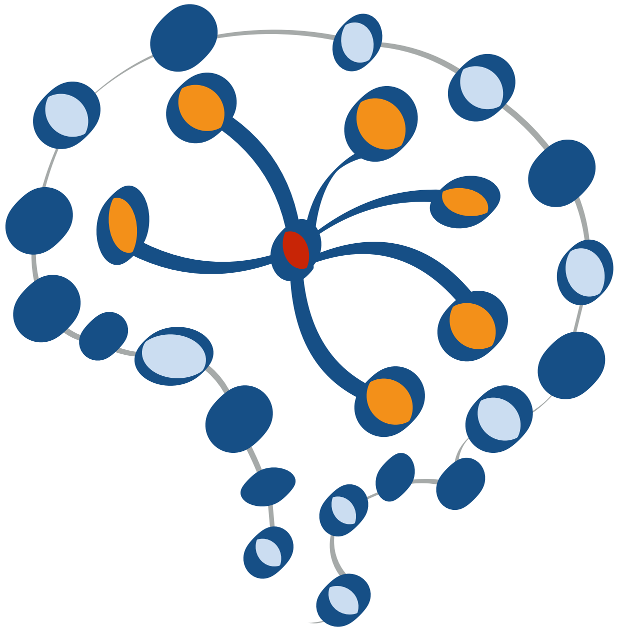
STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching



Lecture 05

Fitting Neurons with Gradient Descent

Let's improve upon the perceptron & learn about a neural network model for which the training always converges

Our Goals

- A learning rule that is more robust than the perceptron: always converges even if the data is not (linearly) separable

This lecture

- Combine multiple neurons and layers of neurons ("deep neural nets") to learn more complex decision boundaries (because most real-world problems are not "linear" problems!)

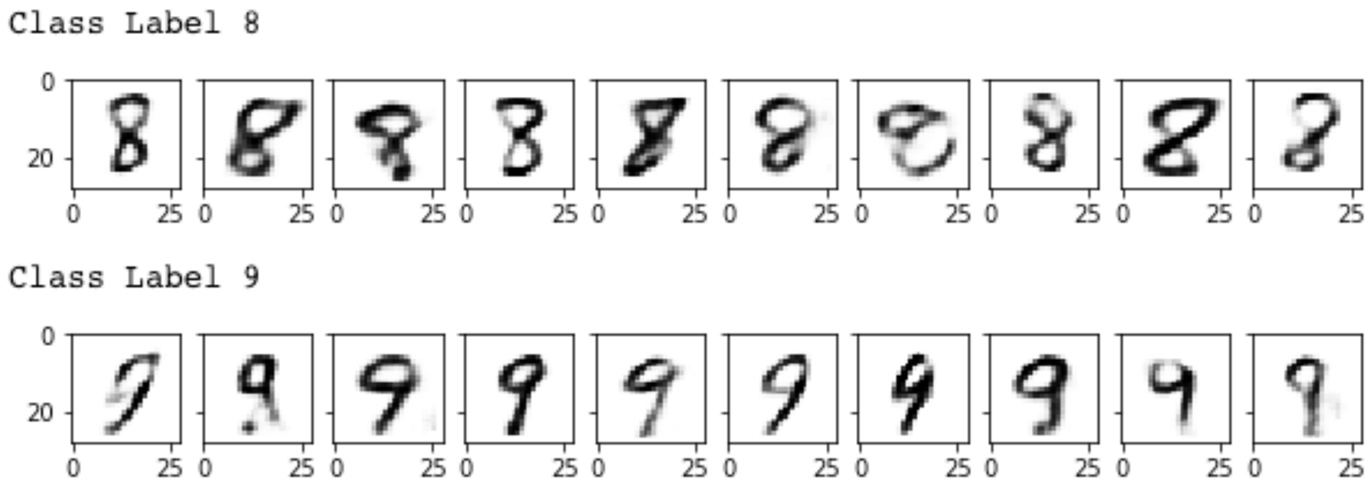
Next lecture(s)

- Handle multiple categories (not just binary) in classification

Next lecture(s)

- Do even fancier things like generating NEW images and text

More towards the end of the course



All based on the same learning algorithm and extensions thereof.
So, this is prob. the most fundamental lecture!

Good news:

- After this lecture, there won't be any "new" mathematical concepts.
- Everything in DL will be extensions & applications of these basic concepts.

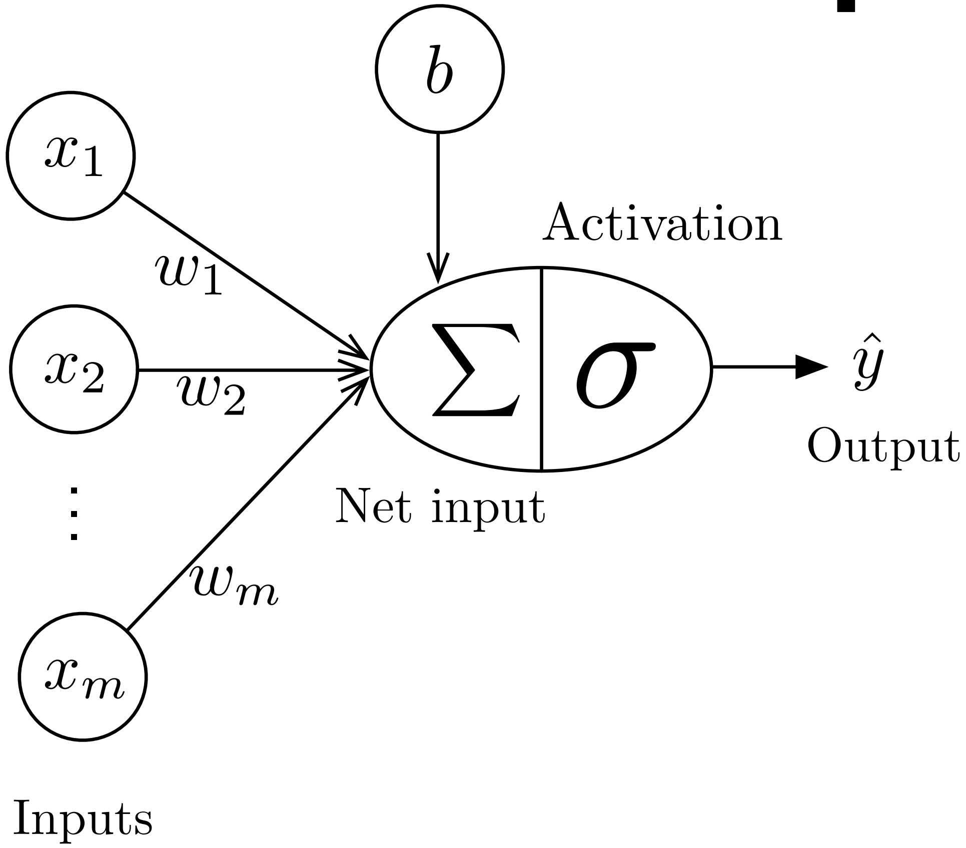
Lecture Overview

1. Online, batch, and minibatch mode
2. Relation between perceptron and linear regression
3. An iterative training algorithm for linear regression
4. (Optional) Calculus refresher I: Derivatives
5. (Optional) Calculus refresher II: Gradients
6. Understanding gradient descent
7. Training an adaptive linear neuron (Adaline)

