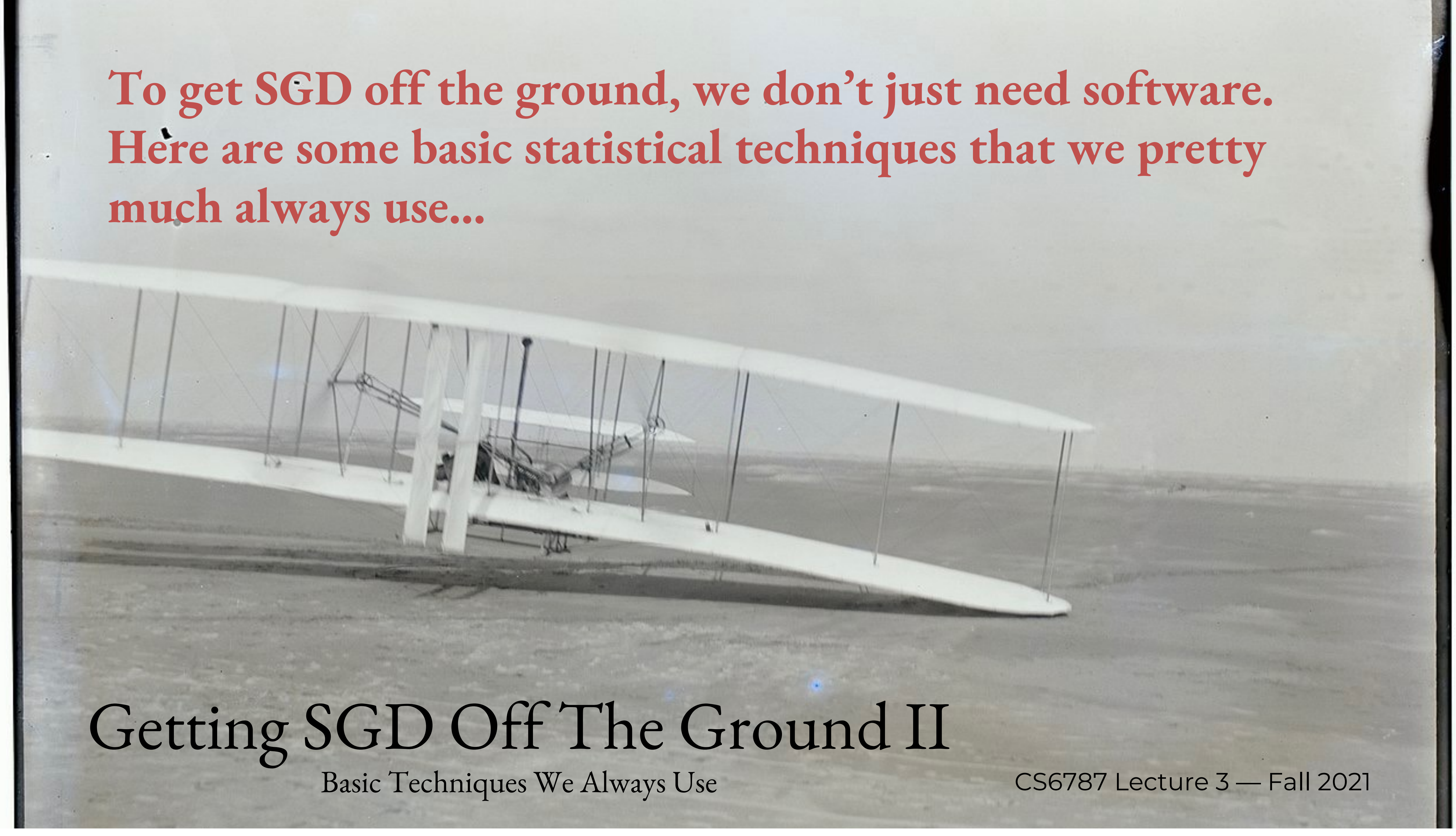


To get SGD off the ground, we don't just need software. Here are some basic statistical techniques that we pretty much always use...

Getting SGD Off The Ground II

Basic Techniques We Always Use

CS6787 Lecture 3 — Fall 2021

Mini-Batching

Gradient Descent vs. SGD

- Gradient descent: **all examples at once**

$$w_{t+1} = w_t - \alpha_t \frac{1}{N} \sum_{i=1}^{N} \nabla f(w_t; x_i)$$

- Stochastic gradient descent: **one example at a time**

$$w_{t+1} = w_t - \alpha_t \nabla f(w_t; x_{i_t})$$

- Is it really **all or nothing**? Can we do something intermediate?

Mini-Batch Stochastic Gradient Descent

- An intermediate approach

$$w_{t+1} = w_t - \alpha_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f(w_t; x_i)$$

where B_t is sampled uniformly from the set of all subsets of {1, … , N} of size b.

  - The b parameter is the **batch size**
  - Typically choose b << N.

- Also called **mini-batch gradient descent**

How does runtime cost of Mini-Batch compare to SGD and Gradient Descent?

- Takes **less time to compute each update** than gradient descent
  - Only needs to sum up b gradients, rather than N

$$w_{t+1} = w_t - \alpha_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f(w_t; x_i)$$

- But takes **more time for each update** than SGD
  - So what's the benefit?

- It's more like gradient descent, so **maybe it converges faster** than SGD?

Advantages of Mini-Batch (reprise)

- Takes **less time to compute each update** than gradient descent
  - Only needs to sum up b gradients, rather than N

$$w_{t+1} = w_t - \alpha_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f(w_t; x_i)$$

- Converges to a **smaller noise ball** than stochastic gradient descent

$$\lim_{T \to \infty} \mathbf{E}\left[\|w_T - w^*\|^2\right] \le \frac{\alpha M}{(2\mu - \alpha\mu^2)b}$$

How to choose the batch size?

- **Mini-batching is not a free win**
  - Naively, compared with SGD, it takes **b** times as much effort to get a **b**-times-as-accurate answer
  - But we could have gotten a **b**-times-as-accurate answer by just running SGD for **b** times as many steps with a step size of $\alpha/\mathbf{b}$.
- But it still makes sense to run it for **systems** and **statistical** reasons
  - Mini-batching exposes more parallelism
  - Mini-batching lets us estimate statistics about the full gradient more accurately
- Another use case for **hyperparameter optimization**

Mini-Batch SGD is very widely used

- Including in basically all neural network training

- **b = 32** is a typical default value for batch size
    - From “Practical Recommendations for Gradient-Based Training of Deep Architectures,” Bengio 2012.

