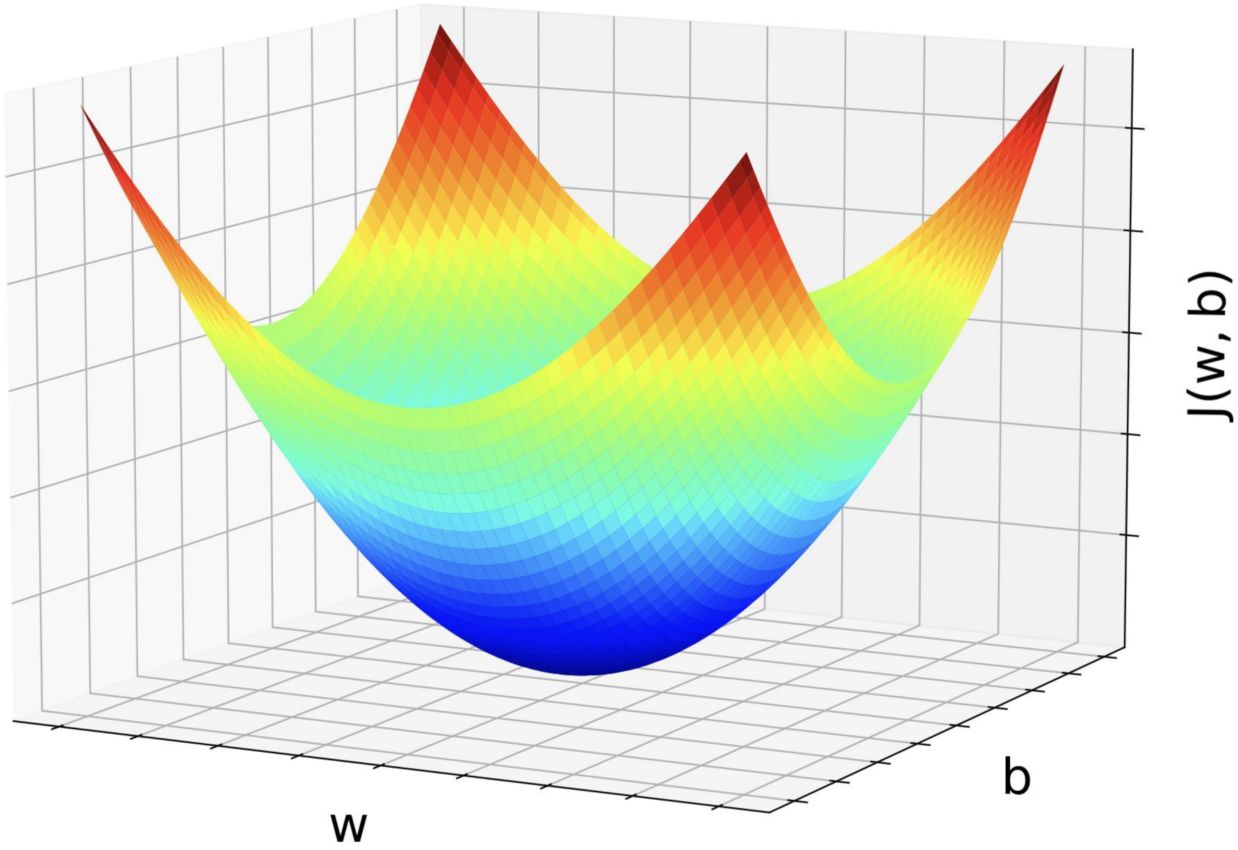
Gradient Descent

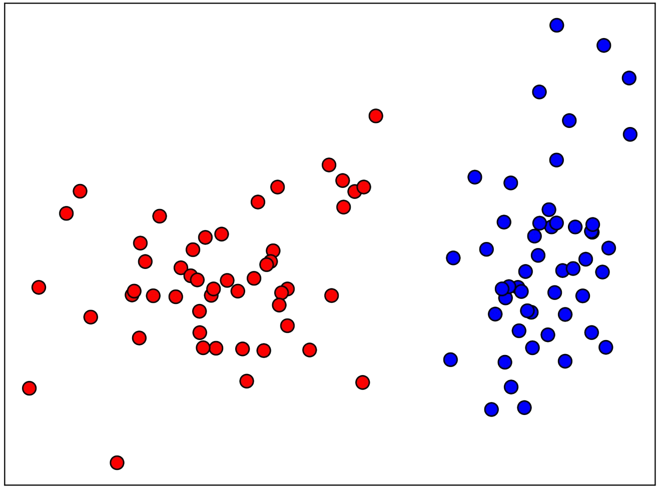
CS344
Deep Learning

WELLESLEY

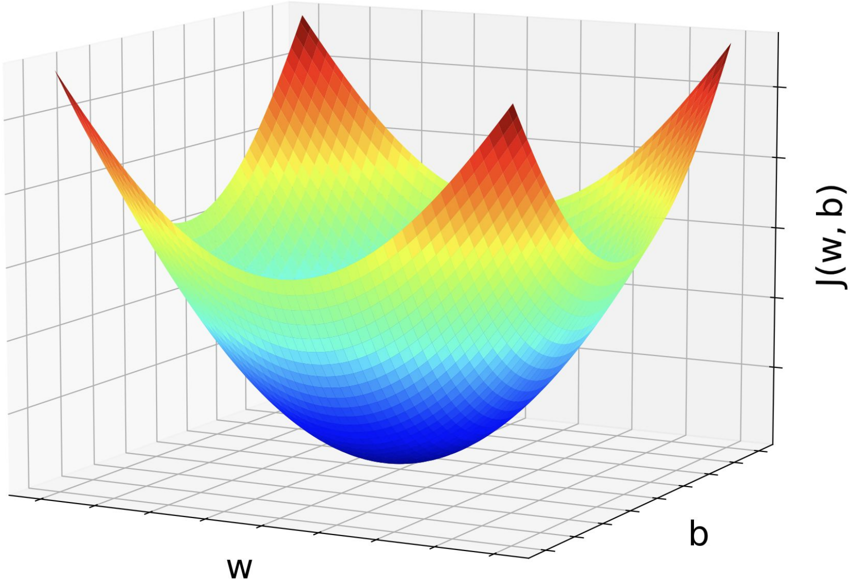
Cost Function



Cost Function

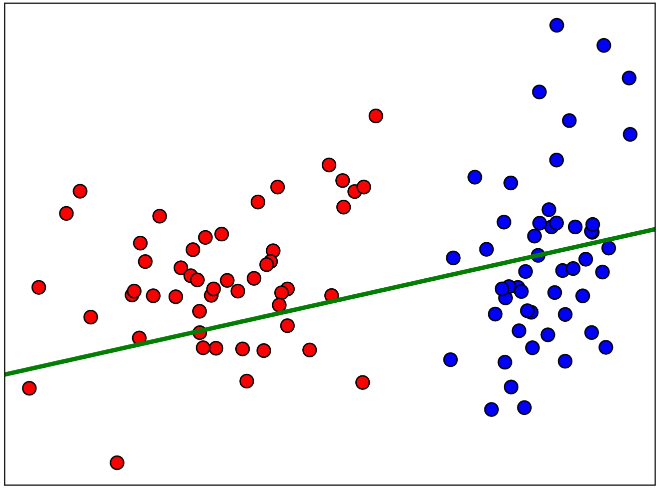


Data and Decision Boundary

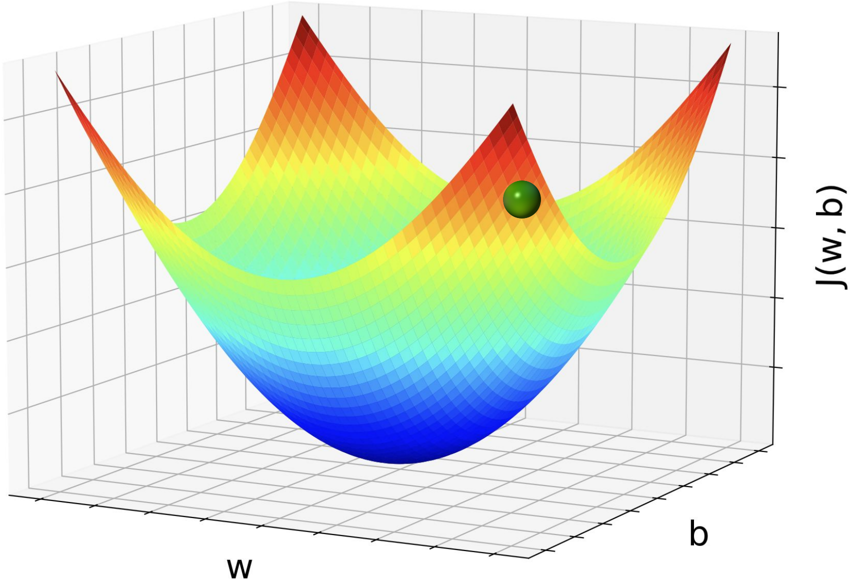


Cost Function

Cost Function

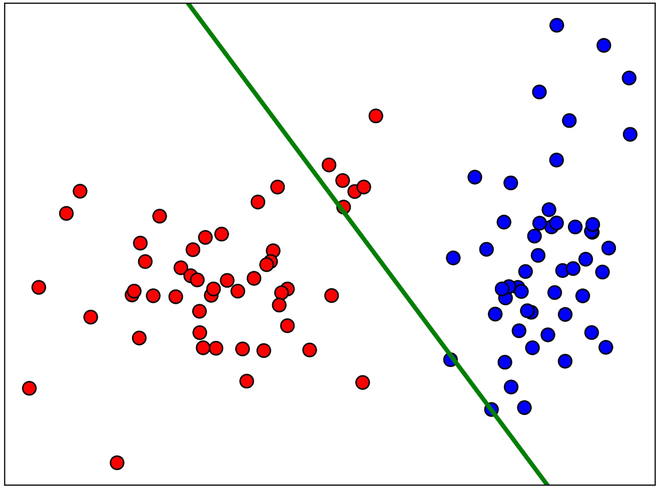


