CS 4342: Class 4

Jacob Whitehill

Linear regression

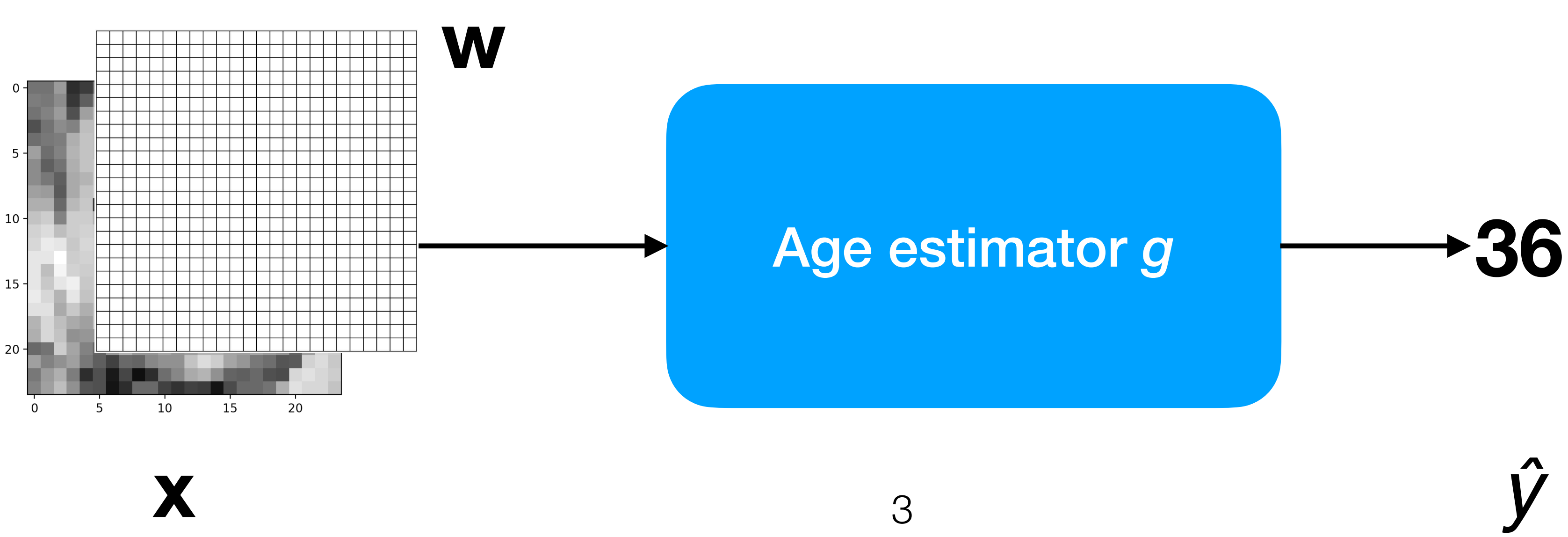
Linear regression

- Linear regression is built as a linear combination of all the inputs **x**:

image pixels

$$\hat{y} = g(\mathbf{x}; \mathbf{w}) = \sum_{j=1}^{m} \mathbf{x}_j \mathbf{w}_j = \mathbf{x}^\top \mathbf{w}$$

- Vector **w** represent an "overlay image" that weights the different pixel intensities of **x**.



Solving for w

- The gradient of f_{MSE} is thus:

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2n} \sum_{i=1}^{n} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \left[\left({\mathbf{x}^{(i)}}^\top \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}^{(i)} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} - y^{(i)} \right)
\end{aligned}
$$

Solving for $\mathbf{w}$

- By setting to 0, splitting the sum apart, and solving, we reach the solution:

$$\frac{1}{n}\sum_{i=1}^{n}\mathbf{x}^{(i)}\left({\mathbf{x}^{(i)}}^\top\mathbf{w} - y^{(i)}\right)$$

$$\begin{aligned}
0 &= \sum_i \mathbf{x}^{(i)}{\mathbf{x}^{(i)}}^\top\mathbf{w} - \sum_i \mathbf{x}^{(i)}y^{(i)} \\
\sum_i \mathbf{x}^{(i)}{\mathbf{x}^{(i)}}^\top\mathbf{w} &= \sum_i \mathbf{x}^{(i)}y^{(i)} \\
\mathbf{w} &= \left(\sum_i \mathbf{x}^{(i)}{\mathbf{x}^{(i)}}^\top\right)^{-1}\sum_i \mathbf{x}^{(i)}y^{(i)}
\end{aligned}$$

Linear regression: matrix notation

- To compute **w**, do *not* use np.linalg.inv.
- Instead, use np.linalg.solve, which avoids explicitly computing the matrix inverse.
- Show age_demo.py.

