# Machine Learning Frameworks Continued!

CS6787 Lecture 3 — Fall 2020

## Common Features of Machine Learning Frameworks

## What do ML frameworks support?

- **Basic tensor operations**
  - Provides the low-level math behind all the algorithms
  - Including support for running them on hardware such as GPUs
- **Automatic differentiation**
  - Used to make it easy to run backprop on any model
- Simple-to-use composable implementations of **systems techniques**
  - Including most of the techniques we will discuss in the remainder of this course

## Tensors

- CS way to think about it: a tensor is a **multidimensional array**

- Math way to think about it: a **tensor is a multilinear map**

$$T : \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \times \cdots \times \mathbb{R}^{d_n} \to \mathbb{R}$$

$T(x_1, x_2, \ldots, x_n)$ is linear in each $x_i$, with other inputs fixed.

  - Here the number **n** is called the *order* of the tensor
  - For example, a **matrix is just a 2nd-order tensor**

# Examples of Tensors in Machine Learning

- The **CIFAR10 dataset** consists of 60000 32x32 color images
  - We can write the training set as a tensor

$$T_{\text{CIFAR10}} \in \mathbb{R}^{32 \times 32 \times 3 \times 60000}$$

- **Gradients** for deep learning can also be tensors
  - Example: fully-connected layer with 100 input and 100 output neurons, and mini-batch size b=32

$$G \in \mathbb{R}^{100 \times 100 \times 32}$$

# Common Operations on Tensors

- **Elementwise operations** — looks like vector sum
  - Example: Hadamard product

$$(A \circ B)_{i_1,i_2,\ldots,i_n} = A_{i_1,i_2,\ldots,i_n} B_{i_1,i_2,\ldots,i_n}$$

- **Broadcast operations** — expand along one or more dimensions
  - Example: $A \in \mathbb{R}^{11\times 1}$, $B \in \mathbb{R}^{11\times 5}$ , then with broadcasting

$$(A + B)_{i,j} = A_{i,1} + B_{i,j}$$

  - Extreme version of this is the **tensor product**

- **Matrix-multiply-like operations** — sum or reduce along a dimension
  - Also called **tensor contraction**

## Broadcasting makes ML easy to write

- Here's how easy it is to write the loss and gradient for logistic regression
  - Doesn't even need to include a for-loop
  - This code is in **Julia** but it would be similar in other languages

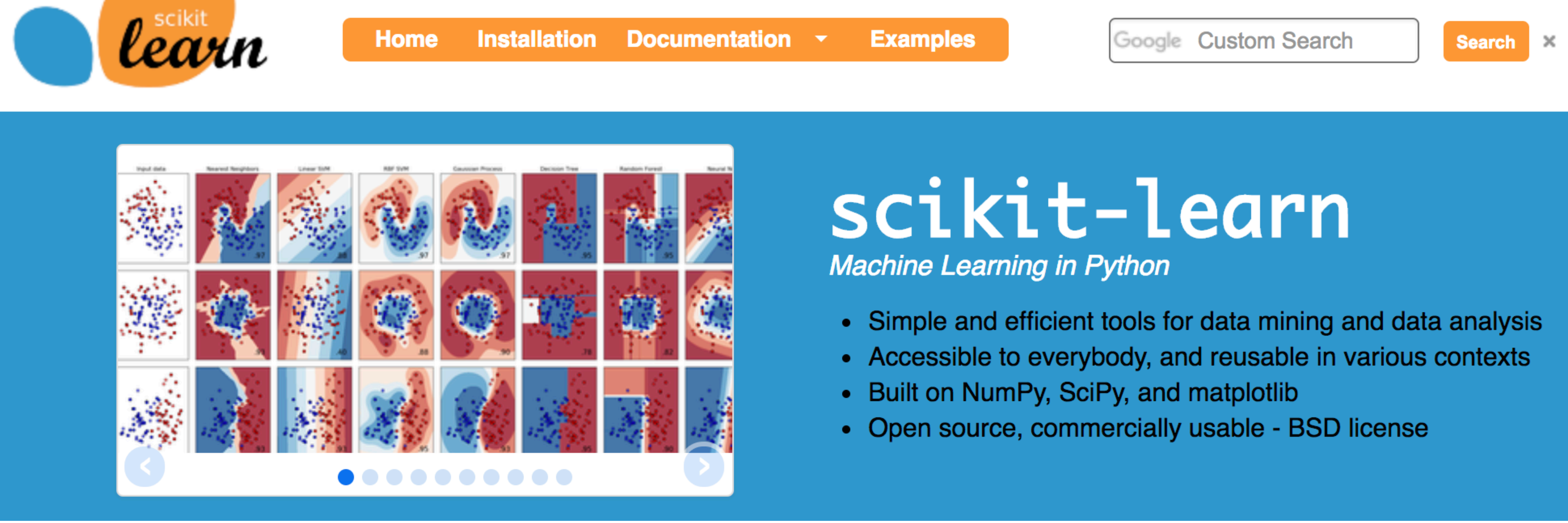
```julia
function logreg_loss(w, X, Y)
    return sum(log(1 + exp(-Y .* (X * w))));
end

function logreg_grad(w, X, Y)
    return -X' * (Y ./ (1 + exp(Y .* (X * w))));
end
```

# Tensors: a systems perspective

- **Loads of data parallelism**
  - Tensors are in some sense the structural embodiment of data parallelism
  - Multiple dimensions → **not always obvious** which one best to parallelize over
- **Predictable linear memory access patterns**
  - Great for locality
- **Many different ways** to organize the computation
  - Creates opportunities for frameworks to **automatically optimize**

## General Machine Learning Frameworks



scikit-learn

*Machine Learning in Python*

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

- **scikit-learn**
  - A broad, full-featured toolbox of machine learning and data analysis tools
  - In **Python**
  - Features support for classification, regression, clustering, dimensionality reduction: including SVM, logistic regression, *k*-Means, PCA

## NumPy and SciPy.org

Sponsored By ENTHOUGHT

- **NumPy**
  - Adds large multi-dimensional array and matrix types (tensors) to python
  - Supports basic numerical operations on tensors, on the CPU

- **SciPy**
  - Builds on NumPy and adds tools for scientific computing
  - Supports optimization, data structures, statistics, symbolic computing, etc.
  - Also has an interactive interface (Jupyter) and a neat plotting tool (matplotlib)

- **Great ecosystem for prototyping systems**

## Theano

- Machine learning library for **python**
  - Created by the University of Montreal
- Supports **tight integration with NumPy**
- But also supports **CPU and GPU integration**
  - Making it very fast for a lot of applications
- **Development has ceased** because of competition from other libraries

## Julia and MATLAB

- **Julia**
  - Relatively new language (8 years old) with growing community
  - Natively **supports numerical computing** and all the tensor ops
  - **Syntax is nicer than Python**, and it's often **faster**
  - Has **Flux**, a library for machine learning that supports backpropagation
  - But **less support from the community** and **less library support**

- **MATLAB**
  - The decades-old standard for numerical computing
  - **Supports tensor computation**, and some people use it for ML
  - But has less attention from the community because it's **proprietary**

## General Big Data Processing Frameworks

# The original: MapReduce/Hadoop

- Invented by Google to handle distributed processing
- People started to use it for **distributed machine learning**
  - And people still use it today
- But it's mostly been **supplanted by other libraries**
  - And for good reason
  - Hadoop does a **lot of disk writes** in order to be robust against failure of individual machines — not necessary for machine learning applications

## Apache Spark

- Open-source **cluster computing framework**
  - Built in **Scala**, and can also embed in **Python**
- Developed by Berkeley AMP lab
  - Now spun off into a company: **DataBricks**
- The original pitch: **100x faster** than Hadoop/MapReduce
- Architecture based on resilient distributed datasets (**RDDs**)
  - Essentially a **distributed fault-tolerant data-parallel array**

## Spark MLLib

- **Scalable machine learning library** built on top of Spark

- Supports most of the same algorithms scikit-learn supports
  - Classification, regression, decision trees, clustering, topic modeling
  - Not primarily a deep learning library

- Major benefit: **interaction with other processing in Spark**
  - SparkSQL to handle database-like computation
  - GraphX to handle graph-like computation

## Apache Mahout

- **Backend-independent** programming environment for machine learning
  - Can support Spark as a backend
  - But also supports basic MapReduce/Hadoop
- Focuses mostly on collaborative filtering, clustering, and classification
  - Similarly to MLLib and scikit-learn
- Also not very deep learning focused

