STATS 507
Data Analysis in Python

Week12-1: Loss Function, Gradient, Stochastic Gradient Descent

Dr. Xian Zhang

Recall the scope of this class

Part 1: Introduction to Python

Data types, functions, classes, objects, functional programming

Part 2: Numerical Computing and Data Visualization

numpy, scipy, scikit-learn, matplotlib, Seaborn

Part 3: Dealing with structured data

pandas, SQL, real datasets

Part4: Intro to Deep Learning

PyTorch, Perceptron, Multi-layer perceptron, SGD, regularization, ConvNets

Slide Credit: Jeffrey Regier

Recap: What's PyTorch

It's a Python-based open-sourced scientific computing package targeted at two sets of audiences:

1) A replacement for **NumPy** to use the power of **GPUs**

2) A **deep learning** research platform that provided maximum flexibility and speed

https://pytorch.org/get-started/locally/

Slide Credit: Jeffrey Regier

Recap: Auto differentiation in PyTorch

```
x = torch.ones(2, 2, requires_grad = True)
```

```
x
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

```
y = x + 2
print(y)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

```
y.grad_fn
```

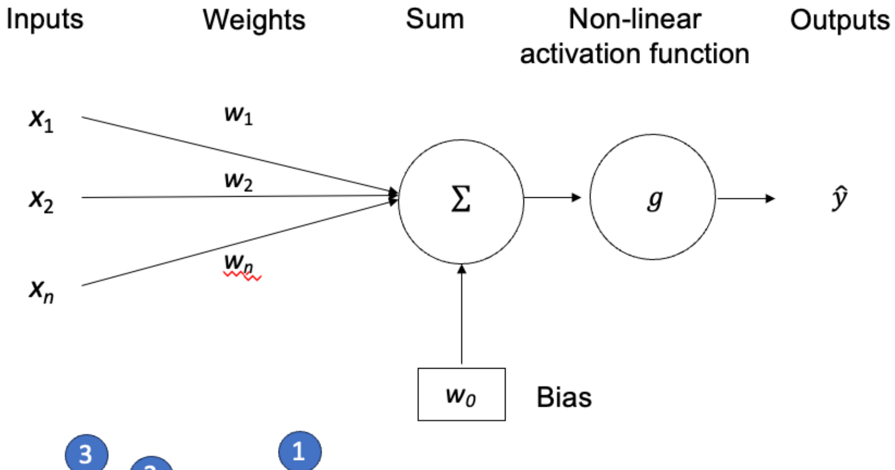
```
<AddBackward0 at 0x14dd86a10>
```

When we call backward(), PyTorch uses these `grad_fns` to compute:

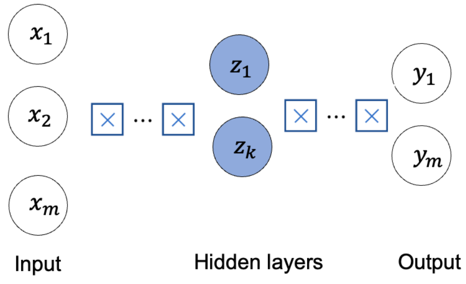
```
x = torch.tensor([1.0], requires_grad=True)
y = x * 2   # grad_fn=<MulBackward0>
z = y + 3   # grad_fn=<AddBackward0>

# When we call backward(), PyTorch uses these grad_fns to compute:
z.backward()
print(x.grad)
```

Slide credit: Christian S. Perone

Recap: The perceptron and DNN



$$\hat{y} = g(w_0 + \sum_{i=1}^{n} x_i w_i) = g(z)$$




```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, num_features, hidden_size1, hidden_size2, num_classes):
        super(MultilayerPerceptron, self).__init__()

        # 1st hidden layer
        self.linear_1 = nn.Linear(num_features, hidden_size1)

        # 2nd linear layer
        self.linear_2 = nn.Linear(hidden_size1, hidden_size2)

        # output layer
        self.linear_out = nn.Linear(hidden_size2, num_classes)

    def forward(self, x):
        x = F.relu(self.linear_1(x))
        x = F.relu(self.linear_2(x))
        logits = self.linear_out(x)
        probas = F.softmax(logits, dim = 1)

        return logits, probas
```

Applying DNN: Will I pass this class?

