Getting SGD Off The Ground

Basic Techniques We Always Use

CS6787 Lecture 2 — Fall 2021

The hidden cost of SGD

- By switching to SGD, we eliminated the costly sum over the dataset

$$w_{t+1} = w_t - \alpha \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla f(w_t, x_i)$$

$$\Downarrow$$

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

- But the cost of **computing the individual gradients** remains, and we'll need to run more steps

Using gradients naively is problematic

- **Hardware efficiency perspective**
  - Need some way to compute gradients efficiently on the underlying hardware
- **Software engineering perspective**
  - If we had to express the gradients by hand, this requires human effort
  - Also makes it **difficult to change the objective**, since we'd need to re-derive the gradient
  - Also makes us **prone to bugs**

How do we address these problems?

- **Automatic differentiation**
  - Compute a gradient automatically — just need to specify the objective
  - Prime example: **backpropagation**
  - Can also decompose the gradient computation into operators that will run efficiently on hardware
- **Machine learning frameworks**
  - Make it easy to express learning tasks in a high-level language
  - Support for building scalable systems

Why Automatic Differentiation?

- There are other classical approaches we have to compute derivatives.

- **Symbolic differentiation**
  - Express the objective function as a mathematical expression, then differentiate symbolically by applying the chain rule and other rules of calculus.

- **Numerical differentiation**

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \text{ for small } \epsilon$$

Why might these be a bad approach for SGD?

How does automatic differentiation work?

- Main idea: **transform the program that computes the objective directly into a program that computes the gradient**.

- There are many ways of doing automatic differentiation
  - Two broad classes: **forward mode** and **reverse mode**

- For most ML applications, we use **backpropagation**, a particular flavor of reverse-mode automatic differentiation
  - One specialized to compute gradients of neural network objectives

Backpropagation

- Start with a computation graph that represents the function to be differentiated
- **Forward pass**: compute through the graph normally, as if we were computing the function value
    - Save all intermediate values used in the computation—**memory cost**
- **Backward pass**: now proceed backwards through the graph, computing gradients of the function value w.r.t. intermediates

Backpropagation: A simple example

- Consider using backpropagation to differentiate the function

$$h(x) = \exp(\sin(\cos(x)))$$

$$h'(x) = \exp(\sin(\cos(x))) \cdot \cos(\cos(x)) \cdot (-\sin(x))$$

- Notice that there's a bunch of redundant expressions here
- Backpropagation will compute, in order:

$$\begin{aligned} a_1 &= \cos(x) \\ a_2 &= \sin(a_1) \\ a_3 &= \exp(a_2) \end{aligned}$$

$$\begin{aligned} g_3 &= \exp(a_2) \\ g_2 &= g_3 \cdot \cos(a_1) \\ f'(x) &= g_2 \cdot (-\sin(x)) \end{aligned}$$

Key thing to know: Automatic differentiation can compute gradients…

- For **any function** that has differentiable components
- To **arbitrary precision**
- Using a **small constant factor of additional compute** compared with the cost to compute the objective

Systems tradeoffs of backpropagation

- Question: **what are some aspects of backpropagation that are beneficial from a systems/hardware efficiency perspective?**

- Question: **what are some aspects of backpropagation that may present an additional systems/hardware efficiency cost?**
  - Relative to the cost of computing the function (but not the gradient).

This solves part of the problem…but there’s still a software engineering challenge.

How do we build software that people can use to reliably and robustly build, train, and deploy machine learning solutions?

The answer: machine learning frameworks

- Goal: **make ML easier**
  - From a software engineering perspective
  - Make the computations more reliable, debuggable, and robust
- Goal: **make ML scalable**
  - To large datasets running on distributed heterogeneous hardware
- Goal: **make ML accessible**
  - So that even people who aren’t ML systems experts can get good performance

ML frameworks come in a few flavors

- **General machine learning frameworks**
  - Goal: make a wide range of ML workloads and applications easy for users
- **General big data processing frameworks**
  - Focus: computing large-scale parallel operations quickly
  - Typically has machine learning as a major, but not the only, application
- **Deep learning frameworks**
  - Focus: fast scalable backpropagation and inference
  - Although typically supports other applications as well

How can we evaluate an ML framework?