## Many more here

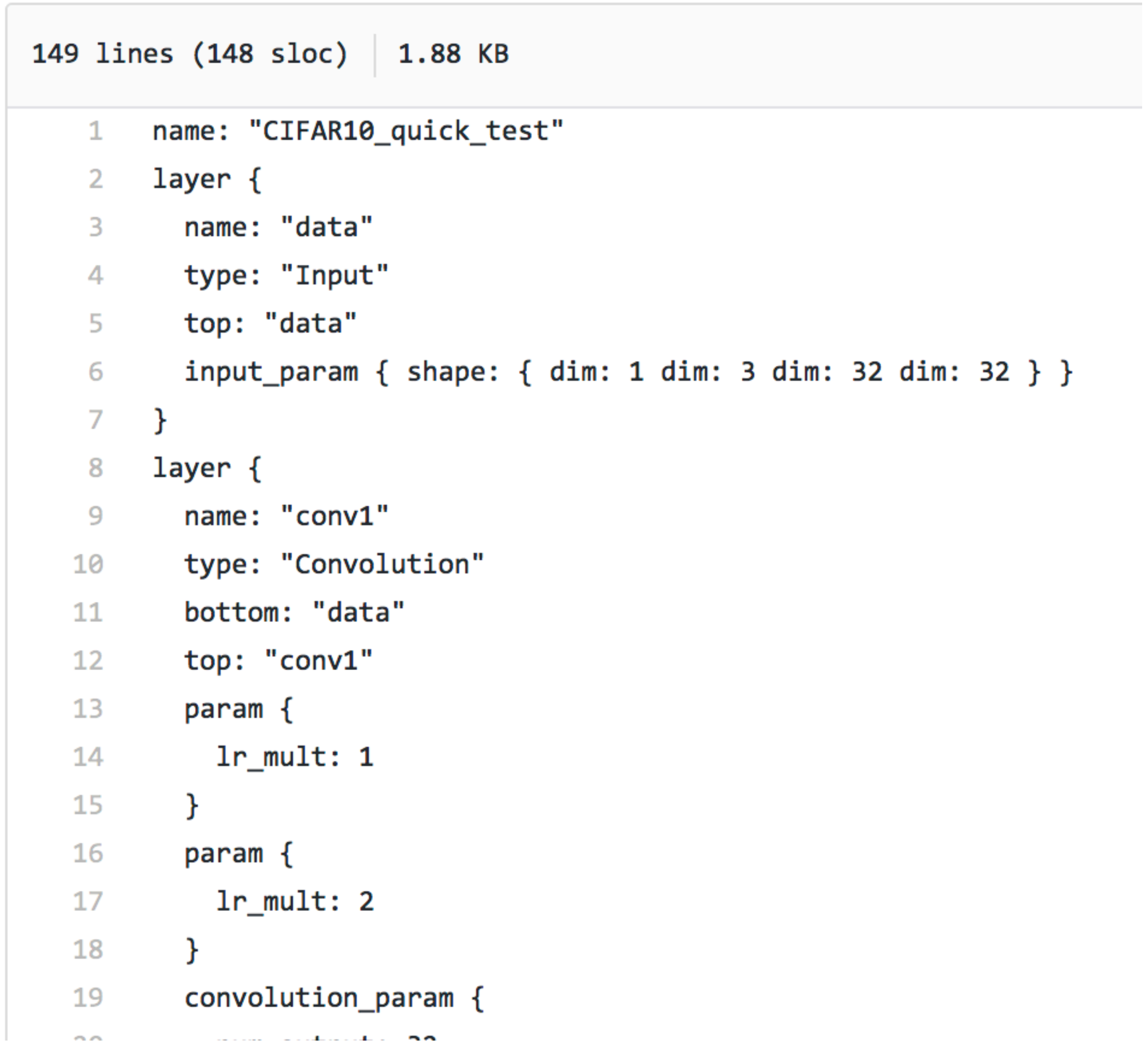
- Lots of very good frameworks for large-scale parallel programming **don't end up becoming popular**
- Takeaway: **important to release code people can use easily**
  - And capture a group of users who can then help develop the framework

## Deep Learning Frameworks

# Caffe

- Deep learning framework
  - Developed by Berkeley AI research
- **Declarative expressions** for describing network architecture
- **Fast** — runs on CPUs and GPUs out of the box
  - And supports a lot of optimization techniques
- **Huge community** of users both in academia and industry

# Caffe code example



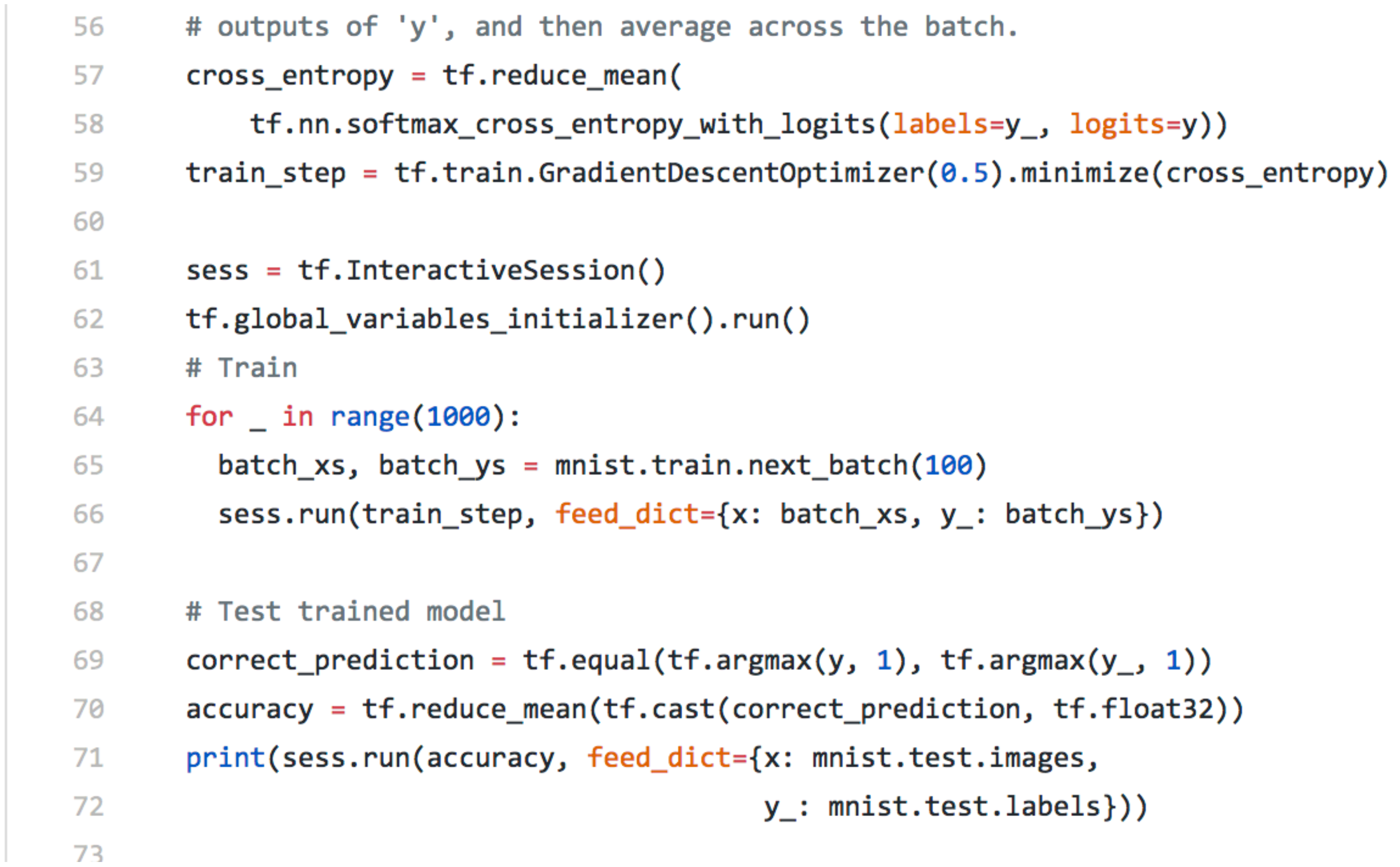
149 lines (148 sloc) | 1.88 KB

```
name: "CIFAR10_quick_test"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 1 dim: 3 dim: 32 dim: 32 } }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
```

## TensorFlow

- End-to-end **deep learning system**
  - Developed by Google Brain
- API primarily in **Python**
  - With support for other languages
- Architecture: build up a computation graph in Python
  - Then the **framework schedules it automatically** on the available resources
  - Although recently TensorFlow has announced an **eager version**
- **Super-popular**, used to be the de facto standard for deep learning

# TensorFlow code example



```python
# outputs of 'y', and then average across the batch.
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
# Train
for _ in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

# Test trained model
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                    y_: mnist.test.labels}))
```