Real world input | Model input | Model | Model output | Real world output

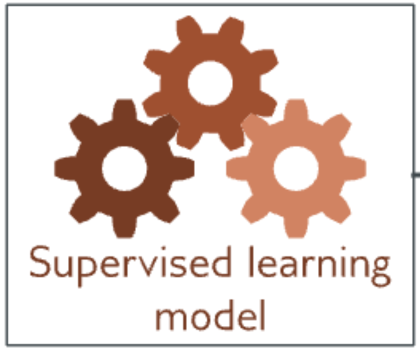
A two layer MLP

Let's start with a simple two feature model

x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

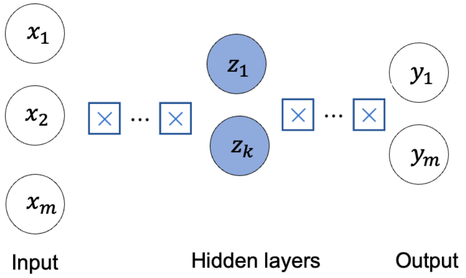


$$\begin{bmatrix} 0.02 \\ 0.98 \end{bmatrix}$$

Positive

Negative

two discrete classes



- Classification or regression?
- Two discrete class-> Binary

1. The Loss Function

2. Training a model

3. DNN in action (in-class practice)

What is loss and what is it for?

A **measure** of how ”correct” (how bad) the model is

Predicted : $f(x^i; \boldsymbol{W})$

Actual : y^i

The **empirical loss** measure the total loss over our entire dataset

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f\left(x^i; \boldsymbol{W}\right), y^i)$$

Also known as:

- objective function
- cost function
- empirical risk

Return a scalar that is smaller when model maps inputs to outputs better

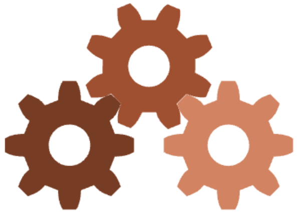
How to construct loss

Over the entire dataset

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\underbrace{f\left(x^i; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^i}_{\text{Actual}})$$

Different learning problem has **different** measures and thus have different loss constructions: **regression**, **classification**, **customized**... (for the scope of this class)

Loss for regression problem

Real world input	Model input	Model	Model output	Real world output
6000 square feet, 4 bedrooms, previously sold for \$235K in 2005, 1 parking spot.	$\begin{bmatrix} 6000 \\ 4 \\ 235 \\ 2005 \\ 1 \end{bmatrix}$	 Supervised learning model	$[340]$	Predicted price is \$340k

