# Optimization for Machine Learning

*Machine Learning Fundamentals for Economists*


## Jesse Perla

*jesse.perla@ubc.ca*

*University of British Columbia*

# Table of contents

# Overview

# Summary

- This lecture continues from the previous lecture on gradients to further explore optimization methods in machine learning, and discusses training pipelines and tooling
- Primary reference materials are:
  - → ProbML Book 1: Introduction
  - → ProbML Book 2: Advanced Topics including Section 6.3
  - → Mark Schmidt's ML Lecture Notes
- We will also give a sense of a standard machine learning pipeline of training, validation, and test data and discuss generalization, logging, etc.

# Why the Emphasis on Optimization and Gradients?

- A huge number of algorithms for economists can be written as optimization problems (e.g., MLE, interpolation) or as something similar in spirit (e.g. Bayesian Sampling, Reinforcement Learning)
- Previous lectures on AD showed ways to find VJPs for extremely complicated functions. **Differentiate everything, no excuses!**
- In practice, **all** problems with high-dimensions parameters or latents require gradients for algorithms to be feasible
- We will soon take a further step: **are (unbiased) estimates of the gradient good enough for many algorithms?**

# Optimization Crash Course

# Optimization Methods

- Learning continuous optimization methods is an enormous project
- See referenced materials and lecture notes
- Here we will give an overview of some key concepts
- Be warned! The details matter, so more study is required if you want to use these methods in practice

# Crash Course in Unconstrained Optimization

$$\min_{\theta} \mathcal{L}(\theta)$$

Will briefly introduce

- First-order methods
- Second-order methods
- Preconditioning
- Momentum
- Regularization

# First-Order Methods

- See ProbML Book 1 Section 8.2
- Armed with reverse-mode AD for $\mathcal{L} : \mathbb{R}^N \rightarrow \mathbb{R}$ we can calculate $\nabla \mathcal{L}(\theta)$ with the same computational order as $\mathcal{L}(\theta)$
- Furthermore, given JVPs we know we can calculate these objective functions for extremely complicated functions (e.g., nested fixed points, and implicit functions)
- Iterative: take $\theta_0$ and provide $\theta_t \rightarrow \theta_{t+1}$
  - → May converge to a stationary point (hopefully close to a global argmin)
  - → If it doesn't converge, the solution may still be an argmin
  - → See references for details on convergence for convex and non-convex problems

# Gradient Descent

- See Mark Schmidt's Notes
- Gradient descent takes $\theta_0$, and stepsize $\eta_t$ and iterates until $\nabla \mathcal{L}(\theta_t)$ is small, or $\theta_t$ stationary

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}(\theta_t)$$

- It is the simplest "first-order" method (i.e., using just the gradient of $\mathcal{L}$)
- Will call $\eta_t$ a "learning rate schedule"
- Think of line-search methods as choosing the stepsize $\eta_t$ optimally. Can help with many of our problems

# When and Where Does This Converge?

- Skipping a million details, see ProbML Book 1 Section 8.2.2 and Mark Schmidt's basic and more advanced notes
- For strictly convex problems this converges to the global minima, though sufficient conditions include Robbins-Monro $\lim_{T\to\infty} \eta_T = 0$ and

$$\lim_{T\to\infty} \frac{\sum_{t=1}^{T} \eta_t}{\sum_{t=1}^{T} \eta_t^2} = 0$$

- For problems that not globally convex this may go to local optima, but if the function is locally strictly convex then it will converge to a local optima
- For other types of functions (e.g., invex) it may still converge to the "right" solution in some important sense

# Preconditioned Gradient Descent

- As we saw analyzing LLS, badly conditioned problems converge slowly with iterative methods
- We can precondition a problem as we did with linear systems, and it has the same stationary point
- Choose some $C_t$ for preconditioned gradient descent

$$\theta_{t+1} = \theta_t - \eta_t C_t \nabla \mathcal{L}(\theta_t)$$

- We saw before that the Hessian tells us the geometry, so the optimal preconditioner must be related to $\nabla^2 \mathcal{L}(\theta_t)$