Overfitting, Generalization Error, and Regularization

Minimizing Training Loss is Not our Real Goal

- Training loss looks like

$$h(w) = \frac{1}{N} \sum_{i=1}^{N} f(w; x_i)$$

- What we actually want to minimize is **expected loss on new examples**
  - Drawn from some real-world distribution ϕ

$$\bar{h}(w) = \mathbf{E}_{x \sim \phi} \left[f(w; x) \right]$$

  - Typically, assume the training examples were drawn from this distribution

Overfitting

- Minimizing the training loss **doesn't generally minimize the expected loss** on new examples
  - They are two different objective functions after all
- Difference between the empirical loss on the training set and the expected loss on new examples is called the **generalization error**
- Even a model that has high accuracy on the training set can have terrible performance on new examples
  - Phenomenon is called **overfitting**

Demo

How to address overfitting

- **Many, many techniques** to deal with overfitting
  - Have varying computational costs
- But this is a systems course…so what can we do **with little or no extra computational cost?**
- Notice from the demo that **some loss functions do better than others**
  - Can we **modify our loss function** to prevent overfitting?

Regularization

- Add an extra **regularization term** to the objective function

- Most popular type: **L2 regularization**

$$h(w) = \frac{1}{N} \sum_{i=1}^{N} f(w; x_i) + \sigma^2 \|w\|_2^2 = \frac{1}{N} \sum_{i=1}^{N} f(w; x_i) + \sigma^2 \sum_{k=1}^{d} x_k^2$$

- Also popular: **L1 regularization**

$$h(w) = \frac{1}{N} \sum_{i=1}^{N} f(w; x_i) + \gamma \|w\|_1 = \frac{1}{N} \sum_{i=1}^{N} f(w; x_i) + \gamma \sum_{k=1}^{d} \|x_k\|$$

Benefits of Regularization

- **Cheap to compute**
  - For SGD and L2 regularization, there's just an extra scaling

$$w_{t+1} = (1 - 2\alpha_t \sigma^2) w_t - \alpha_t \nabla f(w_t; x_{i_t})$$

- **L2 regularization makes the objective strongly convex**
  - This makes it easier to get and prove bounds on convergence

- **Helps with overfitting**

Demo

How to choose the regularization parameter?

- One way is to use an independent **validation set** to estimate the test error, and set the regularization parameter manually so that it is high enough to avoid overfitting
  - This is what we saw in the demo
- But doing this naively can be **computationally expensive**
  - Need to re-run learning algorithm many times
- Yet another use case for **hyperparameter optimization**

More general forms of regularization

- **Regularization** is used more generally to describe anything that helps prevent overfitting
  - By biasing learning by making some models more desirable *a priori*

- Many techniques that give throughput improvements also have a regularizing effect
  - Sometimes: a **win-win** of better statistical and hardware performance

Early Stopping

Asymptotically large training sets

- Setting 1: we have a distribution ϕ and we sample a very large (asymptotically infinite) number of points from it, then run stochastic gradient descent on that training set for only **N** iterations.

- Can our algorithm in this setting overfit?
  - **No, because its training set is asymptotically equal to the true distribution.**

- Can we compute this efficiently?
  - **No, because its training set is asymptotically infinitely large**

Consider a second setting

- Setting 1: we have a distribution ϕ and we sample a very large (asymptotically infinite) number of points from it, then run stochastic gradient descent on that training set for only **N** iterations.

- Setting 2: we have a distribution ϕ and we sample **N** points from it, then run stochastic gradient descent using each of these points exactly once.

- What is the difference between the output of SGD in these two settings?
  - **Asymptotically, there's no difference!**
  - So SGD in Setting 2 will also never overfit

Early Stopping

- Motivation: if we only use each training example once for SGD, then we can’t overfit.
- So if we **only use each example a few times**, we probably won’t overfit too much.
- **Early stopping**: just stop running SGD before it converges.

Benefits of Early Stopping

- **Cheap to compute**
  - Literally just does less work
  - It seems like the technique was designed to make systems run faster
- **Helps with overfitting**

Another class of technique: Acceleration and Momentum

How does the step size affect convergence?

- Let's go back to gradient descent

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

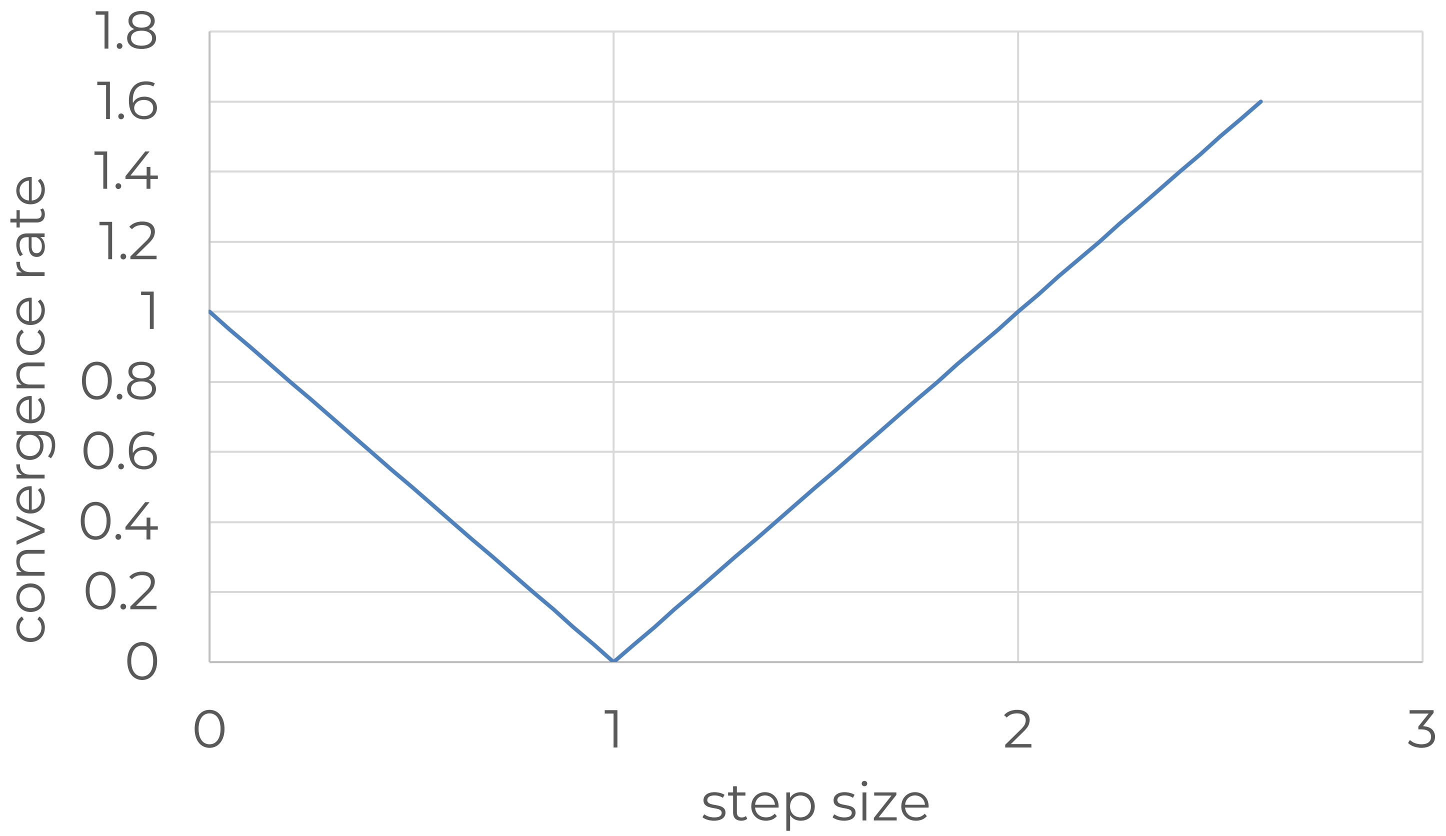
- Simplest possible case: a quadratic function