Linear regression: matrix notation

- Let's define a matrix **X** to contain all the training images:

$$\mathbf{X} = \begin{bmatrix} | & & | \\ \mathbf{x}^{(1)} & \dots & \mathbf{x}^{(n)} \\ | & & | \end{bmatrix}$$

  - In statistics, **X** is called the **design matrix**.

- Let's define vector **y** to contain all the training labels:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

Linear regression: matrix notation

- Using summation notation, we derived:

$$\mathbf{w} = \left(\sum_{i=1}^{n} \mathbf{x}^{(i)} \mathbf{x}^{(i)^\top} \right)^{-1} \left(\sum_{i=1}^{n} \mathbf{x}^{(i)} y^{(i)} \right)$$

- Using matrix notation, we can write the solution as:

$$\mathbf{w} =$$

Linear regression: matrix notation

- Using summation notation, we derived:

$$\mathbf{w} = \left(\sum_{i=1}^{n} \mathbf{x}^{(i)} \mathbf{x}^{(i)^\top}\right)^{-1} \left(\sum_{i=1}^{n} \mathbf{x}^{(i)} y^{(i)}\right)$$

- Using matrix notation, we can write the solution as:

$$\mathbf{w} = \left(\mathbf{X}\mathbf{X}^\top\right)^{-1} \mathbf{X}\mathbf{y}$$

Linear regression: matrix notation

- Once we've "trained" the weights **w**, we can estimate the *y*-value (label) for any **x**.
- We can compute the { $\hat{y}^{(i)}$ } for a set of images { $\mathbf{x}^{(i)}$ } in one-fell-swoop using matrix operations.
- Let's define our design matrix **X** as before:

$$\mathbf{X} = \begin{bmatrix} | & & | \\ \mathbf{x}^{(1)} & \ldots & \mathbf{x}^{(n)} \\ | & & | \end{bmatrix}$$

- Then our estimates of the labels is given by:

$$\hat{\mathbf{y}} = \mathbf{X}^\top \mathbf{w}$$

Linear regression: matrix notation

- Suppose we have *n* images, each with just 2 pixels.

$$\hat{\mathbf{y}} = \mathbf{X}^\top \mathbf{w}$$

Linear regression: matrix notation

- Suppose we have *n* images, each with just 2 pixels.

$$\begin{aligned}\hat{\mathbf{y}} &= \mathbf{X}^\top \mathbf{w} \\ &= \begin{bmatrix} \mathbf{x}_1^{(1)} & \dots & \mathbf{x}_1^{(n)} \\ \mathbf{x}_2^{(1)} & \dots & \mathbf{x}_2^{(n)} \end{bmatrix}^\top \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\end{aligned}$$

This is the index of the *image*.

Linear regression: matrix notation

- Suppose we have *n* images, each with just 2 pixels.

$$
\begin{aligned}
\hat{\mathbf{y}} &= \mathbf{X}^\top \mathbf{w} \\
&= \left[\begin{array}{ccc} \mathbf{x}_1^{(1)} & \ldots & \mathbf{x}_1^{(n)} \\ \mathbf{x}_2^{(1)} & \ldots & \mathbf{x}_2^{(n)} \end{array} \right]^\top \left[\begin{array}{c} w_1 \\ w_2 \end{array} \right]
\end{aligned}
$$

This is the index of the *pixel*.

Linear regression: matrix notation

- Suppose we have *n* images, each with just 2 pixels.

$$
\begin{aligned}
\hat{\mathbf{y}} &= \mathbf{X}^\top \mathbf{w} \\
&= \left[\begin{array}{ccc} \mathbf{x}_1^{(1)} & \dots & \mathbf{x}_1^{(n)} \\ \mathbf{x}_2^{(1)} & \dots & \mathbf{x}_2^{(n)} \end{array}\right]^\top \left[\begin{array}{c} w_1 \\ w_2 \end{array}\right] \\
&= \left[\begin{array}{cc} \mathbf{x}_1^{(1)} & \mathbf{x}_2^{(1)} \\ \vdots & \vdots \\ \mathbf{x}_1^{(n)} & \mathbf{x}_2^{(n)} \end{array}\right] \left[\begin{array}{c} w_1 \\ w_2 \end{array}\right]
\end{aligned}
$$

