Adaptive Methods and Non-convex Optimization

CS6787 Lecture 5 — Spring 2026

Acceleration and Momentum

How does the step size affect convergence?

- Let's go back to gradient descent

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

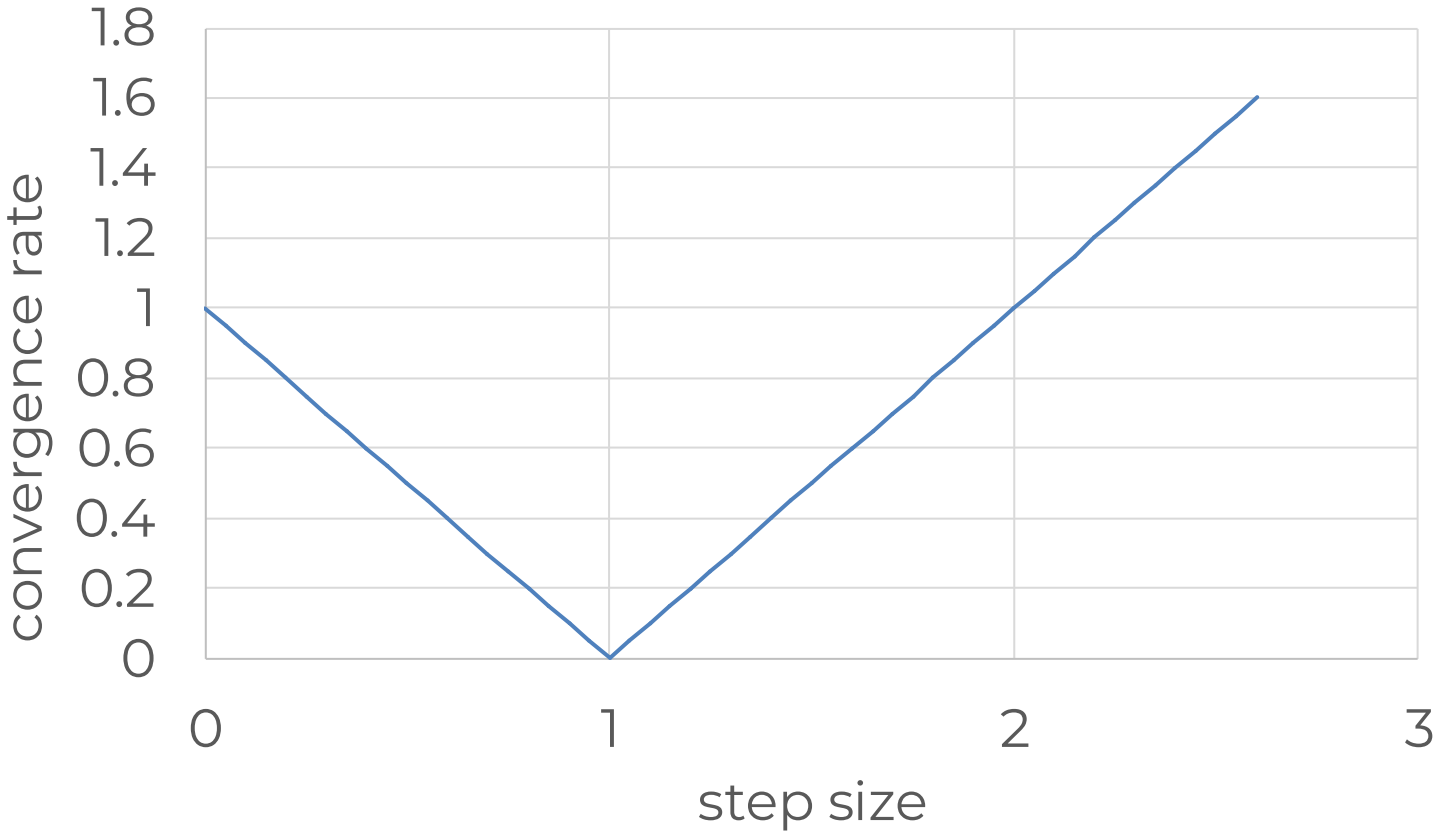
- Simplest possible case: a quadratic function

$$f(x) = \frac{1}{2}x^2$$

$$x_{t+1} = x_t - \alpha x_t = (1 - \alpha)x_t$$

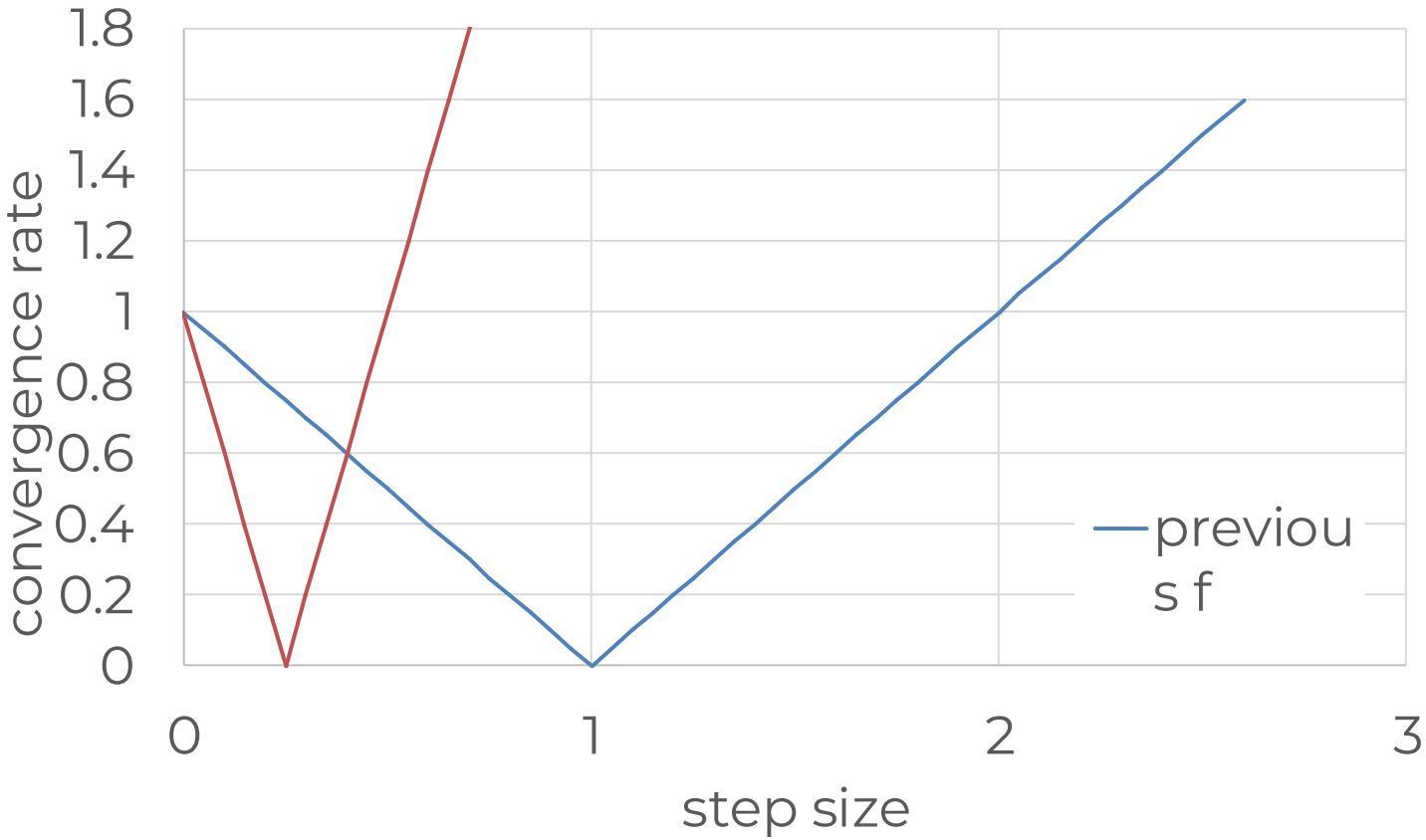
Step size vs. convergence: graphically

$$|x_{t+1} - 0| = |1 - \alpha| \, |x_t - 0|$$



What if the curvature is different?

$$f(x) = 2x^2 \qquad x_{t+1} = x_t - 4\alpha x_t = (1 - 4\alpha)x_t$$



Step size vs. curvature

- For these one-dimensional quadratics, how we should set **the step size depends on the curvature**
  - More curvature → smaller ideal step size

- What about higher-dimensional problems?
  - Let's look at a really simple quadratic that's just a sum of our examples.

$$f(x, y) = \frac{1}{2}x^2 + 2y^2$$

Simple two dimensional problem

$$f(x, y) = \frac{1}{2}x^2 + 2y^2$$

- Gradient descent:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} - \alpha \begin{bmatrix} x_t \\ 4y_t \end{bmatrix}$$

$$= \begin{bmatrix} 1-\alpha & 0 \\ 0 & 1-4\alpha \end{bmatrix} \begin{bmatrix} x_t \\ y_t \end{bmatrix}$$

