[544] PyTorch Optimization

Tyler Caraza-Harter

Learning Objectives

- write a PyTorch optimization loop to find inputs that minimize/optimize an output
- frame model training as an optimization problem minimizing loss
- prepare datasets using DataSet and DataLoader from sources like CSVs

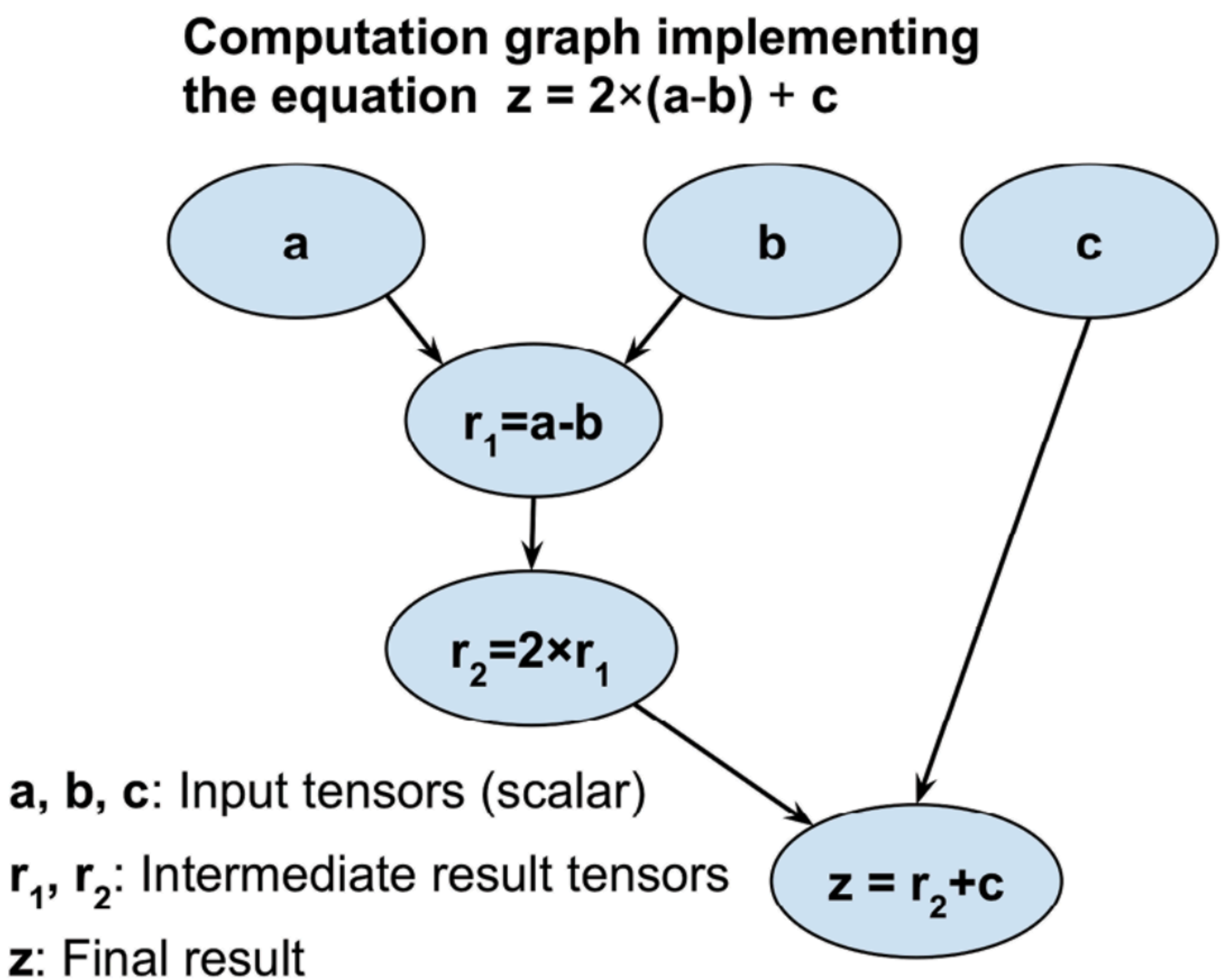
Outline

Optimization

- Calculations as DAGs
- Iterative approach

Machine Learning

- Brief background
- Machine Learning as Optimization



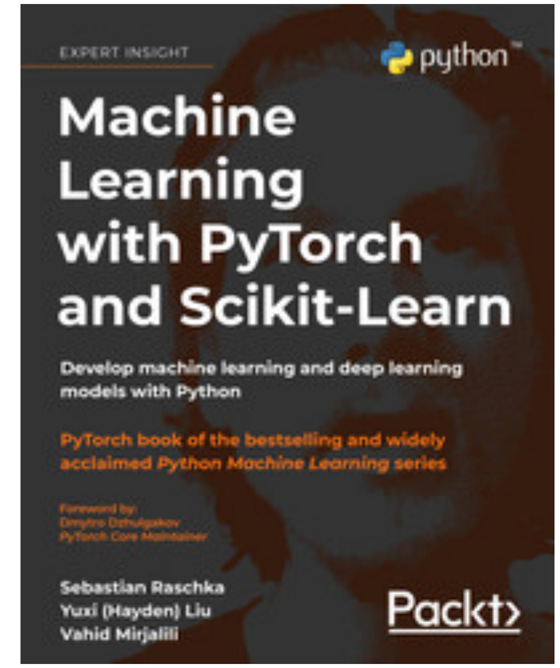


Figure 13.1: How a computation graph works

PyTorch can calculate how small changes in one variable in the DAG impacts another.
Example: if b *increases* by 0.001, z will *decrease* by 0.002. The **gradient** of z with respect to b is -2.