# Second-Order Methods

- See ProbML Book 1 Section 8.3 and Mark Schmidt's Notes
- Adapt $\eta_t C_t$ to use the Hessian (e.g., Newton's Method)

$$\theta_{t+1} = \theta_t - \eta_t \left[\nabla^2 \mathcal{L}(\theta_t)\right]^{-1} \nabla \mathcal{L}(\theta_t)$$

- Second order methods are rarer because the calculating the Hessian is no longer the same computational order as $\mathcal{L}(\theta)$
- See ProbML Book 1 Section 8.3.2 for info on Quasi-Newtonian methods which approximation Hessian using gradients like BFGS

# Momentum

- See ProbML Book 1 Section 8.2.4 and Mark Schmidt's Notes
- Can use "momentum", which speeds up convergence, helps avoid local optima, and moves fast in flat regions
- Momentum will be a common feature of many ML optimizers (e.g. Adam, RMSProp, etc.) as it helps with heavily non-convex problems
- A classic method is called Nesterov Accelerated Gradient (NAG), which is a modification of gradient descent for some $\beta_t \in (0,1)$ (e.g., $0.9$)

$$\hat{\theta}_{t+1} = \theta_t + \beta_t(\theta_t - \theta_{t-1})$$
$$\theta_{t+1} = \hat{\theta}_{t+1} - \eta_t \nabla \mathcal{L}(\hat{\theta}_{t+1})$$

# Does Uniqueness Matter?

- Remember from our previous lecture on Sobolev norms and regularization that we care about functions, not parameters.
- Consider when $\theta$ is used as parameters for a function (e.g. $\hat{f}_\theta$)
  - → Then what does a lack of convergence of the $\theta_t$ or multiplicity with multiple $\theta$ solutions mean?
  - → Maybe nothing! If $\|\hat{f}_{\theta_0} - \hat{f}_{\theta_1}\|_S$ is small, then the functions themselves may be in the same equivalence class. Depends on the norm, of course.
- This topic will be discussed when we consider double-descent curves, but the punchline for now is that the training/optimization is a means to an end (i.e., generalization) and not an end in itself.

# Regularization

- See **Mark Schmidt's Notes**. For LLS this is the ridge regression
- We discussed regularization as a way to deal with multiplicity

$$\min_{\theta} \left[ \mathcal{L}(\theta) + \frac{\alpha}{2} \| \theta \|^2 \right]$$

- Gradient descent becomes (called "weight decay" in ML, and "ridge regression" if objective is LLS)

$$\theta_{t+1} = \theta_t - \eta_t \left[ \nabla \mathcal{L}(\theta_t) + \alpha \theta_t \right]$$

- Mapping of regularized $\theta_t$ to a $f_{\theta_t}$ is subtle if nonlinear

# Stochastic Optimization

# Are Gradients Really that Cheap to Calculate?

- Consider that the objective often involves data (or grid points for interpolation)
  - → Denote $x_n$, and observables $y_n$ for $n = 1, \dots N$
- With VJPs, the computational order of $\nabla_\theta \mathcal{L}(\theta; \{x_n, y_n\}_{n=1}^N)$ may be the same as that of $\mathcal{L}$ itself
- However, keep in mind that reverse-mode requires storing the intermediate values in the "primal" calculation (i.e., $\mathcal{L}(\theta; \{x_n, y_n\}_{n=1}^N)$)
  - → Hence, the memory requirements grow with $N$
  - → This may be a big problem for large datasets or complicated calculations, especially with GPUs which have more limited memory

# Do We Need the Full Gradient?

- In practice, it is impossible to calculate the full gradient for large datasets
- In GD, the gradient provided the direction of steepest descent
- Consider an algorithm with a $g_t$ as an unbiased estimate of the gradient

$$\theta_{t+1} = \theta_t - \eta_t g_t$$
$$\mathbb{E}[g_t] = \nabla \mathcal{L}(\theta_t)$$

  - → Make the $\eta_t$ smaller to deal with noise if this is high-variance
  - → Choose $g_t$ to be far cheaper to calculate than $\nabla \mathcal{L}(\theta_t)$
- Remember: we don't need the actual value of the objective function to optimize it!
- Will also add additional regularization, which can help with generalization

# Stochastic Optimization

- To formalize: Up until now our optimizers have been "deterministic"
- Now we introduce a source of randomness $z \sim q_\theta$, i.e. it might depend on the estimated parameters $\theta$ later with RL/etc.
  - → $z$ could be a source of uncertainty in the environment
  - → $z$ could involve latent variables
  - → $z$ could come from randomness in the optimization process (e.g., using subsets of data to form $g_t$)
- Denote expectations using this distribution as $\mathbb{E}_{z \sim q_\theta}$
- For now, drop the dependence on $\theta$ for simplicity, though it becomes crucial for understanding reinforcement learning/etc.