## PYTORCH

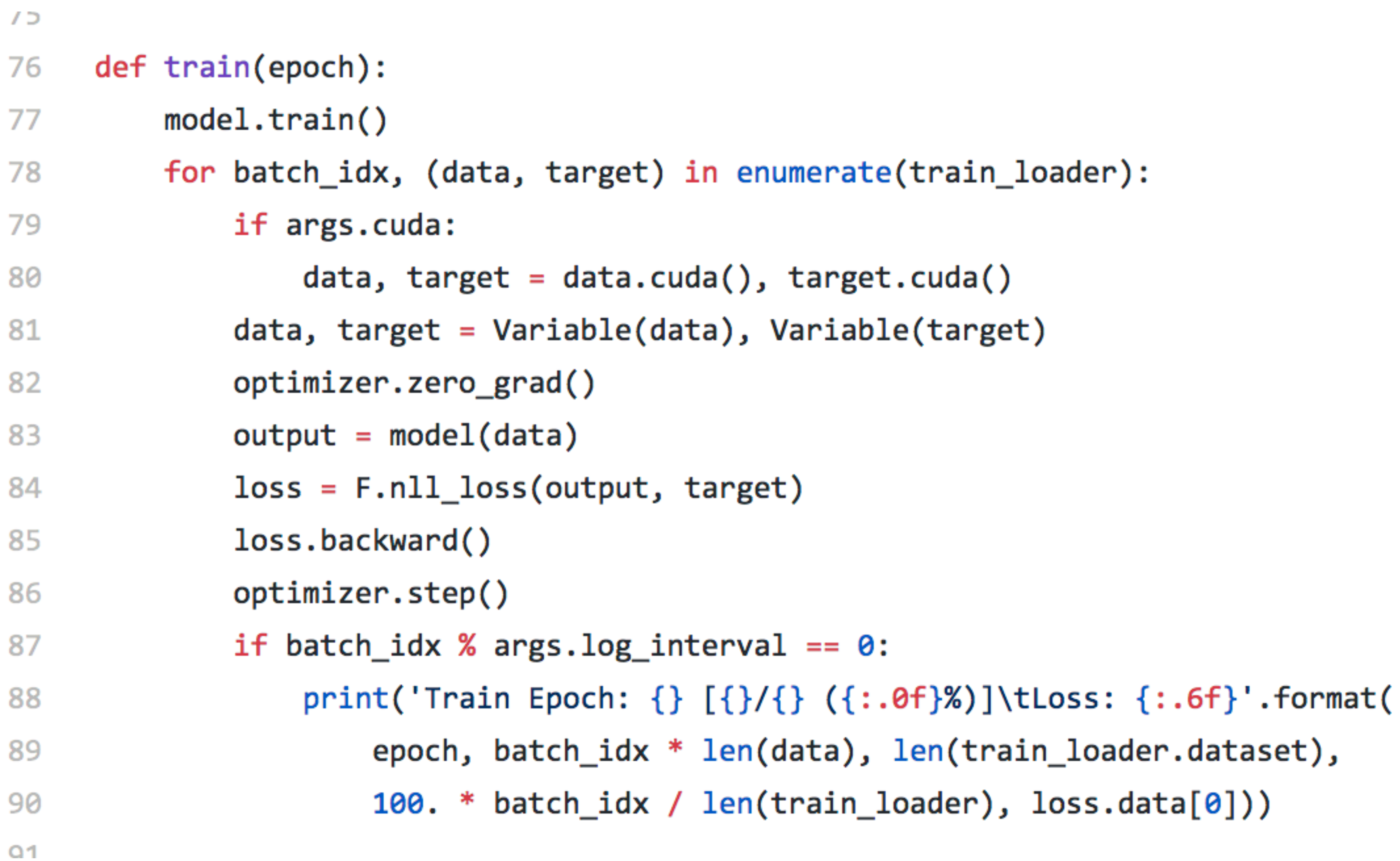
- **Python** package that focuses on
  - **Tensor computation** (like numpy) with strong **GPU acceleration**
  - **Deep Neural Networks** built on a tape-based autograd system

- **Eager computation** out-of-the-box

- Uses a technique called **reverse-mode auto-differentiation**
  - Allows users to change network behavior arbitrarily with zero lag or overhead
  - Fastest implementation of this method

- PyTorch is **very popular — the most common in academia right now**

# PyTorch example



```python
def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if args.cuda:
            data, target = data.cuda(), target.cuda()
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
```

## mxnet

- Deep learning library from **Apache**.

- Scalable C++ backend
  - Support for many frontend languages, including **Python**, Scala, C++, R, Perl…

- Focus on **scalability to multiple GPUs**
  - Sometimes performs better than competing approaches.

# MXNet Example

```
# define network
net = nn.Sequential()
with net.name_scope():
    net.add(nn.Dense(128, activation='relu'))
    net.add(nn.Dense(64, activation='relu'))
    net.add(nn.Dense(10))
```

(from MXNet MNIST tutorial)

```
epoch = 10
# Use Accuracy as the evaluation metric.
metric = mx.metric.Accuracy()
softmax_cross_entropy_loss = gluon.loss.SoftmaxCrossEntropyLoss()
for i in range(epoch):
    # Reset the train data iterator.
    train_data.reset()
    # Loop over the train data iterator.
    for batch in train_data:
        # Splits train data into multiple slices along batch_axis
        # and copy each slice into a context.
```

## …and many other frameworks for ML

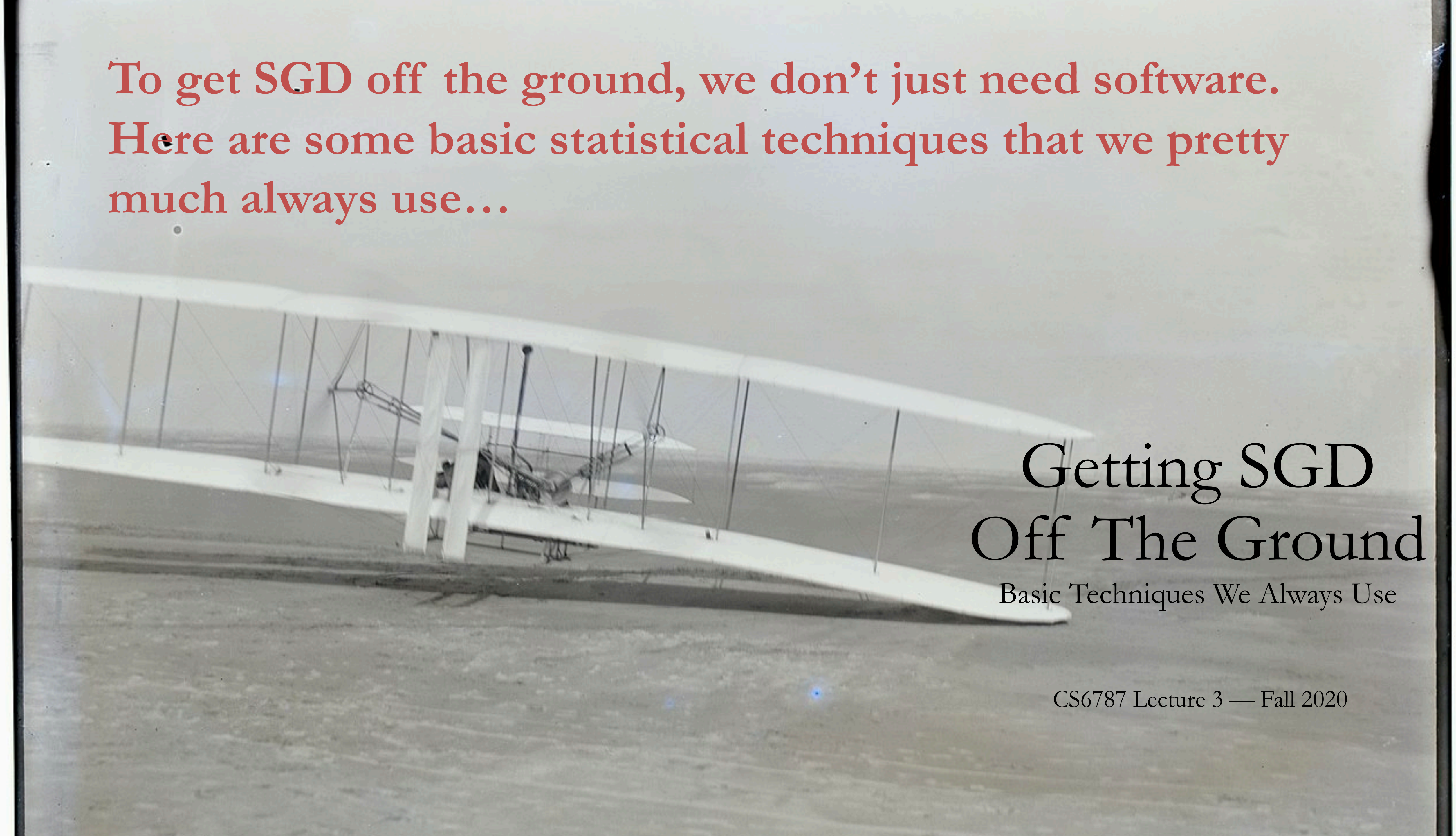
- Theano
- ONNX
- New frameworks will continue to be developed!

## ML Frameworks: Conclusion

- These frameworks are important to know about because they **give you the tools** you can use to build ML software.