Linear regression: matrix notation

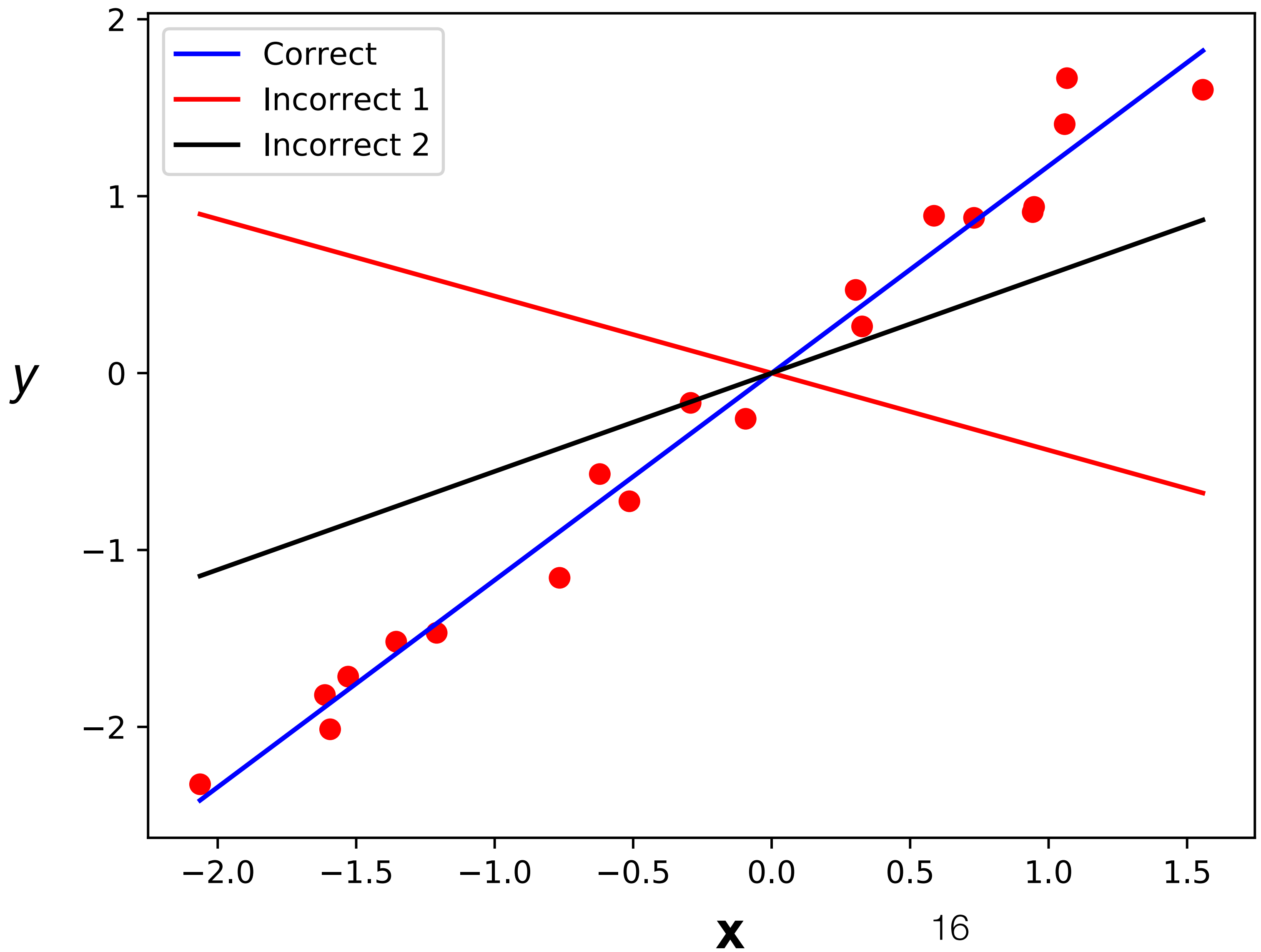
- Suppose we have *n* images, each with just 2 pixels.