# Stochastic Objective

- The full optimization problem is then to minimize this stochastic objective

$$\min_{\theta} \overbrace{\mathbb{E}_{q(z)} \tilde{\mathcal{L}}(\theta, z)}^{\equiv \mathcal{L}(\theta)}$$

- Under appropriate regularity conditions, could use GD on this objective

$$\nabla \mathcal{L}(\theta) = \mathbb{E}_{q(z)} \left[ \nabla \tilde{\mathcal{L}}(\theta, z) \right]$$

- But in practice, it is rare that we can marginalize out the $z$

# Unbiased Draws from the Gradient

- Assume we can sample $z_t \sim q(z)$ IID
- Then with enough regularity the gradient using just $z_t$ is unbiased

$$\mathbb{E}_{q(z)}\left[\nabla\tilde{\mathcal{L}}(\theta_t, z_t)\right] = \nabla\mathcal{L}(\theta_t)$$

- That is, on average $\nabla\tilde{\mathcal{L}}(\theta_t, z_t)$ is in the right direction for minimizing $\mathcal{L}(\theta_t)$
- This basic approach of finding unbiased estimators of the gradient (and finding ways to lower the variance) is at the heart of most ML optimization algorithms

# Stochastic Gradient Descent

- See ProbML Book 1 Section 8.4, ProbML Book 2 Section 6.3, and Mark Schmidt's Notes
- Given the previous slide, given IID samples $z_t \sim q$, the gradient is unbiased and we have the simplest version of stochastic gradient descent (SGD)

$$\theta_{t+1} = \theta_t - \eta_t \nabla \tilde{\mathcal{L}}(\theta_t, z_t)$$

- Which converges to the minima of $\mathbf{min}_\theta \mathcal{L}(\theta)$ under appropriate conditions
- We can layer on all of the other features we discussed (e.g., momentum, preconditioning, etc) with SGD, but some become especially important (e.g. the $\eta_t$ schedule)

# Finite-Sum Objectives

- Consider a special case of the loss function which is the sum of $N$ terms. For example with empirical risk minimization used in LLS/etc.
  - → $z_n \equiv (x_n, y_n)$ are typically data, observables, or grid points
  - → $\ell(\theta, x_n, y_n)$ is a loss function for a single data point (e.g., forecasting using some $f_\theta$)

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \tilde{\mathcal{L}}(\theta, z_n) \equiv \frac{1}{N}\sum_{n=1}^{N} \ell(\theta, x_n, y_n)$$

  - → For example, LLS is $\ell(\theta, x_n, y_n) = \| y_n - \theta \cdot x_n \|_2^2$
- In this case, the randomness of $z_t$ is which data point is chosen

# SGD for Finite-Sum Objectives

- Hence consider sampling $z_t \equiv (x_t, y_t)$ from our data.
  - → In principle, IID with replacement
- Then run SGD on one data point at a time

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \ell(\theta_t, x_t, y_t)$$

- This may converges to the minima of $\mathcal{L}(\theta)$, and potentially the storage requirements for calculations the gradient are radically reduced
- You can guess that the $\eta_t$ parameter is especially sensitive to the variance of the gradient estimate

# Decrease Variance with Multiple Draws

- With a single draw, the variance of the gradient estimate may be high

$$\mathbb{E}\left[\nabla_\theta \ell(\theta_t, x_t, y_t) - \nabla \mathcal{L}(\theta_t)\right]^2$$

- One tool to decrease the variance is just more monte-carlo draws. With finite-sum objectives draw $B \subseteq \{1, \dots N\}$ indices

$$\frac{1}{|B|} \sum_{n \in B} \nabla_\theta \ell(\theta_t, x_n, y_n)$$

- Classic SGD: $|B| = 1$; GD: $B = \{1, \dots N\}$ and in between is called “minibatch SGD”. Usually minibatch is implied with “SGD”

