let $f(x) = \frac{1}{2} x^T A\, x$, can we characterize convergence time of gradient descent more precisely? What does it depend on?

Thm: Let A be a symmetric positive-definite matrix with minimum and maximum eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$ and denote $\kappa = \lambda_{\max} / \lambda_{\min}$ (**condition number**).

The iterates of gradient descent with $\eta = \frac{2}{\lambda_{\max} + \lambda_{\min}}$ satisfy:

$$||x_t|| \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^t ||x_0||$$

$= ||x_t - 0||$, *i.e. distance from optimum*

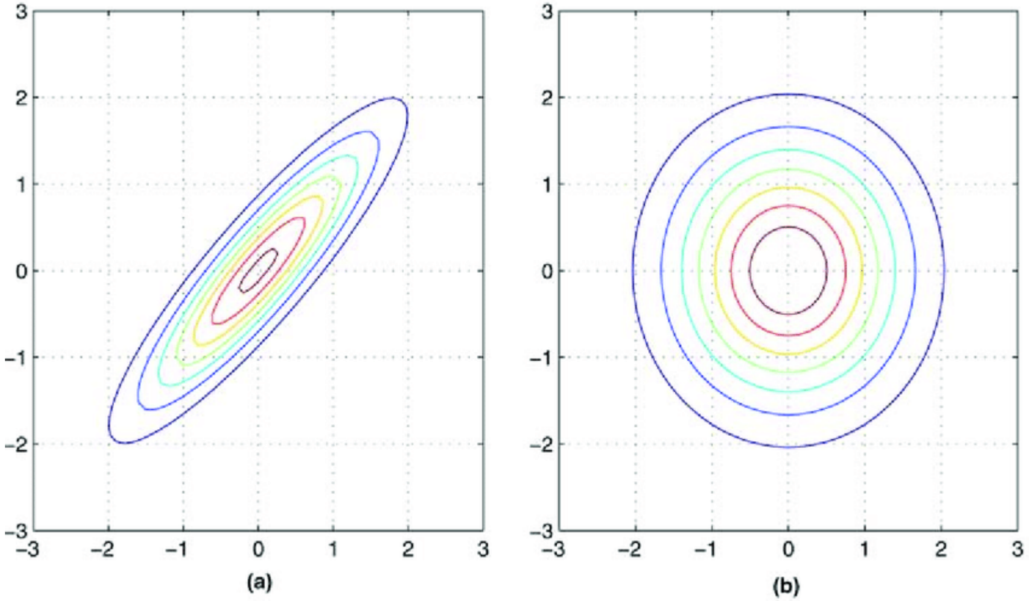
$= ||x_0 - 0||$, *i.e. distance from optimum*

Understanding gradient descent locally

Question: Let $f(x) = \frac{1}{2} x^T A\, x$, can we characterize convergence time of gradient descent more precisely? What does it depend on?

Thm: Let A be a symmetric positive-definite matrix with minimum and maximum eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$ and denote $\kappa = \lambda_{\max} / \lambda_{\min}$ (condition number).

The iterates of gradient descent with $\eta = \frac{2}{\lambda_{\max} + \lambda_{\min}}$ satisfy:



$$||x_t|| \le \left(\frac{\kappa - 1}{\kappa + 1}\right)^t ||x_0||$$

$$= \left(1 - \frac{2}{\kappa + 1}\right)^t ||x_0||$$

κ large => slower convergence

Understanding gradient descent locally

Question: Let $f(x) = \frac{1}{2} x^T A\, x$, can we characterize convergence time of gradient descent more precisely? What does it depend on?

Thm: Let A be a symmetric positive-definite matrix with minimum and maximum eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$ and denote $\kappa = \lambda_{\max} / \lambda_{\min}$ (condition number).

The iterates of gradient descent with $\eta = \frac{2}{\lambda_{\max} + \lambda_{\min}}$ satisfy:

$$||x_t|| \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^t ||x_0||$$

Proof: $||x_{t+1}|| = ||x_t - \eta \nabla f(x_t)||$

$$= ||x_t - \eta A x_t|| = ||(I - \eta A) x_t|| \leq ||I - \eta A||_2 ||x_t||_2$$

$$\leq \max(|1 - \eta \lambda_{max}|, |1 - \eta \lambda_{\min}|) \, ||x_t||_2$$

$$= \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} ||x_t||_2 \quad = \frac{\kappa - 1}{\kappa + 1} ||x_t||_2$$

Fixes to the conditioning problem

What can we do for poorly conditioned problems?

Quadratic problem suggests solution: we can solve it in closed form!!