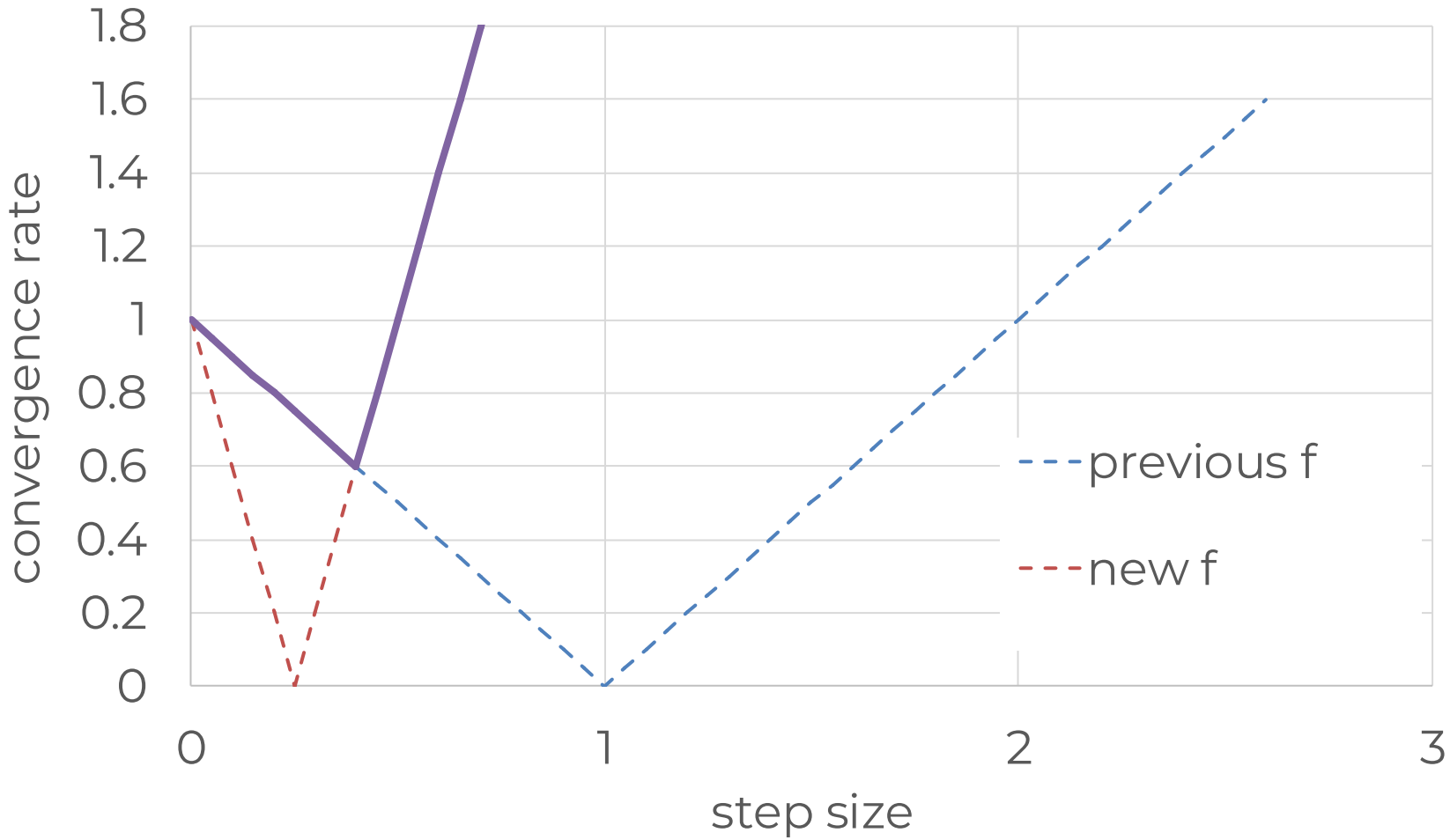
What's the convergence rate?

- Look at the worst-case contraction factor of the update

$$\max_{x,y} \frac{\left\| \begin{bmatrix} 1-\alpha & 0 \\ 0 & 1-4\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right\|}{\left\| \begin{bmatrix} x \\ y \end{bmatrix} \right\|} = \max(|1-\alpha|, |1-4\alpha|)$$

- Contraction is maximum of previous two values.

Convergence of two-dimensional quadratic



What does this example show?

- We'd like to set the step size larger for dimension with less curvature, and smaller for the dimension with more curvature.

- But we can't, because there is **only a single step-size parameter**.

- There's a **trade-off**
  - Optimal convergence rate is **substantially worse than** what we'd get in each scenario individually — individually we converge in one iteration.

For general quadratics

- For PSD symmetric A,

$$f(x) = \frac{1}{2} x^T A x$$

- Gradient descent has update step

$$x_{t+1} = x_t - \alpha A x_t = (I - \alpha A) x_t$$

- What does the convergence rate look like in general?

Convergence rate for general quadratics

$$\begin{aligned}
\max_x \frac{\|(I-\alpha A)x\|}{\|x\|} &= \max_x \frac{1}{\|x\|} \left\| \left(I - \alpha \sum_{i=1}^{n} \lambda_i u_i u_i^T \right) x \right\| \\
&= \max_x \frac{\left\| \sum_{i=1}^{n} (1-\alpha\lambda_i) u_i u_i^T x \right\|}{\left\| \sum_{i=1}^{n} u_i u_i^T x \right\|} \\
&= \max_i |1-\alpha\lambda_i| \\
&= \max(1-\alpha\lambda_{\min}, \alpha\lambda_{\max}-1)
\end{aligned}$$

Optimal convergence rate

- Minimize:

$$\max(1 - \alpha\lambda_{\min}, \alpha\lambda_{\max} - 1)$$

- Optimal value occurs when

$$1 - \alpha\lambda_{\min} = \alpha\lambda_{\max} - 1 \Rightarrow \alpha = \frac{2}{\lambda_{\max} + \lambda_{\min}}$$

- Optimal rate is

$$\max(1 - \alpha\lambda_{\min}, \alpha\lambda_{\max} - 1) = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}}$$

What affects this optimal rate?

$$\begin{aligned}\text{rate} &= \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \\ &= \frac{\lambda_{\max}/\lambda_{\min} - 1}{\lambda_{\max}/\lambda_{\min} + 1} \\ &= \frac{\kappa - 1}{\kappa + 1}.\end{aligned}$$

- Here, κ is called the **condition number** of the matrix **A**.

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$$

- Problems with larger condition numbers converge slower.
  - Called **poorly conditioned**.

Poorly conditioned problems

- Intuitively, these are problems that are **highly curved in some directions but flat in others**

- Happens pretty often in machine learning
  - Measure something unrelated → low curvature in that direction
  - Also affects stochastic gradient descent

- **How do we deal with this?**

Momentum

Motivation

- Can we tell the difference between the curved and flat directions using information that is already available to the algorithm?

- Idea: in the one-dimensional case, if the gradients are **reversing sign**, then the step size is too large
  - Because we're **over-shooting the optimum**
  - And if the gradients stay in the same direction, then step size is too small

- Can we leverage this to make steps smaller when gradients reverse sign and larger when gradients are consistently in the same direction?

Polyak Momentum

- Add extra **momentum term** to gradient descent

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

- Intuition: if current gradient step is in same direction as previous step, then move a little further in that direction.
  - And if it's in the opposite direction, move less far.

- Also known as the **heavy ball method**.

Momentum for 1D Quadratics

$$f(x) = \frac{\lambda}{2}x^2$$

- Momentum gradient descent gives

$$\begin{aligned} x_{t+1} &= x_t - \alpha\lambda x_t + \beta(x_t - x_{t-1}) \\ &= (1 + \beta - \alpha\lambda)x_t - \beta x_{t-1} \end{aligned}$$

Characterizing momentum for 1D quadratics

- Start with $x_{t+1} = (1 + \beta - \alpha\lambda)x_t - \beta x_{t-1}$

- Trick: let $x_t = \beta^{t/2} z_t$

$$\beta^{(t+1)/2} z_{t+1} = (1 + \beta - \alpha\lambda)\beta^{t/2} z_t - \beta \cdot \beta^{(t-1)/2} z_{t-1}$$

$$z_{t+1} = \frac{1 + \beta - \alpha\lambda}{\sqrt{\beta}} z_t - z_{t-1}$$

Characterizing momentum (continued)

- Let

$$u = \frac{1 + \beta - \alpha\lambda}{2\sqrt{\beta}}$$

- Then we get the simplified characterization

$$z_{t+1} = 2uz_t - z_{t-1}$$

- This is a degree-***t*** polynomial in **u**

Chebyshev Polynomials

- If we initialize such that $z_0 = 1, z_1 = u$ then these are a special family of polynomials called the **Chebyshev polynomials**

$$z_{t+1} = 2uz_t - z_{t-1}$$

- Standard notation:

$$T_{t+1}(u) = 2uT_t(u) - T_{t-1}(u)$$

- These polynomials have an important property: for all **t**

$$-1 \leq u \leq 1 \Rightarrow -1 \leq z_t \leq 1$$

To see why they are bounded...