# Minibatch SGD

- Algorithm is to draw $B_t$ indices at each step and execute SGD

$$g_t \equiv \frac{1}{|B_t|} \sum_{n \in B_t} \nabla_\theta \ell(\theta_t, x_n, y_n)$$
$$\theta_{t+1} = \theta_t - \eta_t g_t$$

- Note that we never need to calculate $\mathcal{L}(\theta_t)$ directly, so can write our code to all operate on batches $B_t$
- Then layer other tricks on top (e.g., momentum, preconditioning, etc.)

  → In principle you could also use minibatch with second-order or quasi-newtonian methods but much rarer

# Choosing Batches

- Choosing the $B_t$ process may be tricky. You could sample from $\{1, \dots N\}$
  - → with replacement
  - → without replacement
  - → without replacement after shuffling the data, and then ensure you have gone through all of the data before repeating
  - → etc.
- Just remember the goal: variance reduction on gradient estimates
- You want it to be unbiased in principle (consider partitioning the data into batches and operating sequentially?)
- More art than science in many cases, because it requires many priors

# Stochastic Optimization with $q_\theta$

- Previously, $q(z)$ was independent of $\theta$ (i.e., $\min_\theta \mathbb{E}_{q(z)} \tilde{\mathcal{L}}(\theta, z)$)
- For many problems, the population distribution is dependent on the parameters $\theta$ (e.g., reinforcement learning, variational inference, etc.).

$$\min_\theta \overbrace{\mathbb{E}_{z \sim q_\theta} \tilde{\mathcal{L}}(\theta, z)}^{\equiv \mathcal{L}(\theta)}$$

- if $q_\theta$ we have our previous special case, otherwise

$$\nabla_\theta \mathcal{L}(\theta) = \underbrace{\int \nabla_\theta \tilde{\mathcal{L}}(\theta, z) q_\theta(z) dz}_{= \mathbb{E}_{z \sim q_\theta}\left[\nabla_\theta \tilde{\mathcal{L}}(\theta, z)\right]} + \int \tilde{\mathcal{L}}(\theta, z) \nabla_\theta q_\theta(z) dz$$

# Computational Challenges with Approximation

- Given $\theta$ we can use $B$ samples from $q_\theta(z)$ we can still approximate $\mathbb{E}_{q(z)}\left[\nabla_\theta \tilde{\mathcal{L}}(\theta, z)\right]$ $\approx \frac{1}{|B|}\sum_{z\in B}\nabla_\theta \tilde{\mathcal{L}}(\theta, z)$
- But note that $\int \tilde{\mathcal{L}}(\theta, z)\nabla_\theta q_\theta(z)dz$, requires $\nabla_\theta q_\theta(z)$

  → "Vanilla" monte-carlo approximations which can only sample from $q_\theta$ in order to find gradient estimates of the objective are no longer possible
- There are several common approaches, often used in reinforcement learning, variational inference, etc.
- See ProbML Book 2: Advanced Topics section 6.3.3 for more

# Score Function Estimation (REINFORCE)

- Apply the chain rule to $\log q_\theta(z)$, we find $\nabla_\theta q_\theta(z) = q_\theta(z)\nabla_\theta \log q_\theta(z)$. Rewrite $\int \tilde{\mathcal{L}}(\theta, z)\nabla_\theta q_\theta(z) dz$ as

$$\int \tilde{\mathcal{L}}(\theta, z)(\nabla_\theta \log q_\theta(z)) q_\theta(z) dz = \mathbb{E}_{q(z)}\left[\tilde{\mathcal{L}}(\theta, z)\nabla_\theta \log q_\theta(z)\right]$$

- Then the REINFORCE estimator using $B$ samples from $q_\theta(z)$ is

$$\mathbb{E}_{z\sim q}\left[\tilde{\mathcal{L}}(\theta, z)\nabla_\theta \log q_\theta(z)\right] \approx \frac{1}{|B|}\sum_{z\in B} \tilde{\mathcal{L}}(\theta, z)\nabla_\theta \log q_\theta(z)$$