Optimization: if we want z to be large, decreasing b a little (how much?) is probably a good idea.

Making a small improvement

```
a = torch.tensor(3.0)
b = torch.tensor(4.0)
c = torch.tensor(5.0)

z = 2 * (a - b) + c
```

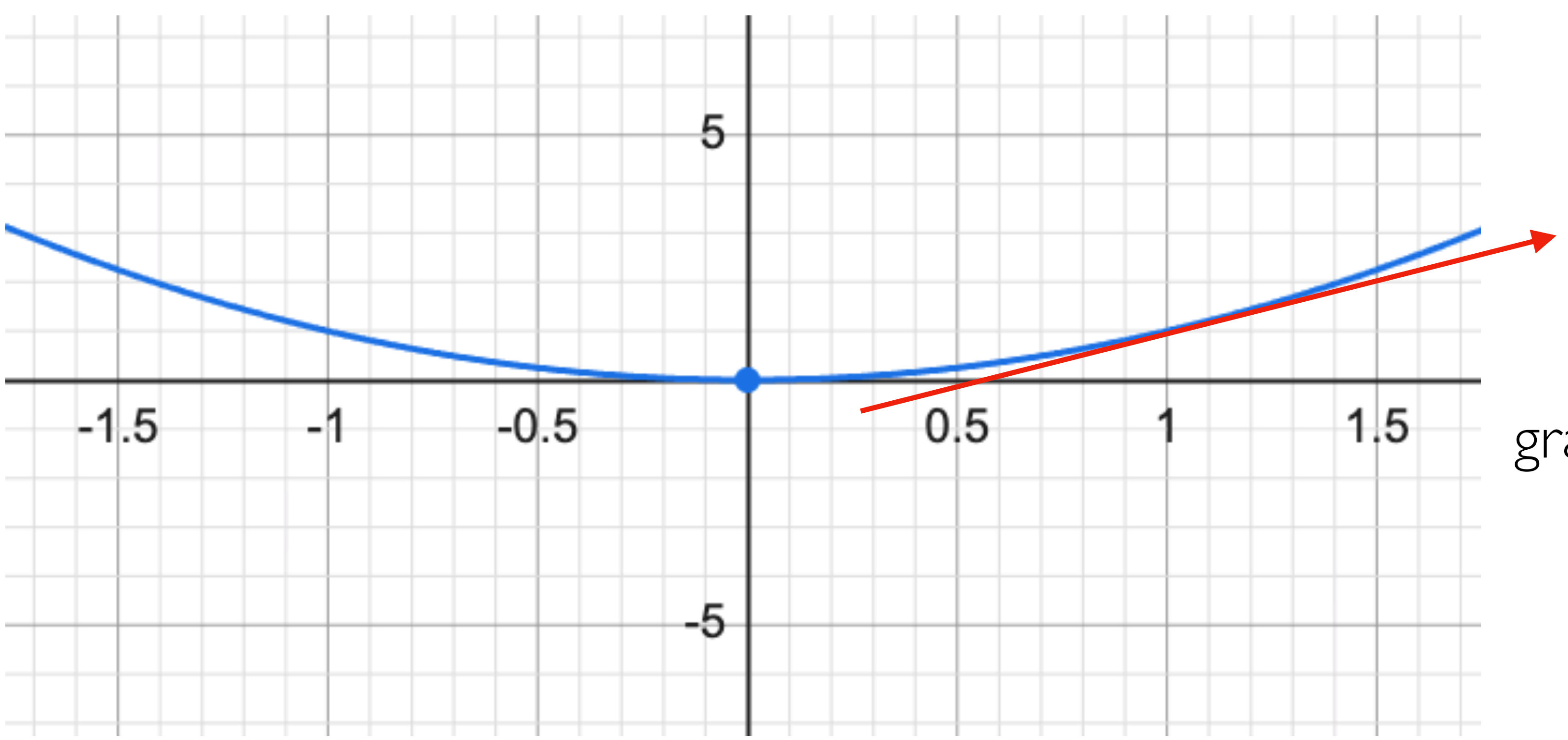
Scenario

- We want z to be large
- We're allowed to change b (but to what?)