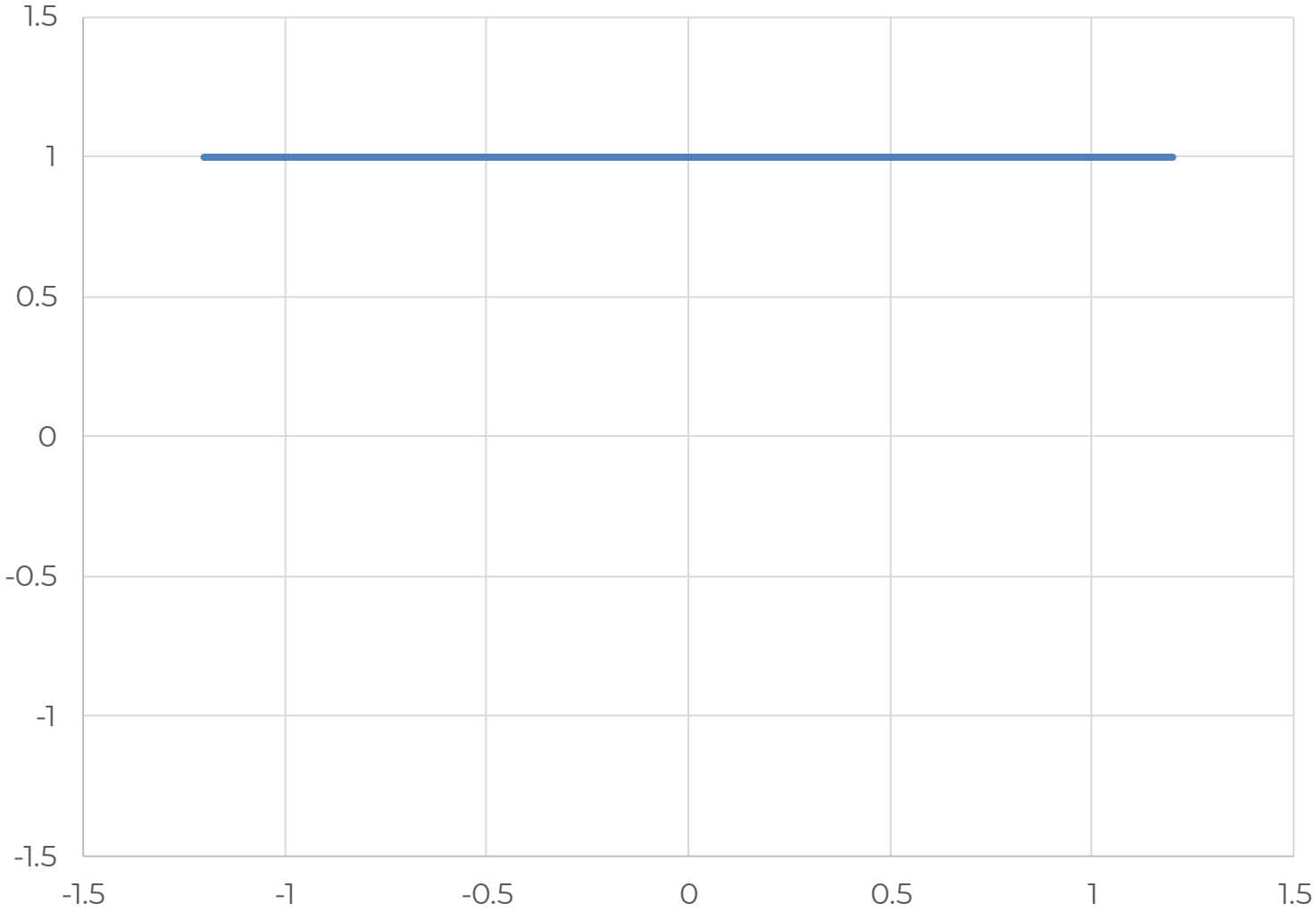
- **Chebyshev polynomials** $z_0 = 1,\ z_1 = u$

$$z_{t+1} = 2uz_t - z_{t-1}$$

- We can write them in terms of trig functions

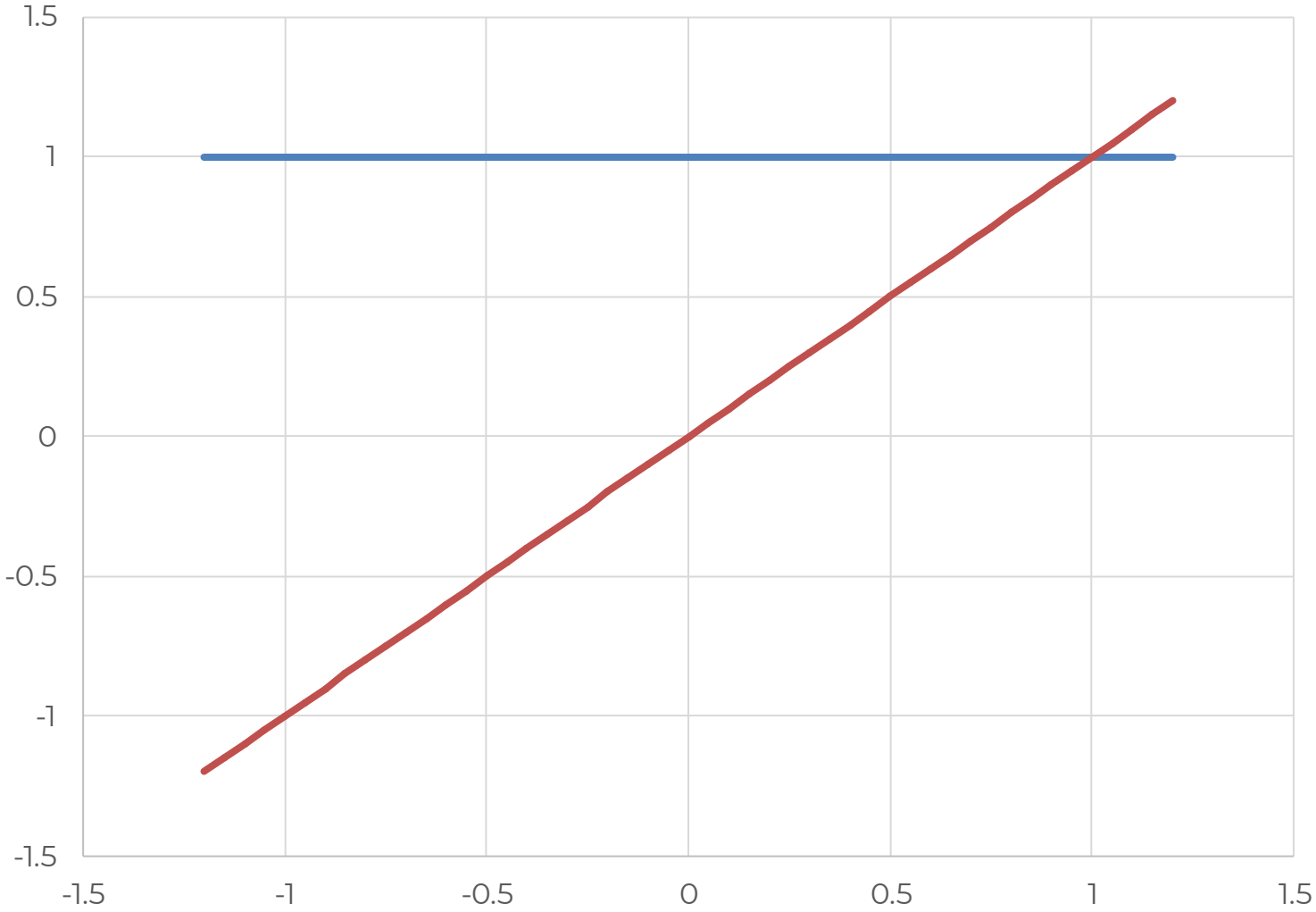
$$\cos((t+1)\theta) = \cos(\theta)\cos(t\theta) - \sin(\theta)\sin(t\theta)$$
$$\cos((t-1)\theta) = \cos(\theta)\cos(t\theta) + \sin(\theta)\sin(t\theta),$$
$$\underbrace{\cos((t+1)\theta)}_{T_{t+1}(u)} = 2\underbrace{\cos(\theta)}_{u}\underbrace{\cos(t\theta)}_{T_t(u)} - \underbrace{\cos((t-1)\theta)}_{T_{t-1}(u)}.$$