  → Requires a differentiable $q_\theta(z)$ calculated local to each $z$

  → This estimator is usually high variance, so there are many tricks to reduce this. See ProbML Book 2 Sections 6.3.4-6.3.7 or RL books for more

# The “Reparameterization Trick”

- Alternatively, if you can rewrite the $q_\theta(z)$ in terms of $\epsilon \sim q_\epsilon$
  - → e.g., $z = g(\epsilon; \theta)$ for some $g(\cdot; \theta)$ differentiable wrt $\theta$
  - → Then the optimization objective is

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}_{q_\epsilon(\epsilon)} \left[ \nabla_\theta \tilde{\mathcal{L}}(\theta, g(\epsilon; \theta)) \right]$$

- This lets us use stochastic gradient approximations sampling $\epsilon$ batches

$$\nabla_\theta \mathcal{L}(\theta) =\approx \frac{1}{|B|} \sum_{\epsilon \in B} \nabla_\theta \tilde{\mathcal{L}}(\theta, g(\epsilon; \theta))$$

# Training Loops

# “Grad Student Descent”

- Those methods for stochastic optimization make deep learning possible. Just swap SGD with slightly fancier algorithms using momentum, tinker with parameters, etc.
- In practice, all of these optimizer settings (e.g., how large for $|B_t|$, $\eta_t$, convergence criteria, etc.) are fragile and require a lot of tuning
  - → Part of a a process called **hyperparameter optimization (HPO)** where you try to find the best non-model parameters for your goals
  - → Same issue with all numerical methods in economics (e.g. convergence criteria of fixed point iteration, initial conditions)
- The concern is not just that it is time-consuming for researchers (and ML “Grad Students”), but that it is easy for priors to sneak in and bias results

# What was our Goal?

- We will address this more formally next lecture, but it is worth stepping back to think about our goals. Loosely:
  - → If we are solving an empirical risk minimization problem (like regressions, etc.) or interpolation, then our goal is to use the "data" to find a function $\hat{f}_\theta$ that is close to the "true" function $f^*$
- Fitting $\hat{f}_\theta$ is easy, but we want it to **generalize** within the true distribution
  - → But we don't know that distribution (hence the "empirical")
  - → So a typical approach is to emulate this by splitting the data we have
  - → But HPO is dangerous because if we are not careful we can "contaminate" our process for finding $\hat{f}_\theta$ using some of the data we intend to check it with. Which might lead to overfitting/etc.

# Splitting the Data

A standard way to do this for Empirical Risk Minimization/Regressions/etc. is to split it into three parts:

1. **Training** data used in fitting our approximations
    - This is just a means to an end in ML and economics
2. **Validation** data used for HPO and checking convergence criteria
    - Be cautious to avoid using it for training
3. **Test** data used to evaluate the generalization performance
    - Ensure we don't accidentally use it in training or validation

Not all problems will have this structure (in particular, a "validation" set).

# Why Separate Validation and Test?

- As we will see in deep learning, with massive over-parameterization you typically can interpolate all of the training data.
  - → Minimizing training loss is a means to an end, which usually ends at zero
- The validation data might be used to check stopping criteria by checking how well the approximation generalizes to data outside of training
- But if we are using it for a stopping criteria or HPO, then is is **contaminated**!
  - → Distorts our picture of generalization if we combine it into test data

# What about Interpolation Problems?

- When simply trying to find interpolating functions which solve functional equations, the risk of prior contamination is less clear
- However, you may still want to separate out validation and test grid points because any data you use for HPO or convergence criteria can't be used to understand generalization.
- For example consider:
  1. Fit until "training" loss is zero
  2. Keep running stochastic optimizer until "validation" loss is zero
- In that case, it crudely interpolating the validation data, which makes it equivalent to training data? Not useful for generalization

  → May find that the model generalized better if you **stopped earlier**

# Level of Abstraction for Optimizers

- While you can setup a standard optimization objective and optimizer, most ML frameworks work at a lower level
- The key reasons are that:
  - → Minibatching (usually just called “batches”) requires more flexibility in implementation to be efficient
  - → Stopping criteria is more complicated with highly overparameterized models
  - → Logging and validation logic requires more flexibility
  - → Often you will want to take a snapshot of the current best solution and continue later for refinement (or to solve in parallel)