$$
\begin{aligned}
\hat{\mathbf{y}} &= \mathbf{X}^\top \mathbf{w} \\
&= \begin{bmatrix} \mathbf{x}_1^{(1)} & \dots & \mathbf{x}_1^{(n)} \\ \mathbf{x}_2^{(1)} & \dots & \mathbf{x}_2^{(n)} \end{bmatrix}^\top \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{x}_1^{(1)} & \mathbf{x}_2^{(1)} \\ \vdots & \vdots \\ \mathbf{x}_1^{(n)} & \mathbf{x}_2^{(n)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{x}_1^{(1)} w_1 + \mathbf{x}_2^{(1)} w_2 \\ \vdots \\ \mathbf{x}_1^{(n)} w_1 + \mathbf{x}_2^{(n)} w_2 \end{bmatrix}
\end{aligned}
$$

Jacob Whitehill, WPI

1-d example

- Linear regression finds the weight vector **w** that minimizes the f_{MSE}. Here's an example where each **x** is just 1-d…



The best **w** is the one such that $f_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}})$ is as small as possible, where each $\hat{y} = \mathbf{x}^\top \mathbf{w}$.

Bias term

- In order to account for target values *y* with non-zero mean, we could add a **bias term** *b* to our model:

$$\hat{y} = \mathbf{x}^\top \mathbf{w} + b$$

- We could then compute the gradient w.r.t. both **w** and *b* and solve.

$$\begin{aligned}
\nabla_{\mathbf{w}} f_{\mathrm{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}, b) &= \nabla_{\mathbf{w}} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(\mathbf{x}^{(i)\top} \mathbf{w} + b - y^{(i)} \right)^2 \right] \\
\nabla_{b} f_{\mathrm{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}, b) &= \nabla_{b} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(\mathbf{x}^{(i)\top} \mathbf{w} + b - y^{(i)} \right)^2 \right]
\end{aligned}$$

Bias term

- Alternatively, we can implicitly include a bias term by augmenting each input vector **x** with a 1 at the end:

$$\tilde{\mathbf{x}} = \left[\begin{array}{c} \mathbf{x} \\ 1 \end{array} \right]$$

- Correspondingly, our weight vector **w** will have an extra component (bias term) at the end.

$$\tilde{\mathbf{w}} = \left[\begin{array}{c} \mathbf{w} \\ b \end{array} \right]$$

Jacob Whitehill, WPI

Bias term

- To see why, notice that:

$$
\begin{aligned}
\hat{y} &= \tilde{\mathbf{x}}^\top \tilde{\mathbf{w}} \\
&= \left[\begin{array}{c} \mathbf{x} \\ 1 \end{array} \right]^\top \left[\begin{array}{c} \mathbf{w} \\ b \end{array} \right] \\
&= \left[\begin{array}{cc} \mathbf{x}^\top & 1 \end{array} \right] \left[\begin{array}{c} \mathbf{w} \\ b \end{array} \right] \\
&= \mathbf{x}^\top \mathbf{w} + b
\end{aligned}
$$

Bias term

- We can find the optimal **w** and *b* based on all the training data using matrix notation.

- First define an augmented design matrix:

$$\tilde{\mathbf{X}} = \left[\begin{array}{ccc} \mathbf{x}^{(1)} & \ldots & \mathbf{x}^{(n)} \\ 1 & \ldots & 1 \end{array} \right]$$

- Then compute:

$$\tilde{\mathbf{w}} = \left(\tilde{\mathbf{X}} \tilde{\mathbf{X}}^\top \right)^{-1} \tilde{\mathbf{X}} \mathbf{y}$$

Fairness in ML

Fairness in ML

- Consider the following definition of ML fairness:
    - The machine's accuracy should be equal across all demographic subgroups on which it is tested.

Exercise

- Suppose we have trained a classifier to perceive whether a person is smiling based on their face image, and suppose its test accuracy (PC) on male & female faces is:
    - Male: 42%; female: 55%
      (You may assume that people from both genders smile with 50% probability.)
- Describe 3 possible reasons for why the test accuracy may differ between male and female faces.

Iterative solution to linear regression

Linear regression

- Linear regression is one of the few ML algorithms that has an analytical solution:

$$\mathbf{w} = \left(\mathbf{X}\mathbf{X}^\top\right)^{-1}\mathbf{X}\mathbf{y}$$

- **Analytical solution**: there is a closed formula for the answer.