Training neural nets: Online, batch, and minibatch modes

1. **Online, batch, and minibatch mode**
2. Relation between perceptron and linear regression
3. An iterative training algorithm for linear regression
4. (Optional) Calculus refresher I: Derivatives
5. (Optional) Calculus refresher II: Gradients
6. Understanding gradient descent
7. Training an adaptive linear neuron (Adaline)

Perceptron Recap



$$\sigma\left(\sum_{i=1}^{m} x_i w_i + b\right) = \sigma\left(\mathbf{x}^T\mathbf{w} + b\right) = \hat{y}$$

$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

$$b = -\theta$$

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

```
1.  Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2.  For every training epoch:
        A.  For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
                (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w} + b)$            ← Compute output (prediction)
                (b)  $\text{err} := (y^{[i]} - \hat{y}^{[i]})$                          ← Calculate error
                (c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$ , $b := b + err$  ← Update parameters
```

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

"On-line" mode

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
   A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
      (a) Compute output (prediction)
      (b) Calculate error
      (c) Update $\mathbf{w}, b$
```

This applies to all common neuron models and (deep) neural network architectures!

There are some variants of it, namely the "batch mode" and the "minibatch mode" which we will briefly go over in the next slides and then discuss more later

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

"On-line" mode

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
        (a) Compute output (prediction)
        (b) Calculate error
        (c) Update $\mathbf{w}, b$
```

Batch mode

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. Initialize $\Delta \mathbf{w} := 0, \ \Delta b := 0$
    B. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
        (a) Compute output (prediction)
        (b) Calculate error
        (c) Update $\Delta \mathbf{w}, \Delta b$
    C. Update $\mathbf{w}, b$ :
        $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, b := +\Delta b$
```

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

"On-line" mode

```
1.  Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2.  For every training epoch:
    A.  For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
        (a)  Compute output (prediction)
        (b)  Calculate error
        (c)  Update $\mathbf{w}, b$
```

In practice, we usually shuffle the dataset prior to each epoch to prevent cycles

Batch mode

```
1.  Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2.  For every training epoch:
    A.  Initialize $\Delta\mathbf{w} := 0, \ \Delta b := 0$
    B.  For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
        (a)  Compute output (prediction)
        (b)  Calculate error
        (c)  Update $\Delta\mathbf{w}, \Delta b$
    C.  Update $\mathbf{w}, b$:
            $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, b := +\Delta b$
```

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

"On-line" mode

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
        (a) Compute output (prediction)
        (b) Calculate error
        (c) Update $\mathbf{w}, b$
```

In practice, we usually shuffle the dataset prior to each epoch to prevent cycles

"On-line" mode II (alternative)

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For for $t$ iterations:
    A. Pick random $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
        (a) Compute output (prediction)
        (b) Calculate error
        (c) Update $\mathbf{w}, b$
```

No shuffling required

(actually, not really stochastic because a fixed training set instead of sampling from the population)

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

Minibatch mode

(mix between on-line and batch)

The most common mode in deep learning. Any ideas why?

```
1.  Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2.  For every training epoch:
    3.  For every minibatch of size k:
        A.  Initialize $\Delta \mathbf{w} := 0, \; \Delta b := 0$
        B.  For every $\{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle, ..., \langle \mathbf{x}^{[i+k]}, y^{[i+k]} \rangle\} :\subset \mathcal{D}$
            (a)  Compute output (prediction)
            (b)  Calculate error
            (c)  Update $\Delta \mathbf{w}, \Delta b$
        C.  Update $\mathbf{w}, b$ :
                $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, b := +\Delta b$
```

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

Minibatch mode

(mix between on-line and batch)