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

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

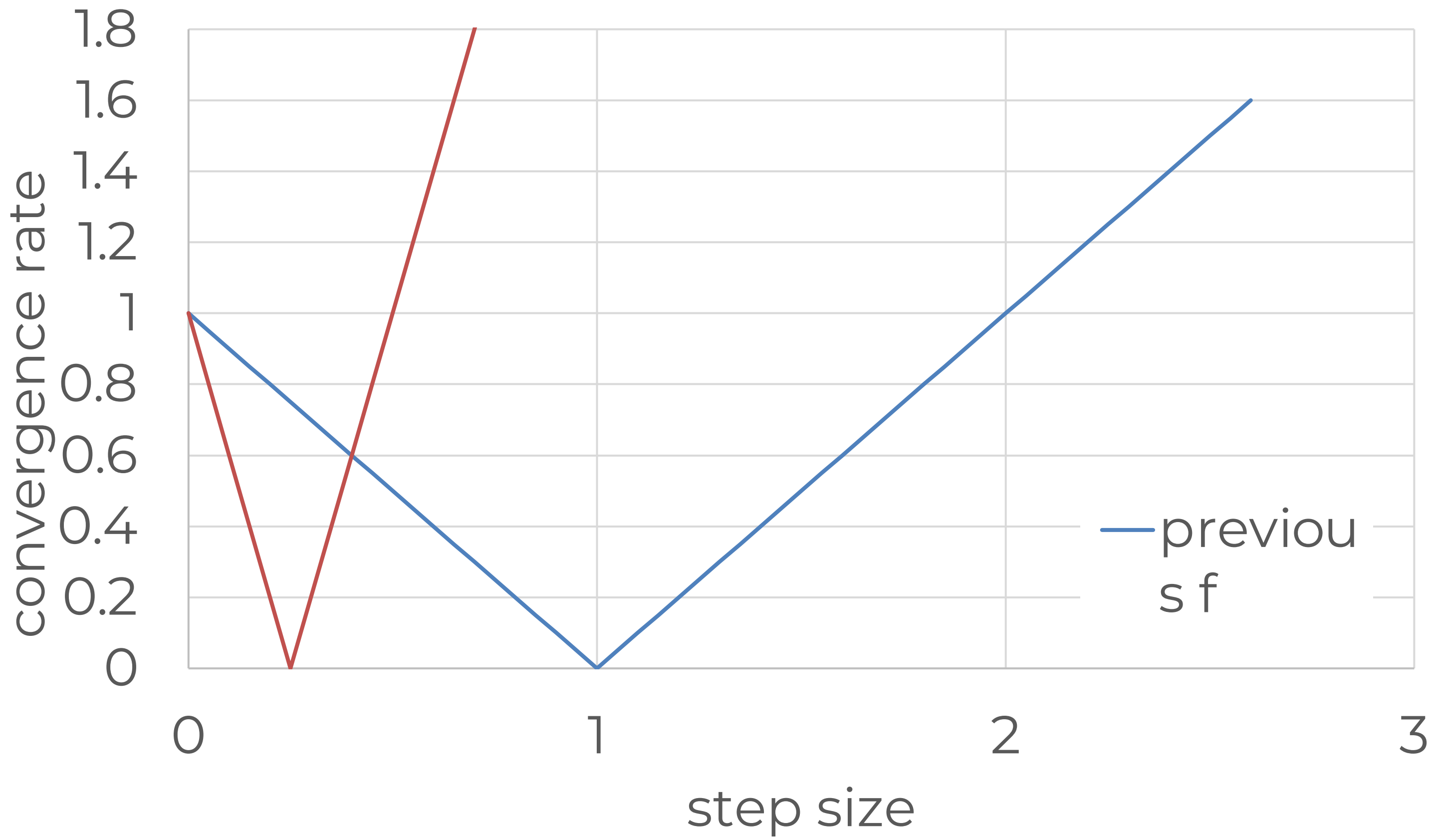
Step size vs. convergence: graphically

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



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:

$$\begin{aligned} \begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} &= \begin{bmatrix} x_t \\ y_t \end{bmatrix} - \alpha \begin{bmatrix} x_t \\ 4y_t \end{bmatrix} \\ &= \begin{bmatrix} 1-\alpha & 0 \\ 0 & 1-4\alpha \end{bmatrix} \begin{bmatrix} x_t \\ y_t \end{bmatrix} \end{aligned}$$