Tracking gradients

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

z = 2 * (a - b) + c
```



gradient (slope at a position)
at x=1 is 2

Calculating gradients

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

z = 2 * (a - b) + c
z.backward()
b.grad
```

fill .grad for all input tensors that we're tracking

-2 (because that's the z-over-b slope at the current location)

Accumulating gradients

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

z = 2 * (a - b) + c
z.backward()
b.grad
```

-2

repeat

```
z = 2 * (a - b) + c
z.backward()
b.grad
```

-4

careful, gradients accumulate in .grad everytime you call backward
(has uses, but not usually what we want)

Taking steps

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

optimizer = ????

z = 2 * (a - b) + c
z.backward()
b.grad
```

-2

```
optimizer.step()
```

step() will make b a little bigger or a little smaller,
depending on gradient, and whether we're minimizing or maximizing

Stochastic Gradient Descent (SGD) Optimizer

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)
```

can change b

what z to be big

```
optimizer = torch.optim.SGD([b], maximize=True,
                            lr=0.1)

z = 2 * (a - b) + c
z.backward()
b.grad
```

-2

```
optimizer.step()
```

Learning rate

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

optimizer = torch.optim.SGD([b], maximize=True,
                            lr=0.1)
```

learning rate specifies how much step() should change b

```
z = 2 * (a - b) + c
z.backward()
b.grad
```

-2

```
optimizer.step()
```

```
b += b.grad * lr
     -2     * 0.1 = -0.2
```

now b is 3.8

(use -= if minimizing)

Clearing gradients (to prep for another step)

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

optimizer = torch.optim.SGD([b], maximize=True,
                            lr=0.1)

z = 2 * (a - b) + c
z.backward()
b.grad  →  -2

optimizer.step()
optimizer.zero_grad()

b.grad  →  0
```

Iteratively improving

```
a = torch.tensor(3.0)
b = torch.tensor(4.0, requires_grad=True)
c = torch.tensor(5.0)

optimizer = torch.optim.SGD([b], maximize=True,
                            lr=0.1)

for epoch in range(10):
    z = 2 * (a - b) + c
    z.backward()

    optimizer.step()
    optimizer.zero_grad()

print(b)
```

each iteration of optimization is called an "epoch"

many small improvements have been made

Demos...

Outline

Optimization

- Calculations as DAGs
- Iterative approach

Machine Learning

- Brief background
- Machine Learning as Optimization

Machine Learning, Major Ideas

Categories of Machine Learning:

- **Reinforcement learning**: agent makes series of actions to maximize reword
- **Unsupervised learning**: looking for general patterns
- **Supervised learning:** train models to predict unknowns (today)

Models are functions that return predictions:

```
def my_model(some_info):
    ...
    return some_prediction
```

numeric (1, 2, 3) is "regression"

categorical (A, B, C) is "classification"

Example:

```
def weather_forecast(temp_today, temp_yesterday):
    ...
    return temp_tomorrow
```

Machine Learning, Major Ideas

Categories of Machine Learning:

- **Reinforcement learning**: agent makes series of actions to maximize reword
- **Unsupervised learning**: looking for general patterns
- **Supervised learning:** train models to predict unknowns (today)

Models are functions that return predictions:

```
def my_model(some_info):
    ...
    return some_prediction
```

computation usually involves some calculations (multiply, add) with various numbers (parameters). Training is finding parameters that result in good predictions for known training data

Example:

```
def weather_forecast(temp_today, temp_yesterday):
    ...
    return temp_tomorrow
```

Goal: Learning from Data

	x1	x2	y
0	2	8	5
1	9	2	6
2	4	1	0
3	7	9	7
4	2	2	3
5	3	4	3
6	3	5	9
7	7	1	4
8	6	6	3
9	4	3	?
10	1	2	?
11	2	9	?

how can the cases where we DO know y help us predict the cases where we do not?

Split Known Cases

	x1	x2	y
0	2	8	5
1	9	2	6
2	4	1	0
3	7	9	7
4	2	2	3
5	3	4	3
6	3	5	9
7	7	1	4
8	6	6	3
9	4	3	?
10	1	2	?
11	2	9	?

train

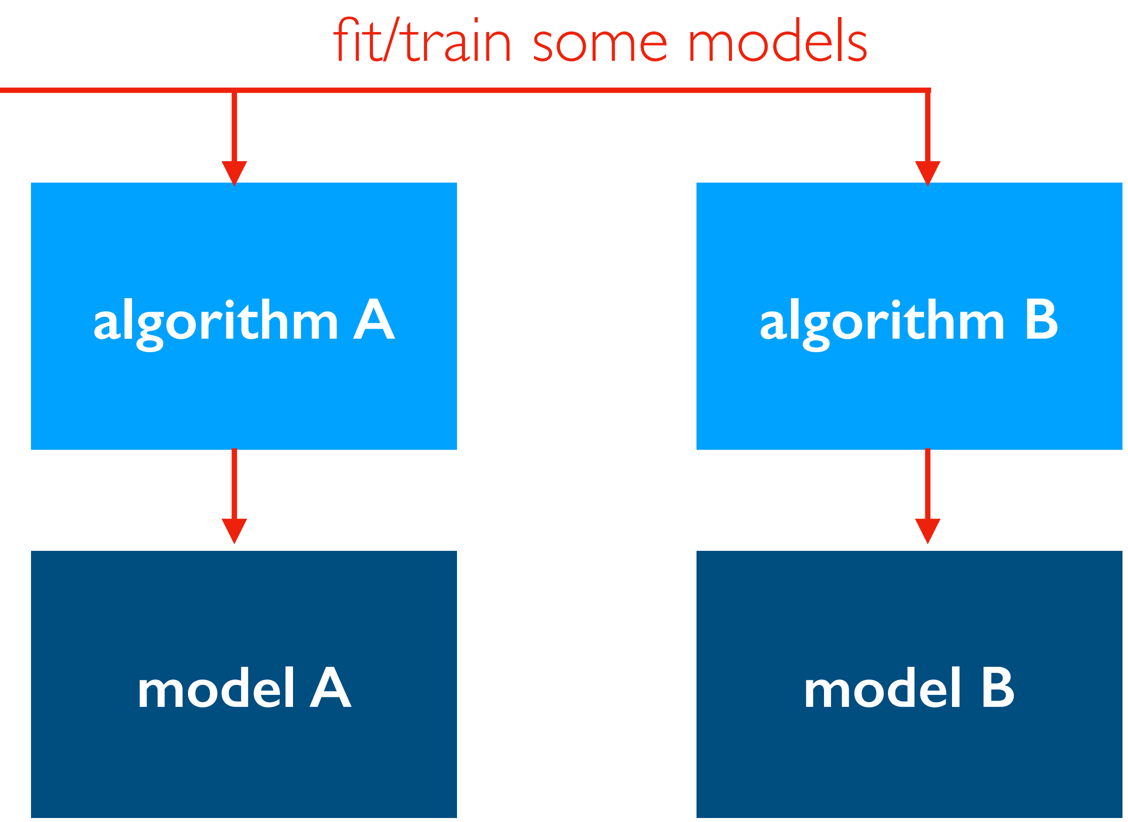
validation

test

random split

Train Models

		x1	x2	y
	0	2	8	5
train	1	9	2	6
	2	4	1	0
	3	7	9	7
validation	4	2	2	3
	5	3	4	3
	6	3	5	9