```
1.  Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2.  For every training epoch:
    3.  For every minibatch of size k:
        A.  Initialize $\Delta \mathbf{w} := 0, \ \Delta b := 0$
        B.  For every $\{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle, ..., \langle \mathbf{x}^{[i+k]}, y^{[i+k]} \rangle\} \subset \mathcal{D}$
            (a)  Compute output (prediction)
            (b)  Calculate error
            (c)  Update $\Delta \mathbf{w}, \Delta b$
        C.  Update $\mathbf{w}, b$ :
                $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, b := +\Delta b$
```

Most commonly used in DL, because

1. Choosing a subset (vs 1 example at a time) takes advantage of vectorization (faster iteration through epoch than on-line)
2. having fewer updates than "on-line" makes updates less noisy
3. makes more updates/ epoch than "batch" and is thus faster

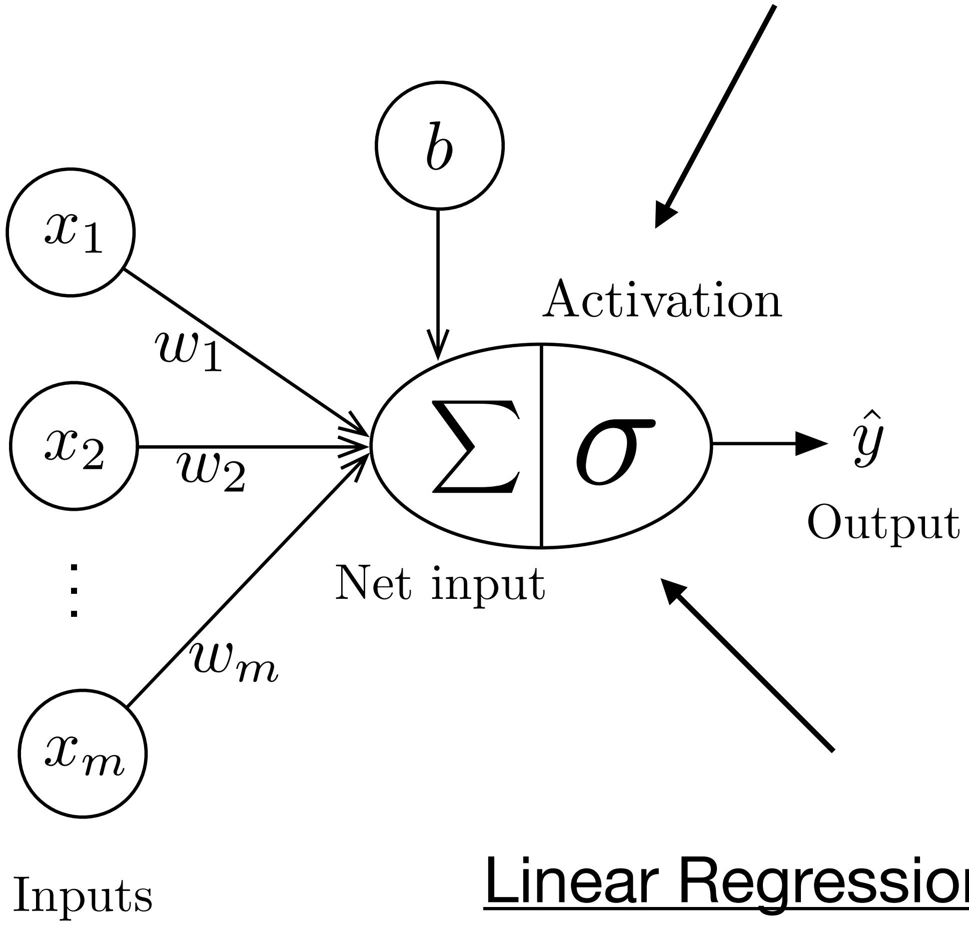
Linear regression as a single-layer neural network

1. Online, batch, and minibatch mode
2. **Relation between perceptron and linear regression**
3. An iterative training algorithm for linear regression
4. (Optional) Calculus refresher I: Derivatives
5. (Optional) Calculus refresher II: Gradients
6. Understanding gradient descent
7. Training an adaptive linear neuron (Adaline)

Linear Regression

<u>Perceptron:</u> Activation function is the threshold function

The output is a binary label $\hat{y} \in \{0, 1\}$



<u>Linear Regression:</u> Activation function is the identity function

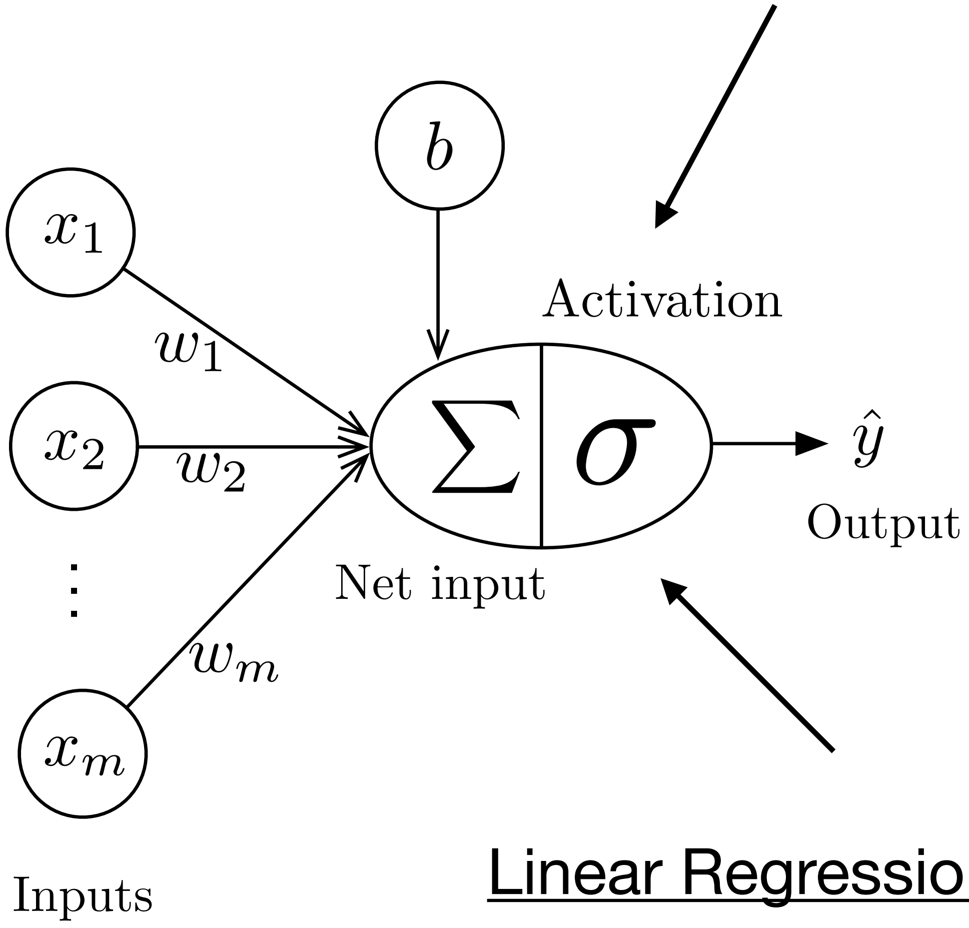
$$\sigma(x) = x$$

The output is a real number $\hat{y} \in \mathbb{R}$

Linear Regression

Perceptron: Activation function is the threshold function

The output is a binary label $\hat{y} \in \{0, 1\}$



You can think of linear regression as a linear neuron!

Linear Regression: Activation function is the identity function

$$\sigma(x) = x$$

The output is a real number $\hat{y} \in \mathbb{R}$

(Least-Squares) Linear Regression

In earlier statistics classes, you probably fit a linear regression model model like this, using the "normal equations" (analytical solution):

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

(assuming that the bias is included in $\mathbf{w}$, and the design matrix has an additional vector of 1's)

(Least-Squares) Linear Regression

In earlier statistics classes, you probably fit a model like this: using the "normal equations:"

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

(assuming that the bias is included in $\mathbf{w}$, and the design matrix has an additional vector of 1's)

- Generally, this is the best approach for linear regression (although,
  the matrix inversion might be problematic on large datasets)
- However, we will now learn about another way to learn these parameters iteratively
- Why? Because this is what we will be doing in deep neural nets later, where we have large datasets, many connections, and non-convex loss functions

Training Linear Regression in an iterative fashion

1. Online, batch, and minibatch mode
2. Relation between perceptron and linear regression
3. **An iterative training algorithm for linear regression**
4. (Optional) Calculus refresher I: Derivatives
5. (Optional) Calculus refresher II: Gradients
6. Understanding gradient descent
7. Training an adaptive linear neuron (Adaline)

(Least-Squares) Linear Regression -- iteratively with "brute force"

- A very naive way to fit a linear regression model (and any neural net) is to start with initializing the parameters to 0's or small random values
- Then, for *k* rounds
  - Choose another random set of weights
  - If the model performs better, keep those weights
  - If the model performs worse, discard the weights

This approach is guaranteed to find the optimal solution for very large *k,* but it would be terribly slow.

(Least-Squares) Linear Regression iteratively

- A very naive way to fit a linear regression model (and any neural net) is to start with initializing the parameters to 0's or small random values
- Then, for *k* rounds
  - Choose another random set of weights
  - If the model performs better, keep those weights
  - If the model performs worse, discard the weights

There's a better way!

- We will analyze what effect a change of a parameter has on the predictive performance (loss) of the model
  then, we change the weight a little bit in the direction that improves the performance (minimizes the loss) the most
- We do this in several (small) steps until the loss does not further decrease

(Least-Squares) Linear Regression

The update rule turns out to be this:

"On-line" mode

Perceptron learning rule

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
   A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
      (a) $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
      (b) $\mathrm{err} := \big(y^{[i]} - \hat{y}^{[i]}\big)$
      (c) $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$
          $b := b + err$
```