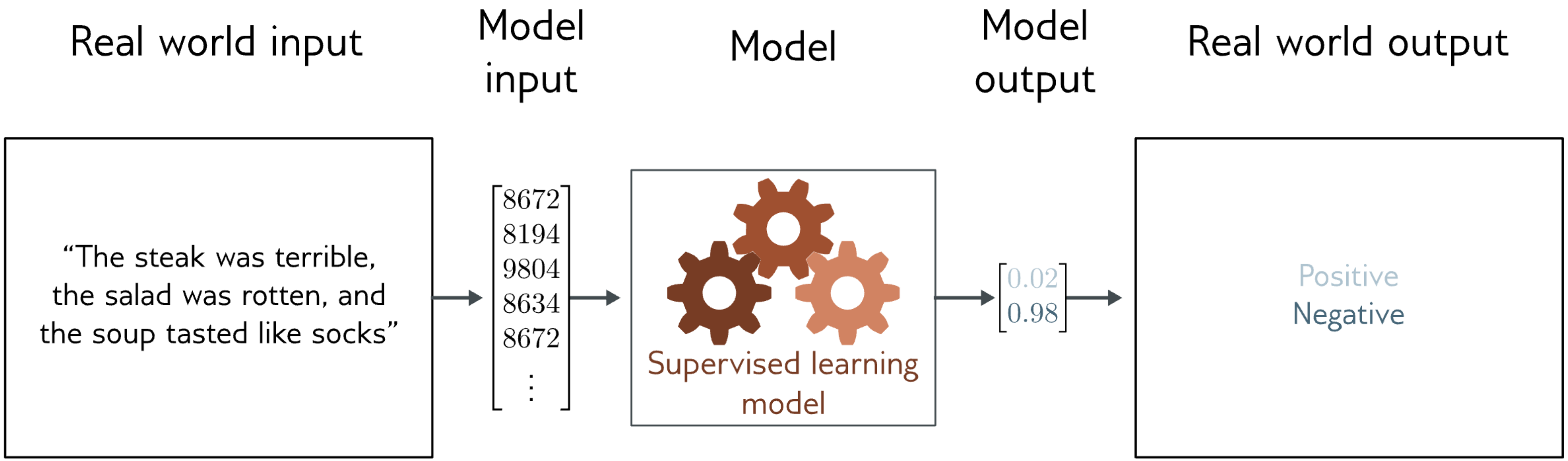
Mean Square Error Loss

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \left(y^i - f\left(x^i; W\right)\right)^2$$

```
mse_loss = nn.MSELoss()
predictions = model(inputs)
loss = mse_loss(predictions, true_outputs)
```

Loss for classification problem

Goal: predict which of the two classes $y \in \{0, 1\}$ the input x belongs to



Binary Cross Entropy Loss

```
loss_func = nn.BCELoss()
predictions = model(inputs)
loss = loss_func(predicited, true_outputs)
```

$$J(W) = -\frac{1}{n} \sum_{i=1}^{n} y^i \log\left(f\left(x^i; W\right)\right) + \left(1 - y^i\right) \log\left(1 - f\left(x^i; W\right)\right)$$

Customized loss function

When standard (built-in) loss functions are not effective we need our own **customized recipe** to define loss functions.

1. Imbalanced data

- Input data
  - penalizing misclassifications of the minority class
- Output data
  - outlier/noise

2. More complicated problem

- Multi-objective learning
- Integrate constraints
- Integrate uncertainty

1. The Loss Function

2. Training a model

3. DNN in action (in-class practice)

What is training

Sometimes also called: fitting the model

Essentially, finding parameters that minimize the loss:

$$J(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f\left(x^i; \boldsymbol{W}\right), y^i)$$

Also formatted as:

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f\left(x^i; \boldsymbol{W}\right), y^i)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

How are we finding parameters: gradient descent

Step1. Compute the derivative of the loss with respect to the parameters

$$\frac{\partial L}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

Step2. Update parameters according to the rule

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial L}{\partial \boldsymbol{W}}$$

Gradient descent as an algorithm

To find the weights:

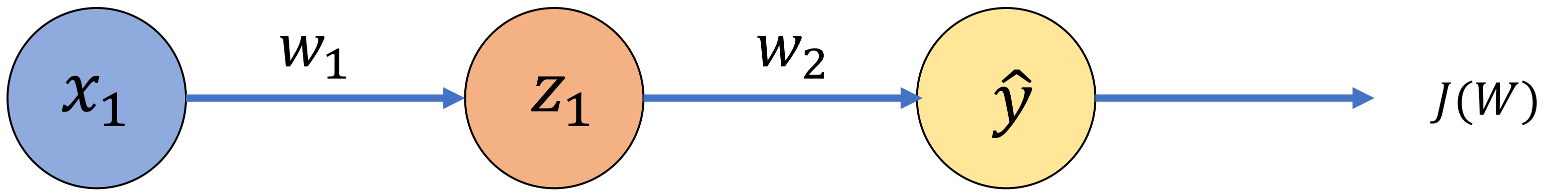
1. Initialize weights randomly $\sim N(0, \sigma^2)$

2. Loop until convergence:
    1. Computer gradient: $\frac{\partial L}{\partial \boldsymbol{W}}$
    2. Update weights: $\boldsymbol{W} \longleftarrow \boldsymbol{W} - \alpha \frac{\partial L}{\partial \boldsymbol{W}}$