If $f(x) = \frac{1}{2} x^T A\, x + b^T x + c$, minimizer is $A^{-1} b$. (Just take derivatives, set to 0.)

What do we do for arbitrary f? Approximate function to second order!!

By Taylor's thm: $f(x + \Delta) \approx f(x) + \Delta^T \nabla f(x) + \frac{1}{2} \Delta^T \nabla^2 f(x) \Delta + \; O(||\Delta||^3)$

Ignoring 3rd and higher order terms, and using the above observation for quadratics:

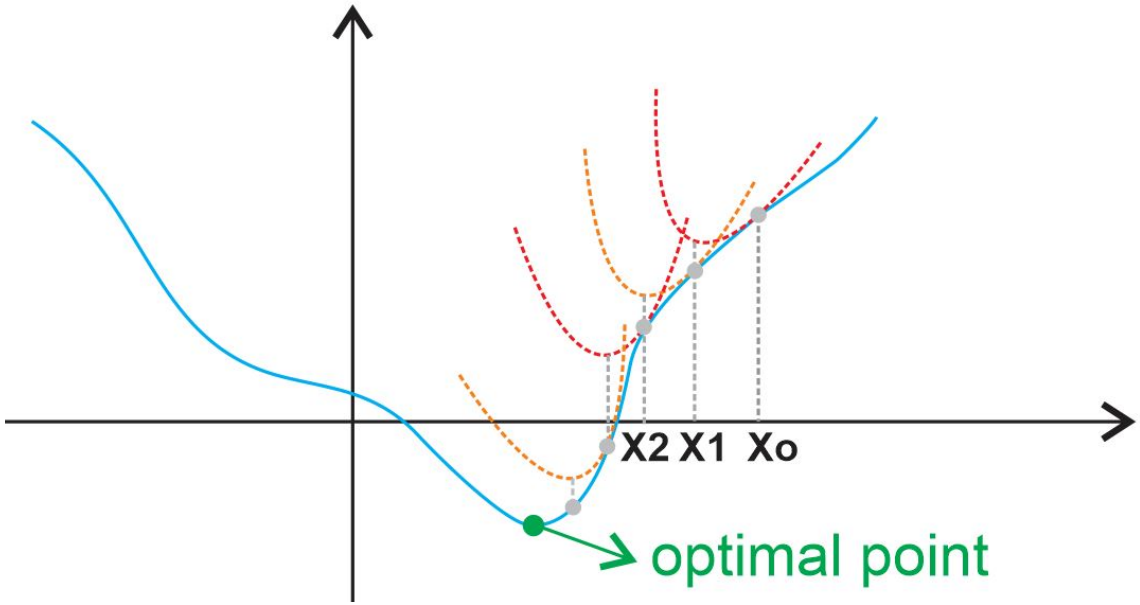
Set $x_{t+1} = x_t - \eta \left(\nabla^2 f(x_t) \right)^{-1} \nabla f(x_t)$

Newton's method.

Fixes to the conditioning problem

Set $x_{t+1} = x_t - \eta\left(\nabla^2 f(x_t)\right)^{-1} \nabla f(x_t)$

Newton's method.



Problem: need to invert a $d \times d$ matrix => d^3runtime. Way too expensive.

Momentum (Polyak '64)

Alternative fix: instead of using the gradient at the current step, use a linear combination of the gradients at prior steps. "Smooths" out zig-zagging, by not relying too much on current gradient.

Linear combination of prior gradients + current one

$$v_{t+1} = -\nabla f(x_t) + \beta v_t$$
$$x_{t+1} = x_t + \eta v_{t+1}$$

Helps provably! For quadratic case, i.e. $f(x) = x^T A\, x$, you can show:

$$||x_t|| \le \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^t ||x_0||$$

Momentum (Nesterov ‘83)

Nesterov acceleration is a *lookahead* variant of momentum, which has provable benefits for *any* convex function. (And is in a certain precise sense, the optimal first-order optimization algorithm).

Evaluate gradient at a “lookahead” point

$$v_{t+1} = -\nabla f(x_t + \beta\, v_t) + \beta\, v_t$$
$$x_{t+1} = x_t + \eta\, v_{t+1}$$

Magical! There’s been a mini cottage industry to “explain” Nesterov acceleration.

Taking gradients of neural networks: backpropagation

The workhorse for training neural networks: an algorithm that for a network with V nodes and E edges calculates the gradient in **linear time** O(V+E).

The name **backpropagation** was introduced by *Rumelhart, Hinton, Williams ‘86*, but so natural that it was rediscovered multiple times (as early as 60s). Algorithm seems to first be mentioned in *Werbos*’ thesis ‘74 in the context of neural networks.

