
CS 232: Artificial Intelligence

Fall 2026

Prof. Carolyn Anderson
Wellesley College

Reminders

- ◆ Please install PyTorch so I can help you with any issues!
- ◆ My help hours: today 3-4 in W423
- ◆ No class next week!
- ◆ Aeka and Amber will hold usual help hours next week (Thursday and Friday)
- ◆ HW 5 released - due Oct. 19th
- ◆ Progress update emails coming soon (will include count of late days used)

Recap:

Logistic Regression Classifiers

How to do classification

For each feature x_i , weight w_i tells us importance of x_i

- (Plus we'll have a bias b)

We'll sum up all the weighted features and the bias

$$z = \left(\sum_{i=1}^n w_i x_i \right) + b$$

$$z = w \cdot x + b$$

$$P(y = 1) = \sigma(w \cdot x + b)$$

$$P(y = 0) = 1 - \sigma(w \cdot x + b)$$

If this sum is
high, we say $y=1$
If low, then $y=0$

Classifying sentiment for input x

count for words
/

Weights: [2.5, -2.5, -3, 4, 0.1]

Bias: 0.2

/
count of neg words

count of pos words
/

Features for radish sandwich review: [2, 0, 1, 0, 3.33]

$$P(y=1) = \sigma(w \cdot x + b) \\ = 0.926$$

↑
 $\log(\text{length})$

$$P(y=0) = 1 - \sigma(w \cdot x + b) \\ = 0.074$$

Really wanted to love this sandwich. Of course, I love the butter and the baguette (and the salt), but the radishes just don't do anything for me.

The two phases of logistic regression

Test: Given a test example x , we compute $p(y | x)$ using learned weights w and b , and return whichever label ($y = 1$ or $y = 0$) is higher probability

Training: we learn weights w and b using stochastic gradient descent and cross-entropy loss.

optimization

loss function

Logistic Regression Example: Pet Picture Classification

Goal: Classify Pet Pictures

- ◆ Dataset: cat + dog pictures
- ◆ Goal: classify a picture as either a cat or a dog
- ◆ Input: grayscale images

Building a Model

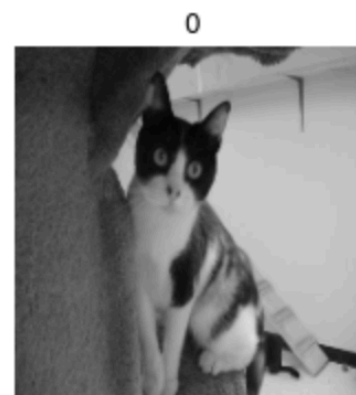
We'll build our model using a machine learning library called **PyTorch**.

PyTorch is a Python library, but most functions are implemented in C (so they are fast!).

PyTorch provides useful abstractions for models:

- ♦ **tensor**: n-dimensional container for data
- ♦ **layer**: apply functions to an input tensor of n dimensions to produce an output tensor of m dimensions.
- ♦ **model**: consist of layers connected together

Example Data



Splitting Our Data

```
class PetImageDataset(Dataset):
    def __init__(self, samples, augment=False):
        self.samples = samples
        self.augment = augment

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, index):
        path, label = self.samples[index]
        with Image.open(path) as image:
            features = prepare_image(image, augment=self.augment)
            # BCELoss expects floating labels shaped like the binary output.
            return features, torch.tensor([label], dtype=torch.float32)

train_dataset = PetImageDataset(train_samples, augment=USE_AUGMENTATION)
val_dataset = PetImageDataset(val_samples)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, num_workers=0)
```

Creating Our Model Architecture

```
class ImageClassifier(nn.Module):
    """A single linear layer for logistic regression on image pixels."""

    def __init__(self, num_features, num_classes=2):
        super().__init__()
        if num_classes == 2:
            self.activation = "sigmoid"
            units = 1
        else:
            self.activation = "softmax"
            units = num_classes
        self.classifier = nn.Linear(num_features, units)

    def forward(self, features):
        outputs = self.classifier(features)
        if self.activation == "sigmoid":
            return torch.sigmoid(outputs)
        return torch.softmax(outputs, dim=1)

def make_model(num_features, num_classes):
    return ImageClassifier(num_features, num_classes)
```

select sigmoid or softmax based on number of classes

of outputs

weights + bias layer - this is the regression bit!