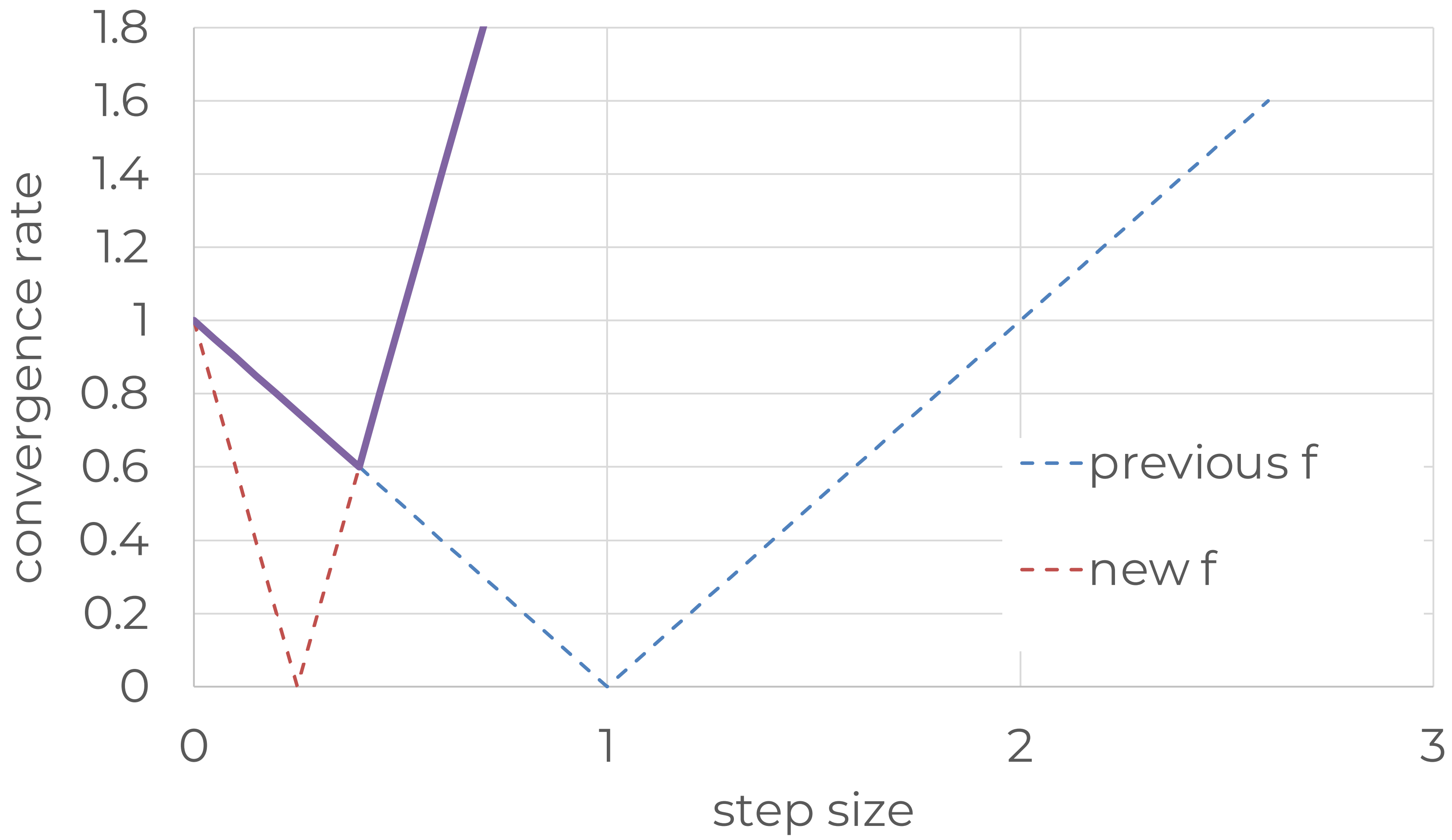
What's the convergence rate?

- 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**

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

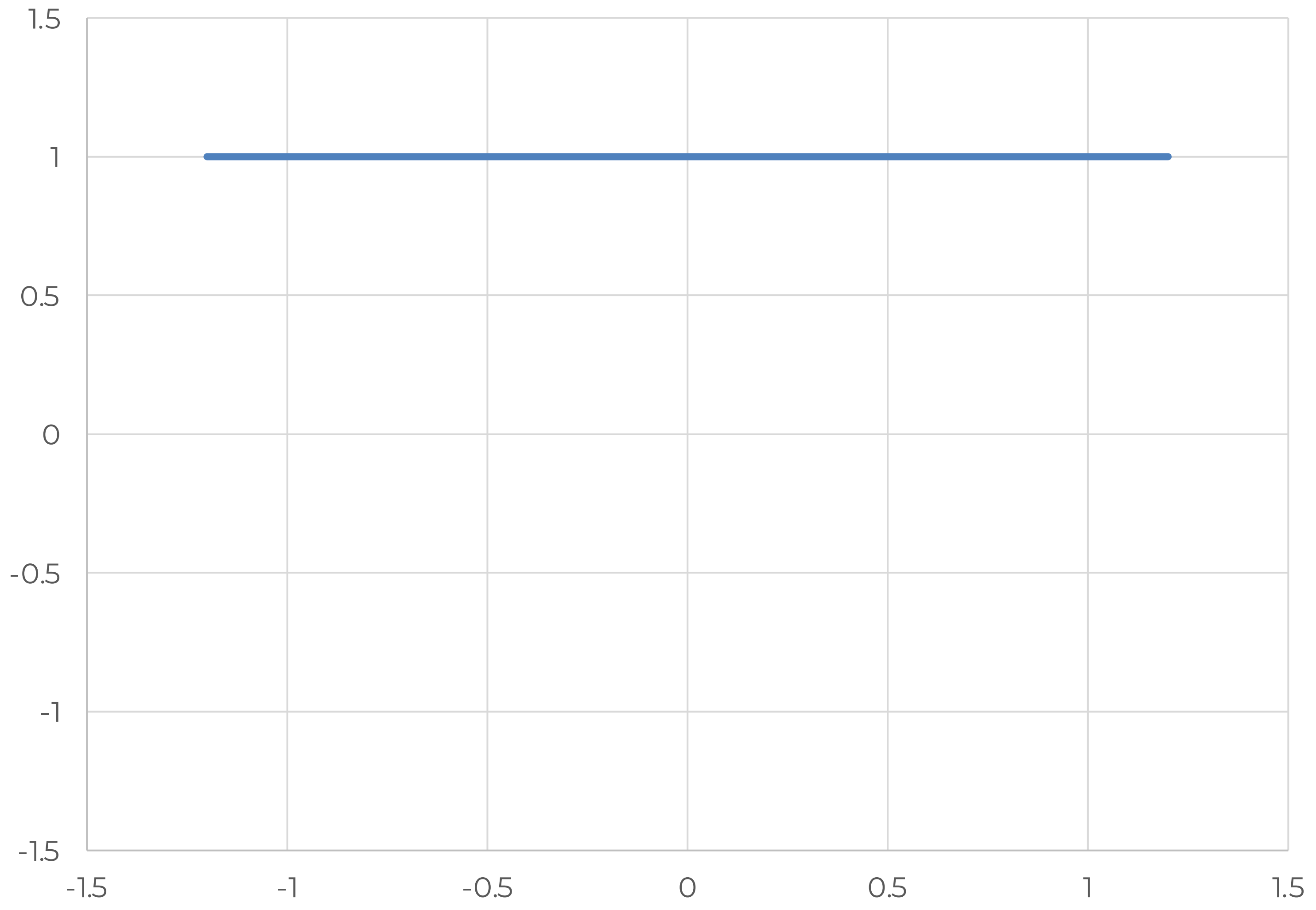
- Standard notation:

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

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

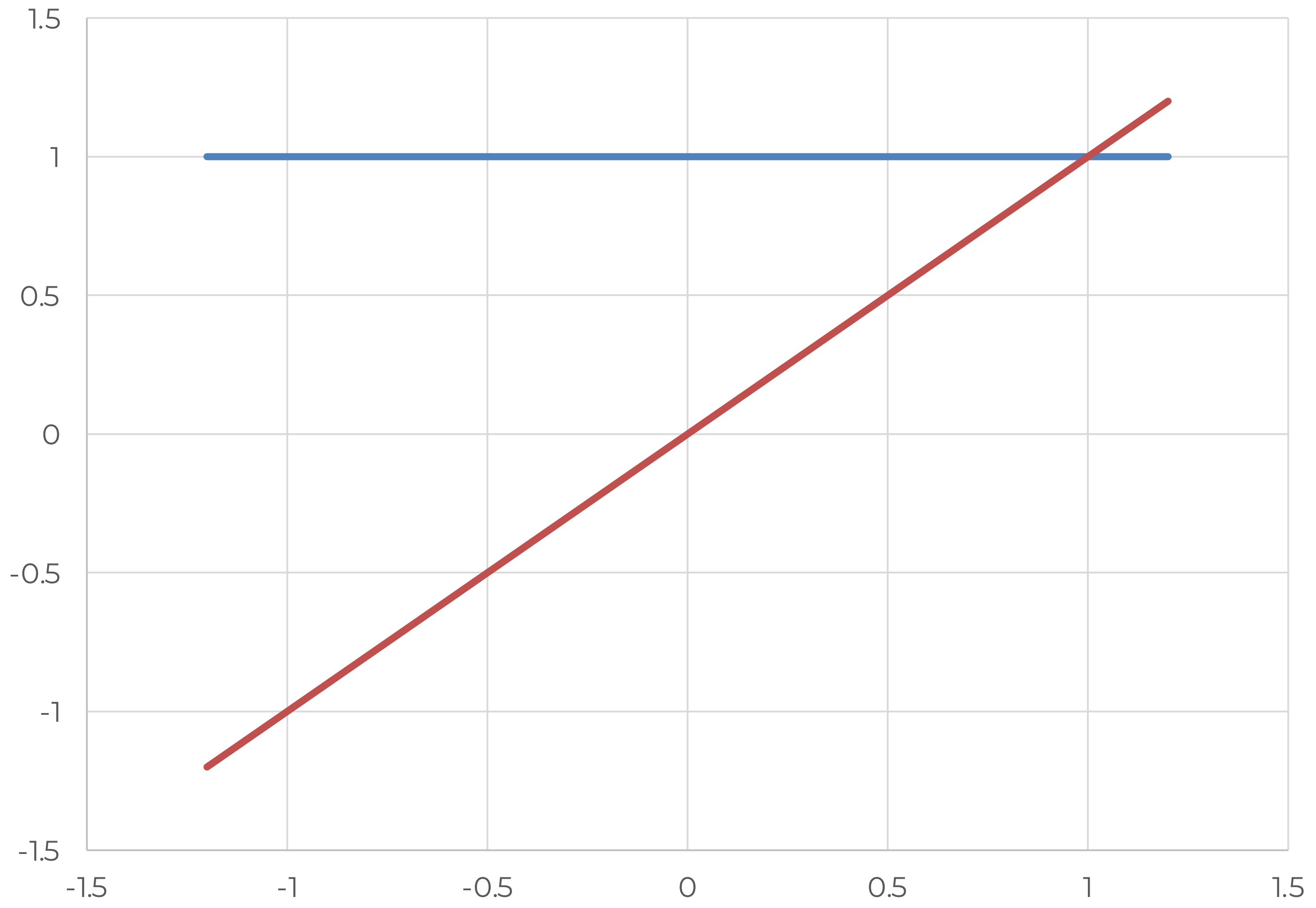
$$-1 \leq u \leq 1 \Rightarrow -1 \leq z_t \leq 1$$