3. Return weights

Backpropagation

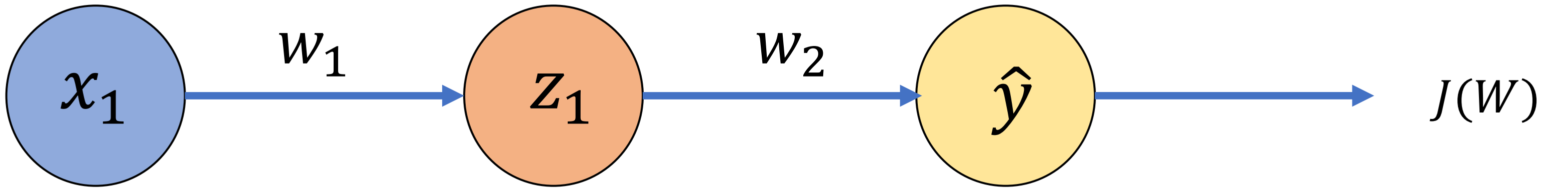
How to compute w_2



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

Backpropagation

How to compute w_1



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Repeat this for very weight in the network using **later layers**

Gradient decent as numerical optimization in PyTorch

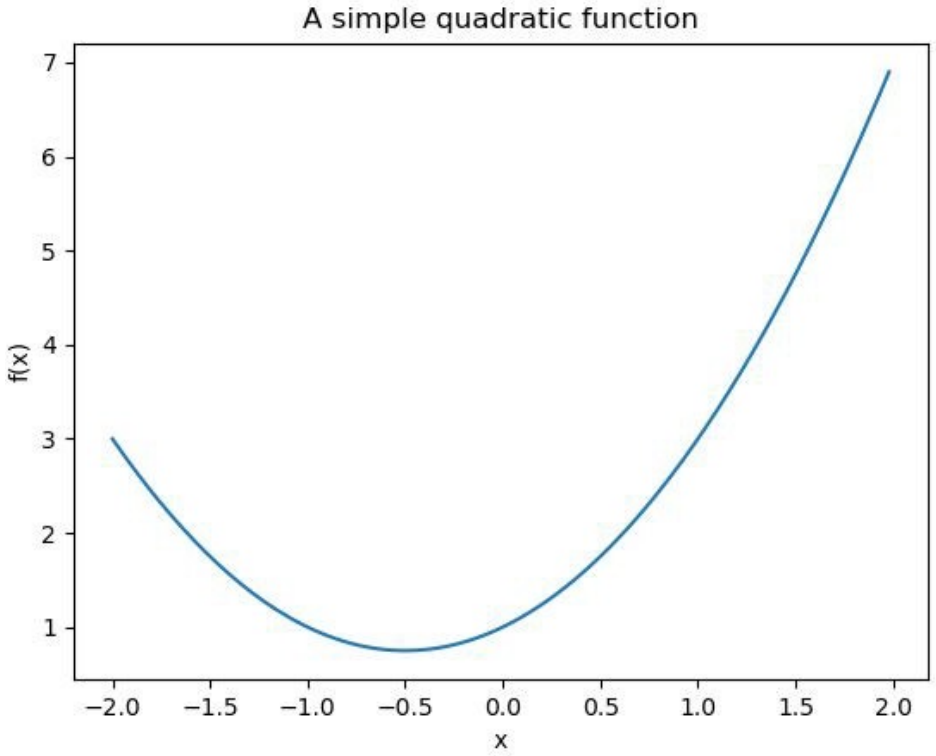
Numerical optimization

Suppose we want to **minimize**

$$f(x) = x^2 + x + 1$$

Easy!

$$\begin{aligned} &\nabla f(x) = 0 \\ \implies\ & 2x + 1 = 0 \\ \implies\ & x = -1/2 \end{aligned}$$