# Steps and Epochs

- There is a great deal of flexibility in how you setup the optimizer
- But a common approach is to randomly shuffle the data, create a set of batches $B_t$ (without replacement), and then iterate through them
- Terminology (when relevant)
  - → Every iteration of SGD for a given batch is a **step**
  - → If you have gone through the entire dataset once, we say that you have completed an **epoch**
- At the end of an epoch is a good time to log, check the validation loss, and potentially stop the training

# Software Components used in ML

Some common software components for optimization are

1. **Autodifferentiation** and libraries of functions provide the approximation class
2. **Data loaders** which will take care of providing batches to the optimizers
3. **Optimizers** are typically iterative, have an internal state, and you can update with one sample of the gradient for that batch
4. **Logging** and visualization tools to track progress because the optimization process may be slow and you want to do HPO
5. **HPO** software using training, validation, and possibly test loss

# Logging and Visualization

- Several tools exist for logging to babysit optimizers, find good hyperparameters, etc. including **Tensorboard**
  - → But we will use **Weights and Biases** (W&B) because it is a market leader, free for academics and seems to be the frontrunner
- Many algorithms and frameworks exist for HPO:
  - → **Weights and Biases** (W&B) has a built-in HPO framework using random search and bayesian optimization
  - → **Optuna** and **Ray Tune** is a popular open-source HPO framework
  - → **Ray Tune** is a popular open-source HPO framework
- HPO frameworks will often use the **command-line** to run new jobs. **Python Fire**

# Broad Frameworks for Machine Learning

- You can just hand-code loops/etc. which seems the best approach for JAX
  - → Even with Pytorch, it isn't obvious that a framework is better ex-post, though ex-ante it can help you try different permutations easily
- **Pytorch Lightning** is a popular framework which will formalize the training loops even across distributed systems and make CLI, HPO, logging, etc. convenient
  - → It remains fairly flexible because it is just wrapping Pytorch
- **Keras** is a similar framework with the ability to target multiple backends (e.g., Pytorch, JAX)
  - → The challenge is that it is much less flexible for non-typical research
- **Hydra** is a framework for more serious engineering code

# Linear Regression with Pytorch

# Linear Regression Examples

- Of course SGD is a terrible way to do a standard linear regression, but it will help us understand the mechanics with a well-understood problem
- These examples simulate data for some: $y_n = x_n \cdot \theta + \sigma\epsilon$ for $\epsilon \sim N(0,1)$
- They then show various features of the optimization pipeline and software to implement the LLS ERM objective

$$\min_{\theta} \frac{1}{N} \sum_{n=1}^{N} \left[ y_n - x_n \cdot \theta \right]^2$$

- Which we note has a finite-sum objective, which lets us use minibatch SGD with gradient estimates

# Packages

- Before showing all of the variations, here we will implement an inline SGD version with minibatches

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader, TensorDataset
from flax import nnx
from jax_dataloader.loaders import DataLoaderJAX
```

# Simulate Data

$$y \sim N(x \cdot \theta, \sigma^2)$$

```
N = 500  # samples
M = 2
sigma = 0.001
theta = torch.randn(M)
X = torch.randn(N, M)
Y = X @ theta + sigma * torch.randn(N)
dataset = TensorDataset(X, Y)
print(dataset[0]) # returns tuples of (x_n, y_n)
```

```
(tensor([-0.0415, -1.5740]), tensor(1.7757))
```

# Dataloaders Provide Batches

- Code which serves up random batches of data for gradient estimates are called “dataloaders”
- The following code shuffles the data and provides batches of size `batch_size`
- It returns a Python generator which can be iterated until it hits the end of the data

```
batch_size = 8
train_loader = DataLoader(dataset,
                 batch_size=batch_size,
                 shuffle=True)
# e.g. iterate and get first element
print(next(iter(train_loader)))
```

```
[tensor([[ 0.6299, -0.0860],
        [ 1.0579,  0.2490],
        [-0.4264,  1.3422],
        [ 1.8625,  0.7344],
        [-1.1870, -0.9154],
        [ 1.1389, -1.5414],
        [-0.1945,  0.3964],
        [-0.9229, -1.5121]]), tensor([ 0.7331,  0.7783,
-1.9794,  1.0244, -0.1371,  2.9250, -0.6530,  0.8181])]
```

# Loss Function for Gradient Descent

- Reminder: need to provide AD-able functions which give a gradient estimate, not necessarily the objective itself!
- In particular, for LLS we simply can find the MSE between the prediction and the data for the batch itself