of inputs

Training Function

```
def train(model, data_loader, optimizer, loss_function, device):
    model.train()
    total_loss = 0.0
    total_correct = 0
    total_items = 0
    for features, labels in data_loader:
        features = features.to(device)
        labels = labels.to(device)
        optimizer.zero_grad(set_to_none=True)
        predictions = model(features)
        loss = loss_function(predictions, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * labels.size(0)
        total_correct += ((predictions >= 0.5) == labels.bool()).sum().item()
        total_items += labels.size(0)
    return total_loss / total_items, total_correct / total_items
```

Homework 5

In HW 5, you will build a regression classifier that takes a recipe name and predicts its course category:

- ◆ 0: Main Dishes
- ◆ 1: Desserts
- ◆ 2: Beverages
- ◆ 3: Soups
- ◆ 4: Salads
- ◆ 5: Condiments and Sauces

You will be able to re-use some of the code for training the pet classifier, but you will need to adapt it for a larger number of classes.

How Does Learning Work?

Learning in Supervised Classification

Supervised classification:

- We know the correct label y (either 0 or 1) for each x .
- But what the system produces is an estimate, $\hat{y}$

We want to set w and b to minimize the **distance** between our estimate $\hat{y}^{(i)}$ and the true $y^{(i)}$.

- We need a distance estimator: a **loss function** or a **cost function**
- We need an optimization algorithm to update w and b to minimize the loss.

Learning components

A loss function:

- cross-entropy loss

An optimization algorithm:

- stochastic gradient descent

The distance between $\hat{y}$ and y

We want to know how far is the classifier output:

$$\hat{y} = \sigma(w \cdot x + b) \quad \text{Guessed label}$$

from the true output:

$$y \quad [= \text{either 0 or 1}] \quad \text{True label}$$

We'll call this difference the **loss**:

$$L(\hat{y}, y) = \text{how much } \hat{y} \text{ differs from the true } y$$

Intuition of negative log likelihood loss = cross-entropy loss

A case of conditional maximum likelihood estimation

We choose the parameters w, b that maximize

- the log probability *numerical stability*
- of the true y labels in the training data
- given the observations x

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

$$\text{Maximize: } p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$$

Since there are only 2 discrete outcomes (0 or 1) we can express the probability $p(y|x)$ from our classifier as:

$$\begin{aligned} \text{Maximize: } \log p(y|x) &= \log [\hat{y}^y (1-\hat{y})^{1-y}] \\ &= y \log \hat{y} + (1-y) \log (1-\hat{y}) \end{aligned}$$

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Maximize:
$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Now take the log of both sides (mathematically handy)

Maximize:
$$\begin{aligned} \log p(y|x) &= \log [\hat{y}^y (1 - \hat{y})^{1-y}] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \end{aligned}$$

(Whatever values maximize $\log p(y|x)$ will also maximize $p(y|x)$)

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Minimize the cross-entropy loss

Minimize: $L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \underbrace{\sigma(\hat{y})}_{\sigma(w \cdot x + b)} + (1 - y) \log (1 - \underbrace{\sigma(\hat{y})}_{\sigma(w \cdot x + b)})]$$

Does this work for our sentiment example?

We want loss to be:

- smaller if the model estimate is close to correct
- bigger if model is confused

Let's first suppose the true label of this is $y=0$ (negative)

Really wanted to love this sandwich. Of course, I love the butter and the baguette (and the salt), but the radishes just don't do anything for me.

Let's see if this works for our sentiment example

$$\hat{y} = 0.926$$

True value is y=0. How well is our model doing?

$$\begin{aligned} p(+|x) = P(y = 1|x) &= -[y \log \hat{y} + (1-y) \log (1-\hat{y})] \\ &= -[\cancel{0 \cdot \log \hat{y}} + \cancel{(1-0)} \cdot \log (1-\hat{y})] \\ &= -\log (1-\hat{y}) \\ &= -\log (1-0.926) \\ &= 2.6 \end{aligned}$$

What if the true label was 1?

$$p(+|x) = P(y = 1|x) = 0.926$$

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \hat{y} + (1-y) \log (1-\hat{y})] \\ &= -[1 \cdot \log \hat{y} + \cancel{(1-1) \log (1-\hat{y})}] \\ &= -\log \hat{y} \\ &= -\log (0.926) \\ &= \underline{0.07} \end{aligned}$$

Let's see if this works for our sentiment example

The loss when model was right: *0.07*

Is lower than the loss when model was wrong: *2.6*

Sure enough, loss was bigger when model was wrong!