- **How popular is it?**
  - Use drives use — ML frameworks have a **snowball effect**
  - Popular frameworks attract more development and eventually more features
- **Who is behind it?**
  - Major companies ensure long-term support
- **What are its features?**
  - Often the least important consideration — unfortunately

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

```
def logreg_loss(w, X, Y):
    return np.log(1.0 + np.exp(-Y * (X @ w))).sum()

def logreg_grad(w, X, Y):
    return -X.T @ (Y / (1.0 + np.exp(Y * (X @ w))))
```

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

NumPy and SciPy.org Sponsored By ENTHOUGHT

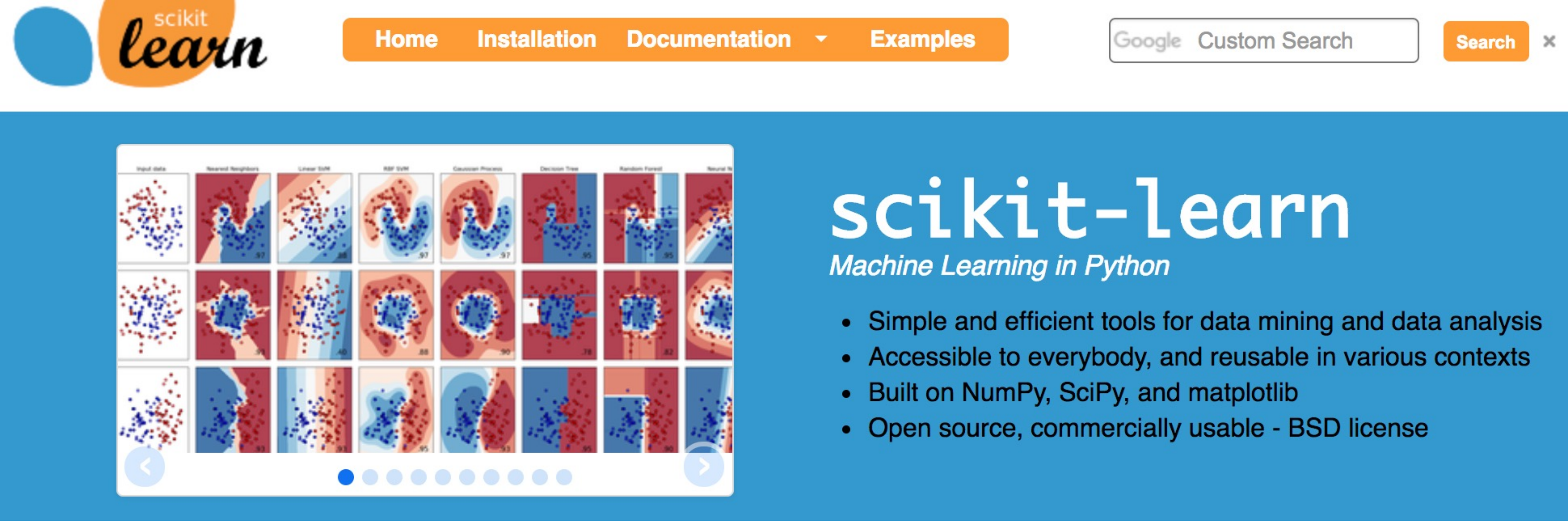
- **NumPy**
  - Adds large multi-dimensional array and matrix types (tensors) to python
  - Supports basic numerical operations on tensors, on the CPU