In **control theory**: Kelley ‘60, Bryson ’61 [cast as dynamic programming];

In **theoretical computer science**: Baur-Strassen lemma ’83 [in the context of algebraic circuits]

Taking gradients of neural networks: backpropagation

The main tool for deriving backprop: **chain rule**

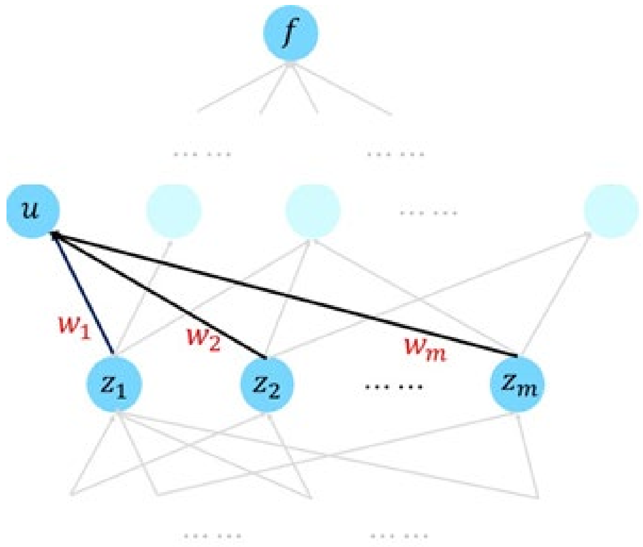
Suppose $f(y) \ = \ f(x_1(y), x_2(y), \dots, x_n(y))$

Then, $\frac{\partial f}{\partial y} \ = \sum_{i=1}^{n} \ \frac{\partial f}{\partial x_i}\frac{\partial x}{\partial y}$

Observation 1: It suffices to take derivatives with respect to node functions.

$$\frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial w_1} = \frac{\partial f}{\partial u}\frac{\partial \sigma(\langle w, z\rangle + b)}{\partial w_1}$$

$$= \frac{\partial f}{\partial u}\sigma'(u)z_1$$



Taking gradients of neural networks: backpropagation

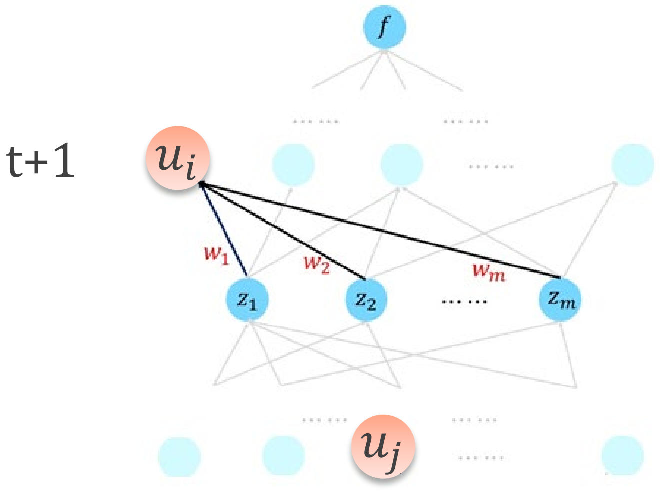
Observation 2: The obvious forward propagation algorithms results in runtime of $\Omega(V^2)$. (Bad! We want O(V+E))

Obvious algorithm? Calculate inductively $\frac{\partial u_i}{\partial u_j}$, for all pairs (u_i, u_j) where u_j is lower than u_i (obviously, this includes $\frac{\partial f}{\partial u}$ which we want)

$$\frac{\partial u_i}{\partial u_j} = \sum_k \frac{\partial u_i}{\partial z_k} \frac{\partial z_k}{\partial u_j}$$

Easy as in prior slide

Have these by inductive hypothesis.

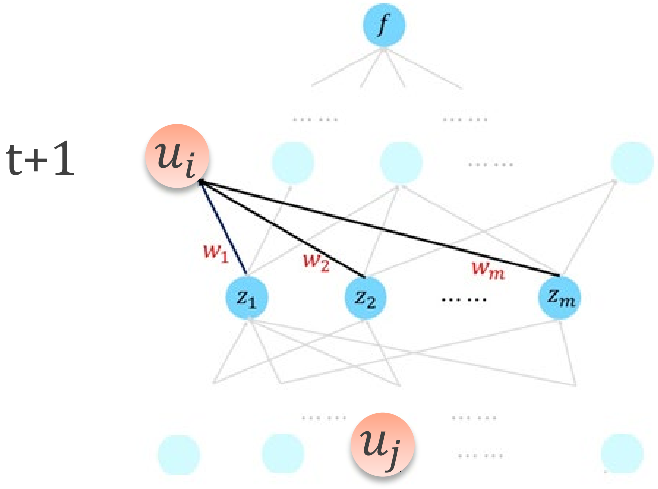


Taking gradients of neural networks: backpropagation

Observation 2: The obvious forward propagation algorithms results in runtime of $\Omega(V^2)$. (Bad! We want O(V+E))