Chebyshev Polynomials



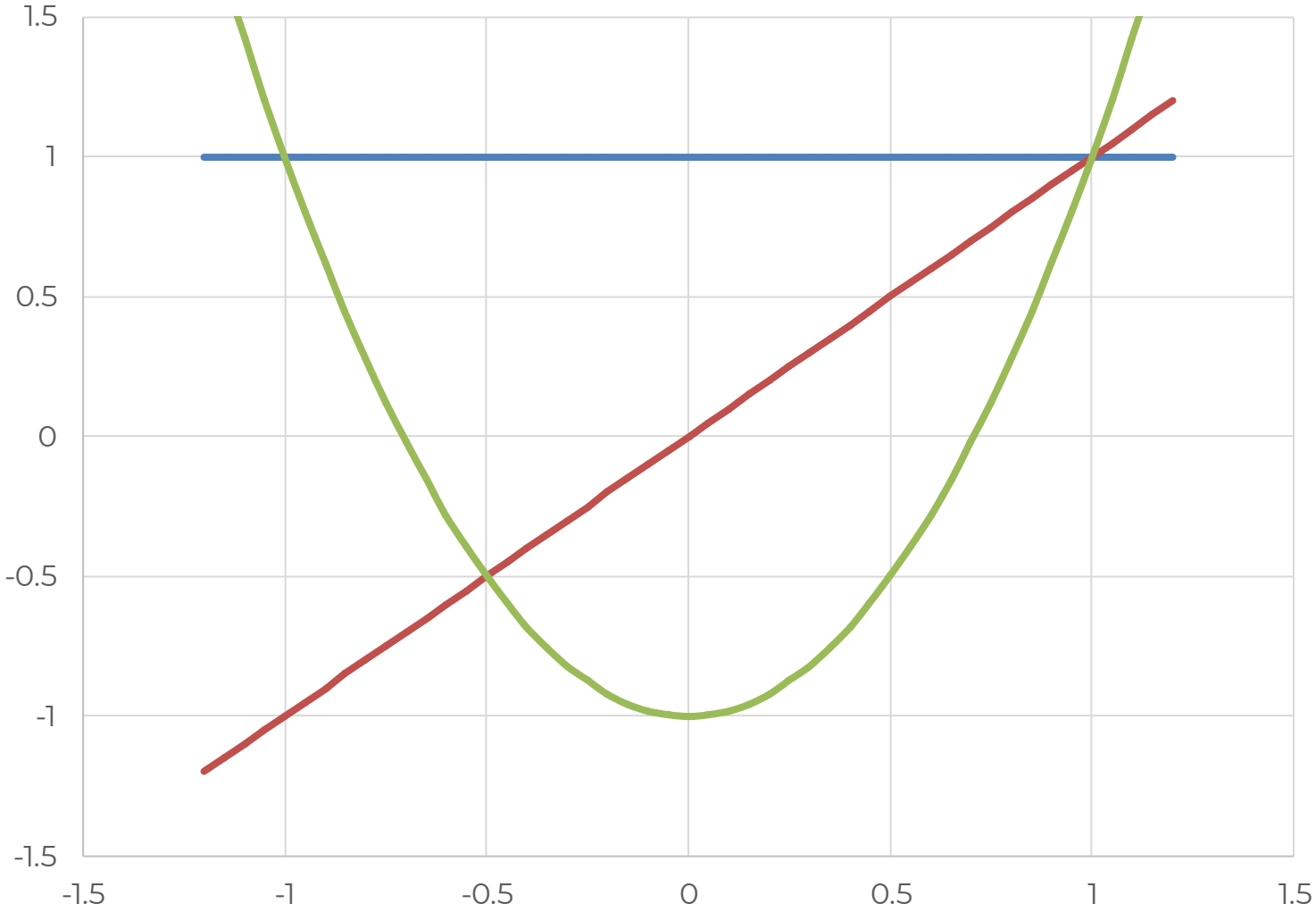
$$T_0(u) = 1$$

Chebyshev Polynomials



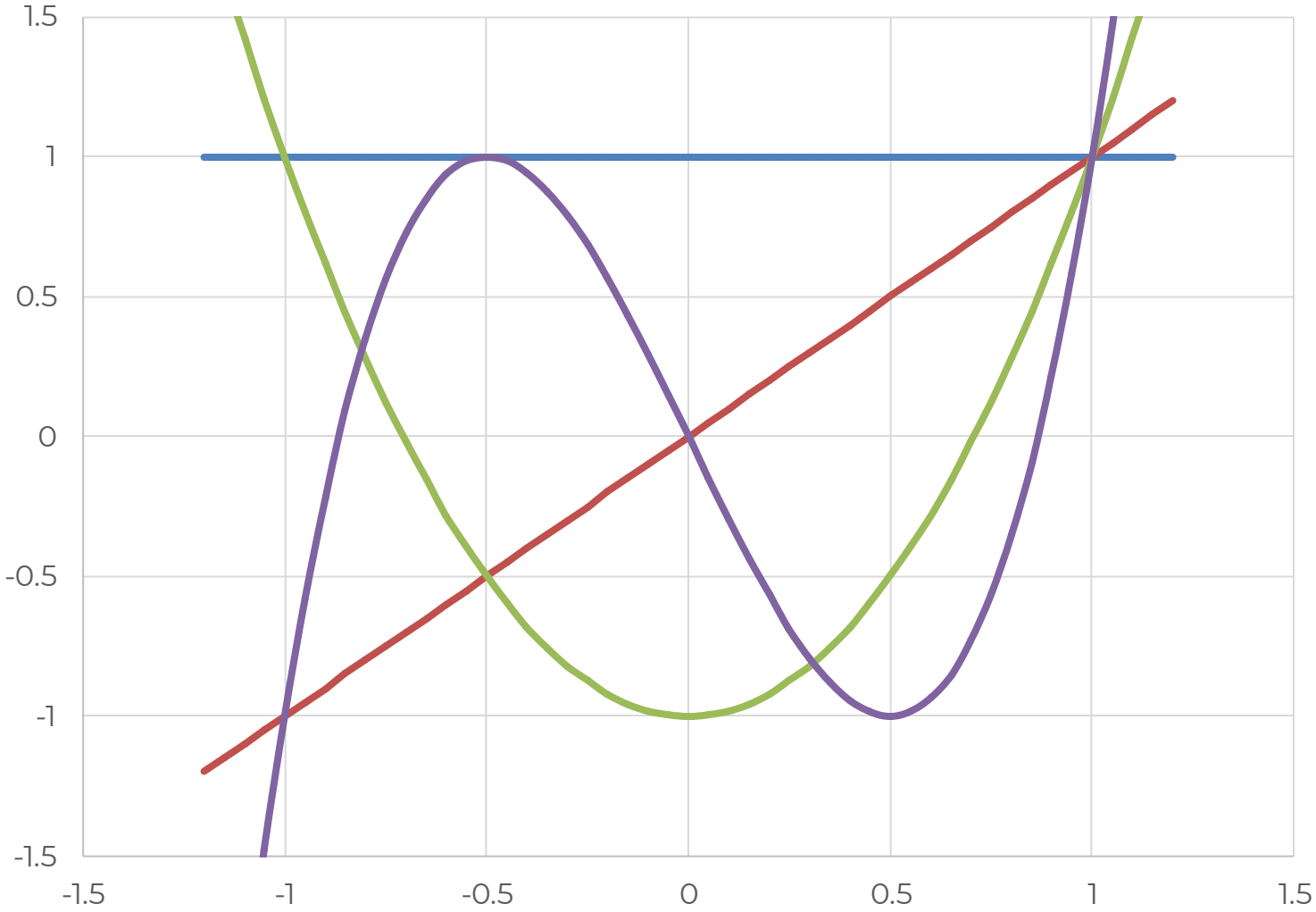
$$T_1(u) = u$$

Chebyshev Polynomials

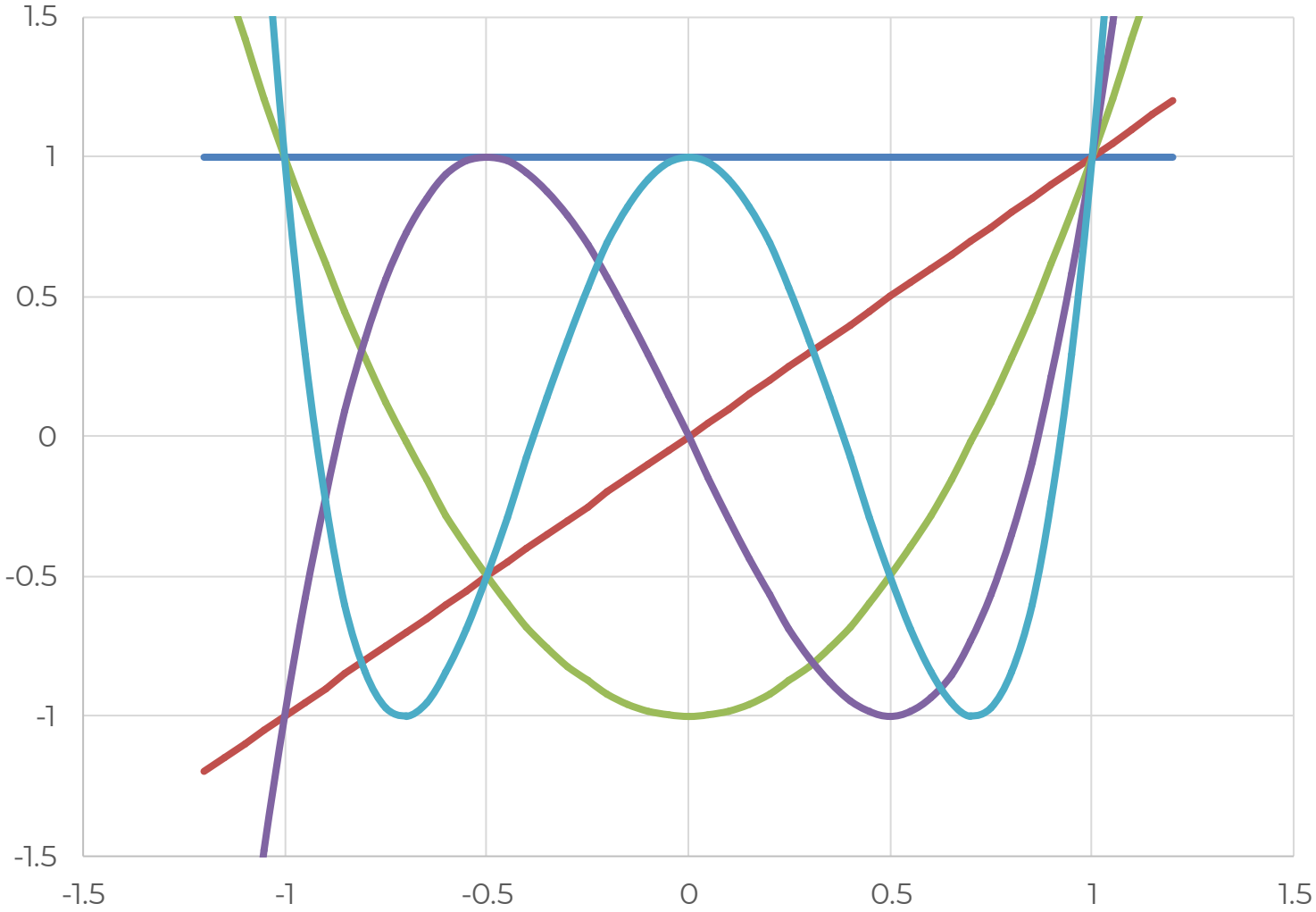


$$T_2(u) = 2u^2 - 1$$

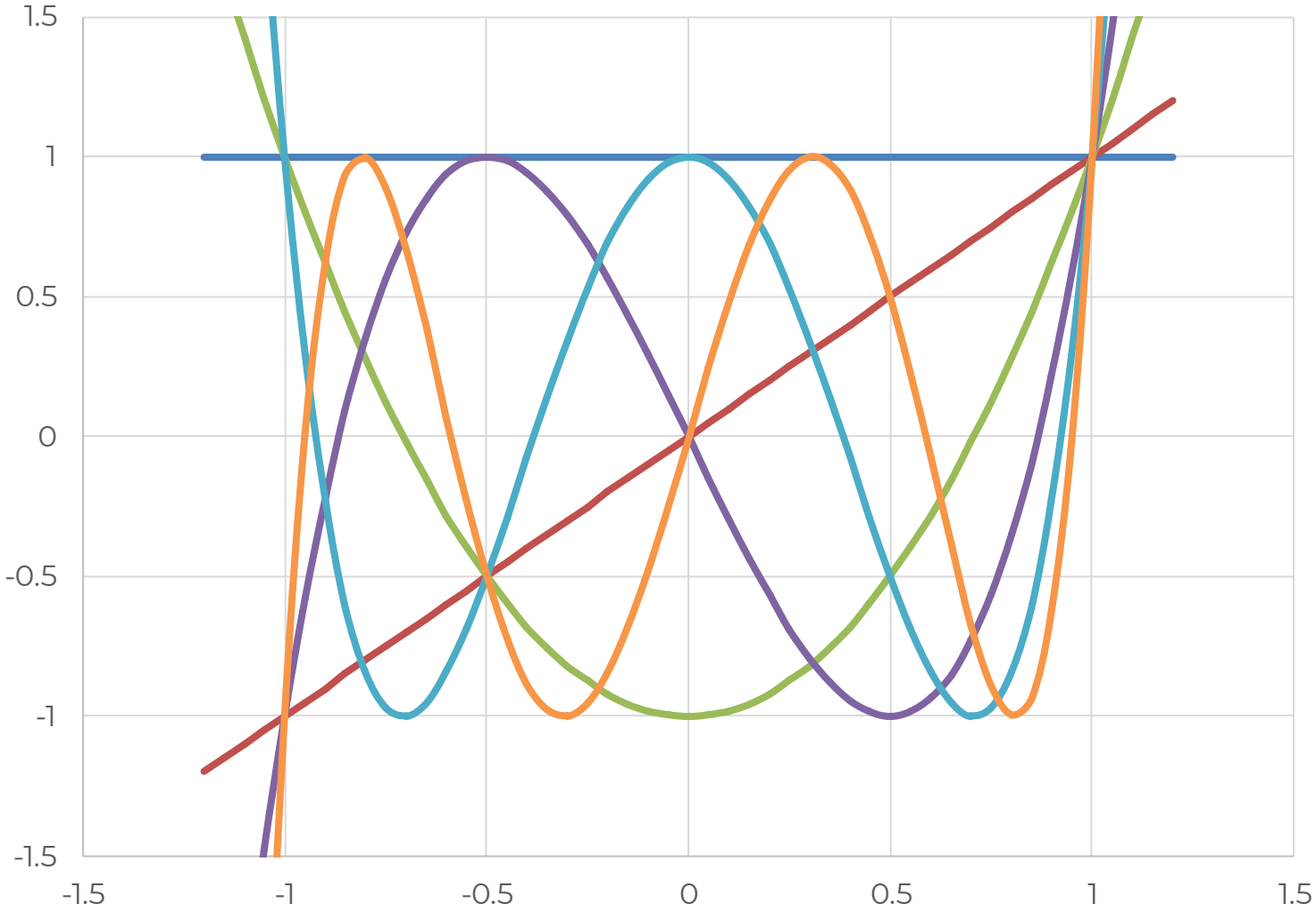
Chebyshev Polynomials



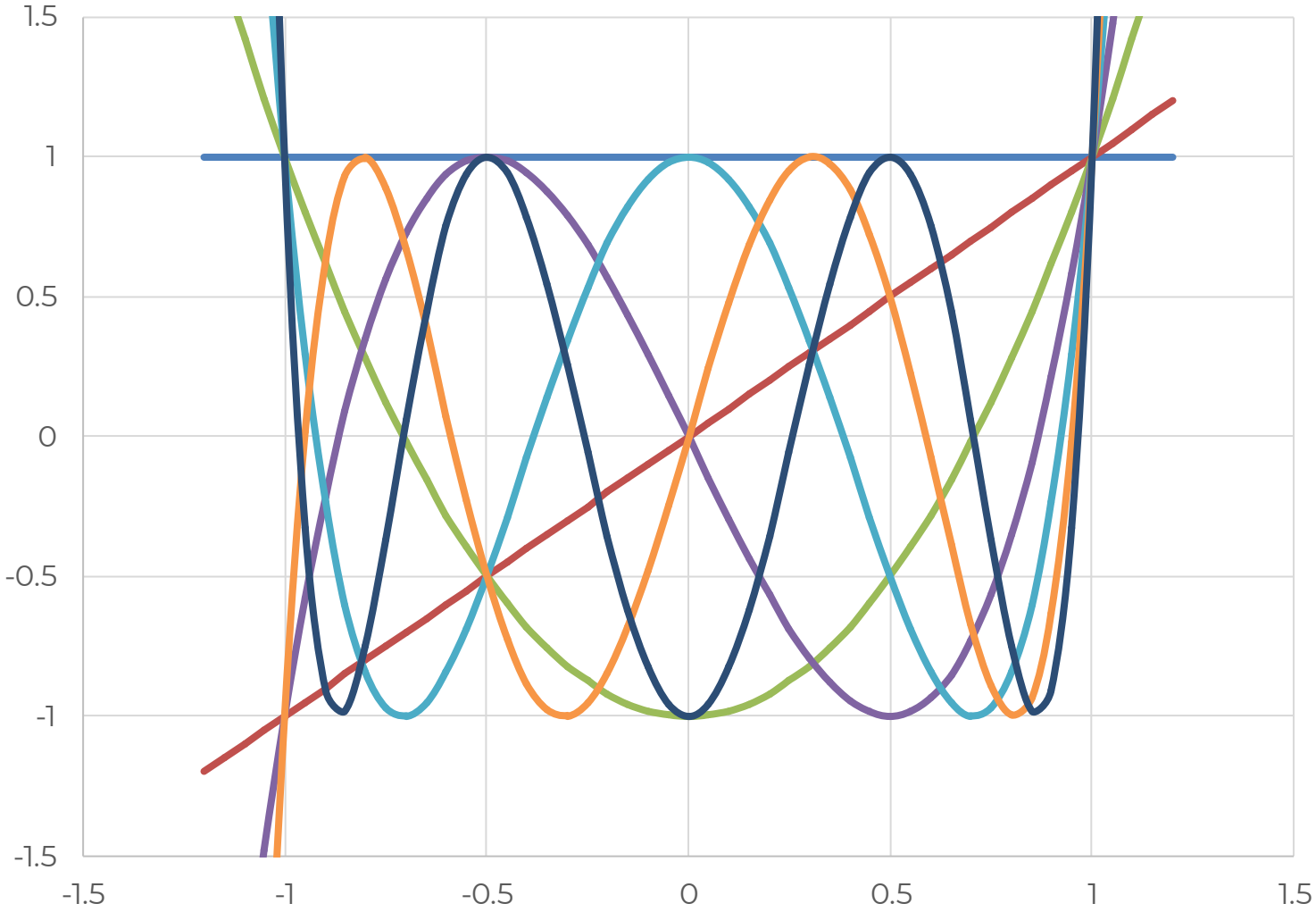
Chebyshev Polynomials



Chebyshev Polynomials



Chebyshev Polynomials



Characterizing momentum (continued)

- What does this mean for our 1D quadratics?
  - Recall that we let $x_t = \beta^{t/2} z_t$

$$x_t = \beta^{t/2} \cdot x_0 \cdot T_t(u)$$
$$= \beta^{t/2} \cdot x_0 \cdot T_t\left(\frac{1+\beta-\alpha\lambda}{2\sqrt{\beta}}\right)$$

- So

$$-1 \leq \frac{1+\beta-\alpha\lambda}{2\sqrt{\beta}} \leq 1 \Rightarrow |x_t| \leq \beta^{t/2} |x_0|$$

Consequences of momentum analysis

- Convergence rate depends **only on momentum parameter β**
  - Not on step size or curvature.

- We **don't need to be that precise in setting the step size**
  - It just needs to be within a window
  - Pointed out in "*YellowFin and the Art of Momentum Tuning*" by Zhang et. al.

- If we have a multidimensional quadratic problem, the **convergence rate will be the same in all directions**
  - This is different from the gradient descent case where we had a trade-off

Choosing the parameters

- How should we **set the step size and momentum parameter** if we only have bounds on λ ?

- Need: $$-1 \leq \frac{1+\beta-\alpha\lambda}{2\sqrt{\beta}} \leq 1$$

- Suffices to have:

$$-1 = \frac{1+\beta-\alpha\lambda_{\max}}{2\sqrt{\beta}} \text{ and } \frac{1+\beta-\alpha\lambda_{\min}}{2\sqrt{\beta}} = 1$$

Choosing the parameters (continued)

- Adding both equations:

$$0 = \frac{2 + 2\beta - \alpha\lambda_{\max} - \alpha\lambda_{\min}}{2\sqrt{\beta}}$$