Chebyshev Polynomials



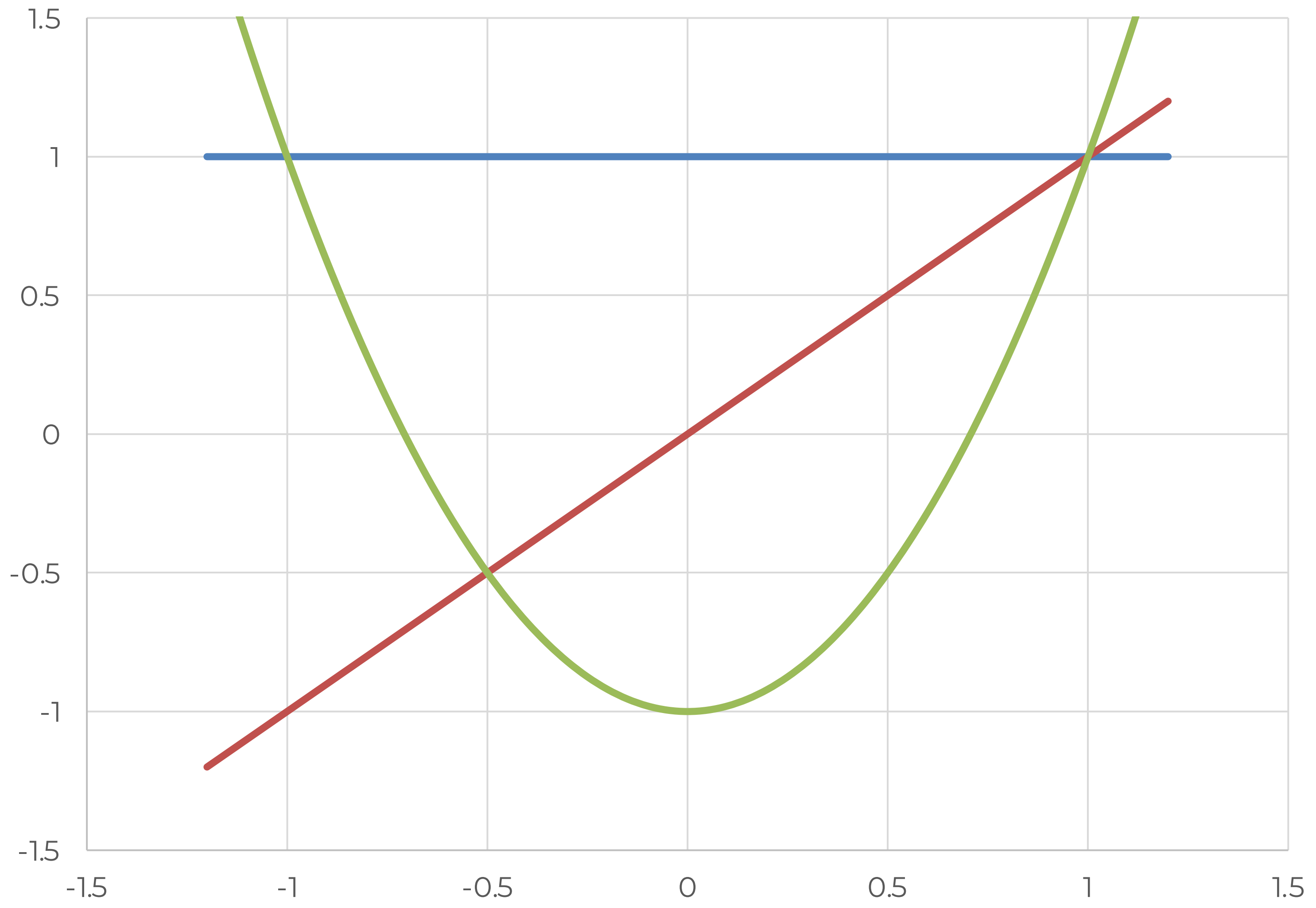
$$T_0(u) = 1$$

Chebyshev Polynomials



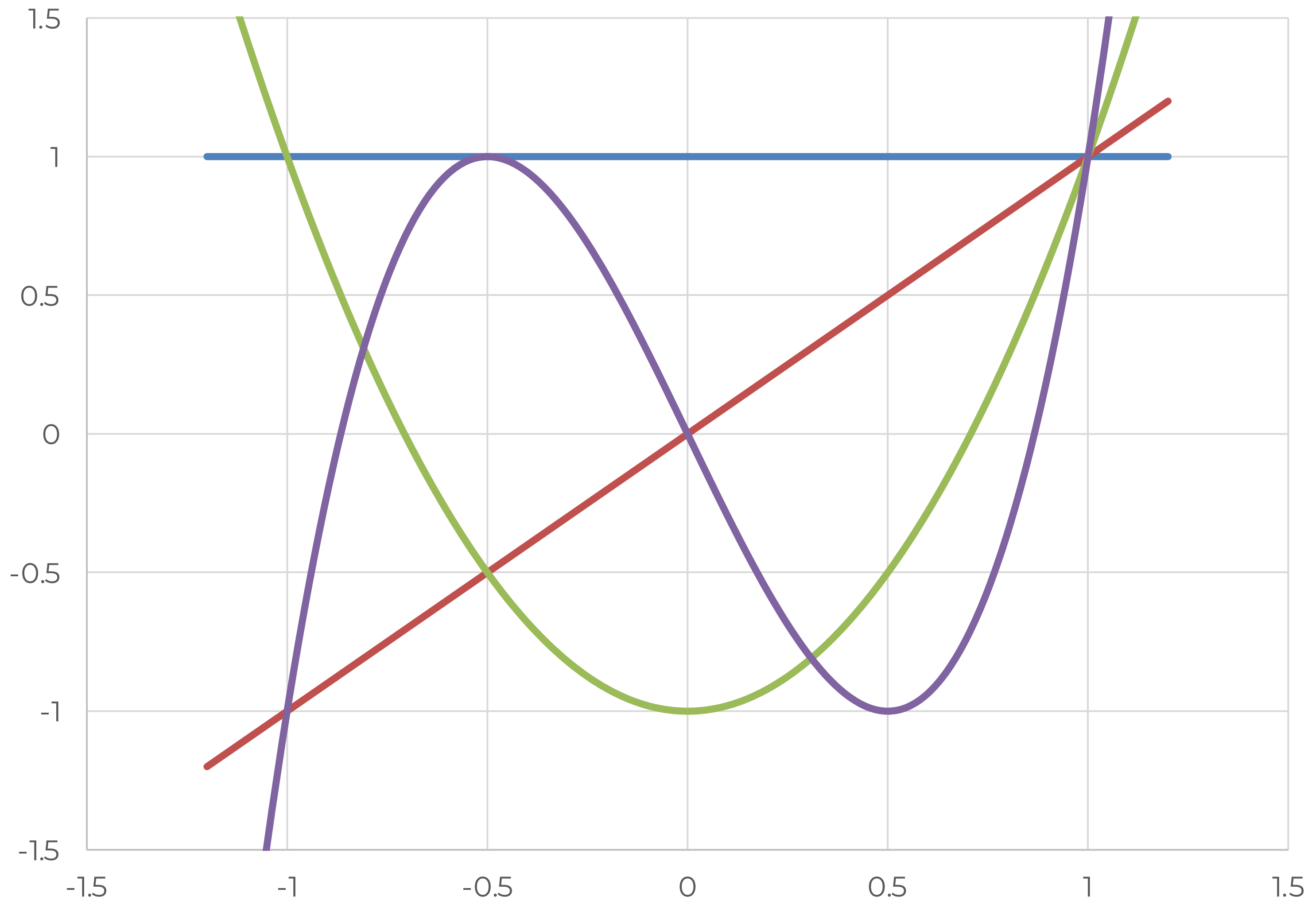
$$T_1(u) = u$$

Chebyshev Polynomials

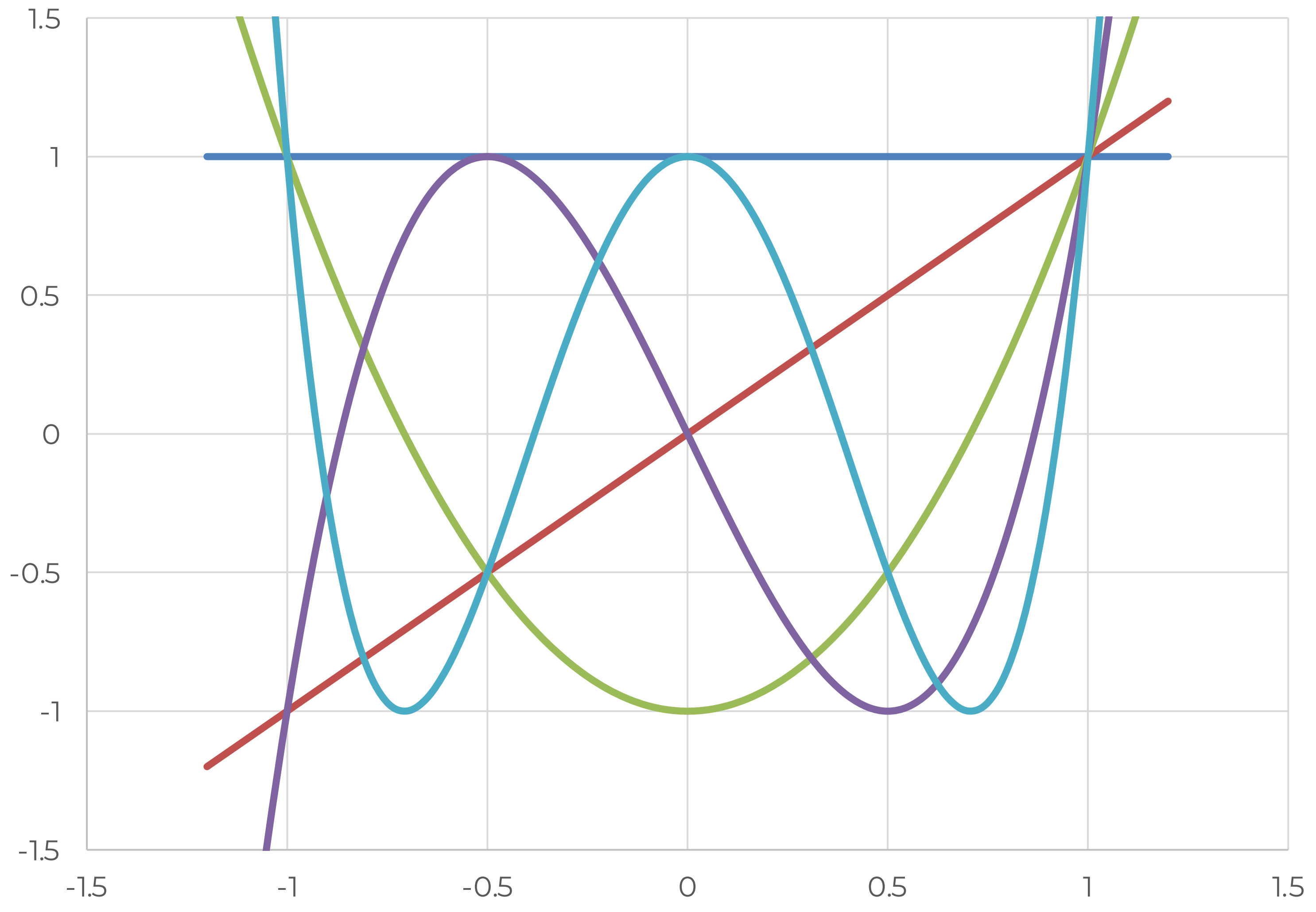


$$T_2(u) = 2u^2 - 1$$

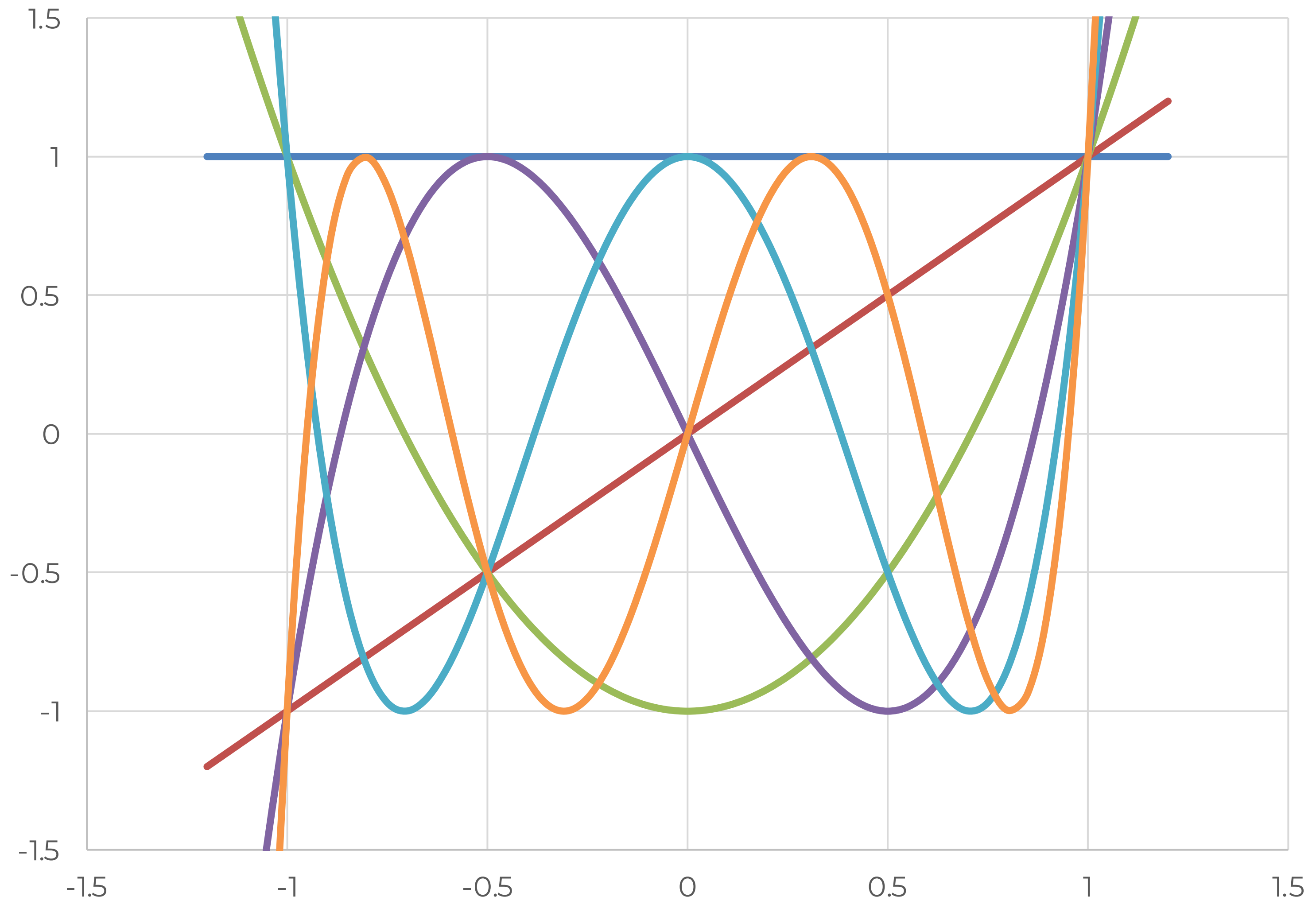
Chebyshev Polynomials



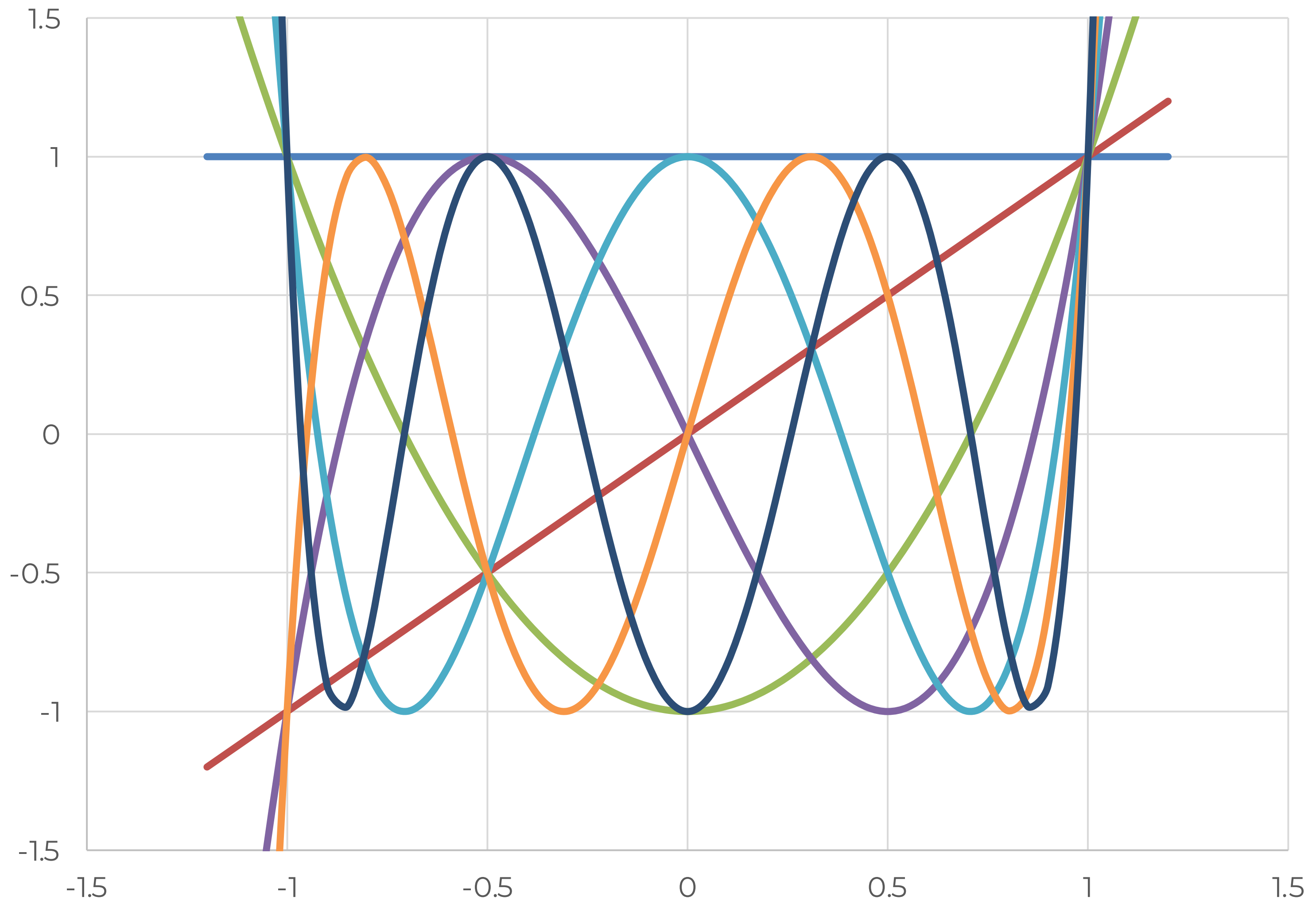
Chebyshev Polynomials



Chebyshev Polynomials



Chebyshev Polynomials



Characterizing momentum (continued)

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

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

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

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

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

Choosing the parameters (continued)

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