What About More Than Two Classes?

If >2 classes we use **multinomial logistic regression**:

The probability of everything must still sum to 1

$$P(\text{positive}|\text{doc}) + P(\text{negative}|\text{doc}) + P(\text{neutral}|\text{doc}) = 1$$

Need a generalization of the sigmoid called **softmax**

- Takes a vector $z = [z_1, z_2, \dots, z_k]$ of k arbitrary values
- Outputs a probability distribution

The softmax function

Turns a vector $z = [z_1, z_2, \dots, z_k]$ of k arbitrary values into probabilities :

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^k \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^k \exp(z_i)}, \dots, \frac{\exp(z_k)}{\sum_{i=1}^k \exp(z_i)} \right]$$

$$[0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Features in binary versus multinomial logistic regression

Binary: positive weight

$$x_5 = \begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases} \quad w_5 = 3.0$$

Multinomial: separate weights for each class:

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	-5.3

Learning Components

A loss function:

- **cross-entropy loss**

An optimization algorithm:

- **stochastic gradient descent**

Optimization

Optimization

Goal: find weights + bias (W,b) that minimize loss.

Let's make explicit that the loss function is parameterized by weights $\Theta=(w,b)$

$$\Theta = (w, b)$$

- And we'll represent $\hat{y}$ as $f(x; \theta)$ to make the dependence on θ more obvious

We want the weights that minimize the loss, averaged over all examples:

The diagram shows the equation for finding the optimal weights $\hat{\Theta}$ by minimizing the average loss over all items. The equation is $\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^m L_{ce}(f(x^i; \Theta), y^i)$. Annotations include: 'new weights' pointing to $\hat{\Theta}$; 'minimize' pointing to argmin ; ' Θ ' pointing to the parameter in argmin ; 'all items' pointing to the summation term $\frac{1}{m} \sum_{i=1}^m$; 'loss' pointing to L_{ce} ; 'feature rep.' pointing to $f(x^i; \Theta)$; 'current weights' pointing to Θ inside the function; ' $\hat{y}$ ' pointing to $f(x^i; \Theta)$; and 'true label' pointing to y^i .

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^m L_{ce}(f(x^i; \Theta), y^i)$$

Annotations in the diagram:

- $\hat{\Theta}$: new weights
- argmin : minimize
- Θ (under argmin): Θ
- $\frac{1}{m} \sum_{i=1}^m$: all items
- L_{ce} : loss
- $f(x^i; \Theta)$: feature rep.
- Θ (inside function): current weights
- $\hat{y}$: $\hat{y}$
- y^i : true label

Optimization

Goal: find weights + bias (W, b) that minimize loss.

Let's make explicit that the loss function is parameterized by weights $\Theta=(w, b)$