$$0 = 2 + 2\beta - \alpha\lambda_{\max} - \alpha\lambda_{\min}$$

$$\alpha = \frac{2 + 2\beta}{\lambda_{\max} + \lambda_{\min}}$$

Choosing the parameters (continued)

- Subtracting both equations:

$$\frac{1+\beta-\alpha\lambda_{\min}-1-\beta+\alpha\lambda_{\max}}{2\sqrt{\beta}}=2$$

$$\frac{\alpha(\lambda_{\max}-\lambda_{\min})}{2\sqrt{\beta}}=2$$

Choosing the parameters (continued)

- Combining these results: $\alpha = \dfrac{2+2\beta}{\lambda_{\max}+\lambda_{\min}}$ $\dfrac{\alpha(\lambda_{\max}-\lambda_{\min})}{2\sqrt{\beta}} = 2$

$$\frac{2+2\beta}{\lambda_{\max}+\lambda_{\min}} \cdot \frac{(\lambda_{\max}-\lambda_{\min})}{2\sqrt{\beta}} = 2$$

$$0 = 1 - 2\sqrt{\beta}\frac{\lambda_{\max}+\lambda_{\min}}{\lambda_{\max}-\lambda_{\min}} + \beta$$

Choosing the parameters (continued)

- Quadratic formula: $0 = 1 - 2\sqrt{\beta}\frac{\lambda_{\max} + \lambda_{\min}}{\lambda_{\max} - \lambda_{\min}} + \beta$

$$\sqrt{\beta} = \frac{\kappa+1}{\kappa-1} - \sqrt{\left(\frac{\kappa+1}{\kappa-1}\right)^2 - 1}$$

$$= \frac{\kappa+1}{\kappa-1} - \sqrt{\frac{4\kappa}{\kappa^2 - 2\kappa + 1}}$$

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

Gradient Descent versus Momentum

- Recall: gradient descent had a convergence rate of

$$\frac{\kappa - 1}{\kappa + 1}$$

- But with momentum, the optimal rate is

$$\sqrt{\beta} = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$$

- This is called **convergence at an accelerated rate**

Adaptive learning rates

- So far, we've looked at update steps that look like

$$w_{t+1} = w_t - \alpha_t \nabla f_t(w_t)$$

- Here, the learning rate/step size is **fixed a priori** for each iteration.

- What if we use a **step size that varies depending on the model**?

- This is the idea of an **adaptive learning rate**.