test	7	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?



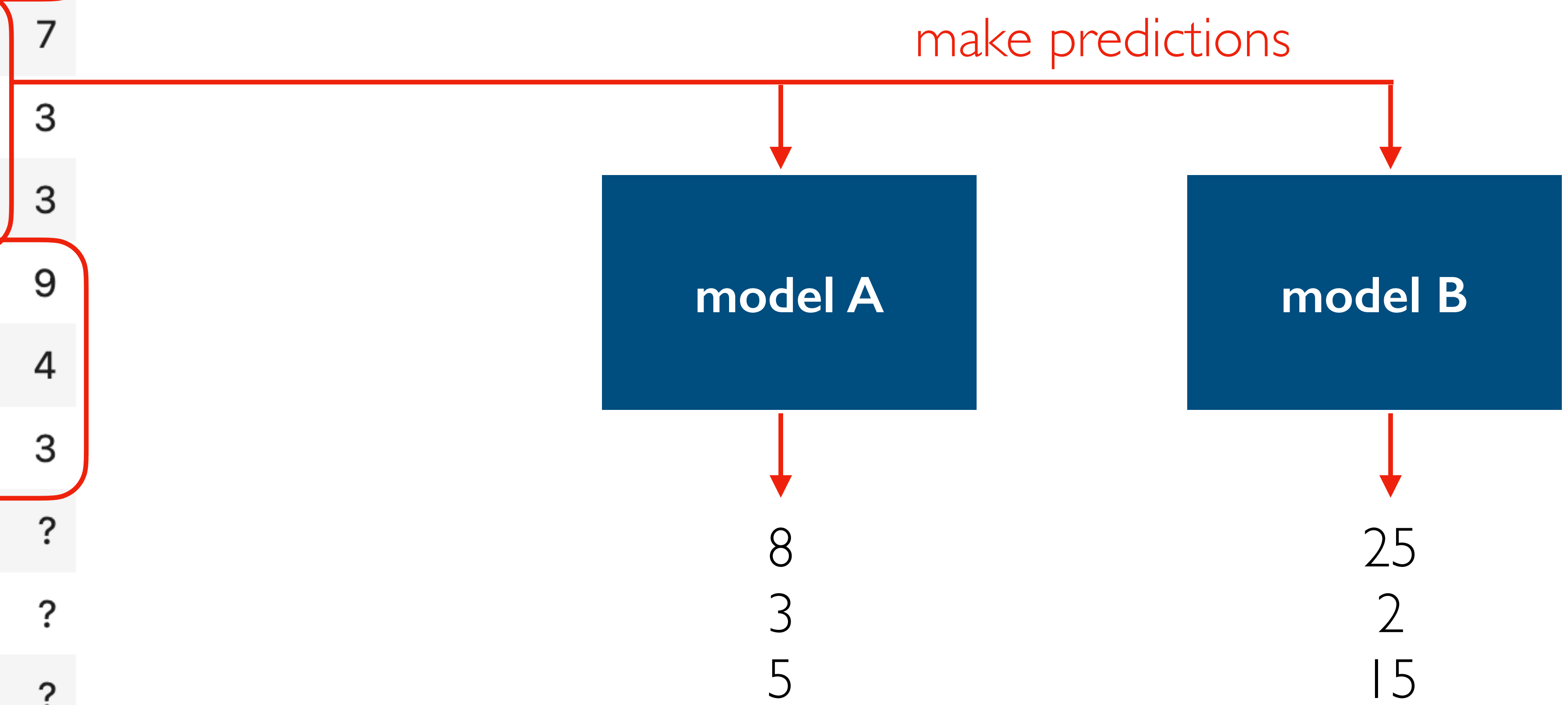
Predict with Models

train

validation

test

	x1	x2	y
0	2	8	5
1	9	2	6
2	4	1	0
3	7	9	7
4	2	2	3
5	3	4	3
6	3	5	9
7	7	1	4
8	6	6	3
9	4	3	?
10	1	2	?
11	2	9	?



Measure Loss

		x1	x2	y
train	0	2	8	5
	1	9	2	6
	2	4	1	0
validation	3	7	9	7
	4	2	2	3
	5	3	4	3
test	6	3	5	9
	7	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?

which model predicts better?

y	p	err	err^2
7	8	1	1
3	3	0	0
3	5	2	4

MSE (mean squared error) is 5/3 = 1.666

Loss functions measure how bad predictions are (MSE is one possible metric)

model A

8
3
5

model B

25
2
15