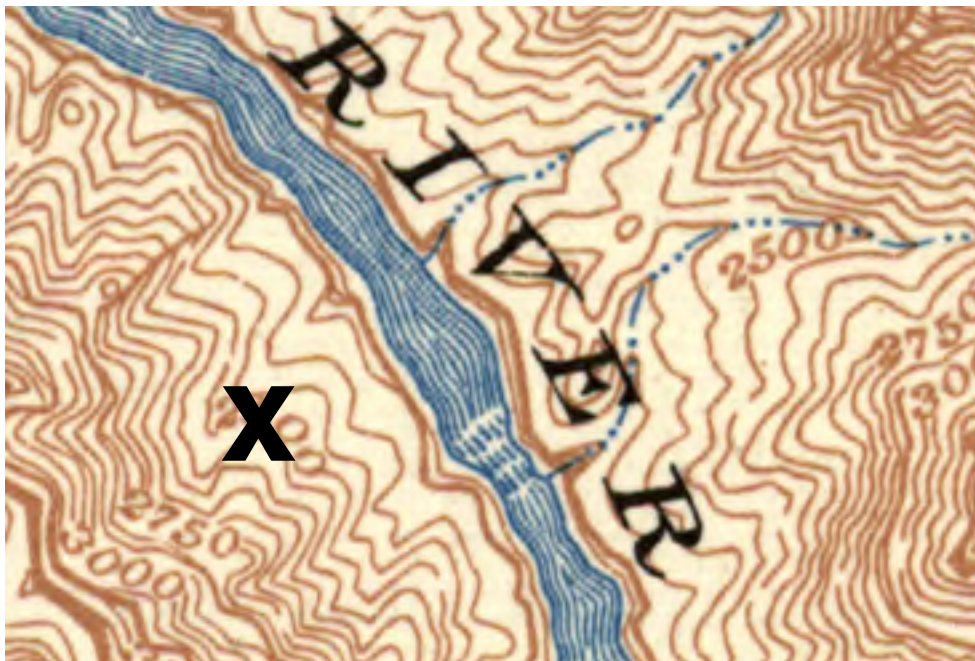
- And we'll represent $\hat{y}$ as $f(x; \theta)$ to make the dependence on θ more obvious

We want the weights that minimize the loss, averaged over all examples:

$$\hat{\theta} = \operatorname{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^m L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

Intuition of gradient descent

How do I get to the bottom of this river canyon?



Look around me 360°

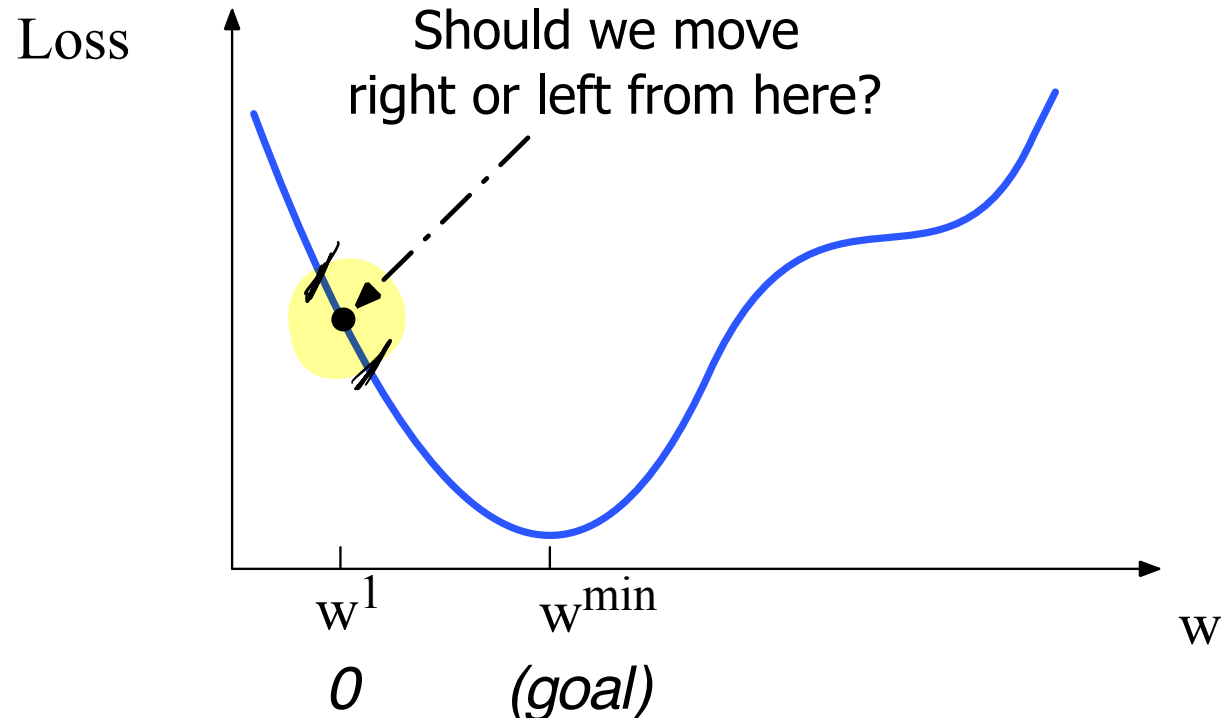
Find the direction of
steepest slope down

Go that way

Our goal: minimize the loss

Q: Given current w , should we make it bigger or smaller?