Stochastic gradient descent

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
   A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
      (a) $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
      (b) $\nabla_{\mathbf{w}}\mathcal{L} = \big(y^{[i]} - \hat{y}^{[i]}\big)\mathbf{x}^{[i]}$
          $\nabla_{b}\mathcal{L} = \big(y^{[i]} - \hat{y}^{[i]}\big)$
      (c) $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}}\mathcal{L})$
          $b := b + \eta \times \underbrace{(-\nabla_{b}\mathcal{L})}_{\text{negative gradient}}$
```

learning rate

negative gradient

(Least-Squares) Linear Regression

The update rule turns out to be this:

"On-line" mode:

Vectorized

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
        (b) $\nabla_{\mathbf{w}}\mathcal{L} = \big(y^{[i]} - \hat{y}^{[i]}\big)\mathbf{x}^{[i]}$
            $\nabla_{b}\mathcal{L} = \big(y^{[i]} - \hat{y}^{[i]}\big)$
        (c) $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}}\mathcal{L})$
            $b := b + \eta \times \underbrace{(-\nabla_{b}\mathcal{L})}_{\text{negative gradient}}$
```

learning rate

$$\Longleftrightarrow$$

For-Loop

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
    B. For weight $j$ in $\{1, ..., m\}$:
            (b) $\frac{\partial \mathcal{L}}{\partial w_j} = \big(y^{[i]} - \hat{y}^{[i]}\big)x_j^{[i]}$
            (c) $w_j := w_j + \eta \times (-\frac{\partial \mathcal{L}}{\partial w_j})$
    C.      $\frac{\partial \mathcal{L}}{\partial b} = \big(y^{[i]} - \hat{y}^{[i]}\big)$
            $b := b + \eta \times (-\frac{\partial \mathcal{L}}{\partial b})$
```

(Least-Squares) Linear Regression

The update rule turns out to be this:

"On-line" mode