Choose best model

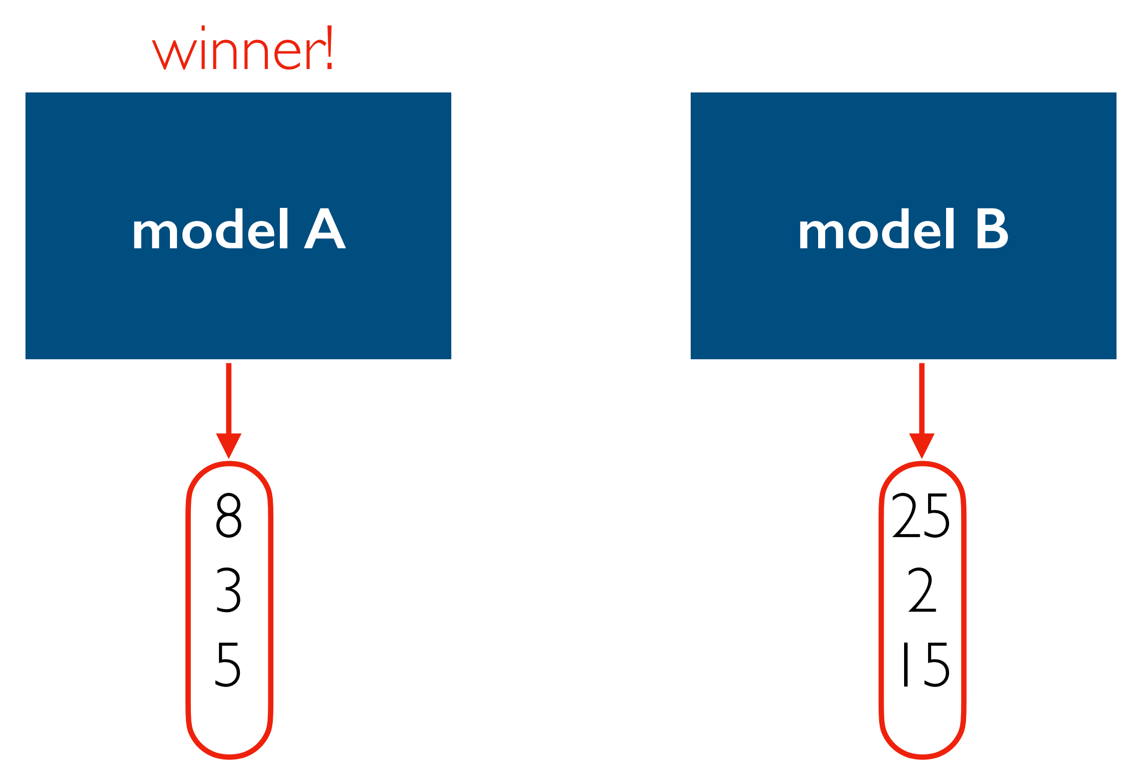
	x1	x2	y
0	2	8	5
1	9	2	6
2	4	1	0
3	7	9	7
4	2	2	3
5	3	4	3
6	3	5	9
7	7	1	4
8	6	6	3
9	4	3	?
10	1	2	?
11	2	9	?

train

validation

test

which model predicts better?



How does the winner do on something new?

		x1	x2	y
train	**0**	2	8	5
	1	9	2	6
	2	4	1	0
validation	**3**	7	9	7
	4	2	2	3
	5	3	4	3
test	**6**	3	5	9
	7	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?

why might the winning model do worse on the test data than the validation data?

winner!

model A

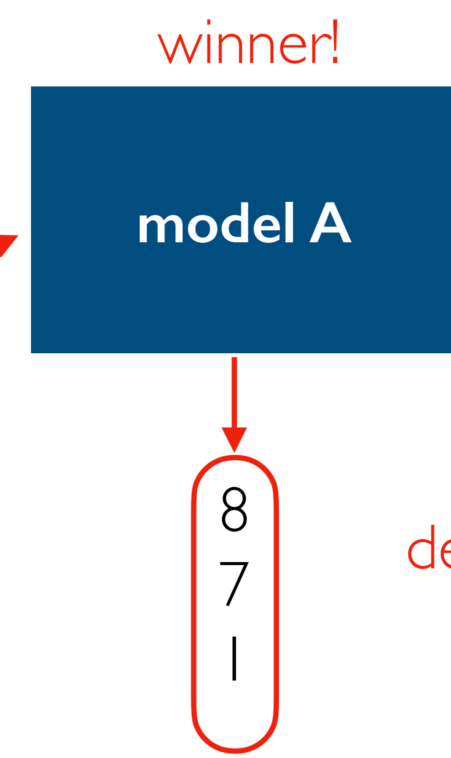
10
3
3

how good does the chosen model do on the test data?