Linear regression

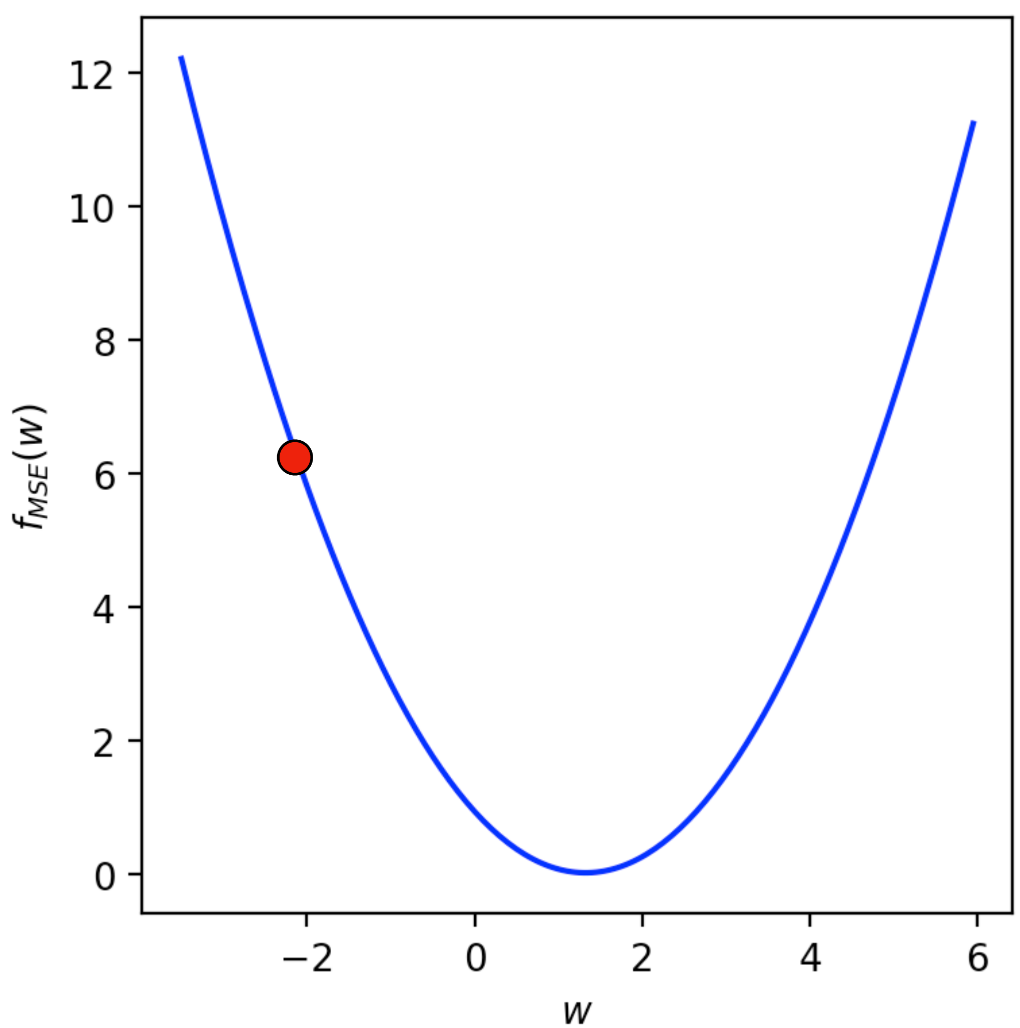
- Alternatively, linear regression can be solved numerically using gradient descent.
- **Numerical solution**: need to iterate (according to some algorithm) many times to *approximate* the optimal value.
- Gradient descent is more laborious to code than the one-shot solution, but it generalizes to a wide variety of ML models.

Gradient descent

- Gradient descent is a **hill climbing algorithm** that uses the gradient (aka slope) to decide which way to “move” **w** to reduce the objective function (e.g., f_{MSE}).