Example: Polyak's step length

- This is an simple step size scheme for **gradient descent** that works when the optimal value is known.

$$\alpha_k = \frac{f(w_k) - f(w^*)}{\|\nabla f(w_k)\|^2}$$

- Can also use this with an estimated optimal value.

Intuition: Polyak's step length

- Approximate the objective with a linear approximation at the current iterate.

$$\hat{f}(w) = f(w_k) + (w - w_k)^T \nabla f(w_k)$$

- Choose the step size that makes the approximation equal to the known optimal value.

$$\begin{aligned} f^* = \hat{f}(w_{k+1}) &= \hat{f}(w_k - \alpha \nabla f(w_k)) \\ &= f(w_k) - \alpha \|\nabla f(w_k)\|^2 \end{aligned} \quad \Rightarrow \alpha = \frac{f(w_k) - f^*}{\|\nabla f(w_k)\|^2}$$

Example: Line search

- Idea: just choose the step size that minimizes the objective.

$$\alpha_k = \arg\min_{\alpha>0} f(w_k - \alpha \nabla f(w_k))$$

- Only works well for **gradient descent**, not SGD.

- Why?
  - SGD moves in random directions that **don't always improve the objective.**
  - Doing line search on full objective is **expensive relative to SGD update.**

Adaptive methods for SGD

- Several methods exist
  - AdaGrad
  - AdaDelta
  - RMSProp
  - Muon
  - Adam

- You'll see Adam in one of next Wednesday's papers

AdaGrad

Adaptive gradient descent

Per-parameter adaptive learning rate schemes

- Main idea: set the **learning rate per-parameter** dynamically at each iteration based on observed statistics of the past gradients.

$$(w_{t+1})_j = (w_t)_j - \alpha_{j,t} \left(\nabla f(w_t; x_t)\right)_j$$

  - Where the step size now depends on the parameter index **j**
  - Corresponds to a multiplication of the gradient by a diagonal scaling matrix.

- There are many different schemes in this class

AdaGrad: One of the first adaptive methods

- **AdaGrad:** Adaptive subgradient methods for online learning and stochastic optimization
  - J Duchi, E Hazan, Y Singer
  - Journal of Machine Learning Research, 2011

- High level approach: can use **history of sampled gradients** to choose the step size for the next SGD step to be inversely proportional to the usual magnitude of gradient steps in that direction
  - On a per-parameter basis.

AdaGrad

Algorithm 1 AdaGrad

input: learning rate factor η, initial parameters w_0
initialize $t \leftarrow 0$
loop
sample a stochastic gradient $g_t \leftarrow \nabla f(w_t; x_t)$
update model: for all $j \in \{1, \ldots, d\}$

$$(w_{t+1})_j \leftarrow (w_t)_j - \frac{\eta}{\sqrt{\sum_{k=0}^{t} (g_t)_j^2 + \epsilon}} \cdot g_j$$

$t \leftarrow t + 1$
end loop

Can think of this as like the norm of the gradients in the jth parameter.

Memory-efficient implementation of AdaGrad

Algorithm 1 AdaGrad

input: learning rate factor η, initial parameters $w_0 \in \mathbb{R}^d$, small number ϵ
initialize $t \leftarrow 0$
initialize $r \leftarrow 0 \in \mathbb{R}^d$
loop
  sample a stochastic gradient $g_t \leftarrow \nabla f(w_t; x_t)$
  accumulate second moment estimate $r_j \leftarrow r_j + (g_t)_j^2$ for all $j \in \{1, \ldots, d\}$
  update model: for all $j \in \{1, \ldots, d\}$

$$(w_{t+1})_j \leftarrow (w_t)_j - \frac{\eta}{\sqrt{r_j} + \epsilon} \cdot g_j$$

  $t \leftarrow t + 1$
end loop

Important thing to notice: step size is monotonically decreasing!

AdaGrad for Non-convex Optimization

- **What problems might arise when using AdaGrad for non-convex optimization?**
  - Think about the step size always decreasing. Could this cause a problem?

- **If you do think of a problem that might arise, how could you change AdaGrad to fix it?**

RMSProp

Algorithm 1 RMSProp

input: learning rate factor η, initial parameters $w_0 \in \mathbb{R}^d$,
initialize $t \leftarrow 0$
initialize $r \leftarrow 0 \in \mathbb{R}^d$
loop
sample a stochastic gradient $g_t \leftarrow \nabla f(w_t; x_t)$
accumulate second moment estimate $r_j \leftarrow \rho \cdot r_j + (1-\rho)\,(g_t)_j^2$ for all $j \in \{1, \ldots, d\}$
update model: for all $j \in \{1, \ldots, d\}$

$$(w_{t+1})_j \leftarrow (w_t)_j - \frac{\eta}{\sqrt{r_j} + \epsilon} \cdot g_j$$

$t \leftarrow t + 1$
end loop

Just replaces the gradient accumulation of AdaGrad with an exponential moving average.

A systems perspective

- **What is the computational cost of AdaGrad, RMSProp, and Adam?**
  - **How much additional memory is required compared to baseline SGD?**
  - **How much additional compute is used?**

Adam

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1$^{\text{st}}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize 2$^{\text{nd}}$ moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Muon optimizer

- Momentum SGD but with an approximate projection of the update onto the set of orthogonal matrices.

$$\mathbf{M}_t = \mu \mathbf{M}_{t-1} + \nabla \mathcal{L}_t(\mathbf{W}_{t-1})$$
$$\mathbf{O}_t = \text{Newton-Schulz}(\mathbf{M}_t)^1$$
$$\mathbf{W}_t = \mathbf{W}_{t-1} - \eta_t \mathbf{O}_t$$

$$M_t = \mu M_{t-1} + \nabla \mathcal{L}_t(W_{t-1})$$
$$O_t = \arg \min_{\hat{O} | \hat{O}^T \hat{O} = I} \|\hat{O} - M_t\|_F^2$$
$$W_t = W_{t-1} - \eta_t O_t$$

- Lots of strong empirical results!
- Very useful because it **eliminates the Adam second-moment buffer.**

Adaptive methods, summed up

- Generally useful when we can expect there to be **different scales for different parameters**
  - But can even work well when that doesn't happen, as we saw in the demo.
- Very commonly used class of methods for training ML models.
- We'll see more of this when we study **Adam** on Wednesday
  - Adam is basically RMSProp + Momentum.

Algorithms other than SGD

Machine learning is not just SGD

- Once a model is trained, we need to use it to classify new examples
  - This **inference task** is not computed with SGD
- There are other algorithms for optimizing objectives besides SGD
  - **Stochastic coordinate descent**
  - **Derivative-free optimization**
- There are other common tasks, such as sampling from a distribution
  - **Gibbs sampling** and other Markov chain Monte Carlo methods
  - And we sometimes use this together with SGD → called **contrastive divergence**

Why understand these algorithms?

- They represent a significant fraction of machine learning computations
  - **Inference** in particular is huge
- You may want to use them **instead of SGD**
  - But you don't want to suddenly pay a computational penalty for doing so because you don't know how to make them fast
- **Intuition from SGD** can be used to make these algorithms faster too
  - And vice-versa

Review — We've covered many methods

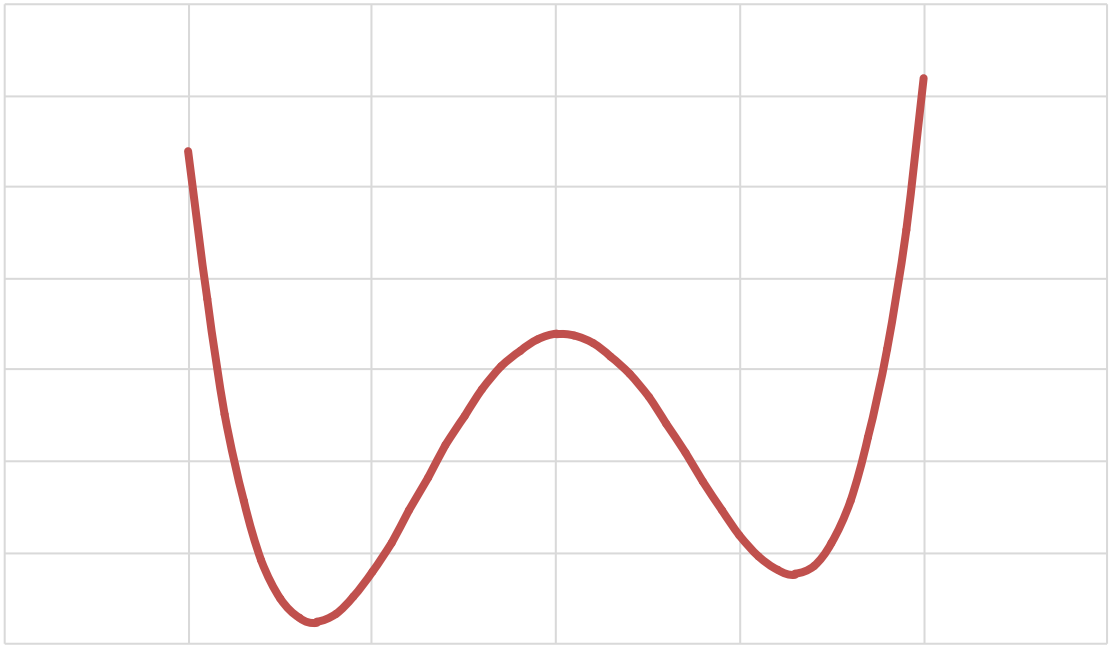
- Stochastic gradient descent
- Mini-batching
- Momentum