Data and Decision Boundary

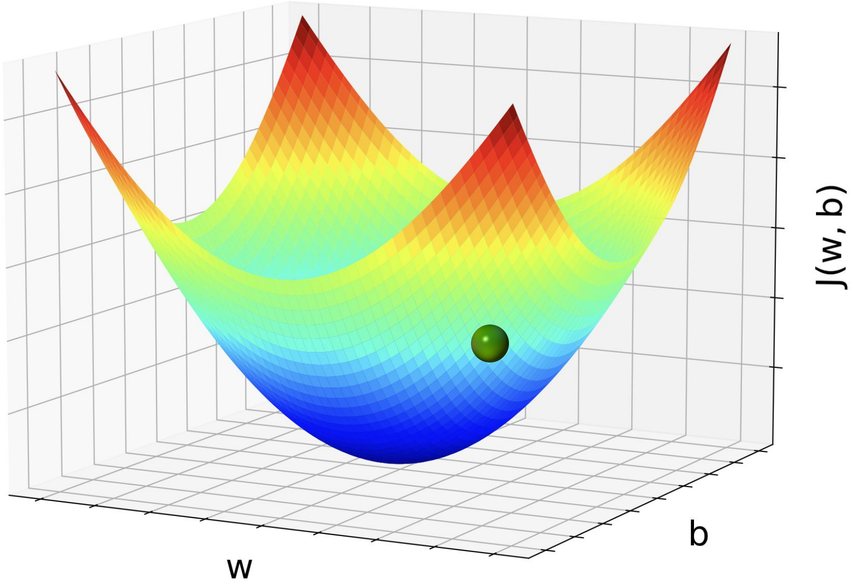


Cost Function

Cost Function

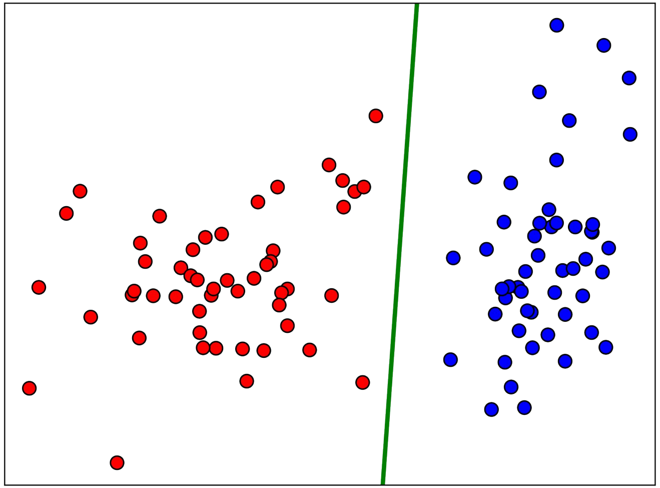


Data and Decision Boundary

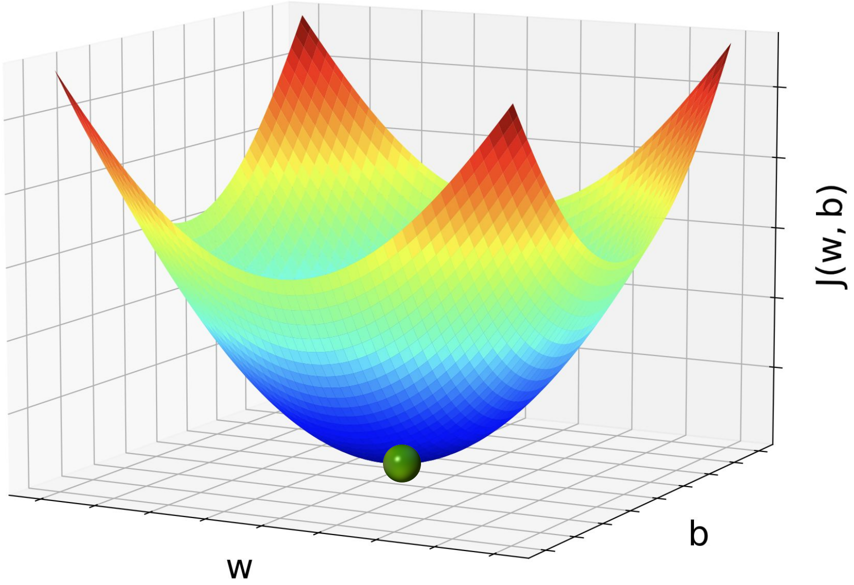


Cost Function

Cost Function



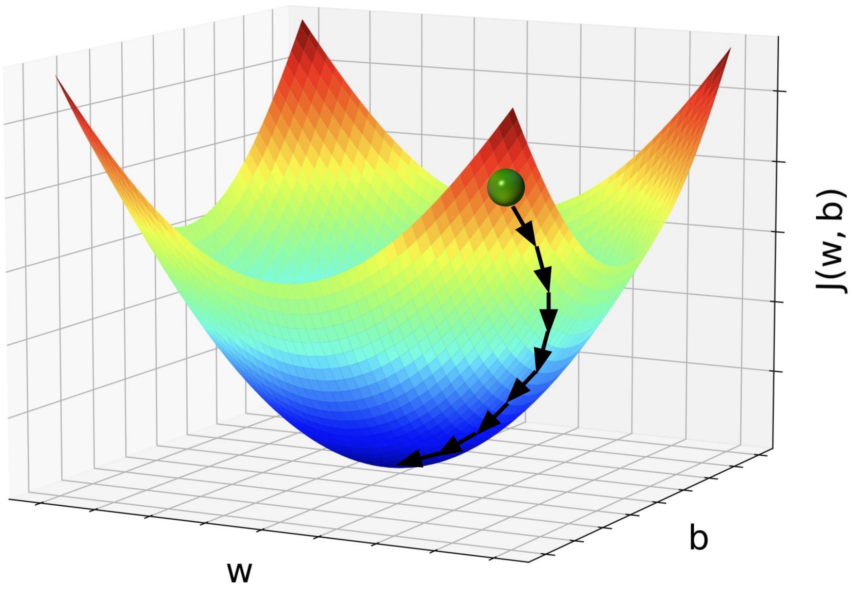
Data and Decision Boundary



Cost Function

Gradient Descent

We want to find parameters ***w*** and *b* that minimize the cost, J(***w***, *b*)