models that do good on train data but bad on validation/test data have "overfitted"

Deploy!

		x1	x2	y
	0	2	8	5
train	**1**	9	2	6
	2	4	1	0
	3	7	9	7
validation	**4**	2	2	3
	5	3	4	3
	6	3	5	9
test	**7**	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?



deploy the model. Use it for predicting real unknowns!

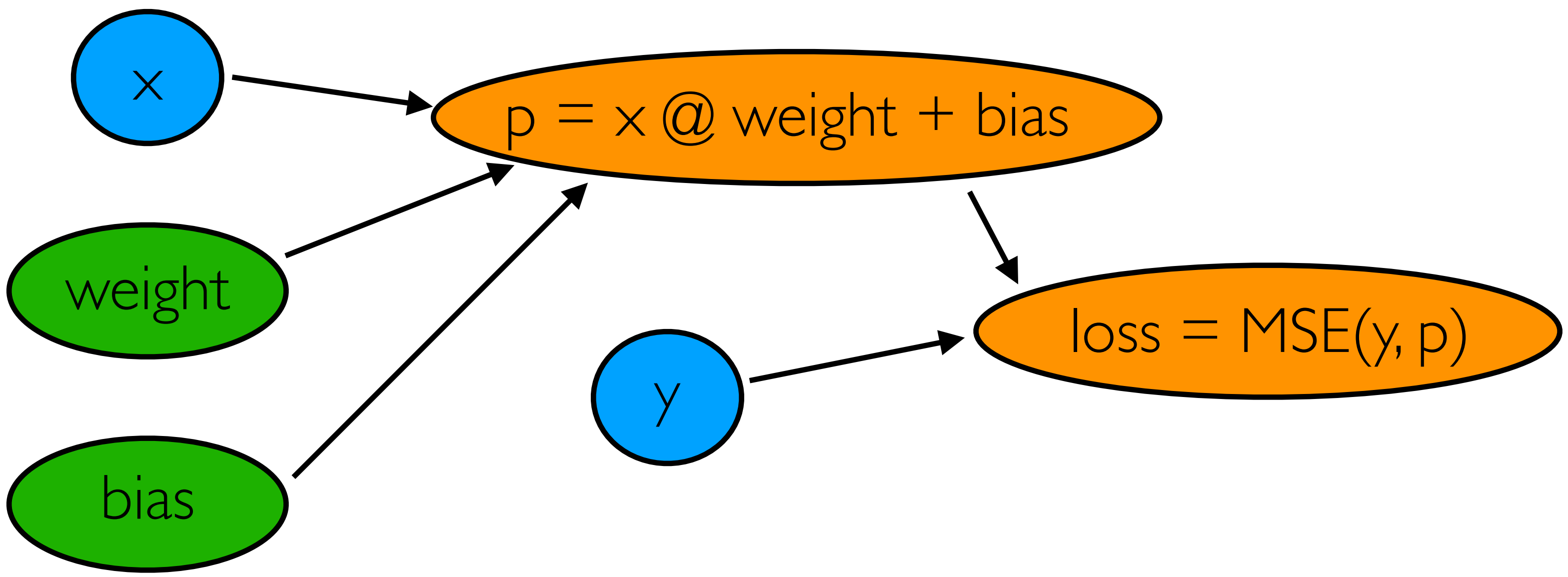
Outline

Optimization

- Calculations as DAGs
- Iterative approach

Machine Learning

- Brief background
- Machine Learning as Optimization



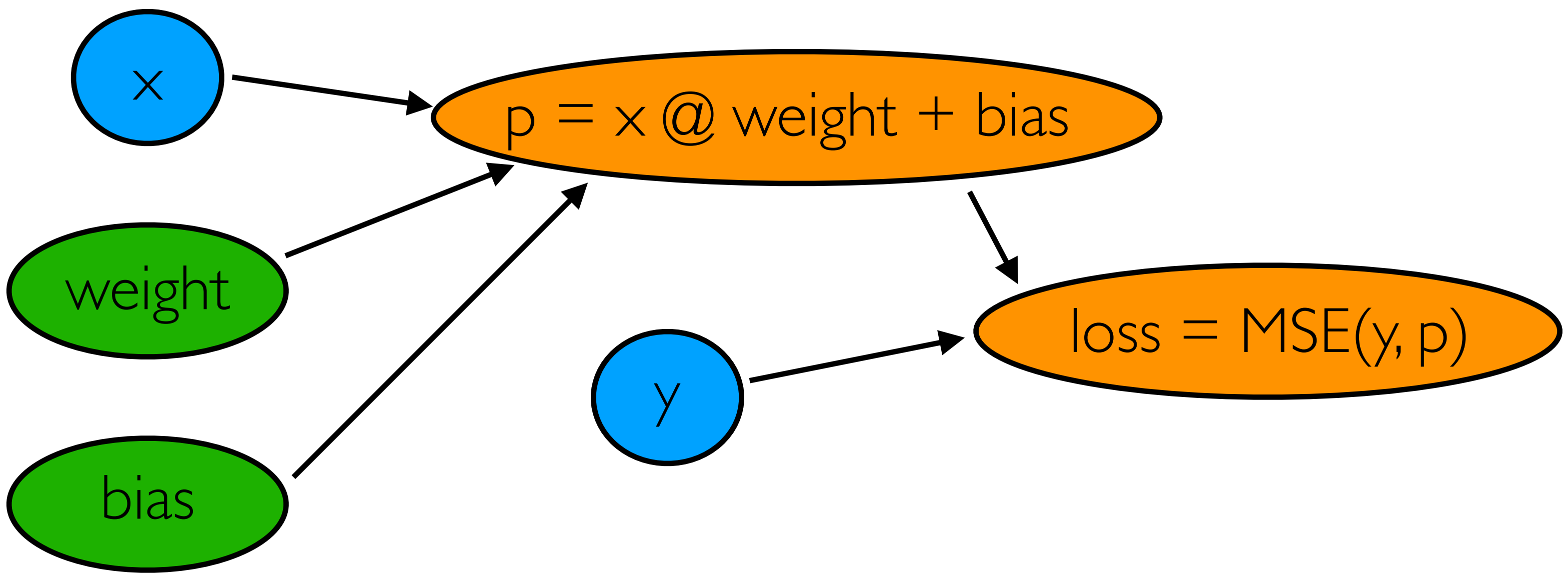
x and y are known (these are matrices/vectors).
what should weight and bias (parameters) be?