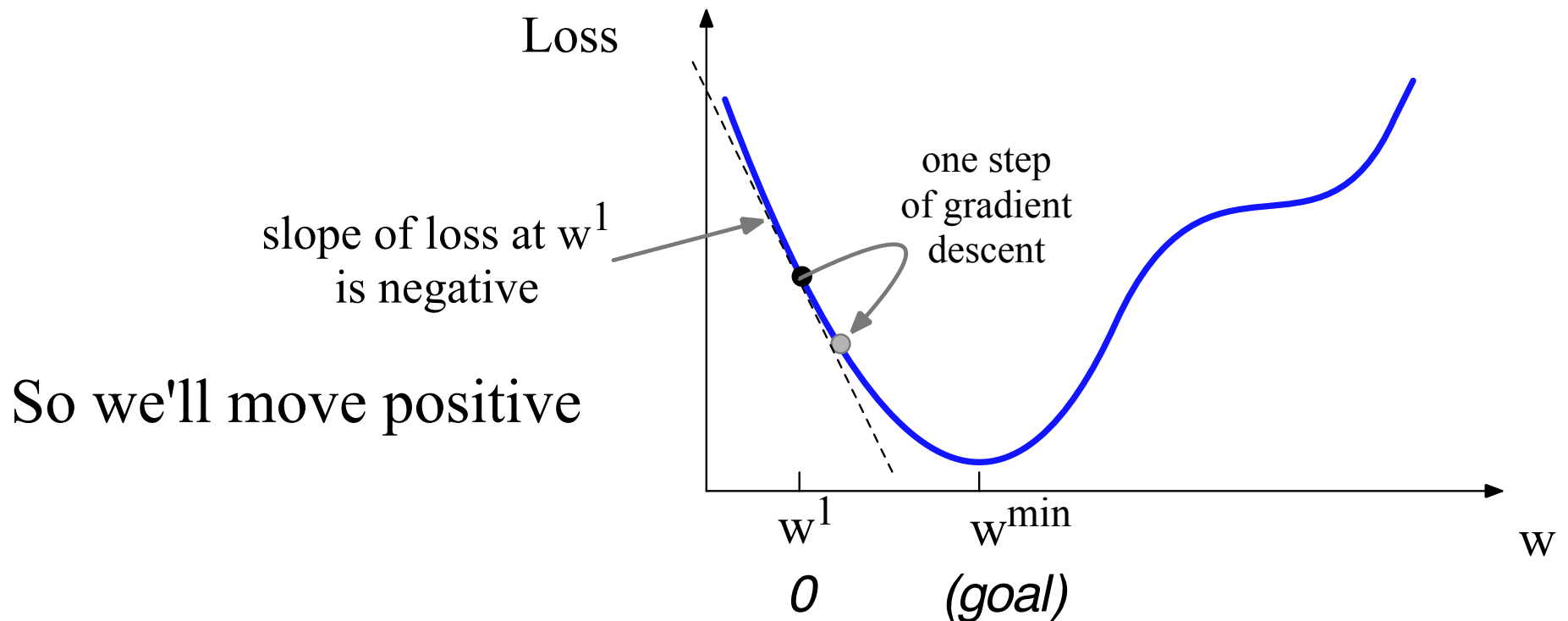
A: Move w in the reverse direction from the slope of the function



Our goal: minimize the loss

Q: Given current w , should we make it bigger or smaller?