- Most of you will be using an ML framework to do your course project.

- In previous years, I've taught ML frameworks at the end of CS6787, because they rely so much on the other techniques we discuss…
  - …but in the interest of giving you access to the tools as soon as possible, I've moved this content up this year.
  - We'll still come back to ML frameworks later in the semester and talk about their systems aspects in more detail.



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

## Basic Techniques We Always Use

CS6787 Lecture 3 — Fall 2020

## 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 $\{1, \ldots, N\}$ 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?

## Mini-Batch SGD Converges

- Start by breaking up the update rule into expected update and noise

$$\begin{aligned} w_{t+1} - w^* = w_t - w^* - \alpha_t \left( \nabla h(w_t) - \nabla h(w^*) \right) \\ - \alpha_t \frac{1}{|B_t|} \sum_{i \in B_t} \left( \nabla f(w_t; x_i) - \nabla h(w_t) \right) \end{aligned}$$

- Second moment bound

$$\begin{aligned} \mathbf{E}\left[ \| w_{t+1} - w^* \|^2 \right] = \mathbf{E}\left[ \| w_t - w^* - \alpha_t \left( \nabla h(w_t) - \nabla h(w^*) \right) \|^2 \right] \\ + \alpha_t^2 \mathbf{E}\left[ \left\| \frac{1}{|B_t|} \sum_{i \in B_t} \left( \nabla f(w_t; x_i) - \nabla h(w_t) \right) \right\|^2 \right] \end{aligned}$$

# Mini-Batch SGD Converges (continued)

Let $\Delta_i = \nabla f(w_t; x_i) - \nabla h(w_t)$

$$\mathbf{E}\left[\left\|\frac{1}{|B_t|}\sum_{i \in B_t}\left(\nabla f(w_t; x_i) - \nabla h(w_t)\right)\right\|^2\right]$$

$$= \mathbf{E}\left[\left\|\frac{1}{|B_t|}\sum_{i \in B_t}\Delta_i\right\|^2\right]$$

## Mini-Batch SGD Converges (continued)

- Because we sampled B uniformly at random, for **i ≠ j**

$$\mathbf{E}\left[\beta_i \beta_j\right] = \mathbf{P}\left(i \in B \wedge j \in B\right) = \mathbf{P}\left(i \in B\right)\mathbf{P}\left(j \in B | i \in B\right) = \frac{b}{N} \cdot \frac{b-1}{N-1}$$

$$\mathbf{E}\left[\beta_i^2\right] = \mathbf{P}\left(i \in B\right) = \frac{b}{N}$$

- So we can bound our square error term as

$$\mathbf{E}\left[\left\| \frac{1}{|B_t|} \sum_{i \in B_t} \left(\nabla f(w_t; x_i) - \nabla h(w_t)\right) \right\|^2\right] = \frac{1}{|B_t|^2} \mathbf{E}\left[\sum_{i=1}^{N} \sum_{j=1}^{N} \beta_i \beta_j \Delta_i^T \Delta_j\right]$$

$$= \frac{1}{b^2} \mathbf{E}\left[\sum_{i \neq j} \frac{b(b-1)}{N(N-1)} \Delta_i^T \Delta_j + \sum_{i=1}^{N} \frac{b}{N} \|\Delta_i\|^2\right]$$

# Mini-Batch SGD Converges (continued)

$$\mathbf{E}\left[\left\|\frac{1}{|B_t|}\sum_{i\in B_t}\left(\nabla f(w_t;x_i)-\nabla h(w_t)\right)\right\|^2\right]=\frac{1}{bN}\mathbf{E}\left[\frac{b-1}{N-1}\sum_{i\neq j}\Delta_i^T\Delta_j+\sum_{i=1}^{N}\|\Delta_i\|^2\right]$$

## Mini-Batch SGD Converges (continued)

$$\mathbf{E}\left[\left\|\frac{1}{|B_t|}\sum_{i\in B_t}(\nabla f(w_t;x_i)-\nabla h(w_t))\right\|^2\right]=\frac{N-b}{b(N-1)}\mathbf{E}\left[\frac{1}{N}\sum_{i=1}^{N}\|\Delta_i\|^2\right]$$

- Compared with SGD, **squared error term decreased by a factor of b**

# Mini-Batch SGD Converges (continued)

- Recall that SGD converged to a noise ball of size

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

- Since mini-batching decreases error term by a factor of **b**, it will have

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

  - **Noise ball smaller** by the same factor!

## 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 $\phi$

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

  - 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 $\phi$ 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 $\phi$ 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 $\phi$ 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**

# Our First Hyperparameters: Mini-batching, Regularization, and Momentum

CS6787 Lecture 3 — Fall 2020

## Where we left off

- We talked about how we can compute gradients easily and efficiently using ML frameworks.

- We talked about **overfitting**, which can negatively impact generalization from the training set to the test set.

- We saw in the paper we read that **early stopping** is one way that overfitting can be prevented.
  - It's an important technique, but I won't cover it in today's lecture since we already covered it in the paper.

## 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**
  - E.g. the linear loss function did better than the polynomial loss function
  - 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

## Another class of technique: Acceleration and Momentum

## How does the step size affect convergence?

- Let's go back to gradient descent

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

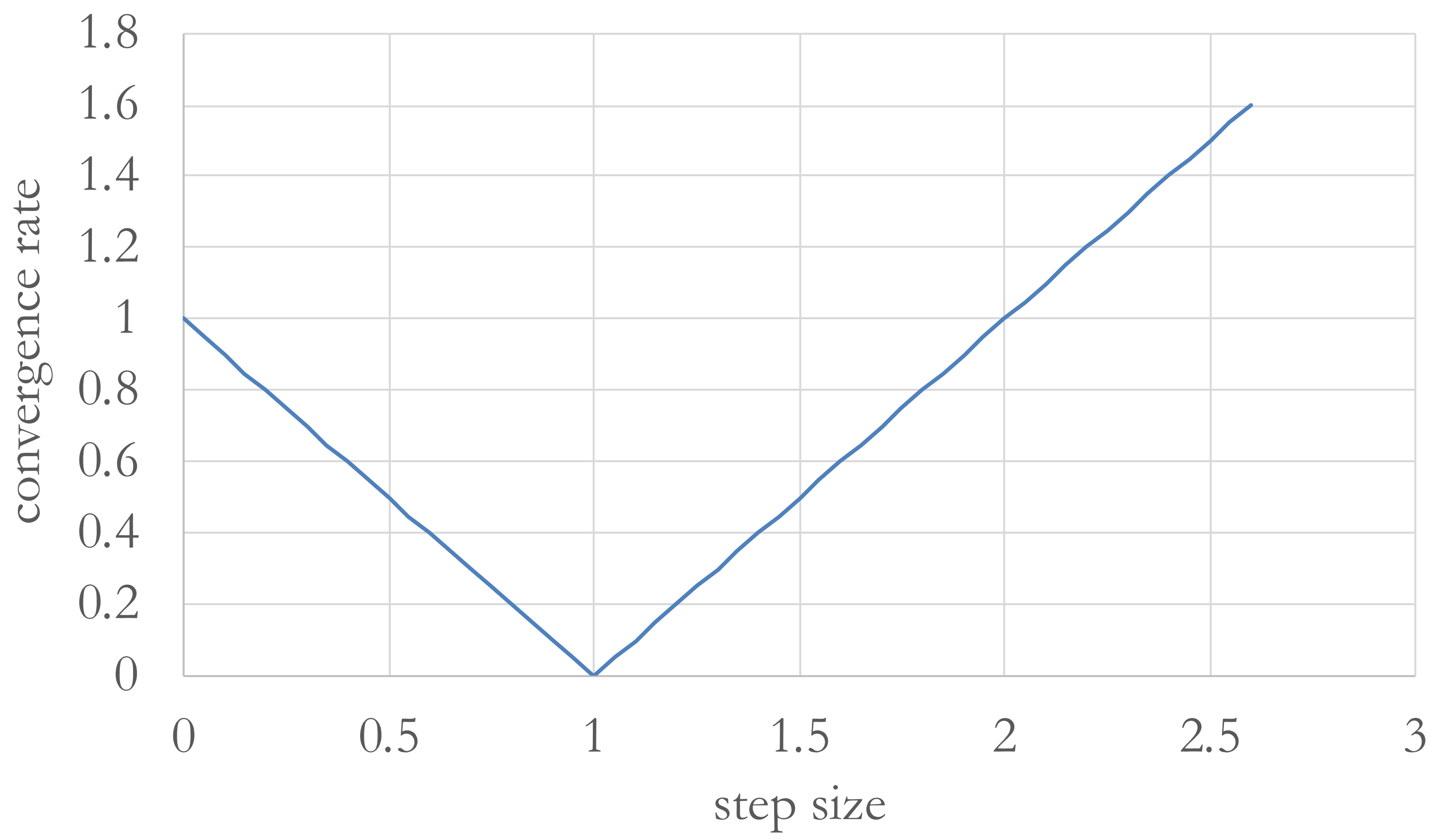
- Simplest possible case: a quadratic function