<u>Gradient Descent Algorithm</u>

- ❖ Initialize ***w*** and *b* (e.g., to 0)
- ❖ Repeat until converge:
  - ➢ Update ***w*** and *b* to reduce the cost J(***w***, *b*)

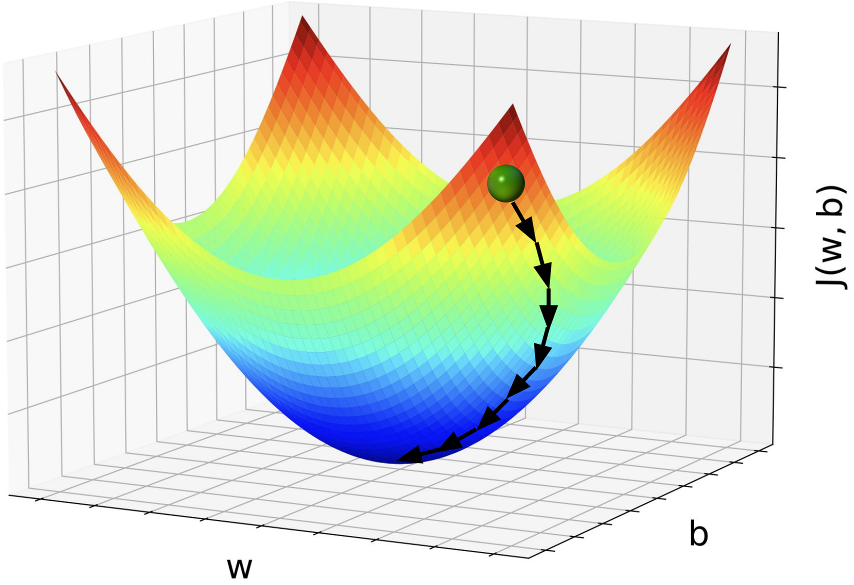
Gradient Descent Algorithm

Repeat until converge:

$$\mathbf{w} = \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} J(\mathbf{w}, b)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(\mathbf{w}, b)$$