```
def residuals(model, X, Y):  # batches or full data
    Y_hat = model(X).squeeze()
    return ((Y_hat - Y) ** 2).mean()
```

# Hypothesis Class

- The “Hypothesis Class” for our ERM approximation is linear in this case.
- Anything with differentiable parameters is a sub-class of the `nn.Module` in Pytorch. Special case of Neural Networks
- In this case, we can just use a prebuilt linear approximation from $\mathbb{R}^M \to \mathbb{R}^1$ without an affine constant term.
- The underlying parameters will have a random initialization, which becomes **crucial** with overparameterized models (but wouldn’t be important here)

```
model = nn.Linear(M, 1, bias=False)  # random initialization
print(model)
```

```
Linear(in_features=2, out_features=1, bias=False)
```

# Optimizer

- First-order optimizers take steps using gradient estimates
- In many ML applications you will want control over the process, so will manually call the function to collect the gradient estimate then call the optimizer to take the next step
- Here we will just use SGD with a fixed learning rate
- Note that the optimizer is constructed to look directly at the `parameters` for your underlying model(s)

```
optimizer = optim.SGD(
    model.parameters(), lr=0.001
)
print(optimizer)
```

```
SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

# Training Loop

- Finally, we can loop for multiple "epochs" of passes through the data with the dataloader, calling the optimizer to update each time
- The `for ... in train_loader:` will repeat until the end of the data and continue to the next epoch (i.e., pass through data)
- Each batch updates a `step` using the optimizer, which is unaware of epochs/batches/etc.

```
for epoch in range(300):
    for X_batch, Y_batch in train_loader:
        optimizer.zero_grad()
        loss = residuals(model, X_batch, Y_batch)  # primal
        loss.backward()  # backprop/reverse-mode AD
        # Now the model.parameters have gradients updated, so...
        optimizer.step()  # Update the optimizers internal parameters
print(f"||theta - theta_hat|| = {torch.norm(theta - model.weight.squeeze())}")
```

```
||theta - theta_hat|| = 8.908539894036949e-05
```

# More Pytorch Linear Regression Examples

# Gradient Descent

- See examples/linear_regression_pytorch_gd.py
- Simulates data and shows the basic training loop
- Using the “full batch” to calculate the residuals

# Stochastic Gradient Descent

- See examples/linear_regression_pytorch_sgd.py
- This takes the existing code, but adds in code to calculate gradient estimates using minibatches
- Note that this is creating batches by shuffling the data and the going through it `batch_size` chunks at a time
- When it gets to the end of that data, it is the end of the `epoch`

# Adam + Bells and Whistles

- See examples/linear_regression_pytorch_adam.py
- This extends the previous version and adds in the full train/val/test datasplit
  - → The `val_loss` is collected and displayed at the end of each epoch
- It also shows a learning rate scheduler and a few utilities for logging and early stopping

# Logging with Weights and Biases

- See examples/linear_regression_pytorch_logging.py
- This adds in support for Weights and Biases, and also demonstrates the use of a custom `nn.Module` for the hypothesis class
    1. Go to wandb.ai and create an account, ideally linked to your github
    2. Ensure you have installed the packages with `pip install -r requirements.txt`
    3. Run `wandb login` in terminal to connect to your account
- You will then be able to run these files and see results on wandb.ai

# Sweeps with W&B

- See **examples/linear_regression_pytorch_sweep.yaml** for a **sweep file** which provides a HPO experiment to run and log
  - → Executes variations on `--model.batch_size=32` and `--model.lr=0.001` etc
  - → Tries to choose them to minimize the `val_loss` as a HPO objective
  - → First, create the sweep, `wandb sweep lectures/examples/linear_regression_pytorch_sweep.yaml`
  - → Then run `wandb agent <sweep_id>` with returned sweep id
  - → Call `wandb agent <sweep_id>` on multiple computers to run in parallel
- Any CLI implementation can be used with these sorts of frameworks