Numerical optimization

Now consider

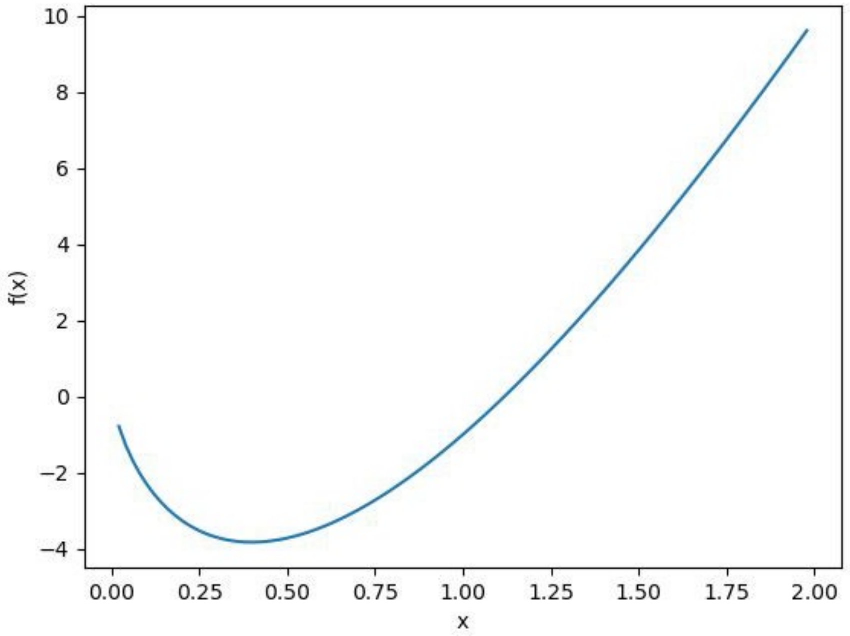
$$f(x) = -x^2 + 10x\log(x)$$

We know

$$\nabla f(x) = -2x + 10\log(x) + 10$$

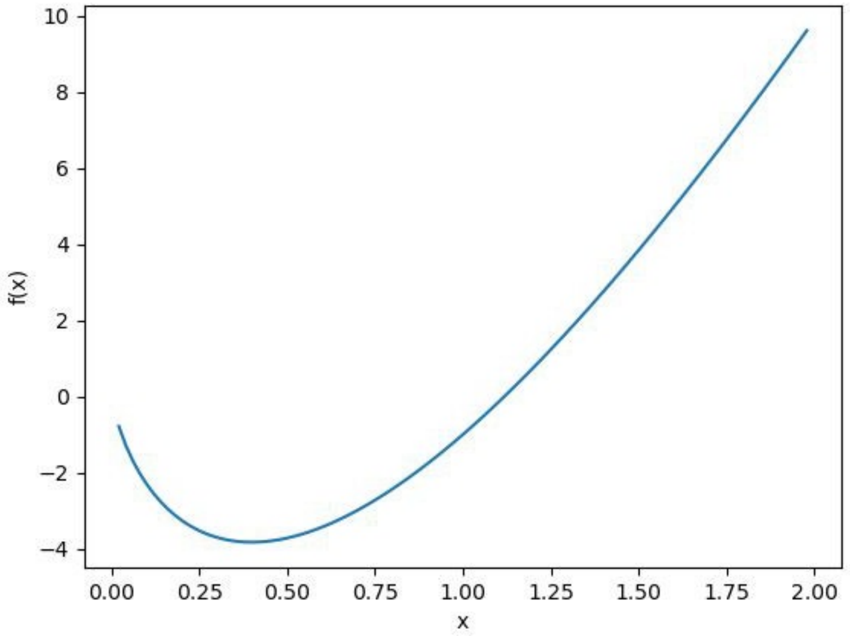
But what value of x sets

$$\nabla f(x) = 0 \quad ?$$



`backward()` evaluates gradient at current x

In [82]:
```
import torch

def f(x):
    return -x**2 + 10 * x * x.log()

x = torch.ones(1, requires_grad=True)
y = f(x)
y.backward()
x.grad
```
Out[82]:
```
tensor([8.])
```



Use auto-differentiation of PyTorch, and the gradient is always evaluated at the current value of x.