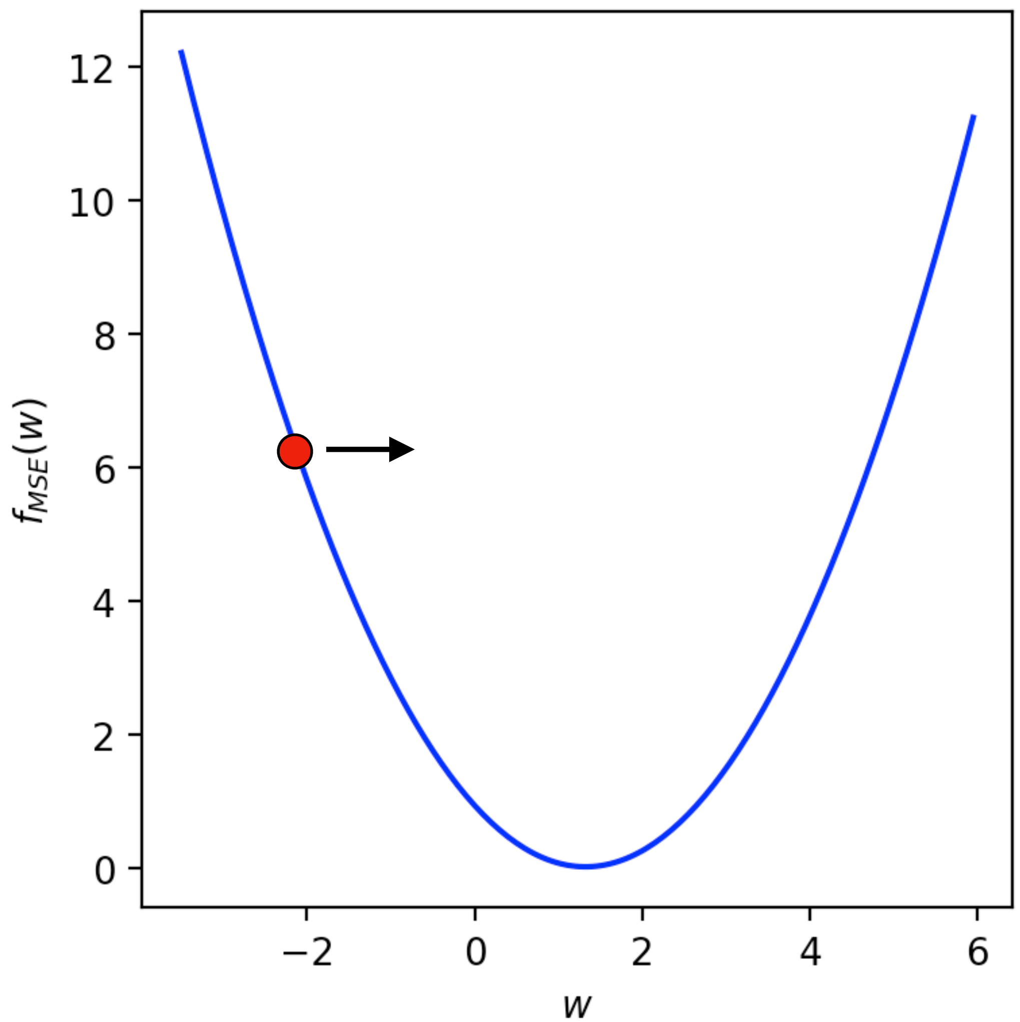
Gradient descent

- Suppose we just guess an initial value for **w** (e.g., -2.1).
- How can we make it better — increase it or decrease it?