- **Nice convergence proofs that give us a rate**

- But **only for convex problems!**

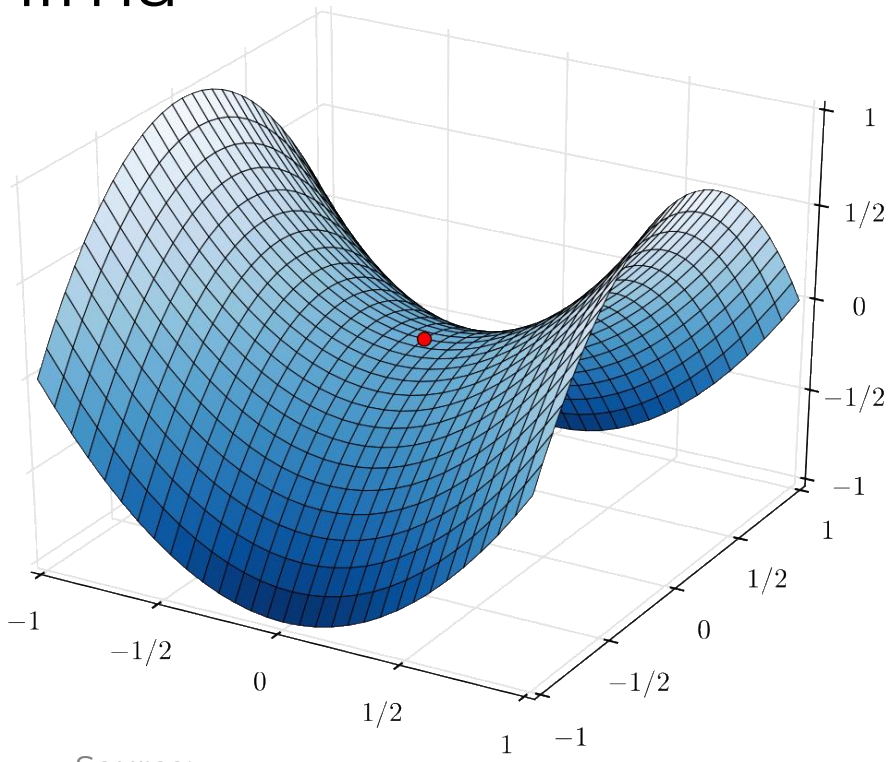
Non-Convex Problems

- Anything that's **not convex**



What makes non-convex optimization hard?

- Potentially many local minima
- Saddle points
- Very flat regions
- Widely varying curvature



Source: https://commons.wikimedia.org/wiki/File:Saddle_point.svg

But is it actually that hard?

- Yes, non-convex optimization is at least **NP-hard**
  - Can encode most problems as non-convex optimization problems

- Example: subset sum problem
  - **Given a set of integers, is there a non-empty subset whose sum is zero?**
  - Known to be NP-complete.

- How do we encode this as an optimization problem?

Subset sum as non-convex optimization

- Let $\mathbf{a_1}$, $\mathbf{a_2}$, …, $\mathbf{a_n}$ be the input integers
- Let $\mathbf{x_1}$, $\mathbf{x_2}$, …, $\mathbf{x_n}$ be 1 if $\mathbf{a_i}$ is in the subset, and 0 otherwise
- Objective:

$$\text{minimize} \ \left(a^T x\right)^2 + \sum_{i=1}^{n} x_i^2 (1 - x_i)^2$$

- **What is the optimum if subset sum returns true? What if it's false?**

So non-convex optimization is pretty hard

- There can't be a general algorithm to solve it efficiently in all cases

- Downsides: **theoretical guarantees are weak** or nonexistent
  - Depending on the application
  - There's usually no theoretical recipe for setting hyperparameters

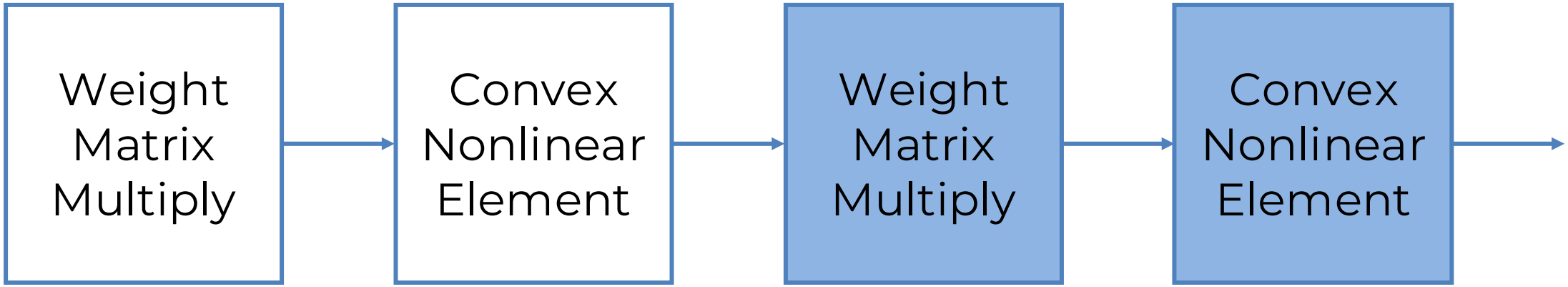
- Upside: an endless array of **problems to try to solve better**
  - And gain theoretical insight about
  - And improve the performance of implementations

Examples of non-convex problems

- Matrix completion, principle component analysis
- Low-rank models and tensor decomposition
- Maximum likelihood estimation with hidden variables
  - Usually non-convex
- The big one: **deep neural networks**

Why are neural networks non-convex?

- They're often made of convex parts!
  - **This by itself would be convex**.



- **Composition of convex functions is not convex**
  - So deep neural networks also aren't convex

Why do neural nets need to be non-convex?

- Neural networks are **universal function approximators**
  - With enough neurons, they can learn to approximate any function arbitrarily well
- To do this, they need to be able to approximate non-convex functions
  - **Convex functions can't approximate non-convex** ones well.
- Neural nets also have many symmetric configurations
  - For example, exchanging intermediate neurons
  - This symmetry means they can't be convex. **Why?**

How to solve non-convex problems?

- Can use many of the **same techniques as before**
  - Stochastic gradient descent
  - Mini-batching
  - SVRG
  - Momentum

- There are also specialized methods for solving non-convex problems
  - Alternating minimization methods
  - Branch-and-bound methods
  - These generally **aren't very popular for machine learning** problems

Varieties of theoretical convergence results

- Convergence **to a stationary point**
- Convergence **to a local minimum**
- **Local convergence** to the global minimum
- **Global convergence** to the global minimum

Non-convex
Stochastic Gradient Descent

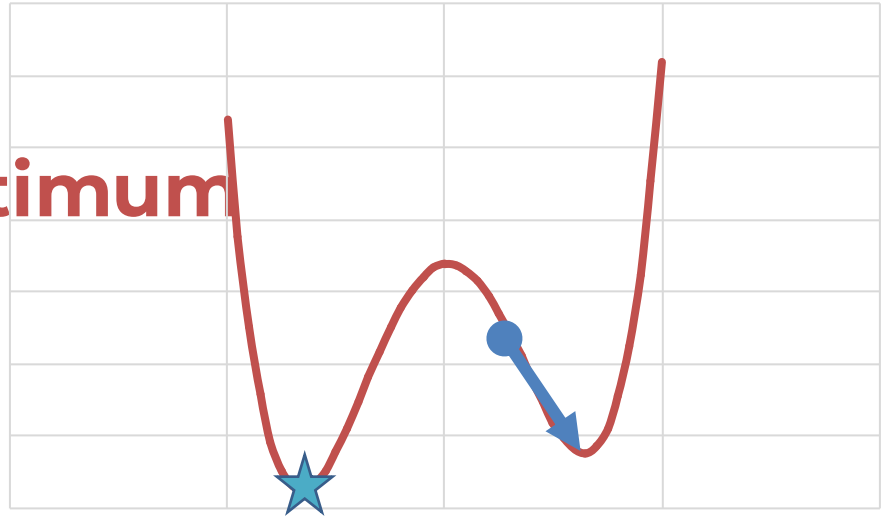
Stochastic Gradient Descent

- The update rule is the same for non-convex functions

$$w_{t+1} = w_t - \alpha_t \nabla \tilde{f}_t(w_t)$$

- Same intuition of **moving in a direction that lowers objective**

- **Doesn't necessarily go towards optimum**
  - Even in expectation



Non-convex SGD: A Systems Perspective

- It's **exactly the same as the convex case**!
- The **hardware doesn't care** whether our gradients are from a convex function or not
- This means that all our intuition about computational efficiency from the convex case directly applies to the non-convex case
- But **does our intuition about statistical efficiency also apply?**

When can we say SGD converges?

- First, we need to decide what type of convergence we want to show
  - Here I'll just show **convergence to a stationary point**, the weakest type

- Assumptions:
  - Second-differentiable objective $-LI \preceq \nabla^2 f(x) \preceq LI$
  - Lipschitz-continuous gradients
  - Noise has bounded variance $\mathbf{E}\left[\left\|\nabla \tilde{f}_t(x) - \tilde{f}(x)\right\|^2\right] \leq \sigma^2$
  - But **no convexity assumption**!