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

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

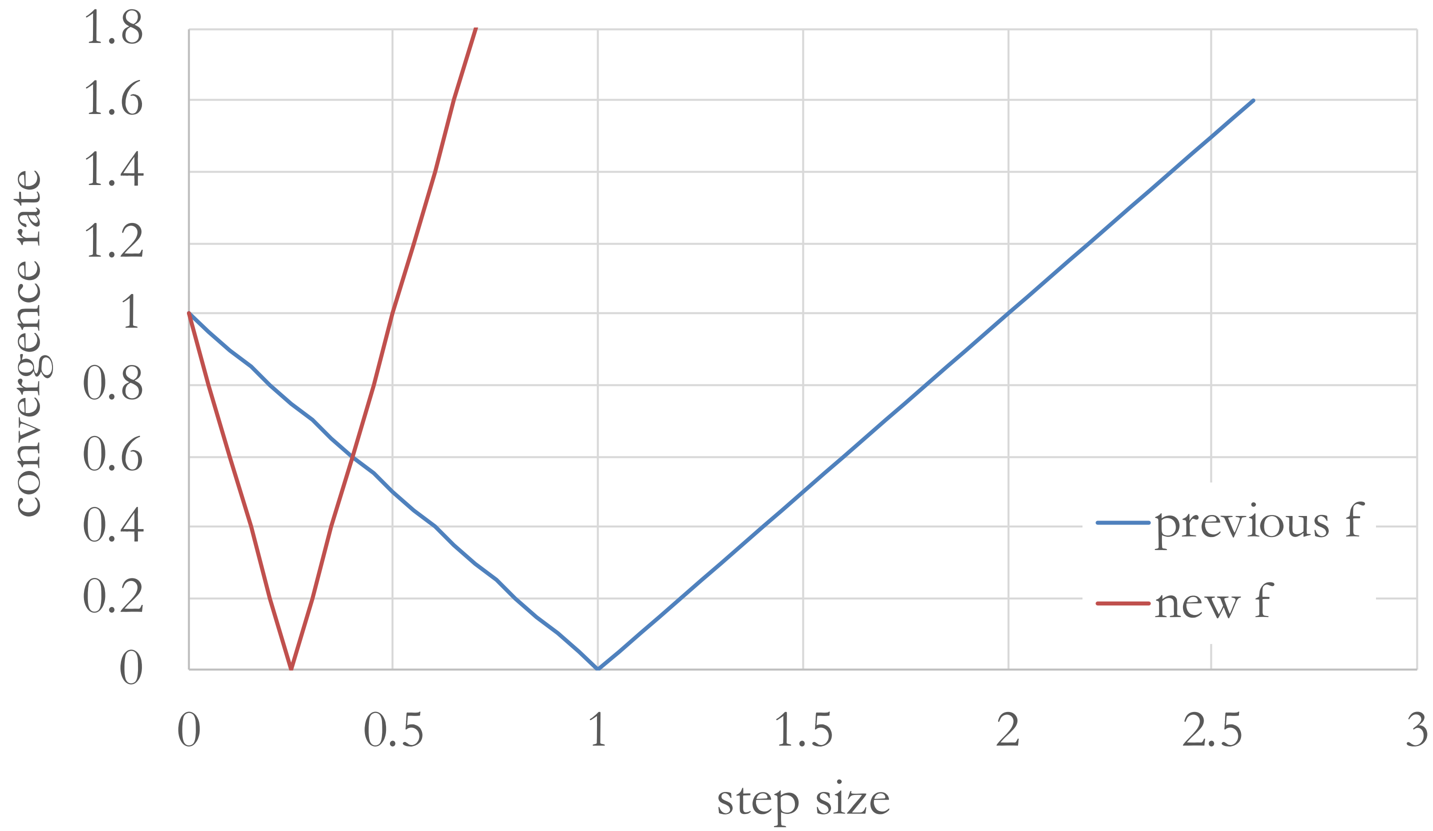
## Step size vs. convergence: graphically

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



# 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 $\rightarrow$ 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:

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

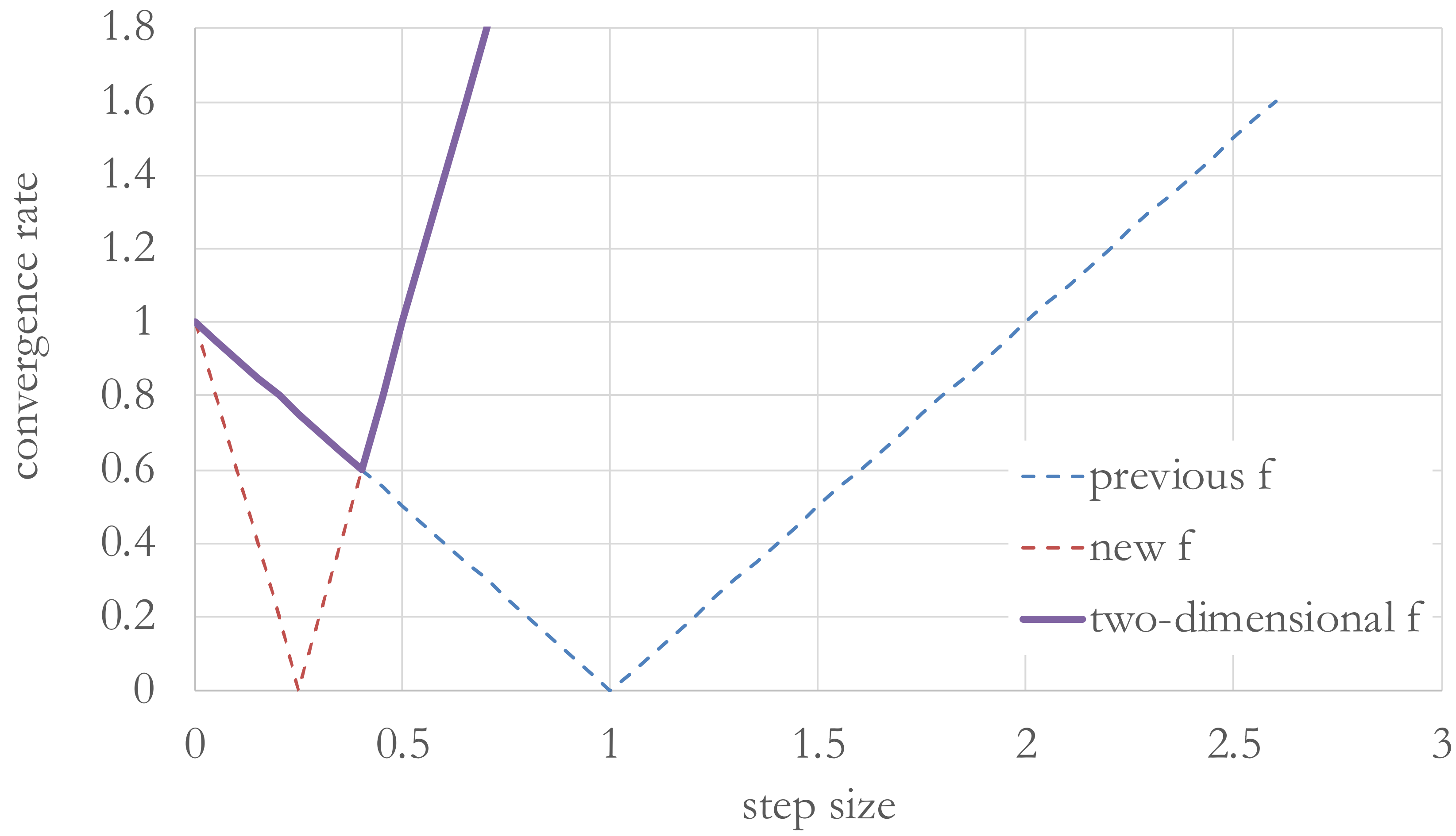
## What's the convergence rate?

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

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

- 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, $\kappa$ is called the **condition number** of the matrix $\mathbf{A}$.