```
1.  Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2.  For every training epoch:
    A.  For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a)  $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
    B.  For weight $j$ in $\{1, ..., m\}$:
        (b)  $\frac{\partial \mathcal{L}}{\partial w_j} = \big(y^{[i]} - \hat{y}^{[i]}\big)x_j^{[i]}$
        (c)  $w_j := w_j + \boxed{\eta \times (-\frac{\partial \mathcal{L}}{\partial w_j})}$
    C.  $\frac{\partial \mathcal{L}}{\partial b} = \big(y^{[i]} - \hat{y}^{[i]}\big)$
        $b := b + \boxed{\eta \times (-\frac{\partial \mathcal{L}}{\partial b})}$
```

Coincidentally, this appears almost to be the same as the perceptron rule, except that the prediction is a real number and we have a learning rate

This learning rule (from the previous slide)
is called (stochastic) gradient descent.
So, how did we get there?

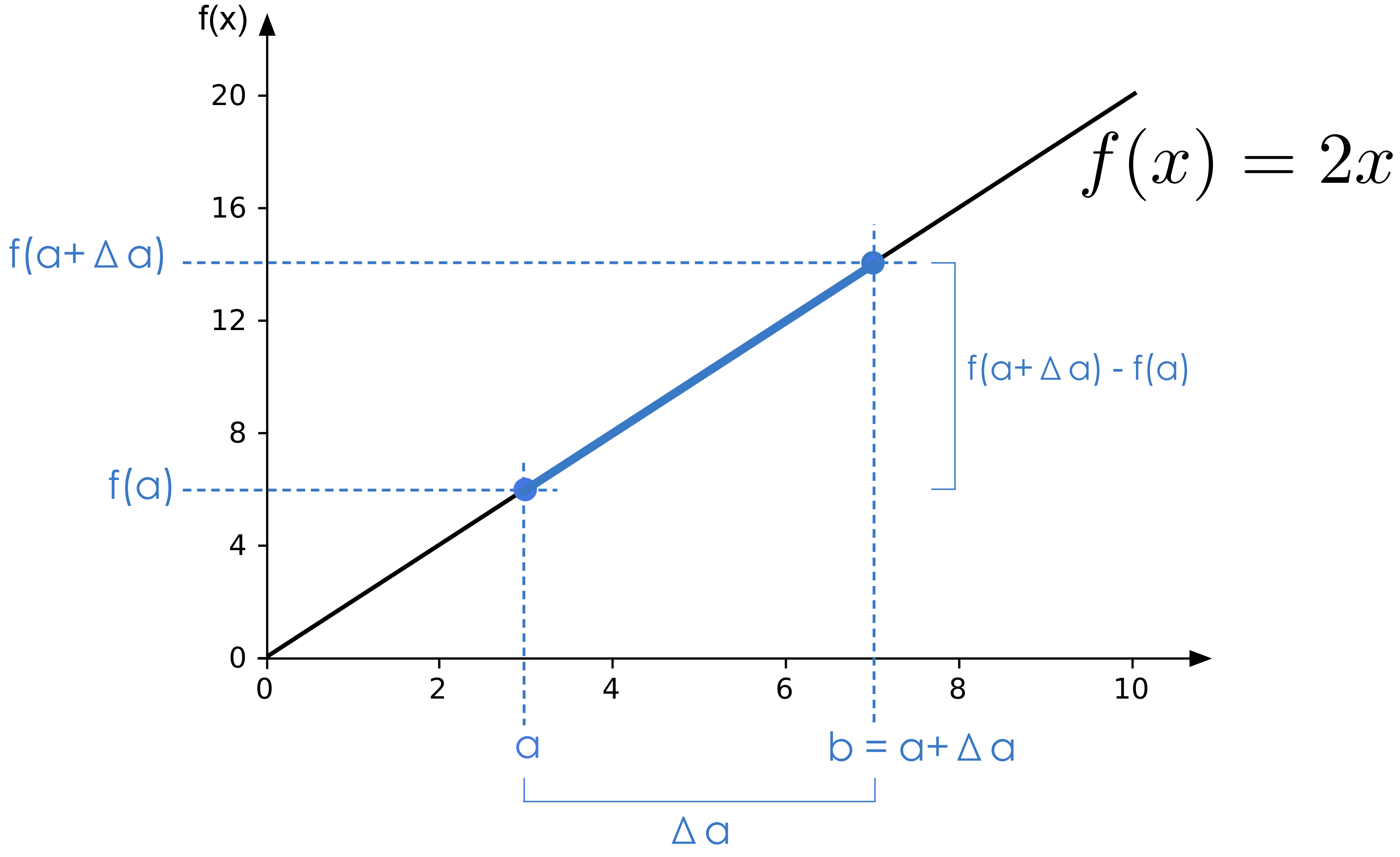
First, let's briefly cover relevant background info ...
(Optional section)

Recapping Derivative Rules

1. Online, batch, and minibatch mode
2. Relation between perceptron and linear regression
3. An iterative training algorithm for linear regression
4. **(Optional) Calculus refresher I: Derivatives**
5. (Optional) Calculus refresher II: Gradients
6. Understanding gradient descent
7. Training an adaptive linear neuron (Adaline)

Differential Calculus Refresher

Derivative of a function = "rate of change" = "slope"



$$\text{Slope} = \frac{f(a + \Delta a) - f(a)}{a + \Delta a - a} = \frac{f(a + \Delta a) - f(a)}{\Delta a}$$

Function Derivative

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Example 1: $f(x) = 2x$

$$\begin{aligned}\frac{df}{dx} &= \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{2x + 2\Delta x - 2x}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{2\Delta x}{\Delta x} \\ &= \lim_{\Delta x \to 0} 2.\end{aligned}$$

Numerical vs Analytical/Symbolical Derivatives

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

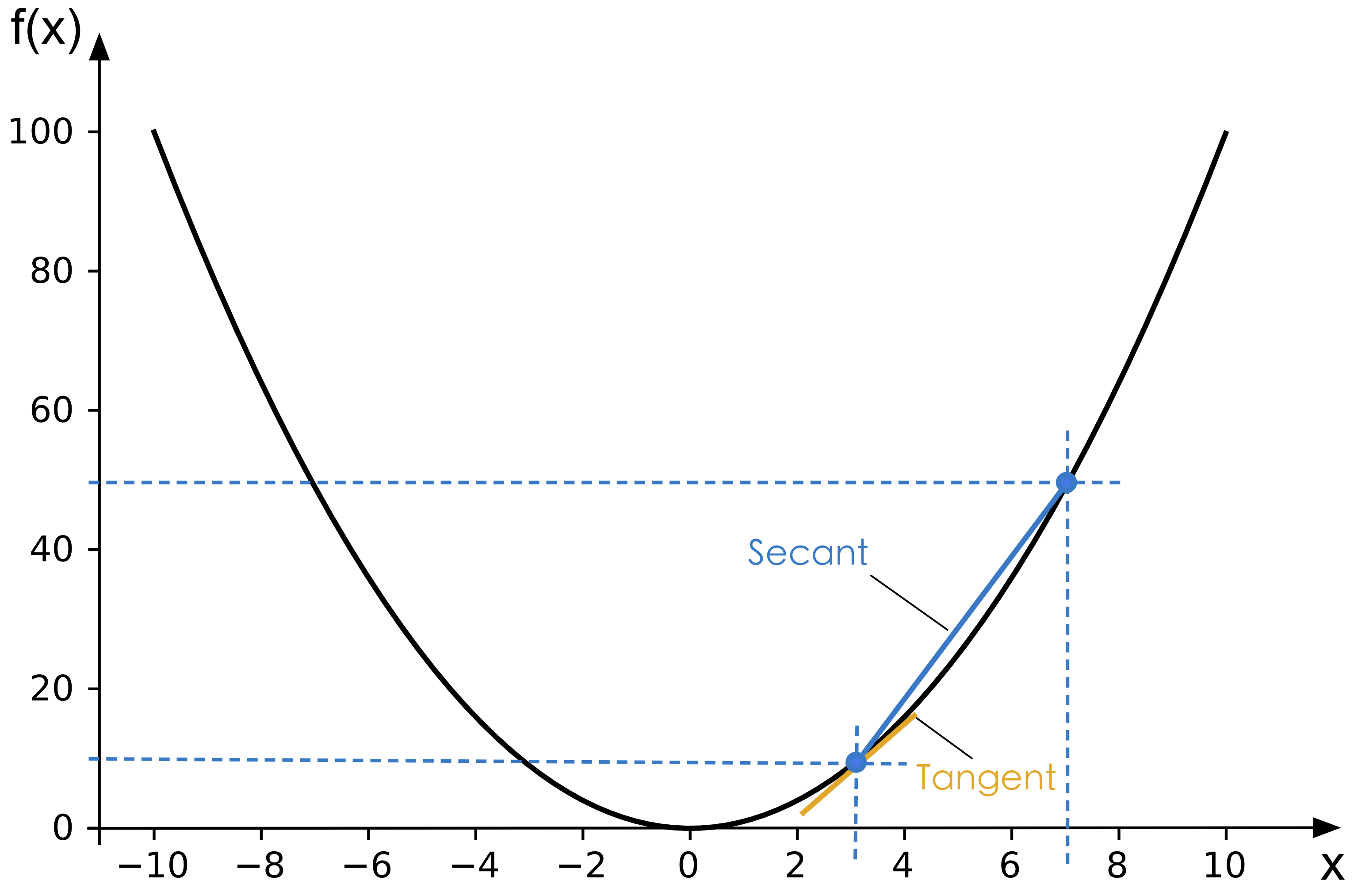
Example 2: $f(x) = x^2$

$$\begin{aligned}
\frac{df}{dx} &= \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{x^2 + 2x\Delta x + (\Delta x)^2 - x^2}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{2x\Delta x + (\Delta x)^2}{\Delta x} \\
&= \lim_{\Delta x \to 0} 2x + \Delta x.
\end{aligned}$$

Numerical vs Analytical/Symbolical Derivatives

Conceptually, we obtained the derivative $\frac{d}{dx}x^2 = 2x$

By approximating the slope (tangent) by a secant between two points (as before)