- **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**

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**



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

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

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**

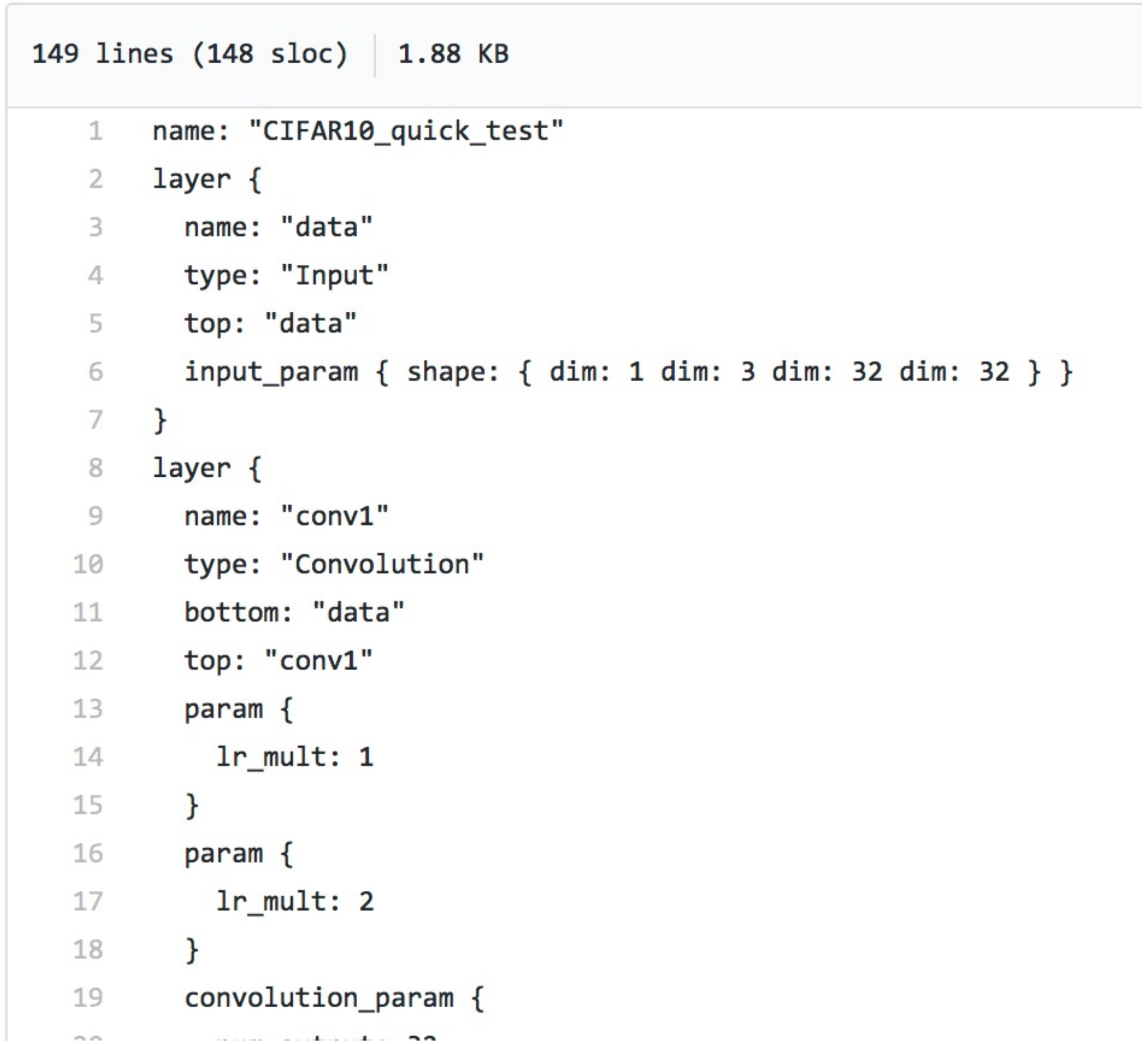
- Takeaway: **important to release code people can use easily**
  - And capture a group of users who can then help develop the framework