Obvious algorithm? Calculate inductively $\frac{\partial u_i}{\partial u_j}$, for all pairs (u_i, u_j) where u_j is lower than u_i (obviously, this includes $\frac{\partial f}{\partial u}$ which we want)

$$\frac{\partial u_i}{\partial u_j} = \sum_k \frac{\partial u_i}{\partial z_k} \frac{\partial z_k}{\partial u_j}$$



Bad *– this will end up with $\Omega(V^2)$ algorithm.*

Taking gradients of neural networks: backpropagation

Observation 3: The better way to do this is in a backward fashion.

Message passing algorithm [dynamic programming]: each node u receives messages (real numbers) from its neighbors on top. Let their sum be S. The node passes to downward neighbors z: $S\frac{\partial u}{\partial z}$. Proceed from top to down.

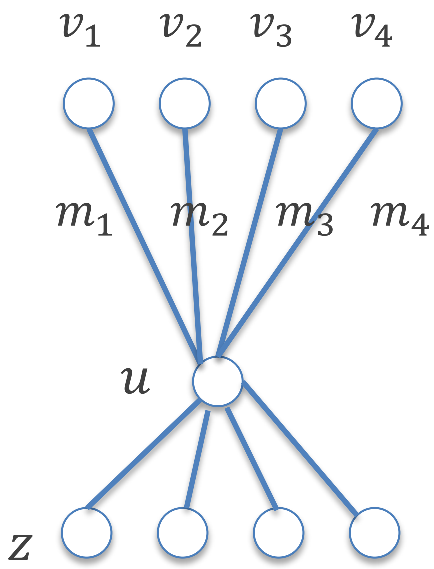
Claim: The sum S of the messages that each node u computes is equal to $\frac{\partial f}{\partial u}$.

Proof: By induction.

Suppose u is at layer t, and inductive hypothesis holds for layers t+1 and above. Sum of messages to u satisfies:

$$S = \sum_k m_k = \sum_k \frac{\partial f}{\partial v_i}\frac{\partial v_i}{\partial u} = \frac{\partial f}{\partial u}$$

Inductive hypothesis *Def. of messages*



Taking gradients of neural networks: backpropagation

Observation 3: The better way to do this is in a backward fashion.

Message passing algorithm [dynamic programming]: each node u receives messages (real numbers) from its neighbors on top. Let their sum be S. The node passes to downward neighbors z: $S\frac{\partial u}{\partial z}$. Proceed from top to down.

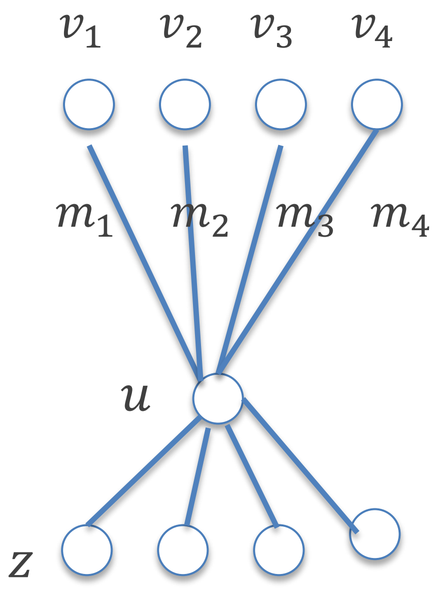
Amount of work: each node u needs to sum its upward neighbor messages (at most deg(u) of them), and pass a message to its downward neighbors (at most deg(u) of them).

Each downward message just takes an extra $\frac{\partial u}{\partial z}$ calculation (easy const. time), so each node does O(deg(u)) amount of work.

Hence, **total amount of work** for all nodes is $O(\sum_u \deg(u)) = O(E)$

Amount of memory: for calculating $\frac{\partial u}{\partial z}$, we need the activation values of intermediate nodes – so **memory** $O(V)$.

[Important! If recalculating these, runtime would be quadratic]


v_1
v_2
v_3
v_4
m_1
m_2
m_3
m_4
u
z