A Cheatsheet for You (1)

	Function $f(x)$	Derivative with respect to x
1	a	0
2	x	1
3	ax	a
4	x^2	$2x$
5	x^a	ax^{a-1}
6	a^x	$\log(a)a^x$
7	$\log(x)$	$1/x$
8	$\log_a(x)$	$1/(x\log(a))$
9	$\sin(x)$	$\cos(x)$
10	$\cos(x)$	$-\sin(x)$
11	$\tan(x)$	$\sec^2(x)$

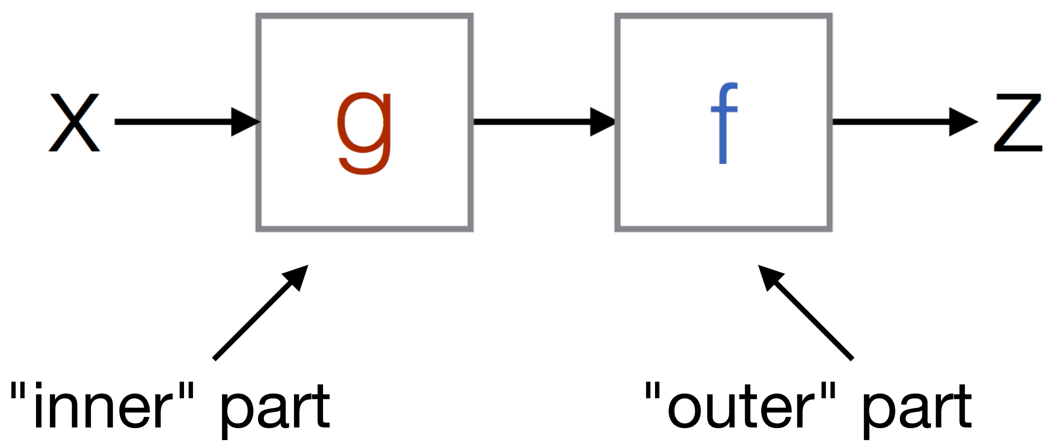
A Cheatsheet for You (2)

	Function	Derivative
Sum Rule	$f(x) + g(x)$	$f'(x) + g'(x)$
Difference Rule	$f(x) - g(x)$	$f'(x) - g'(x)$
Product Rule	$f(x)g(x)$	$f'(x)g(x) + f(x)g'(x)$
Quotient Rule	$f(x)/g(x)$	$[g(x)f'(x) - f(x)g'(x)]/[g(x)]^2$
Reciprocal Rule	$1/f(x)$	$-[f'(x)]/[f(x)]^2$
Chain Rule	$f(g(x))$	$f'(g(x))g'(x)$

Chain Rule & "Computation Graph" Intuition

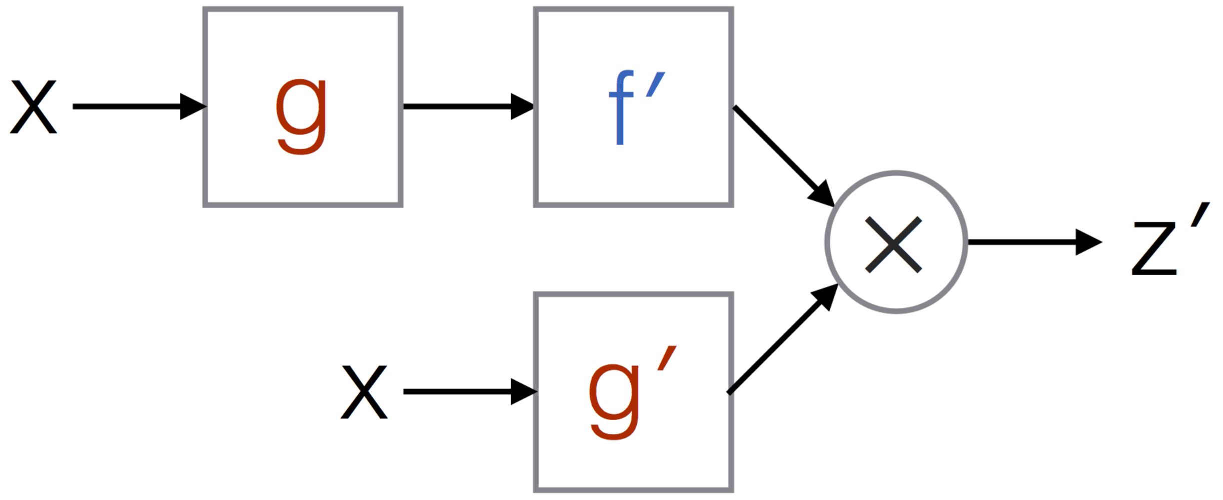
$$F(x) = f\,(\,g(x)\,) = z$$

Decomposition of some (nested) function:



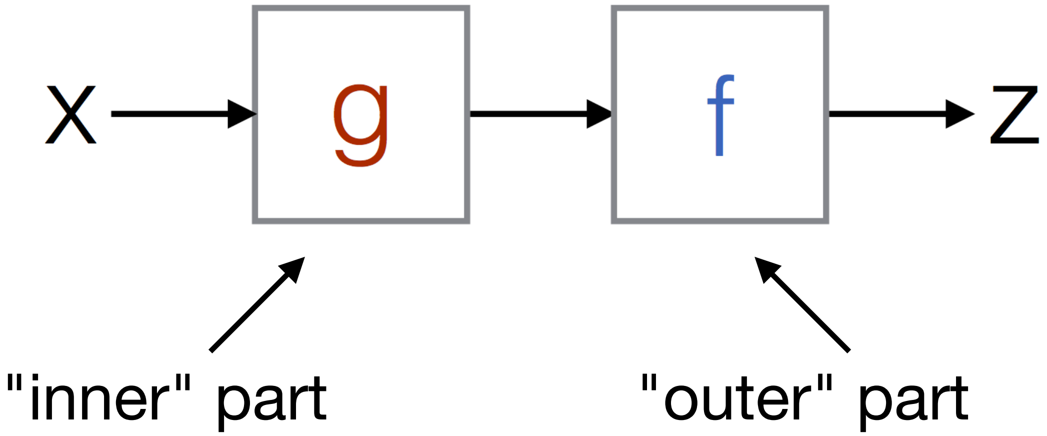
$$F'(x) = f'(\,g(x)\,)\,g'(x) = z'$$

Derivative of that nested function:



Chain Rule & "Computation Graph" Intuition

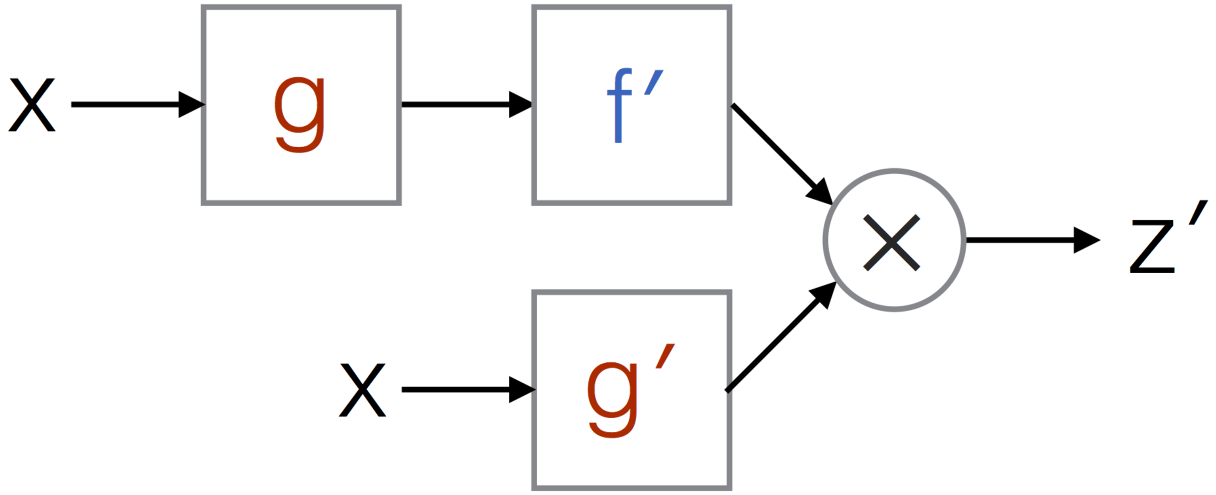
$$F(x) = f\,(\,g(x)\,) = z$$



Later, we will see that PyTorch can do that automatically for us :) (PyTorch literally keeps a computation graph in the background)

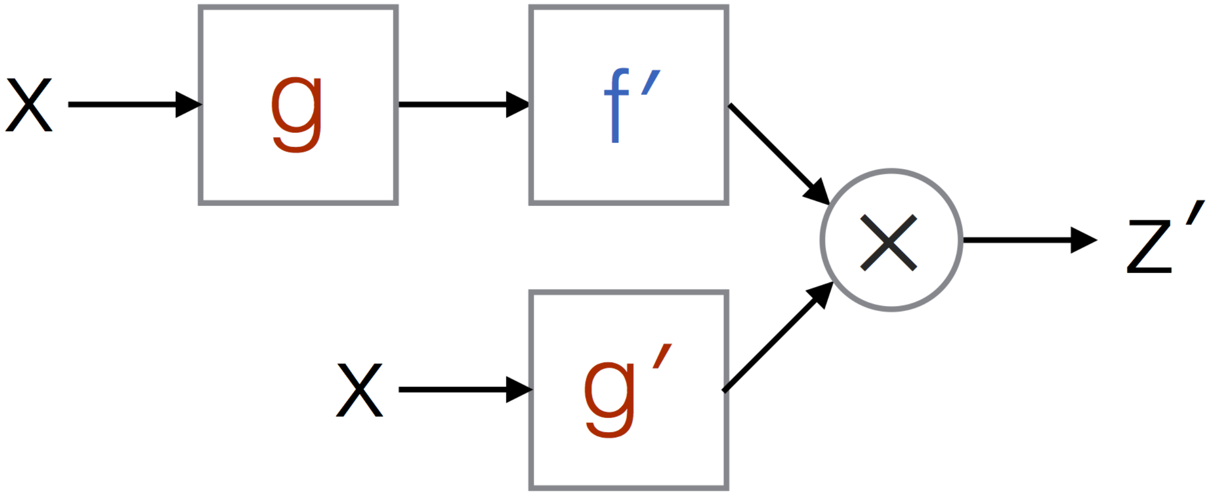
$$F'(x) = f'(\,g(x)\,)\,g'(x) = z'$$

Also, PyTorch can compute the derivatives of most (differentiable) functions automatically



Chain Rule & "Computation Graph" Intuition

$$F'(x) = f'(\, g(x)\,)\, g'(x) = z'$$



In text, for efficiency, we will mostly use the Leibniz notation:

$$\frac{d}{dx}\left[f(g(x))\right] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Chain Rule Example

$$\frac{d}{dx}\left[f(g(x))\right] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Example: $f(x) = \log(\sqrt{x})$

substituting $$\frac{df}{dx} = \frac{d}{dg}\log(g) \cdot \frac{d}{dx}\sqrt{x}$$

with $\frac{d}{dg}\log(g) = \frac{1}{g} = \frac{1}{\sqrt{x}}$ and $\frac{d}{dx}x^{1/2} = \frac{1}{2}x^{-1/2} = \frac{1}{2\sqrt{x}}$

leads us to the solution $$\frac{df}{dx} = \frac{1}{\sqrt{x}} \cdot \frac{1}{2\sqrt{x}} = \frac{1}{2x}$$

Chain Rule for Arbitrarily Long Function Compositions

$$F(x) = f(g(h(u(v(x)))))$$

$$\begin{aligned}\frac{dF}{dx} = \frac{d}{dx}F(x) &= \frac{d}{dx}f(g(h(u(v(x))))) \\ &= \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}\end{aligned}$$

Chain Rule for Arbitrarily Long Function Compositions

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x)))))$$

$$\boxed{= \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}}$$

Also called "reverse mode" as we start with the outer function. In neural nets, this will be from right to left.