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

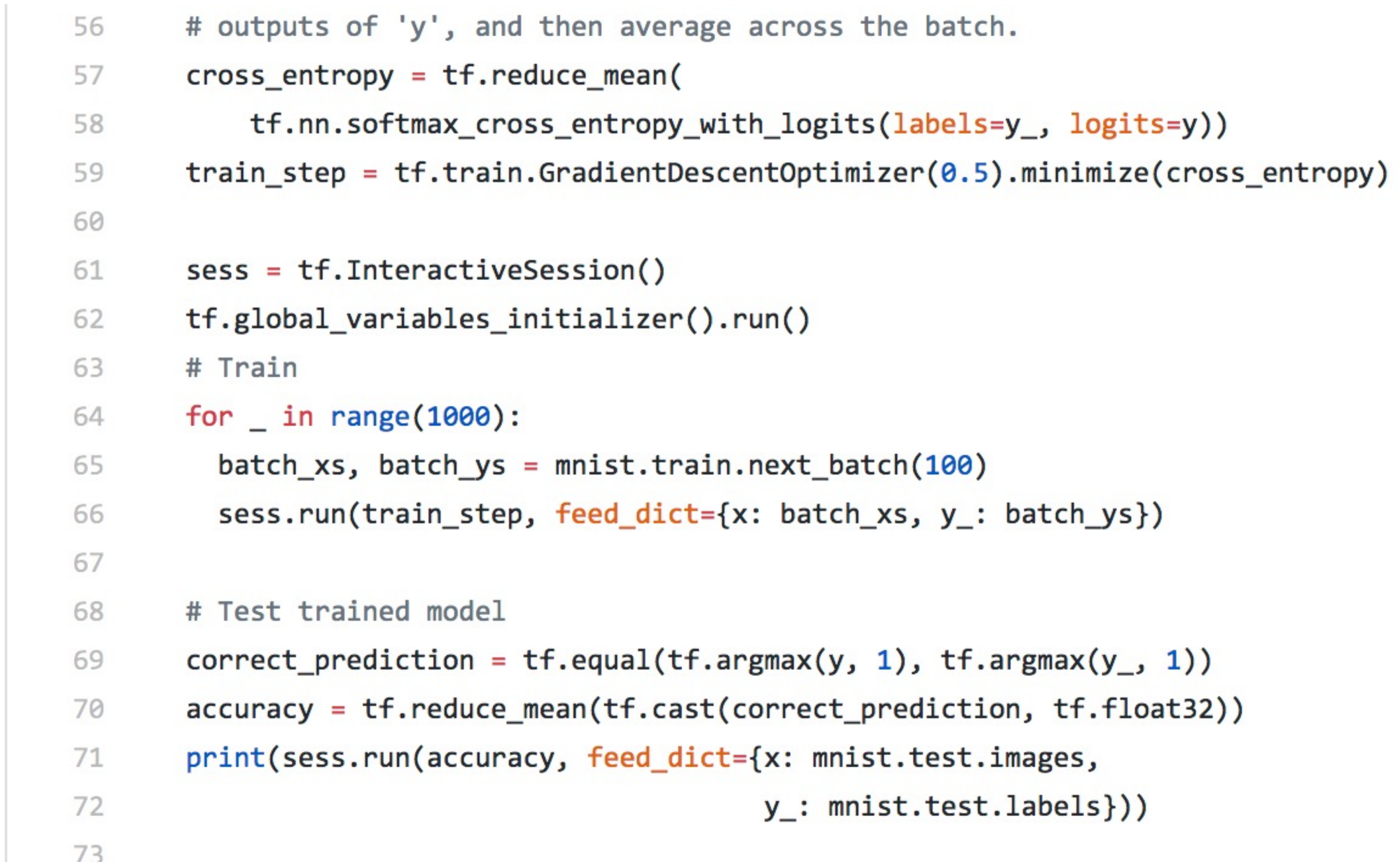

149 lines (148 sloc) | 1.88 KB

```
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**, still very popular for deploying ML

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

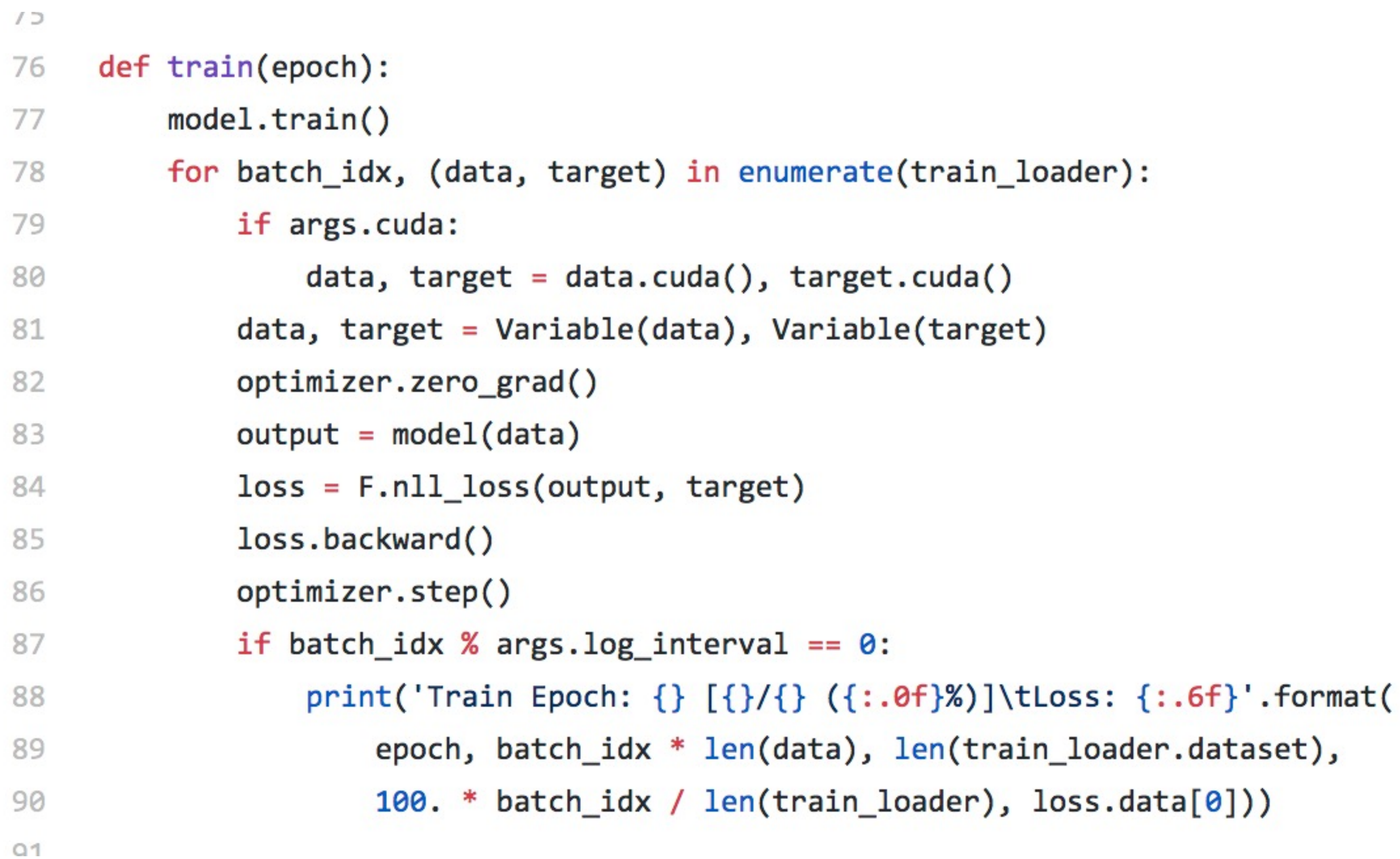
- **Python** package that focuses on
  - **Tensor computation** (like numpy) with strong **GPU acceleration**
  - **Deep Neural Networks** built on a tape-based autograd system

- **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 **the most popular framework for ML research**

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

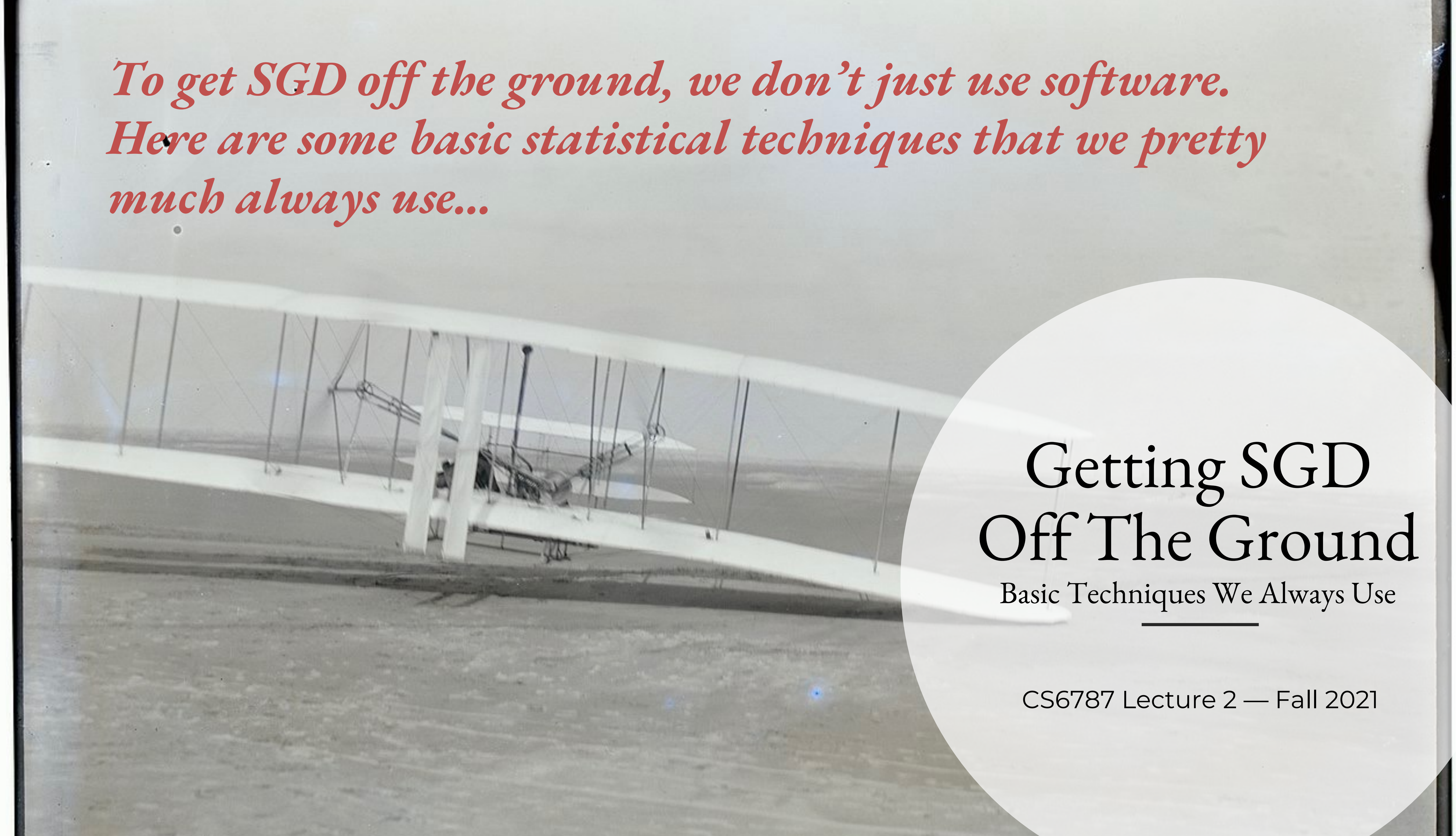
- Theano
- ONNX
- Jax
- New frameworks will continue to be developed!

ML Frameworks: Conclusion

- We use ML frameworks to both make ML code run more efficiently and make it easier to express learning procedures

- 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.



To get SGD off the ground, we don't just use 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 2 — 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?

Mini-Batch SGD Converges

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

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

- Second moment bound

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

Mini-Batch SGD Converges (continued)

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

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

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

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)

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

- 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 at most

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

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

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

  - **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 — we'll see this come up in *Batch Normalization*
- Another use case for **hyperparameter optimization**

Mini-Batch SGD is very widely used

- Including in basically all neural network training

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

- Nowadays larger batch sizes are more common
    - **b = 64, b = 128**
    - Some of this change is driven by systems considerations!

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, we 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**

Questions?

- Upcoming things
  - Please fill out the **paper assignment survey** tonight!