α is the ***step size*** or ***learning rate***

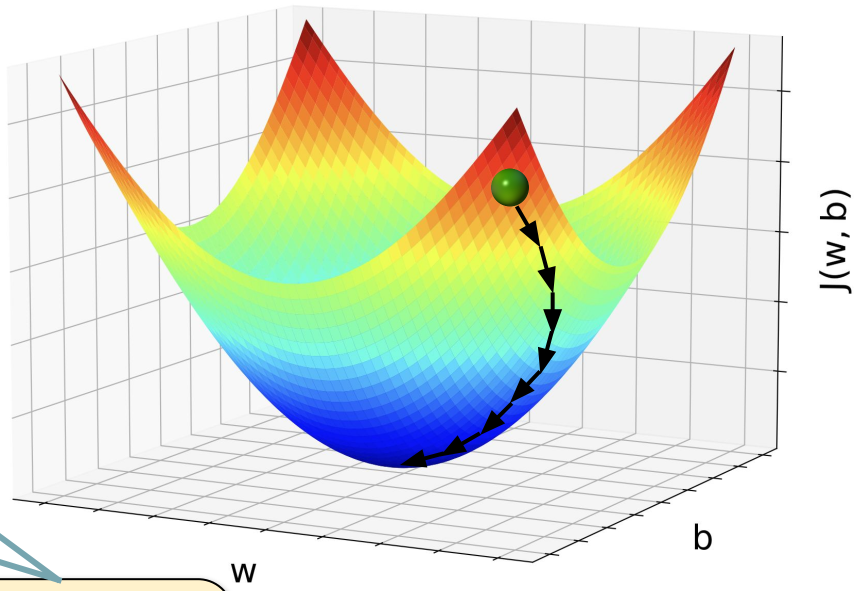


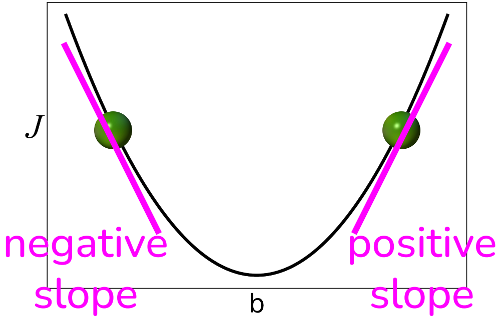
Gradient Descent Algorithm

Repeat until converge:

$$\mathbf{w} = \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} J(\mathbf{w}, b)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(\mathbf{w}, b)$$