```
def model(data):
    return data @ weight + bias

p = model(x)
loss = MSE(y, p)
```

MSE (means squared error) measures how different predictions are from real values, so we want small loss (optimization).

If gradient of loss with respect to weight is positive, then decrease weight.



epoch 0

scan

	x1	x2	y
1	2	3	8
4	8	1	7
0	7	8	0
3	3	5	5
2	9	8	5
5	2	9	0

update params

epoch 1

scan

	x1	x2	y
1	2	3	8
4	8	1	7
0	7	8	0
3	3	5	5
2	9	8	5
5	2	9	0

update params

epoch 2

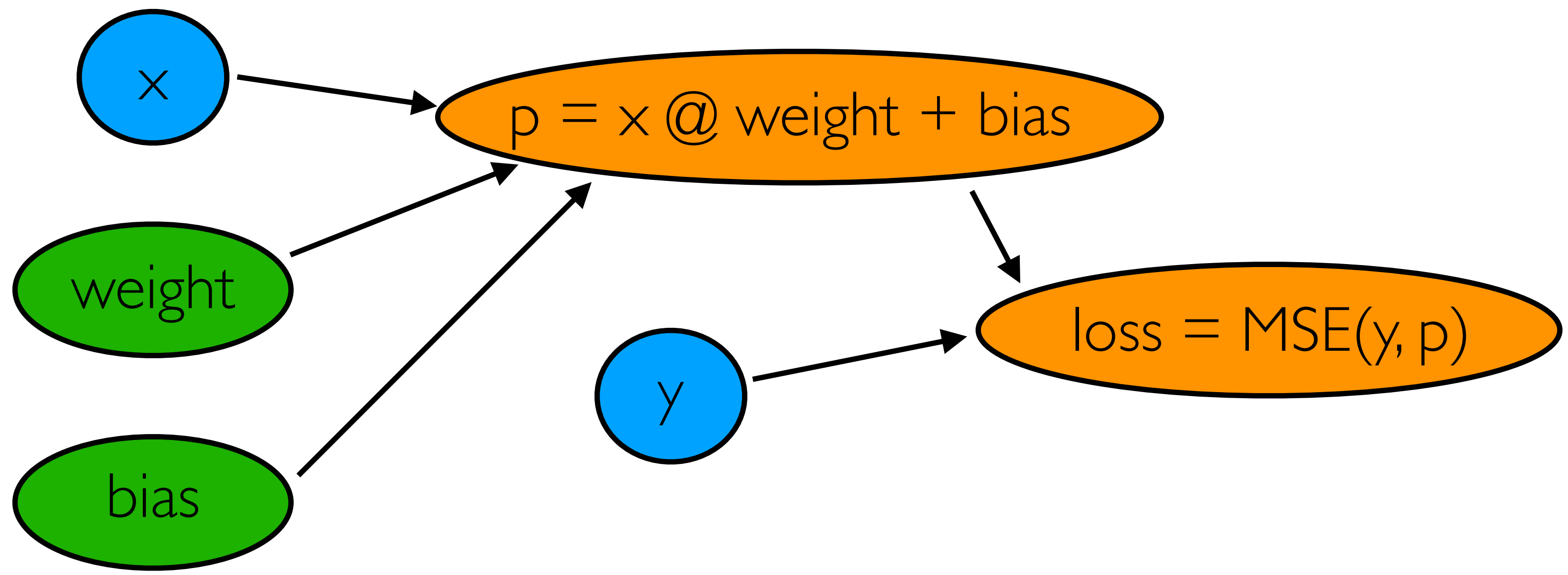
scan

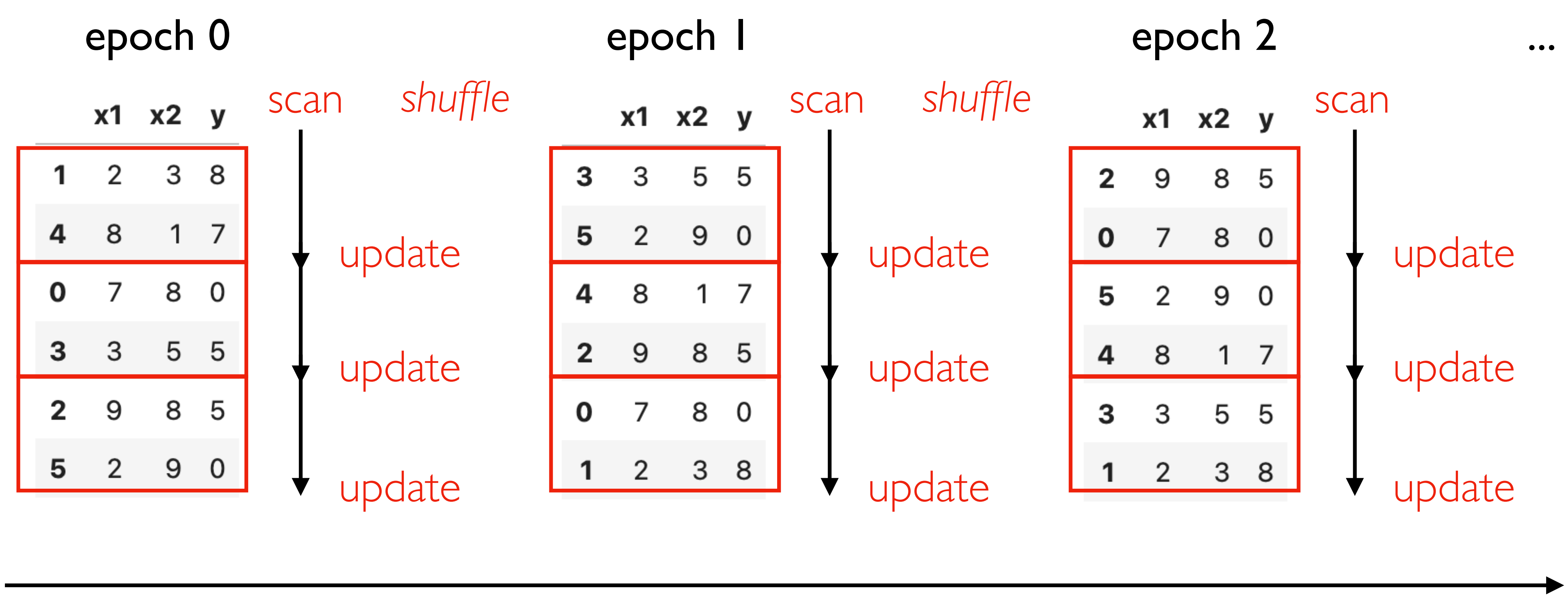
	x1	x2	y
1	2	3	8
4	8	1	7
0	7	8	0
3	3	5	5
2	9	8	5
5	2	9	0

update params

...

gradient descent. slow (consider all data each update), and data might not fit in RAM





stochastic gradient descent. shuffle each time, process in small batches that fit in memory

Demos...