Gradient descent

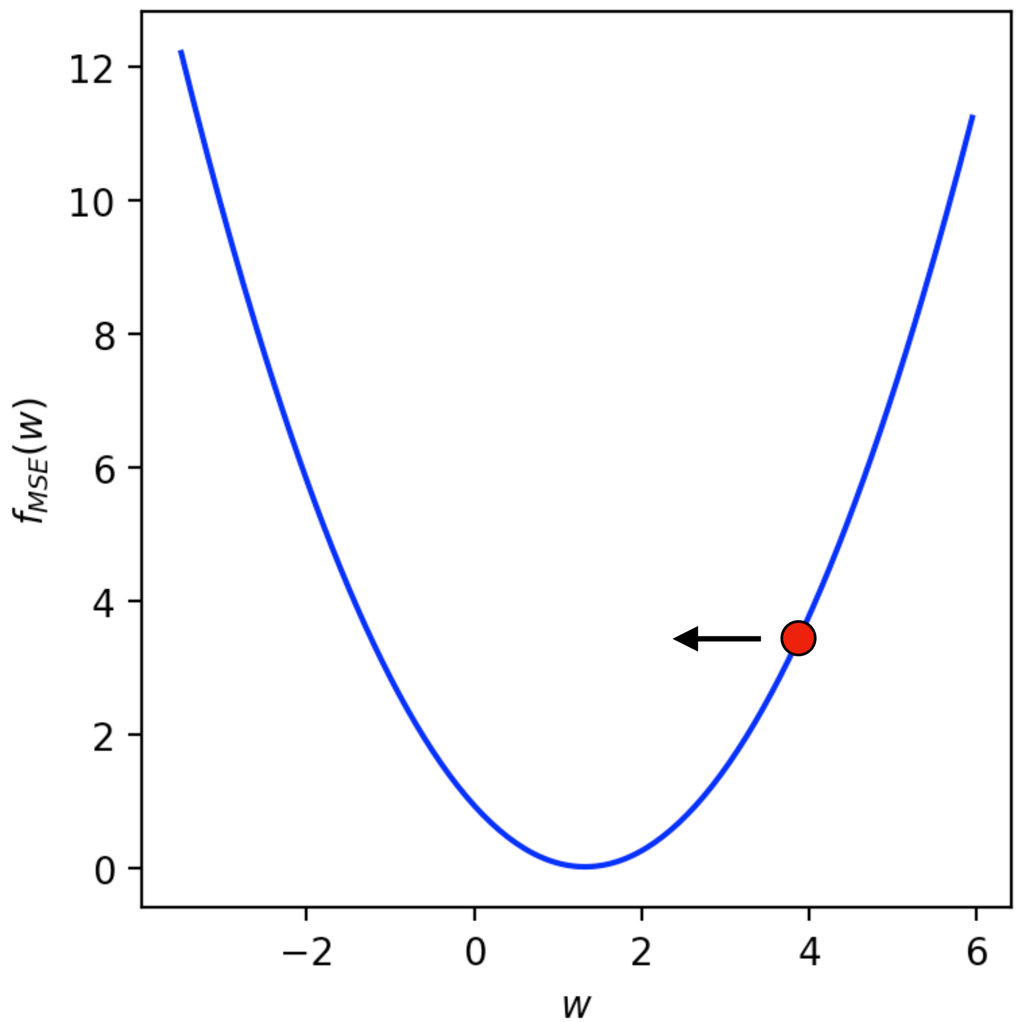
- Suppose we just guess an initial value for **w** (e.g., -2.1).
- How can we make it better — **increase** it or decrease it?
  - What does the **slope** of f_{MSE} tell us to do?



The slope at f_{MSE}(-2.1) is *negative*, i.e., we can *decrease* our cost by *increasing* w.

Gradient descent

- Or maybe our initial guess for **w** was 3.9.
- How can we make it better — increase it or **decrease** it?
  - What does the **slope** of f_{MSE} tell us to do?



The slope at f_{MSE}(3.9) is *positive*, i.e., we can *decrease* our cost by *decreasing* w.

Gradient descent

- How do we know the slope? Compute the **gradient** of f_{MSE} w.r.t. **w**:

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2n} \sum_{i=1}^{n} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \left[\left({\mathbf{x}^{(i)}}^\top \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}^{(i)} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} - y^{(i)} \right) \\
&= \frac{1}{n} \mathbf{X} \left(\mathbf{X}^\top \mathbf{w} - \mathbf{y} \right)
\end{aligned}
$$

Gradient descent

- How do we know the slope? Compute the **gradient** of f_{MSE} w.r.t. **w**:

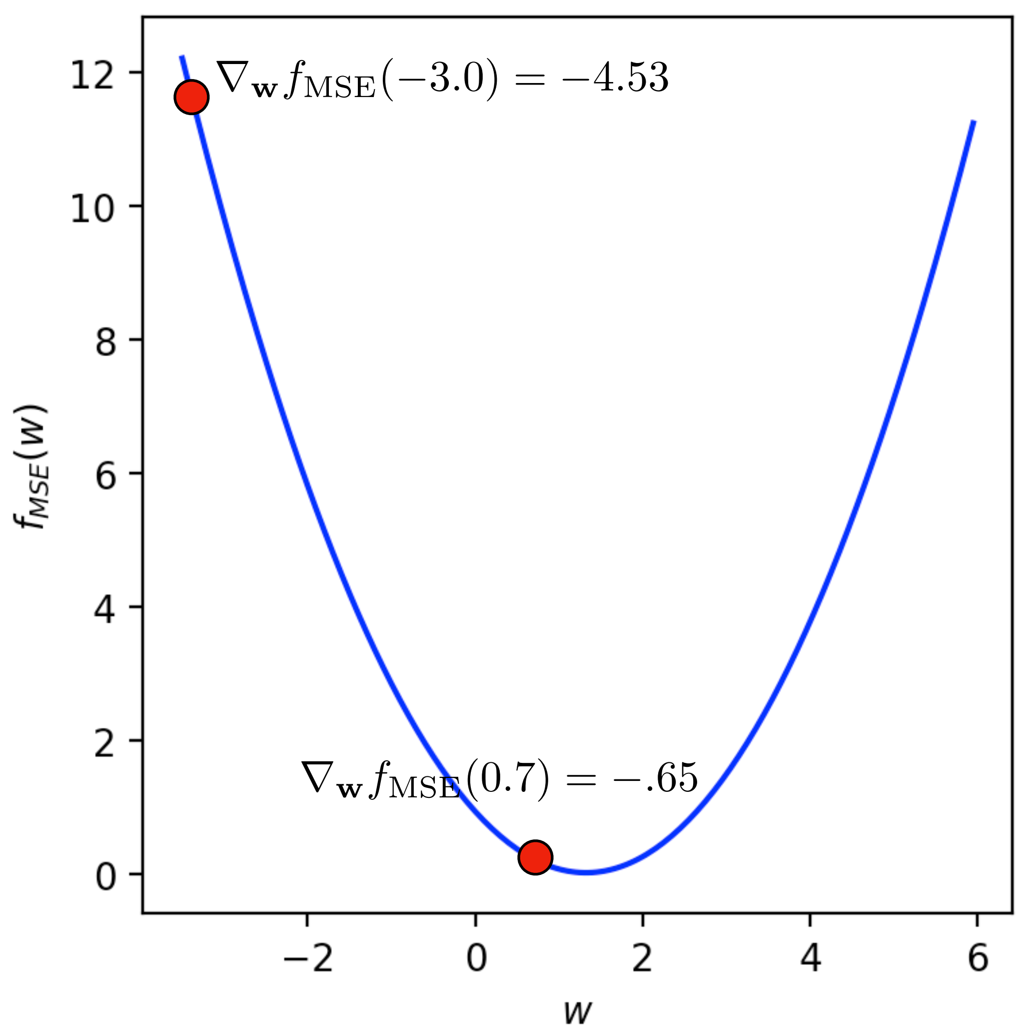
$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \left[\left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}^{(i)} \left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right) \\
&= \frac{1}{n} \mathbf{X} \left(\mathbf{X}^\top \mathbf{w} - \mathbf{y} \right)
\end{aligned}
$$

- Then plug in the current value of **w**.
  (Note that **X** and **y** are computed from the data and are constant.)

Gradient descent

- How *far* do we “move” left or right?

    - Notice that, in the graph below, the **magnitude** of the slope (aka gradient) gives an indication of how far we need to go to reach the optimal **w**.



Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.

  Python: `w = 0.01 * np.random.randn(M)  # Just an example!`

Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.
- Compute the gradient: $\nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$

Gradient descent algorithm

- Set **w** to random values; call this initial choice **w**$^{(0)}$.
- Compute the gradient: $\nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$
- Update **w** by moving opposite the gradient, multiplied by a **step size** ε. $\mathbf{w}^{(1)} \leftarrow \mathbf{w}^{(0)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$

Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.
- Compute the gradient: $\nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$
- Update **w** by moving opposite the gradient, multiplied by a **step size** ε.
- Repeat…

$$\mathbf{w}^{(1)} \leftarrow \mathbf{w}^{(0)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$$

$$\mathbf{w}^{(2)} \leftarrow \mathbf{w}^{(1)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(1)})$$

$$\mathbf{w}^{(3)} \leftarrow \mathbf{w}^{(2)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(2)})$$

...

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(t-1)})$$

Python: `w = w - EPS * gradient(w, X, y)`

Gradient descent algorithm

- How many iterations to run?
- Two alternative strategies:
  - Train for a fixed number of iterations T.

Gradient descent algorithm

- How many iterations to run?
- Two alternative strategies:
  - Train for a fixed number of iterations *T*.
  - Train until the difference in training cost diminishes below a threshold δ:

$$|f(\mathbf{w}^{(t-1)}) - f(\mathbf{w}^{(t)})| < \delta$$

Gradient descent demos

- 1-d
- 2-d