We could also start from the inner parts ("forward mode")

$$\frac{dv}{dx} \cdot \frac{du}{dv} \cdot \frac{dh}{du} \cdot \frac{dg}{dh} \cdot \frac{df}{dg}$$

- Backpropagation (covered later) is basically "reverse" mode auto-differentiation
- It is cheaper than forward mode if we work with gradients, since then we have matrix-"vector" multiplications instead of matrix multiplications

Gradients: Derivatives of Multivariable Functions

1. Online, batch, and minibatch mode
2. Relation between perceptron and linear regression
3. An iterative training algorithm for linear regression
4. (Optional) Calculus refresher I: Derivatives
5. **(Optional) Calculus refresher II: Gradients**
6. Understanding gradient descent
7. Training an adaptive linear neuron (Adaline)

Gradients: Derivatives of Multivariable* Functions

***note that in some fields, the terms "multivariable" and "multivariate" are used interchangeably,**
but here, we really mean "multivariable" because "multivariate" means "multiple outputs", which is
not the case here -- similarly, in most DL applications output one prediction value, or one prediction value per training example

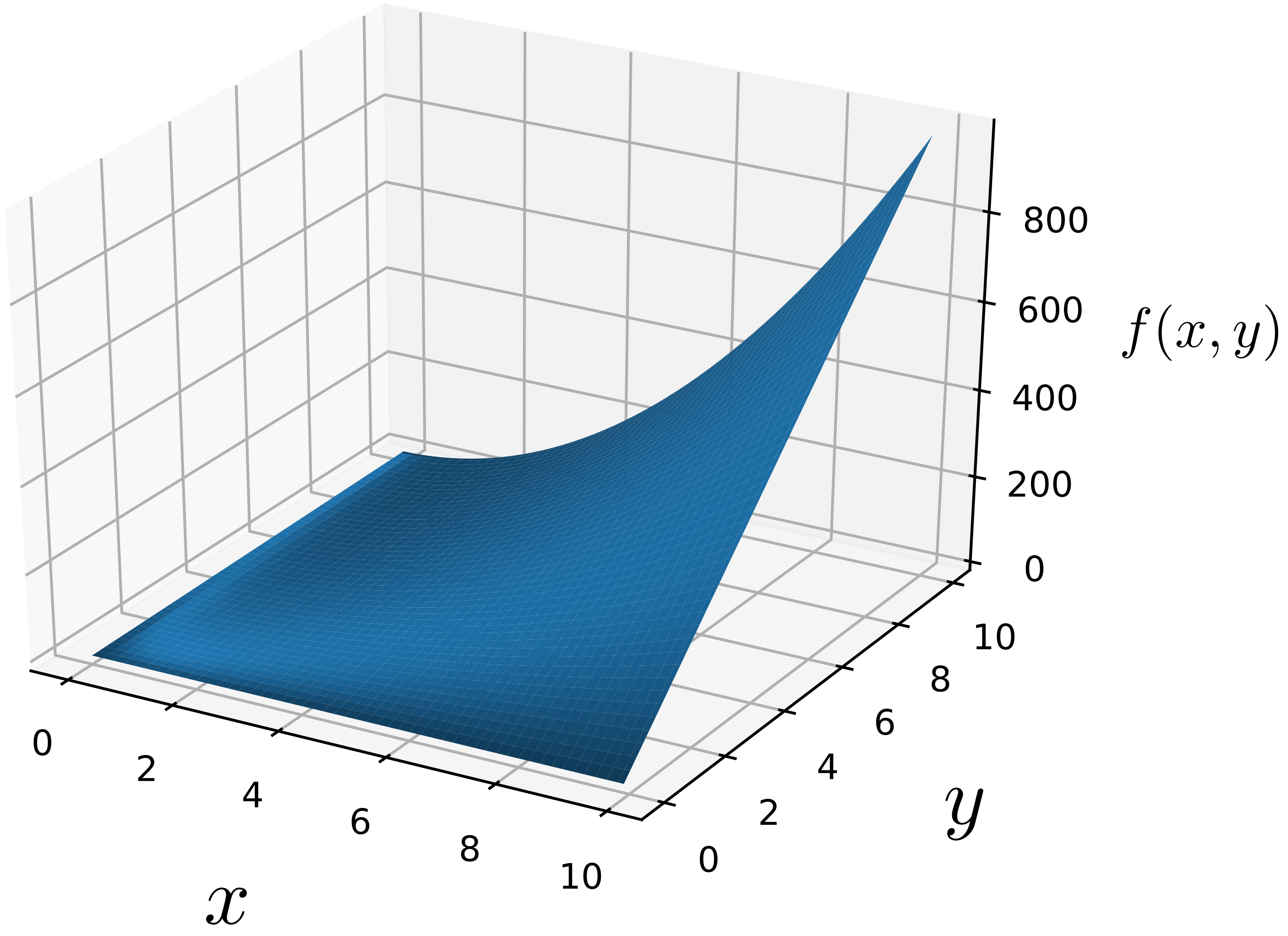
$$f(x, y, z, ...)$$

$$\nabla f = \begin{bmatrix} \partial f / \partial x \\ \partial f / \partial y \\ \partial f / \partial z \\ \vdots \end{bmatrix}$$

For gradients, we use the "partial" symbol to denote partial derivatives; more of a notational convention and the concept is the same as before when we were computing ordinary derivatives (denoted them as "d")

Gradients: Derivatives of Multivariable Functions

Example: $f(x, y) = x^2 y + y$



Gradients: Derivatives of Multivariable Functions

Example: $f(x, y) = x^2 y + y$

$$\nabla f(x, y) = \begin{bmatrix} \partial f / \partial x \\ \partial f / \partial y \end{bmatrix},$$

where

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x} x^2 y + y = 2xy$$

(via the power rule and constant rule), and

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y} x^2 y + y = x^2 + 1.$$

So, the gradient of the function f is defined as

$$\nabla f(x, y) = \begin{bmatrix} 2xy \\ x^2 + 1 \end{bmatrix}.$$

Gradients & the Multivariable Chain Rule

Suppose we have a composite function like this:

$$f\big(g(x), h(x)\big)$$

Remember the regular chain rule for a single input:

$$\frac{d}{dx}\big[f(g(x))\big] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

For two inputs, we now have

$$\frac{d}{dx}\big[f(g(x), h(x))\big] = \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial h} \cdot \frac{dh}{dx}$$