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

- 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

$$z_{t+1} = 2 u z_t - z_{t-1}$$

- 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**

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

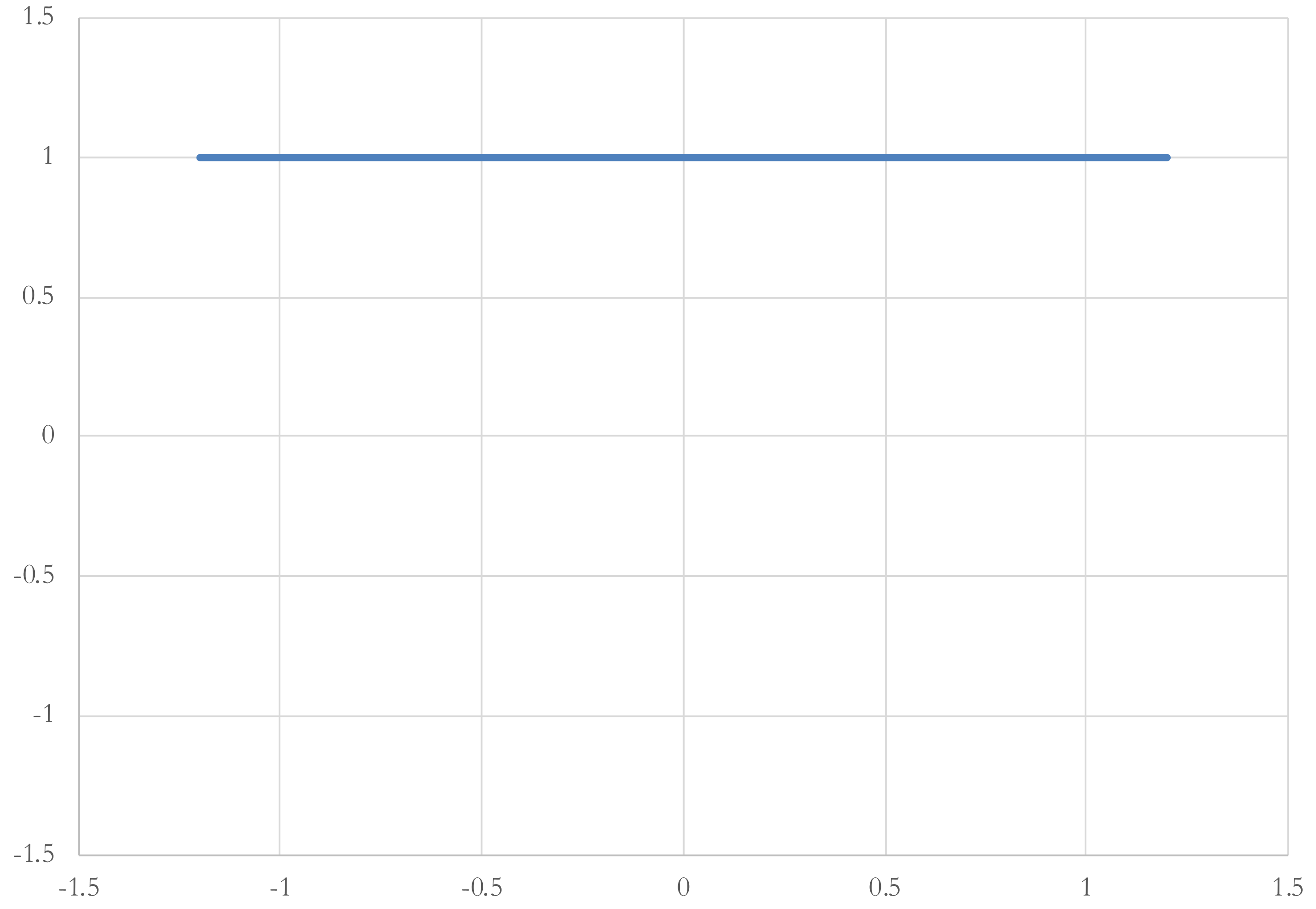
- Standard notation:

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

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

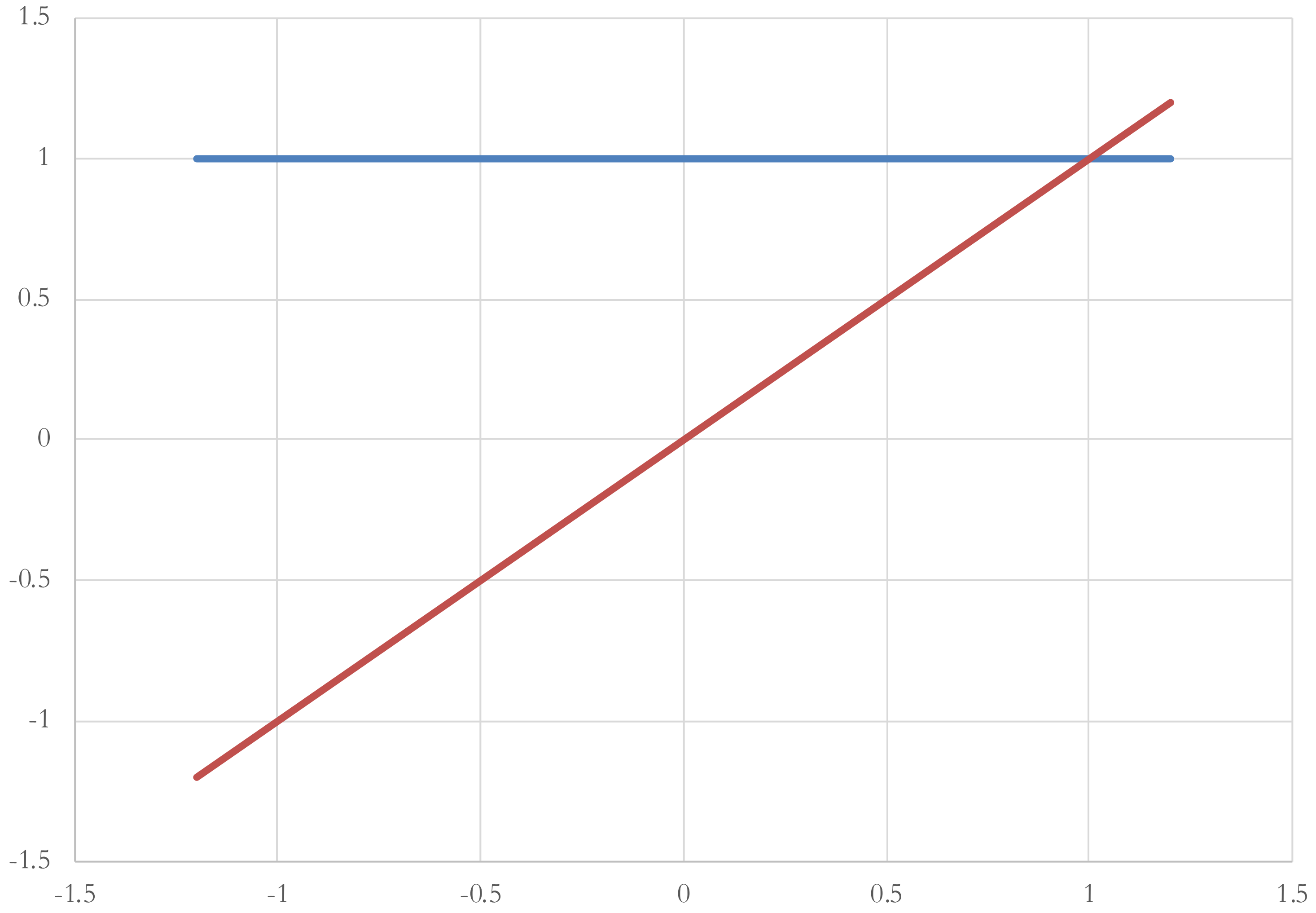
$$-1 \leq u \leq 1 \Rightarrow -1 \leq z_t \leq 1$$