$$\begin{aligned}\sqrt{\beta} &= \frac{\kappa+1}{\kappa-1} - \sqrt{\left(\frac{\kappa+1}{\kappa-1}\right)^2 - 1} \\ &= \frac{\kappa+1}{\kappa-1} - \sqrt{\frac{4\kappa}{\kappa^2 - 2\kappa + 1}} \\ &= \frac{\kappa+1}{\kappa-1} - \frac{2\sqrt{\kappa}}{\kappa-1} = \frac{(\sqrt{\kappa}-1)^2}{\kappa-1} = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\end{aligned}$$

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**

Demo

Setting the parameters

- How do we set the momentum in practice for machine learning?
- One method: **hyperparameter optimization**
- Another method: just set $\beta = 0.9$
  - Works across a range of problems
  - Actually quite popular in deep learning

Nesterov momentum

What about more general functions?

- Previous analysis was for quadratics

- Does this work for general convex functions?

- Answer: **not in general**
  - We need to do something slightly different

Nesterov Momentum

- Slightly different rule

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

- Main difference: separate the momentum state from the point that we are calculating the gradient at.

Nesterov Momentum Analysis

- Converges at an accelerated rate **for ANY convex problem**

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

- Optimal assignment of the parameters:

$$\alpha = \frac{1}{\lambda_{\max}}, \beta = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$$

Nesterov Momentum is Also Very Popular

- People use it in practice for deep learning all the time
- Significant speedups in practice

Demo

What about SGD?

- All our above analysis was for **gradient descent**
- But momentum still produces empirical improvements when used with stochastic gradient descent
- And we'll see how in one of the papers we're reading on **Monday!**