Iterative gradient-based solvers

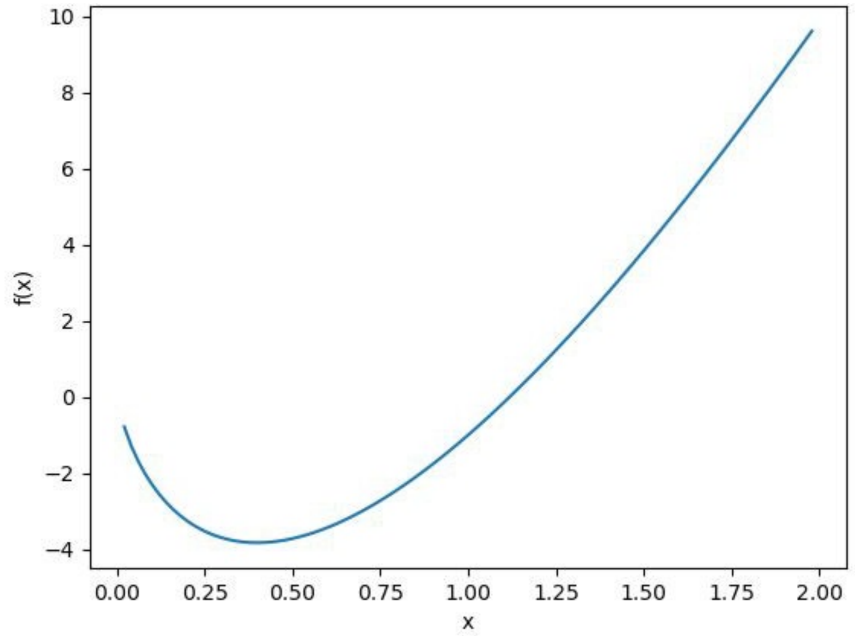
```
for i in range(20):
    x.grad.zero_()
    y = f(x)
    y.backward()
    print("x=%.3f  fx=%.3f  dfdx=%.3f" % (x, y, x.grad))
    step_size = 0.02
    with torch.no_grad():
        x -= step_size * x.grad
```

`x.grad.zero()` clears (set to zero) the gradients from previous backward passes. Without clearing, gradients would accumulate (add up) across iterations

Need to make sure x.grad is not `NONE`.

```
x=1.000  fx=-1.000  dfdx=8.000
x=0.840  fx=-2.170  dfdx=6.576
x=0.708  fx=-2.944  dfdx=5.137
x=0.606  fx=-3.404  dfdx=3.775
x=0.530  fx=-3.645  dfdx=2.595
x=0.478  fx=-3.756  dfdx=1.669
x=0.445  fx=-3.801  dfdx=1.012
x=0.425  fx=-3.817  dfdx=0.587
x=0.413  fx=-3.823  dfdx=0.330
x=0.406  fx=-3.824  dfdx=0.182
x=0.403  fx=-3.825  dfdx=0.099
x=0.401  fx=-3.825  dfdx=0.054
x=0.400  fx=-3.825  dfdx=0.029
x=0.399  fx=-3.825  dfdx=0.016
x=0.399  fx=-3.825  dfdx=0.008
x=0.399  fx=-3.825  dfdx=0.005
x=0.398  fx=-3.825  dfdx=0.002
x=0.398  fx=-3.825  dfdx=0.001
x=0.398  fx=-3.825  dfdx=0.001
x=0.398  fx=-3.825  dfdx=0.000
```

To update x in-place, we need to temporarily disables gradient tracking



In-class practice

Minimizing our function with gradient descent using PyTorch

Stochastic Gradient descent

1. Initialize weights randomly $\sim N(0, \sigma^2)$

2. Loop until convergence:
   1. Pick a single data point
   2. Computer gradient: $\boxed{\frac{\partial L}{\partial \boldsymbol{W}}}$
   3. Update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial L}{\partial \boldsymbol{W}}$

3. Return weights

$\frac{\partial L_i}{\partial \boldsymbol{W}}$

Can be very computationally expensive

Stochastic Gradient descent

1. Initialize weights randomly $\sim N(0, \sigma^2)$

2. Loop until convergence:
   1. Pick batch of **B** data points
   2. Computer gradient: $\boxed{\frac{\partial L}{\partial \boldsymbol{W}}}$ $\quad \frac{\partial L}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{i=1}^{B} \frac{\partial L_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
   3. Update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial L}{\partial \boldsymbol{W}}$