Convergence of Non-Convex SGD

- Start with the update rule:

$$w_{t+1} = w_t - \alpha_t \nabla \tilde{f}_t(w_t)$$

- At the next time step, by Taylor's theorem, the objective will be

$$f(w_{t+1}) = f(w_t - \alpha_t \nabla \tilde{f}_t(w_t))$$

$$= f(w_t) - \alpha_t \nabla \tilde{f}_t(w_t)^T \nabla f(w_t) + \frac{\alpha_t^2}{2} \nabla \tilde{f}_t(w_t)^T \nabla^2 f(y_t) \nabla \tilde{f}_t(w_t)$$

$$\geq f(w_t) - \alpha_t \nabla f_t(w_t)^T \nabla f(w_t) + \frac{\alpha_t^2}{2} \| \nabla f_t(w_t) \|$$

Convergence (continued)

- Taking the expected value

$$\mathbf{E}\left[f(w_{t+1})|w_t\right] \leq f(w_t) - \alpha_t \mathbf{E}\left[\nabla \tilde{f}_t(w_t)^T \nabla f(w_t) \middle| w_t\right] + \frac{\alpha_t^2 L}{2} \mathbf{E}\left[\left\|\nabla \tilde{f}_t(w_t)\right\|^2 \middle| w_t\right]$$

Convergence (continued)

- So now we know how the expected value of the objective evolves.

$$\mathbf{E}\left[f(w_{t+1})|w_t\right] \leq f(w_t) - \left(\alpha_t - \frac{\alpha_t^2 L}{2}\right)\|\nabla f(w_t)\|^2 + \frac{\alpha_t^2 \sigma^2 L}{2}.$$

- If we set α small enough that 1 - αL/2 > 1/2, then

$$\mathbf{E}\left[f(w_{t+1})|w_t\right] \leq f(w_t) - \frac{\alpha_t}{2}\|\nabla f(w_t)\|^2 + \frac{\alpha_t^2 \sigma^2 L}{2}.$$

Convergence (continued)

- Now taking the full expectation,

$$\mathbf{E}\left[f(w_{t+1})\right] \leq \mathbf{E}\left[f(w_t)\right] - \frac{\alpha_t}{2}\mathbf{E}\left[\left\|\nabla f(w_t)\right\|^2\right] + \frac{\alpha_t^2 \sigma^2 L}{2}.$$

- And summing up over an epoch of length **T**

$$\mathbf{E}\left[f(w_T)\right] \leq f(w_0) - \sum_{t=0}^{T-1} \frac{\alpha_t}{2}\mathbf{E}\left[\left\|\nabla f(w_t)\right\|^2\right] + \sum_{t=0}^{T-1} \frac{\alpha_t^2 \sigma^2 L}{2}.$$

Convergence (continued)

- Now we need to decide how to set the step size α_t
  - Let's just **choose a constant step size** for simplicity

- So our bound on the objective becomes

$$\mathbf{E}[f(w_t)] \leq f(w_0) - \frac{\alpha}{2} \sum_{t=0}^{T-1} \mathbf{E}\left[\|\nabla f(w_t)\|^2\right] + \frac{\alpha^2 \sigma^2 LT}{2}$$

Convergence (continued)

- Rearranging the terms,

$$\frac{\alpha}{2}\sum_{t=0}^{T-1}\mathbf{E}\left[\|\nabla f(w_t)\|^2\right] \leq f(w_0) - \mathbf{E}[f(w_t)] + \frac{\alpha^2\sigma^2 LT}{2}$$

$$\leq f(w_0) - \left(\min_w f(w)\right) + \frac{\alpha^2\sigma^2 LT}{2}$$

$$\leq f(w_0) - f^* + \frac{\alpha^2\sigma^2 LT}{2}$$

Now, we're kinda stuck

- How do we use the bound on this term to say something useful?

$$\frac{\alpha}{2} \sum_{t=0}^{T-1} \mathbf{E}\left[\|\nabla f(w_t)\|^2\right]$$

- Idea: rather than outputting $\mathbf{w_T}$, instead output some randomly chosen $\mathbf{w_i}$ from the history.
  - You might recall this trick from the proof in the SVRG paper.

$$\text{Let } z_T = w_t \text{ with probability } 1/T \text{ for all } t \in \{0, \ldots, T-1\}.$$

Using our randomly chosen output

Let $z_T = w_t$ with probability $1/T$ for all $t \in \{0, \ldots, T-1\}$.

- So the expected value of the gradient at this point is

$$\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] = \sum_{t=0}^{T-1} \mathbf{E}\left[\|\nabla f(w_t)\|^2\right] \cdot \mathbf{P}(z_T = w_t)$$

$$= \frac{1}{T} \sum_{t=0}^{T-1} \mathbf{E}\left[\|\nabla f(w_t)\|^2\right]$$

Convergence (continued)

- Substituting this back into our previous bound gives us

$$\frac{\alpha T}{2}\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] \leq f(w_0) - f^* + \frac{\alpha^2\sigma^2 LT}{2}$$

- And this simplifies to

$$\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] \leq \frac{2(f(w_0) - f^*)}{\alpha T} + \frac{\alpha\sigma^2 L}{2}$$

Convergence (continued)

- Now, if we know that we are going to run for **T** iterations, we can set the step size to minimize this expression

$$\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] \leq \frac{2(f(w_0) - f^*)}{\alpha T} + \frac{\alpha \sigma^2 L}{2}$$

$$\alpha = \frac{c}{\sqrt{T}} \Rightarrow \mathbf{E}\left[\|\nabla f(z_T)\|^2\right] \leq \frac{1}{\sqrt{T}} \cdot \left(\frac{2(f(w_0) - f^*)}{c} + \frac{c\sigma^2 L}{2}\right)$$

Convergence Takeaways

- So even **non-convex SGD converges!**
  - In the sense of getting to points where the **gradient is arbitrarily small**
- But this **doesn't mean it goes to a local minimum**!
  - Doesn't rule out that it goes to a saddle point, or a local maximum.
  - Doesn't rule out that it goes to a region of very flat but nonzero gradients.
- Certainly **doesn't mean that it finds the global optimum**
- And the theoretical **rate here was slow** $1/\sqrt{T}$

Strengthening these theoretical results

Convergence to a local minimum

- Under stronger conditions, can prove that SGD **converges to a local minimum**
  - For example using the **strict saddle property** (Ge et al 2015)

- Using even stronger properties, can prove that **SGD converges to a local minimum with an explicit convergence rate** of **1/T**

- **But, it's unclear whether common classes of non-convex problems, such as neural nets, actually satisfy these stronger conditions.**

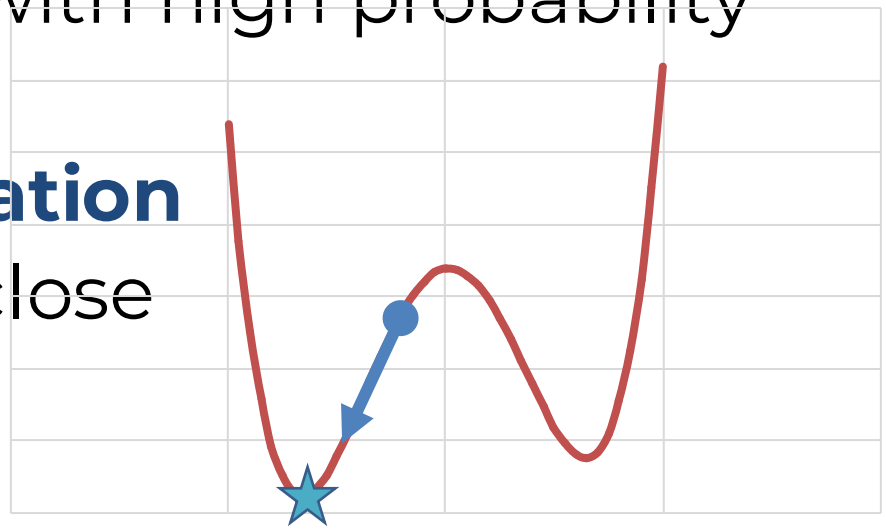
Strengthening these theoretical results

Local convergence to the global minimum

- Another type of result you'll see are **local convergence results**

- Main idea: if we **start close enough to the global optimum**, we will converge there with high probability

- Results often give **explicit initialization scheme** that is guaranteed to be close
  - But it's often **expensive to run**
  - And **limited to specific problems**



Strengthening these theoretical results

Global convergence to a global minimum

- The strongest result is **convergence no matter where we initialize**
  - Like in the convex case
- To prove this, we need a **global understanding of the objective**
  - So it can only apply to a **limited class of problems**
- For many problems, we **know empirically that this doesn't happen**
  - Deep neural networks are an example of this

Other types of results

- Bounds on generalization error
  - Roughly: we can't say it'll converge, but we can say that it won't overfit
- Ruling out "spurious local minima"
  - Minima that **exist in the training loss, but not in the true/test loss**.
- Results that use the Hessian to escape from saddle points
  - By using it to find a descent direction, but rarely enough that it doesn't damage the computational efficiency

Deep Learning as Non-Convex Optimization

Or, “what could go wrong with my non-convex learning algorithm?”

Lots of Interesting Problems are Non-Convex

- Including deep neural networks
- Because of this, we almost always **can't prove convergence** or anything like that when we run backpropagation (SGD) on a deep net
- But can we use intuition from PCA and convex optimization to understand **what could go wrong when we run non-convex optimization** on these complicated problems?

What could go wrong?
We could converge to a bad local minimum

- Problem: we converge to a local minimum which is bad for our task
  - Very rare for overparameterized neural networks. **Why?**

- One way to debug: re-run the system with **different initialization**
  - Hopefully it will converge to some other local minimum which might be better

- Another way to debug: **add extra noise to gradient updates**
  - Sometimes called “stochastic gradient Langevin dynamics”
  - Intuition: extra noise pushes us out of the steep potential well

What could go wrong?
We could converge to a saddle point

- Problem: we converge to a saddle point, which is not locally optimal

- Upside: **usually doesn't happen with plain SGD**
  - Because noisy gradients push us away from the saddle point
  - But can happen with more sophisticated SGD-like algorithms

- One way to debug: find the **Hessian** and compute a descent direction

What could go wrong?
We get stuck in a region of low gradient magnitude

- Problem: we converge to a region where the gradient's magnitude is small, and then stay there for a very long time
  - Might not affect asymptotic convergence, but very bad for real systems

- One way to debug: use specialized techniques like batchnorm
  - There are many **methods for preventing this problem for neural nets**

- Another way to debug: design your network so that it doesn't happen
  - Networks using a **RELU activation** designed to avoid this problem
  - Layer/batch normalization and residual connections help here too

What could go wrong?
Due to high curvature, we do huge steps and diverge

- Problem: we go to a region where the gradient's magnitude is very large, and then we make a series of very large steps and diverge
  - Especially bad for real systems using floating point arithmetic

- One way to debug: use adaptive step size
  - Like we did for PCA
  - **Adam** (which we'll discuss on Wednesday) does this sort of thing

- A simple way to debug: just limit the size of the step
  - But this can lead to the low-gradient-magnitude issue

Takeaway

- Non-convex optimization is **hard to write theory about**
- But it's **just as easy to compute SGD on**
  - This is why we're seeing a renaissance of empirical computing
- We can use the techniques we have discussed to **get speedup here too**
- We can **apply intuition from the convex case and from simple problems like PCA** to learn how these techniques work