# Chebyshev Polynomials



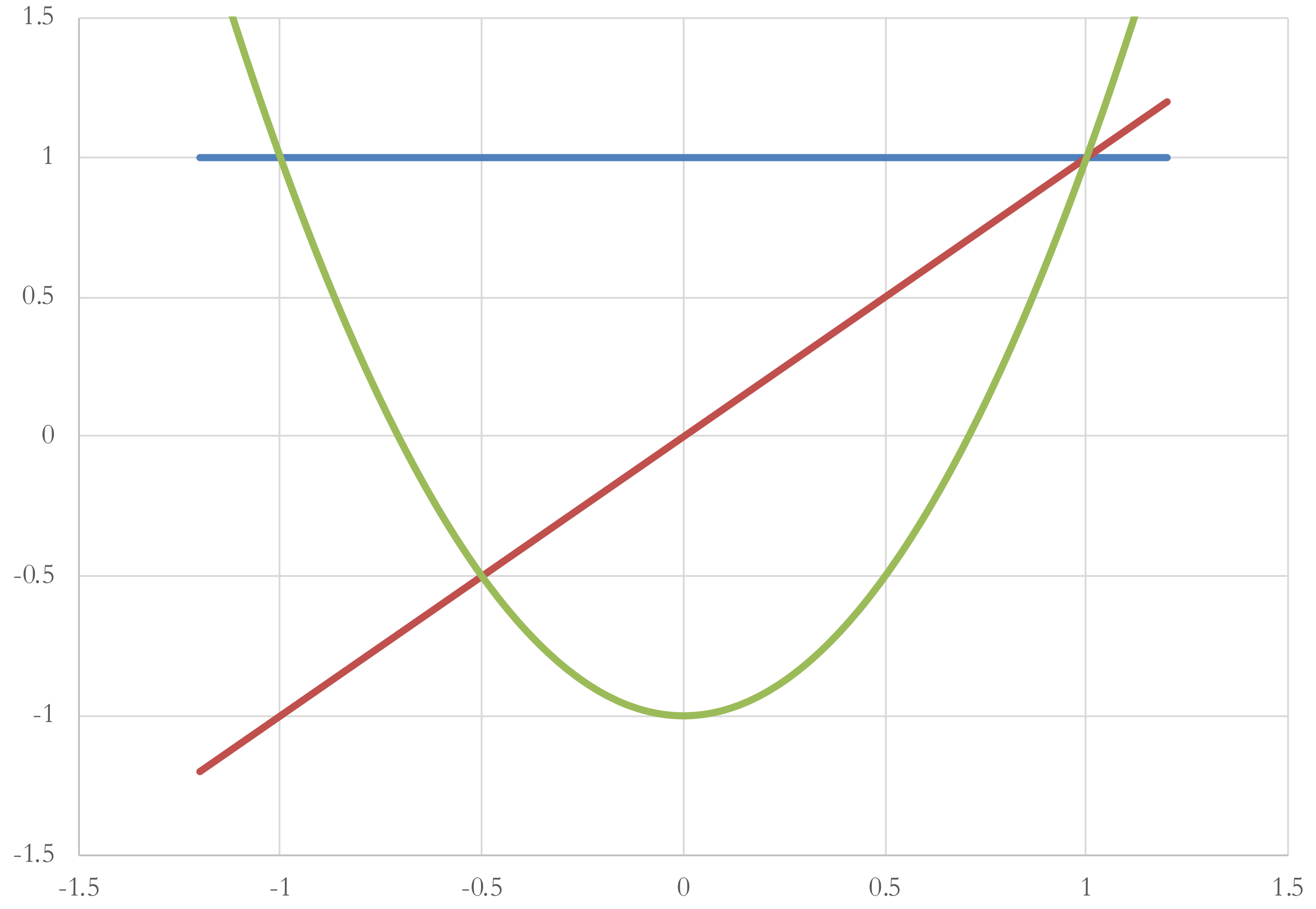
$$T_0(u) = 1$$

## Chebyshev Polynomials



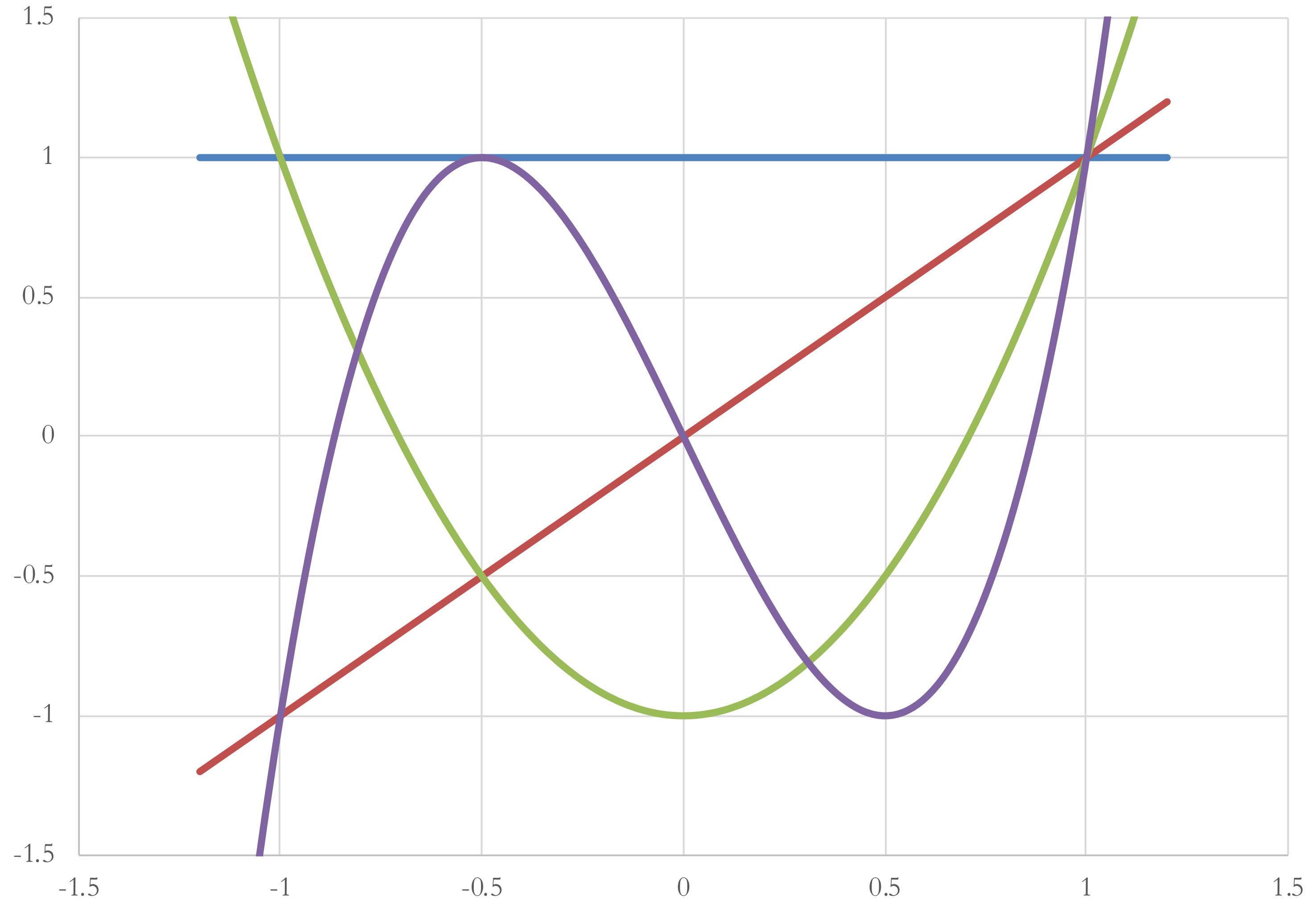
$$T_1(u) = u$$

# Chebyshev Polynomials

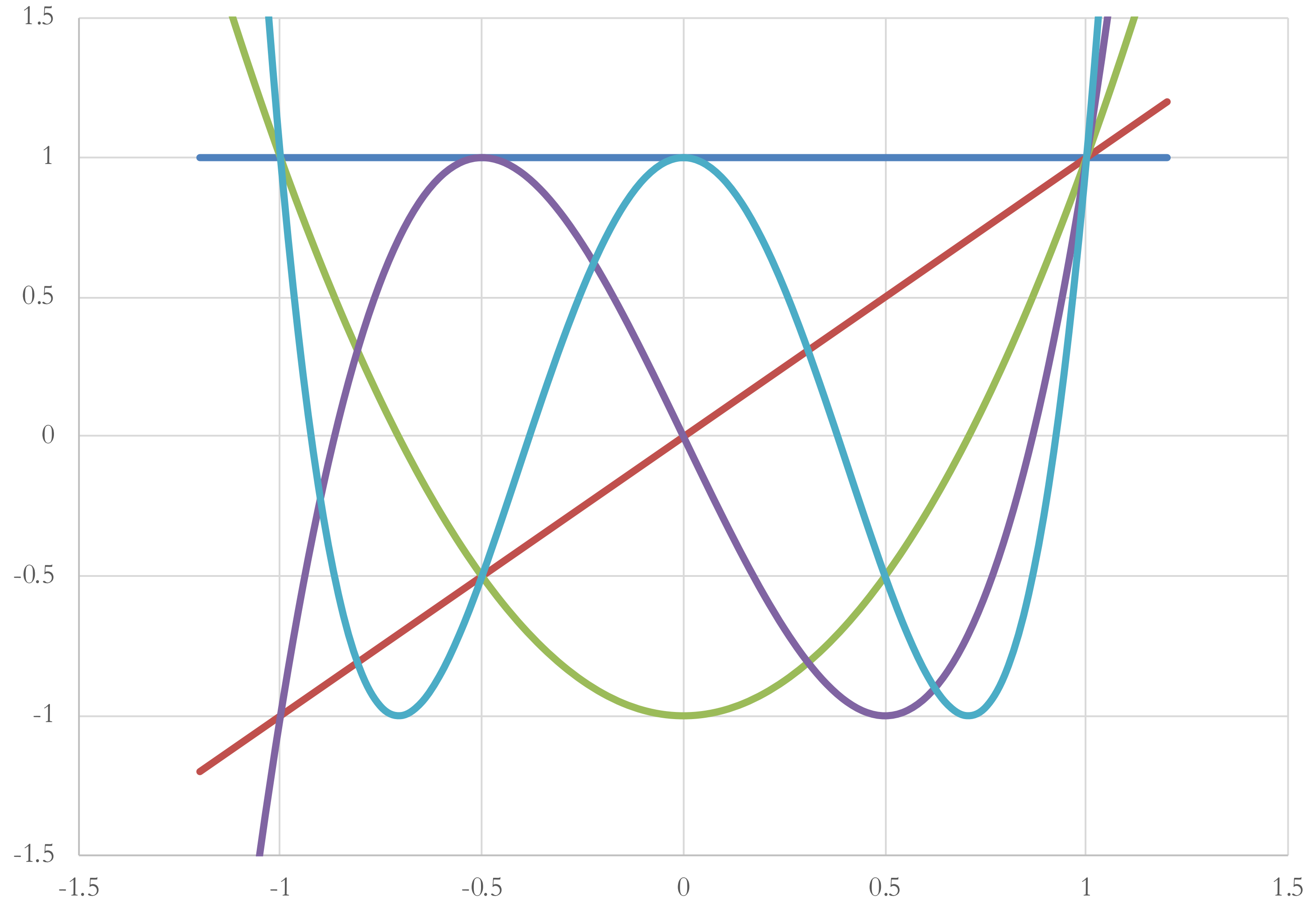


$$T_2(u) = 2u^2 - 1$$

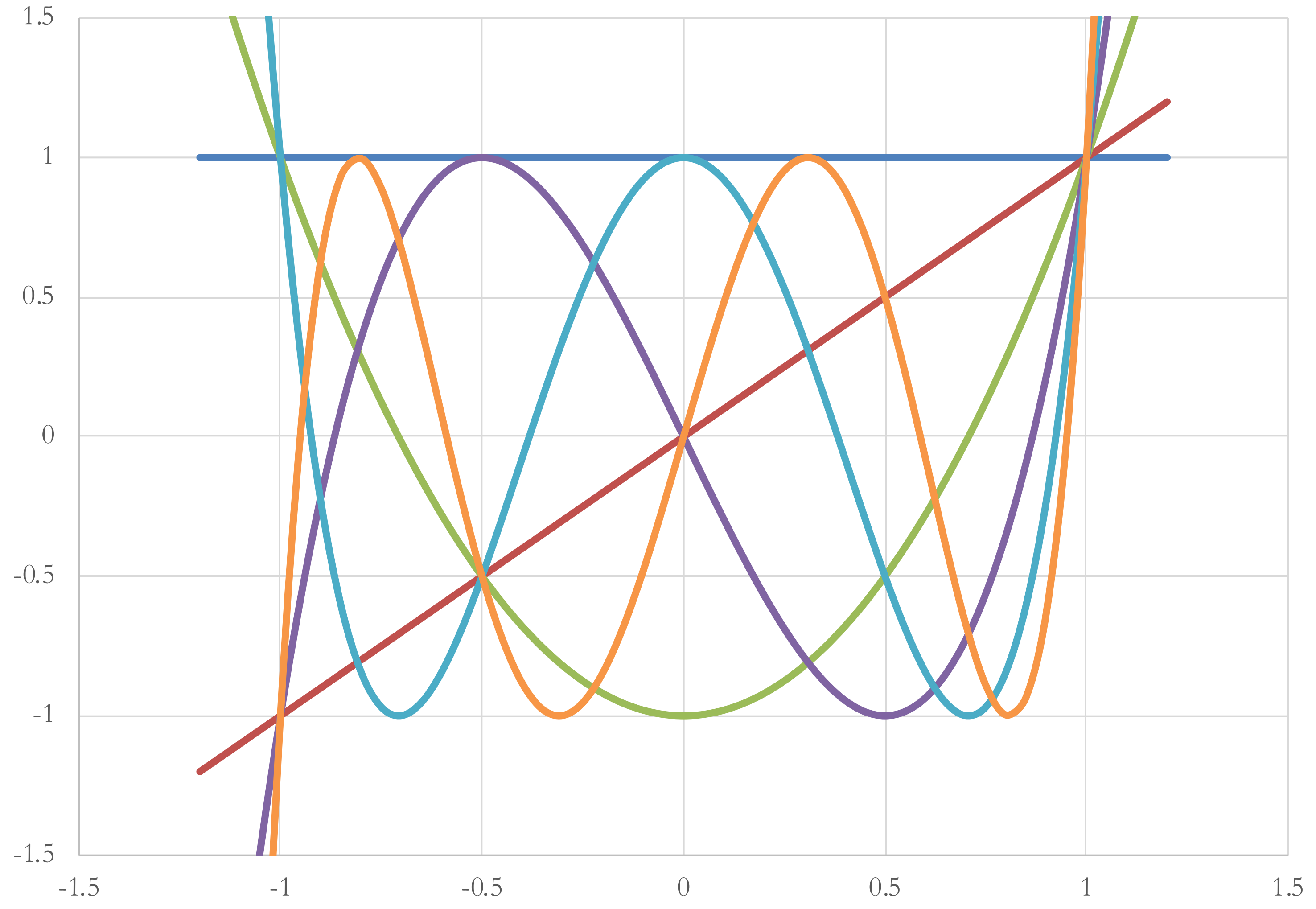