The partial derivative ∂ indicates the ***slope***, i.e., the ***direction*** to step

Computing Derivatives: Chain Rule

$f(x) = x^2$

$f'(x) = 2x$

$g(x) = 2x+1$

$g'(x) = 2$

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

$h'(x) = 2 \cdot (2x+1) \cdot 2$

$h'(x) = f'(g(x)) \cdot g'(x)$

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Computing Derivatives: Chain Rule

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

$$\hat{y} = g(z) = \frac{1}{1 + e^{-z}}$$

$$L = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y})$$

$$J = \frac{1}{m} \sum_{i=1}^{m} L$$

Computing Derivatives: Chain Rule

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

$$\mathbf{x} \qquad \frac{dz}{dw}$$

$$a = g(z) = \frac{1}{1 + e^{-z}}$$

$$a(1 - a) \qquad \frac{da}{dz}$$

$$L = -y\log(a) - (1 - y)\log(1 - a)$$

$$\frac{-y}{a} + \frac{1 - y}{1 - a} \qquad \frac{dL}{da}$$

$$J = \frac{1}{m}\sum_{i=1}^{m} L$$

$$\frac{dL}{dw} = \frac{dL}{da} \cdot \frac{da}{dz} \cdot \frac{dz}{dw} = (a - y) \cdot x$$

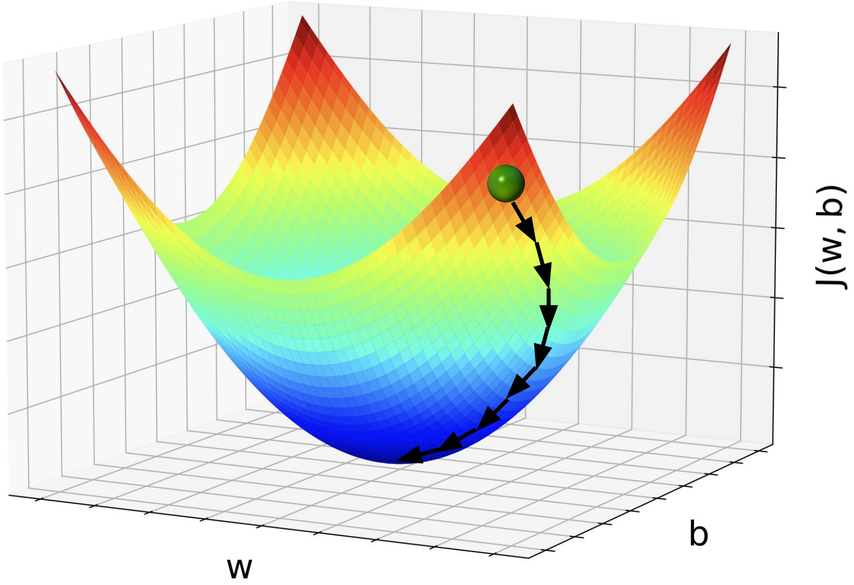
$$\frac{dJ}{dw} = \frac{1}{m}\sum_{i=1}^{m} \frac{dL}{dw} = \frac{1}{m}\sum_{i=1}^{m}(a - y) \cdot x$$

Gradient Descent Algorithm

Repeat until converge:

$$\mathbf{w} = \mathbf{w} - \alpha \boxed{\frac{\partial}{\partial \mathbf{w}} J(\mathbf{w}, b)}$$

$$b = b - \alpha \boxed{\frac{\partial}{\partial b} J(\mathbf{w}, b)}$$

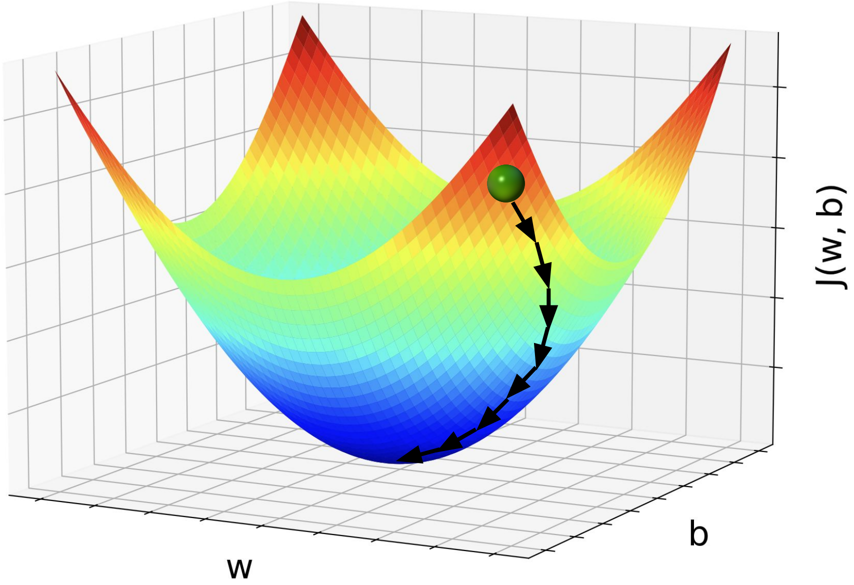


Gradient Descent Algorithm

Repeat until converge:

$$\mathbf{w} = \mathbf{w} - \alpha \boxed{\frac{1}{m}\sum_{i=1}^{m}}(a - y) \cdot x$$

$$b = b - \alpha \boxed{\frac{1}{m}\sum_{i=1}^{m}}(a - y)$$



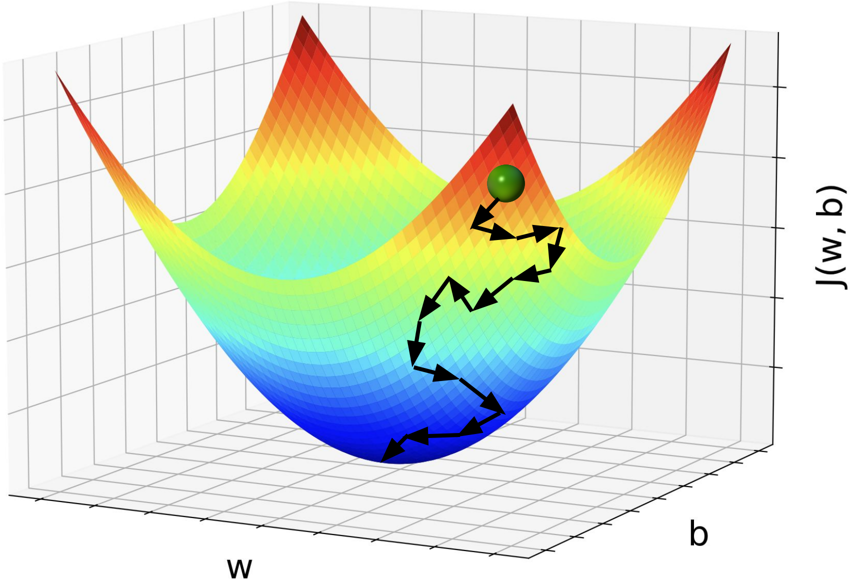
With ***batch*** gradient descent,
we consider ***all*** data points each
time we update a weight parameter

Gradient Descent Algorithm

Repeat until converge:

$$\mathbf{w} = \mathbf{w} - \alpha\ (a - y) \cdot x$$

$$b = b - \alpha\ (a - y)$$



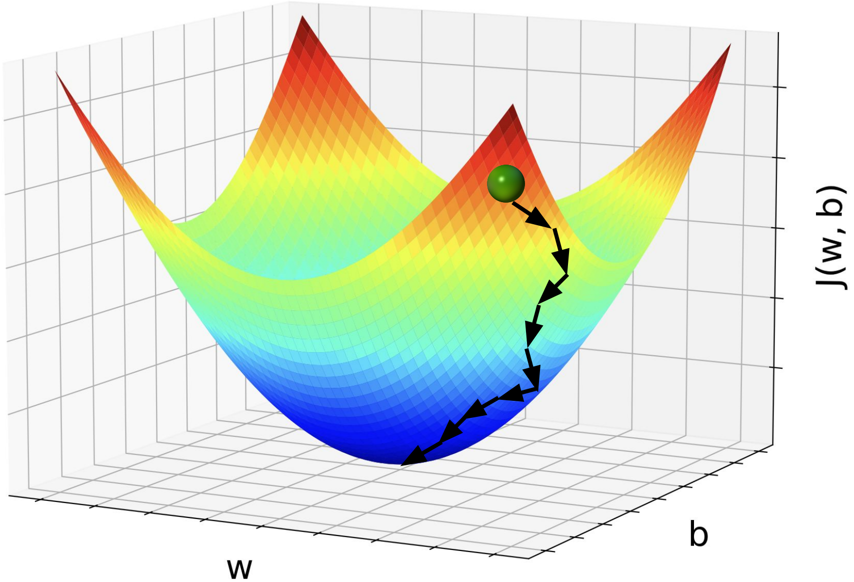
With ***stochastic*** gradient descent, we consider a ***single*** data point each time we update a weight parameter

Gradient Descent Algorithm

Repeat until converge:

$$\mathbf{w} = \mathbf{w} - \alpha \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} (a - y) \cdot x$$

$$b = b - \alpha \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} (a - y)$$



With ***mini-batch*** gradient descent, we consider a ***small batch*** of data points each time we update a weight parameter

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = new array of shape (m, 1)
For i = 1 to m:
    Z[i] = np.dot(X[i], w) + b
```

$$X_{(m,2)} = \begin{bmatrix} 3 & 2 \\ 9 & 1 \\ 5 & 4 \\ 2 & 3 \\ 0 & 4 \\ \dots & \dots \\ 3 & 1 \end{bmatrix} \qquad \mathbf{w}_{(2,1)} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \qquad b = 1 \qquad Z_{(m,1)} = \begin{bmatrix} 2 \\ -6 \\ 4 \\ 5 \\ 9 \\ \dots \\ 0 \end{bmatrix}$$

Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

Guideline: ***avoid looping*** over arrays

`numpy` has a built in function for everything

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = new array of shape (m, 1)
For i = 1 to m:
    Z[i] = np.dot(X[i], w) + b
```

$$X_{(m,\,2)} = \begin{bmatrix} 3 & 2 \\ 9 & 1 \\ 5 & 4 \\ 2 & 3 \\ 0 & 4 \\ \cdots & \cdots \\ 3 & 1 \end{bmatrix} \qquad w_{(2,\,1)} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \qquad b = 1 \qquad Z_{(m,\,1)} = \begin{bmatrix} 2 \\ -6 \\ 4 \\ 5 \\ 9 \\ \cdots \\ 0 \end{bmatrix}$$

Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

Guideline: ***avoid looping*** over arrays

numpy has a built in function for everything

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = new array of shape (m, 1)
For i = 1 to m:
    Z[i] = np.dot(X[i], w) + b
```

$$X_{(m,2)} = \begin{bmatrix} 3 & 2 \\ 9 & 1 \\ 5 & 4 \\ 2 & 3 \\ 0 & 4 \\ \dots & \dots \\ 3 & 1 \end{bmatrix} \qquad w_{(2,1)} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \qquad b = 1 \qquad Z_{(m,1)} = \begin{bmatrix} 2 \\ -6 \\ 4 \\ 5 \\ 9 \\ \dots \\ 0 \end{bmatrix}$$

Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

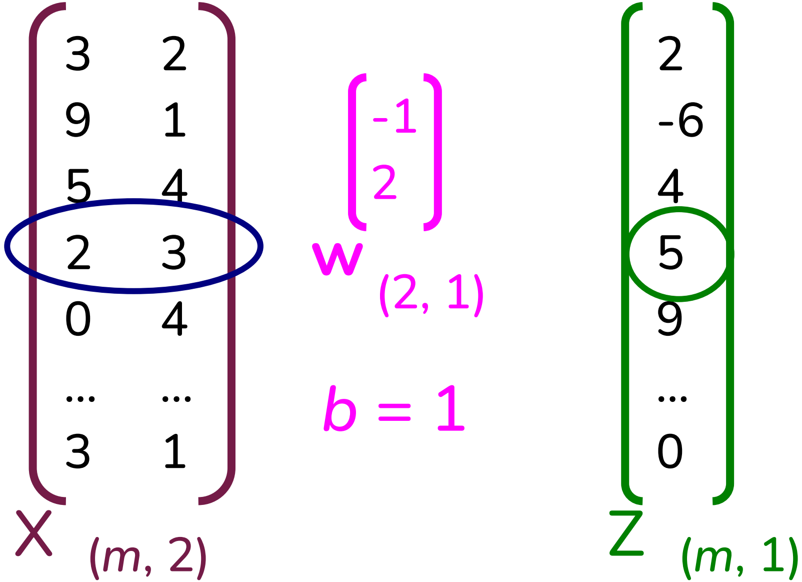
Guideline: ***avoid looping*** over arrays

`numpy` has a built in function for everything

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = new array of shape (m, 1)
For i = 1 to m:
    Z[i] = np.dot(X[i], w) + b
```



Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

Guideline: ***avoid looping*** over arrays

numpy has a built in function for everything

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = new array of shape (m, 1)
For i = 1 to m:
    Z[i] = np.dot(X[i], w) + b
```

$$X_{(m,2)} = \begin{bmatrix} 3 & 2 \\ 9 & 1 \\ 5 & 4 \\ 2 & 3 \\ 0 & 4 \\ \dots & \dots \\ 3 & 1 \end{bmatrix} \qquad \mathbf{w}_{(2,1)} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \qquad b = 1 \qquad Z_{(m,1)} = \begin{bmatrix} 2 \\ -6 \\ 4 \\ 5 \\ 9 \\ \dots \\ 0 \end{bmatrix}$$

Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

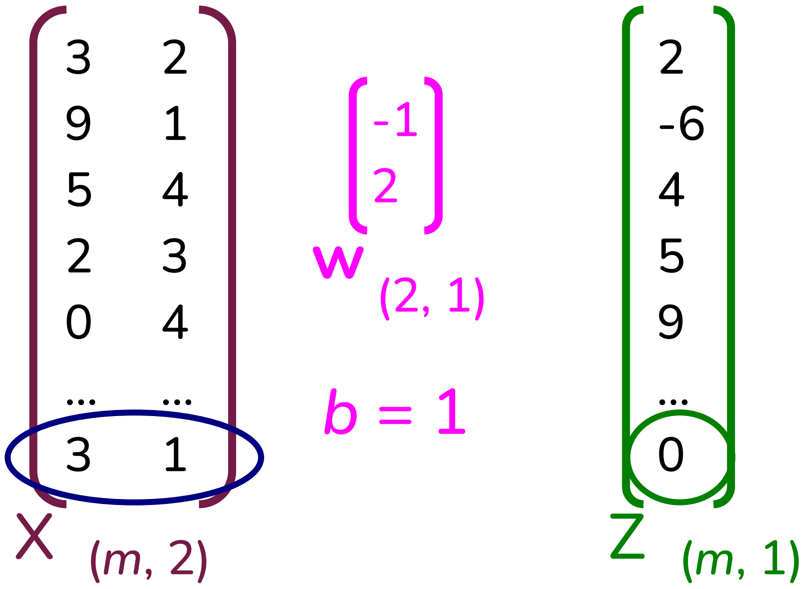
Guideline: ***avoid looping*** over arrays

`numpy` has a built in function for everything

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = new array of shape (m, 1)
For i = 1 to m:
    Z[i] = np.dot(X[i], w) + b
```



Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

Guideline: ***avoid looping*** over arrays

numpy has a built in function for everything

Vectorization

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

```
Z = np.dot(X, w) + b
```

$$X_{(m,2)} = \begin{pmatrix} 3 & 2 \\ 9 & 1 \\ 5 & 4 \\ 2 & 3 \\ 0 & 4 \\ \dots & \dots \\ 3 & 1 \end{pmatrix} \quad \mathbf{w}_{(2,1)} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \quad b = 1 \quad Z_{(m,1)} = \begin{bmatrix} 2 \\ -6 \\ 4 \\ 5 \\ 9 \\ \dots \\ 0 \end{bmatrix}$$

Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

Guideline: ***avoid looping*** over arrays

`numpy` has a built in function for everything

Vectorization

$$a = \text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

```
A = sigmoid(Z)
```

$$Z = \begin{bmatrix} 2 \\ -6 \\ 4 \\ 5 \\ 9 \\ \dots \\ 0 \end{bmatrix} \quad (m, 1) \qquad A = \begin{bmatrix} 0.881 \\ 0.002 \\ 0.982 \\ 0.993 \\ 0.999 \\ \dots \\ 0.5 \end{bmatrix} \quad (m, 1)$$

Vectorization speeds up execution time dramatically

Highly optimized libraries ***parallelize*** computation

Guideline: ***avoid looping*** over arrays

`numpy` has a built in function for everything

Training (Fitting)

<u>Gradient Descent Algorithm</u>

- ❖ Initialization
- ❖ Repeat until convergence:
  - ➢ Forward propagation
  - ➢ Calculate cost
  - ➢ Backpropagation

Training refers to learning the parameters (*w* and *b*) of the model from the training data*

*Assumes ***X*** refers to training data with *m* rows and *d* columns

Training

Gradient Descent Algorithm

- ❖ Initialization
- ❖ Repeat until convergence:
  - ➢ Forward propagation
  - ➢ Calculate cost
  - ➢ Backpropagation

Initialize parameters ***w*** and *b*

- ➔ Create (d, 1) array ***w*** of random numbers
- ➔ Create variable *b* set to 0

Training

Loop for *max_iter* iterations

Gradient Descent Algorithm

- ❖ Initialization
- ❖ Repeat until convergence:
  - ➢ Forward propagation
  - ➢ Calculate cost
  - ➢ Backpropagation

Training

Gradient Descent Algorithm

- Initialization
- Repeat until convergence:
  - **Forward propagation**
  - Calculate cost
  - Backpropagation

Compute activations

→ $Z = X \cdot w + b$

→ $A = g(Z) = \text{sigmoid}(Z)$

Training

Gradient Descent Algorithm

- Initialization
- Repeat until convergence:
  - Forward propagation
  - Calculate cost
  - Backpropagation

Cost using current values of $\boldsymbol{w}$ and b

$$\frac{1}{m}\sum_{i=1}^{m} -\boldsymbol{y}\log(\boldsymbol{A}) - (1-\boldsymbol{y})\log(1-\boldsymbol{A})$$

Training

Gradient Descent Algorithm

- ❖ Initialization
- ❖ Repeat until convergence:
  - ➢ Forward propagation
  - ➢ Calculate cost
  - ➢ Backpropagation

Compute gradients

- ➔ $dZ = A - y$
- ➔ $dW = X^T \cdot dZ / m$
- ➔ $db = \frac{1}{m}\sum_{i=1}^{m} dZ$

Update parameters

- ➔ $W = W - \alpha \cdot dW$
- ➔ $b = b - \alpha \cdot db$

Testing

Testing refers to evaluating the trained model with testing data*

- Make predictions
- Assess how well predictions correspond to known labels

*Assumes ***X*** refers to testing data

Testing

- ❖ Make predictions
- ❖ Assess how well predictions correspond to known labels

<u>Predict activations</u>

- ➔ A = Forward propagation of X
- ➔ Predictions = Binarize (round) A

Testing

❖ Make predictions

❖ Assess how well predictions correspond to known labels

Score model

➔ Calculate percentage of predictions that correctly match labels

Overfitting

- ❖ *Overfitting* is one of the most common problems in ML
- ❖ Model learns properties specific to the training data that don't generalize to new (testing) data
- ❖ Performance is much better on training data than on testing data

Z





TS

Regularization

Smaller values for the parameters ***w*** lead to more generalizable models and are less prone to overfitting

To incentivize small values for ***w***, modify the cost function so that it:

1) Fits the training data well

and

2) Penalizes large values for ***w***

λ is regularization parameter

$$J = \frac{1}{m}\sum_{i=1}^{m} L + \frac{\lambda}{2m}\sum_{j=1}^{d} w_j^2$$

$$\frac{dJ}{dw} = \frac{1}{m}\sum_{i=1}^{m}(a-y)\cdot x + \frac{\lambda}{m}w$$