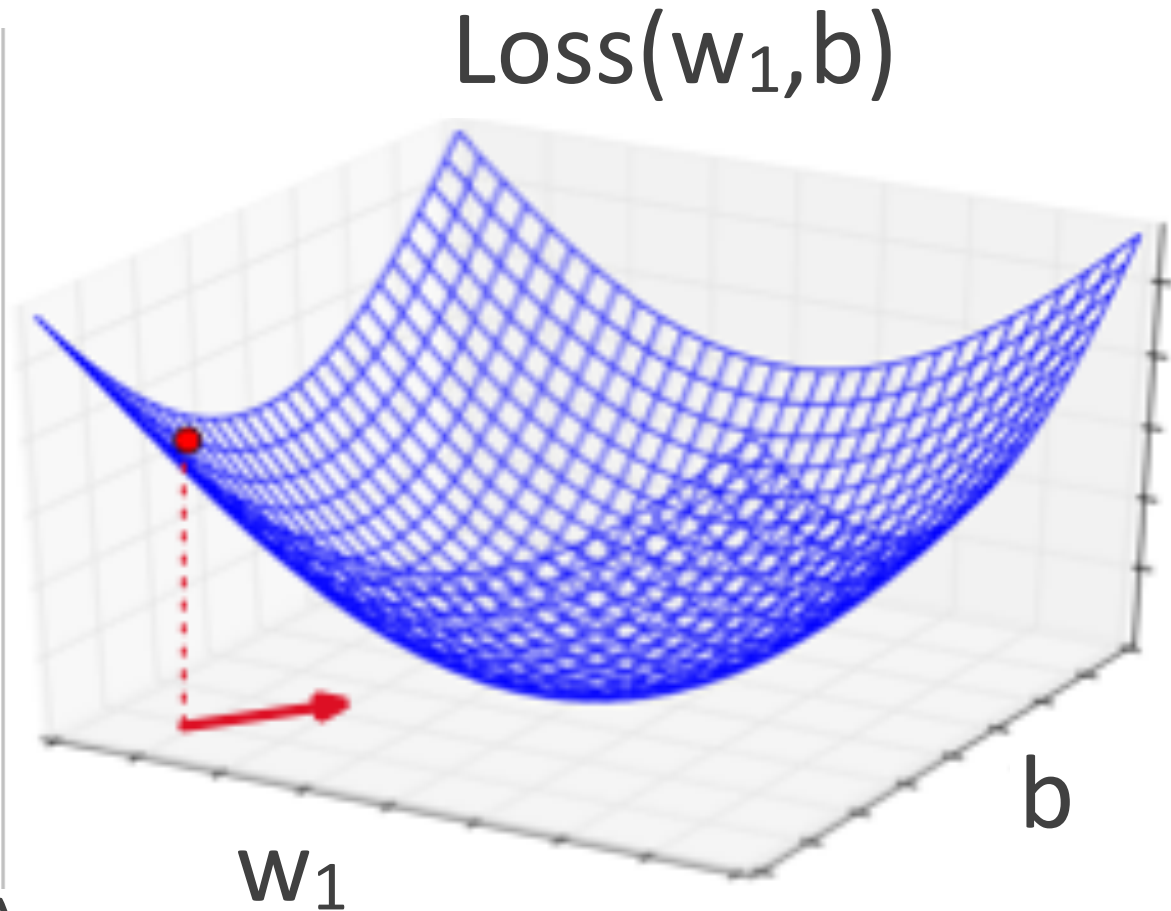
A: Move w in the reverse direction from the slope of the function



Two dimensions: w and b

The **gradient** is a **vector** that expresses the directional components of the sharpest slope along each of the N dimensions.

Here, it has two dimensions (w_1 and b)



Gradients

The **gradient** of a function of many variables is a vector pointing in the direction of the greatest increase in a function.

Gradient Descent: Find the gradient of the loss function at the current point and move in the **opposite** direction.

How much do we move in that direction ?

- The value of the gradient (slope) weighted by a **learning rate** η
- Higher learning rate means we make a bigger change to w at each step

Gradient: $\frac{\partial L(f(x; \theta), y)}{\partial w}$

Weight update:

$$w^{t+1} = w^t - \eta \cdot \frac{\partial L(f(x; \theta), y)}{\partial w}$$

Overfitting

A model that perfectly match the training data has a problem.

It will also **overfit** to the data, modeling noise

- A random word that perfectly predicts y (it happens to only occur in one class) will get a very high weight.

A good model should be able to **generalize**

Training a Supervised Machine Learning Classifier

For n epochs:

- For every x, y in training dataset:
 - Predict y using current weights
 - Compute loss using loss function
 - Update the weights according to loss using optimization algorithm
- Evaluate on validation set
 - Stop training if loss on validation set stops decreasing