3. Return weights

Setting the learning rate

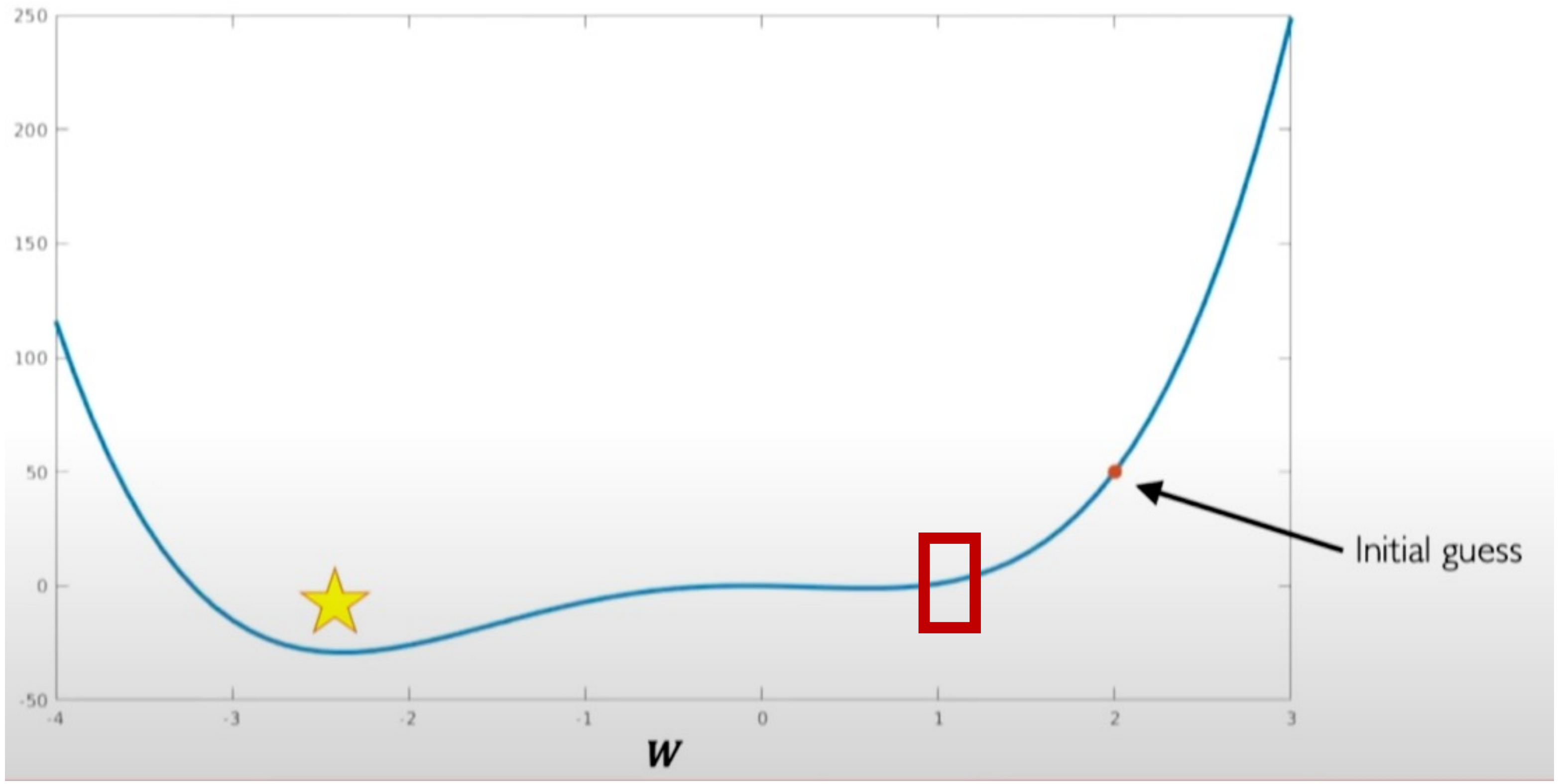
Remember optimization through gradient descent:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boxed{\alpha} \frac{\partial L}{\partial \boldsymbol{W}}$$

How can we set this learning rate?