# Chebyshev Polynomials



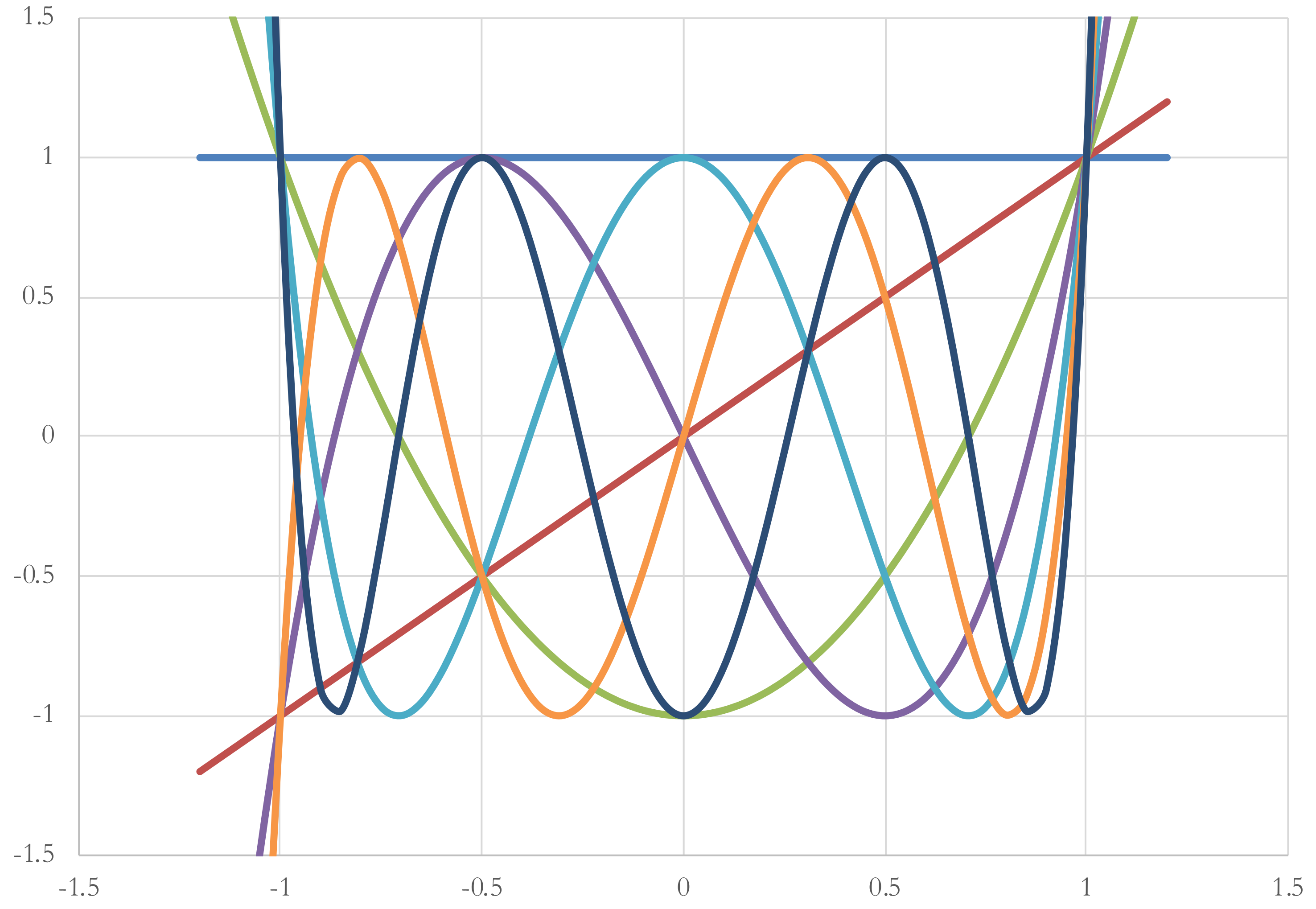
# Chebyshev Polynomials



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

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

## 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 **Wednesday**