Setting the learning rate

Small learning rate converges slowly and gets stuck in false local minima

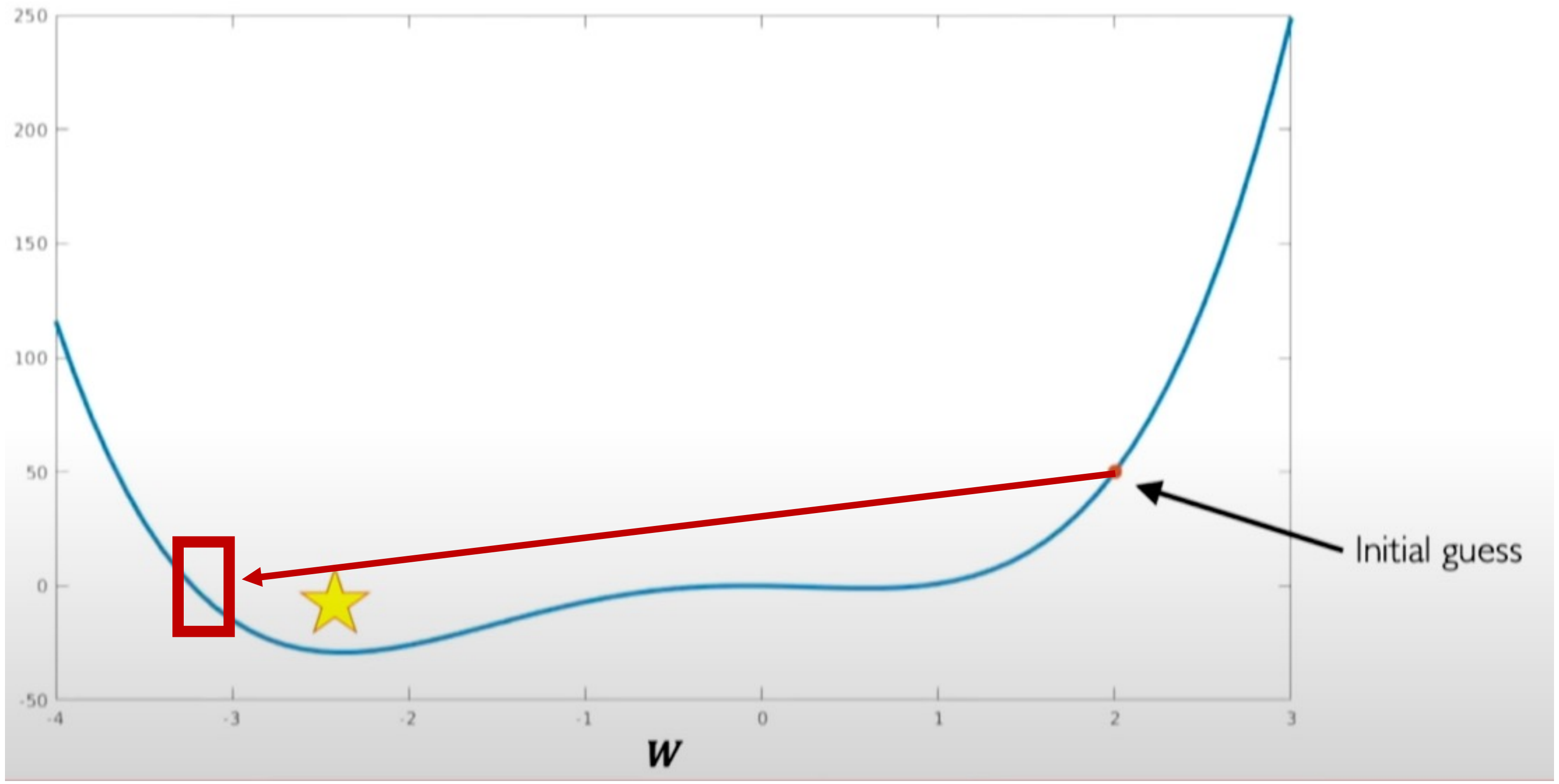


Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

Setting the learning rate

Large learning rate overshoot, become unstable and diverge.



Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

Setting the learning rate

Stable learning rates converges smoothly and avoid local minima

How? - You set the learning rate

1. Try lots of different learning rates and see what work "just right"

2. Something smarter: design an adaptive rate that "**adapts**" to the landscape.

Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

Gradient Descent Algorithms

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adafactor`	Implements Adafactor algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

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

Ref: https://www.geeksforgeeks.org/gradient-descent-algorithm-and-its-variants/

https://pytorch.org/docs/stable/optim.html#module-torch.optim

Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

Put it all together: training our first DNN

In-class practice

Will I pass Stats 507? – A Simple DNN

Other things

HW7 due this week.

HW8 out.

Final project guideline out (**start early**)

Coming next:

Introduce batch size with DataLoader

Deep Sequential model (RNN, LSTM, Transformer…)