Gradients & the Multivariable Chain Rule

$$f\big(g(x), h(x)\big)$$

$$\frac{d}{dx}\big[f(g(x), h(x))\big] = \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial h} \cdot \frac{dh}{dx}$$

Example:

$$f\big(g, h\big) = g^2 h + h$$

where $g(x) = 3x$, and $h(x) = x^2$

$$\frac{\partial f}{\partial g} = 2gh \qquad \frac{\partial f}{\partial h} = g^2 + 1$$

$$\frac{dg}{dx} = \frac{d}{dx} 3x = 3 \qquad \frac{dh}{dx} = \frac{d}{dx} x^2 = 2x$$

$$\begin{aligned}\frac{d}{dx}\big[f(g(x))\big] &= [2gh \cdot 3] + [(g^2 + 1) \cdot 2x] \\ &= 2xg^2 + 6gh + 2x\end{aligned}$$

Gradients & the Multivariable Chain Rule in Vector Form

$$f\big(g(x), h(x)\big)$$

$$\begin{aligned}\frac{d}{dx}\big[f(g(x), h(x))\big] &= \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial h} \cdot \frac{dh}{dx} \\ &= \nabla f \cdot \mathbf{v}'(x).\end{aligned}$$

Where

$$\mathbf{v}(x) = \begin{bmatrix} g(x) \\ h(x) \end{bmatrix} \qquad \mathbf{v}'(x) = \frac{d}{dx}\begin{bmatrix} g(x) \\ h(x) \end{bmatrix} = \begin{bmatrix} dg/dx \\ dh/dx \end{bmatrix}$$

Putting it together:

$$\nabla f \cdot \mathbf{v}'(x) = \begin{bmatrix} \partial f/\partial g \\ \partial f/\partial h \end{bmatrix} \cdot \begin{bmatrix} dg/dx \\ dh/dx \end{bmatrix} = \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial h} \cdot \frac{dh}{dx}$$

The Jacobian (Matrix)

$$\mathbf{f}(x_1, x_2, ..., x_m) = \left[\begin{array}{c} f_1\left(x_1, x_2, x_3, \cdots x_m\right) \\ f_2\left(x_1, x_2, x_3, \cdots x_m\right) \\ f_3\left(x_1, x_2, x_3, \cdots x_m\right) \\ \vdots \\ f_m\left(x_1, x_2, x_3, \cdots x_m\right) \end{array}\right]$$

$$J\left(x_1, x_2, x_3, \cdots x_m\right) = \left[\begin{array}{ccccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \cdots & \frac{\partial f_3}{\partial x_m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial x_3} & \cdots & \frac{\partial f_m}{\partial x_m} \end{array}\right]$$

The Jacobian (Matrix)

$$\mathbf{f}(x_1, x_2, \ldots, x_m) = \begin{bmatrix} f_1\left(x_1, x_2, x_3, \cdots x_m\right) \\ f_2\left(x_1, x_2, x_3, \cdots x_m\right) \\ f_3\left(x_1, x_2, x_3, \cdots x_m\right) \\ \vdots \\ f_m\left(x_1, x_2, x_3, \cdots x_m\right) \end{bmatrix}$$

$$J\left(x_1, x_2, x_3, \cdots x_m\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \cdots & \frac{\partial f_3}{\partial x_m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial x_3} & \cdots & \frac{\partial f_m}{\partial x_m} \end{bmatrix} \begin{matrix} \left(\nabla f_1\right)^\top \\ \\ \\ \\ \\ \end{matrix}$$

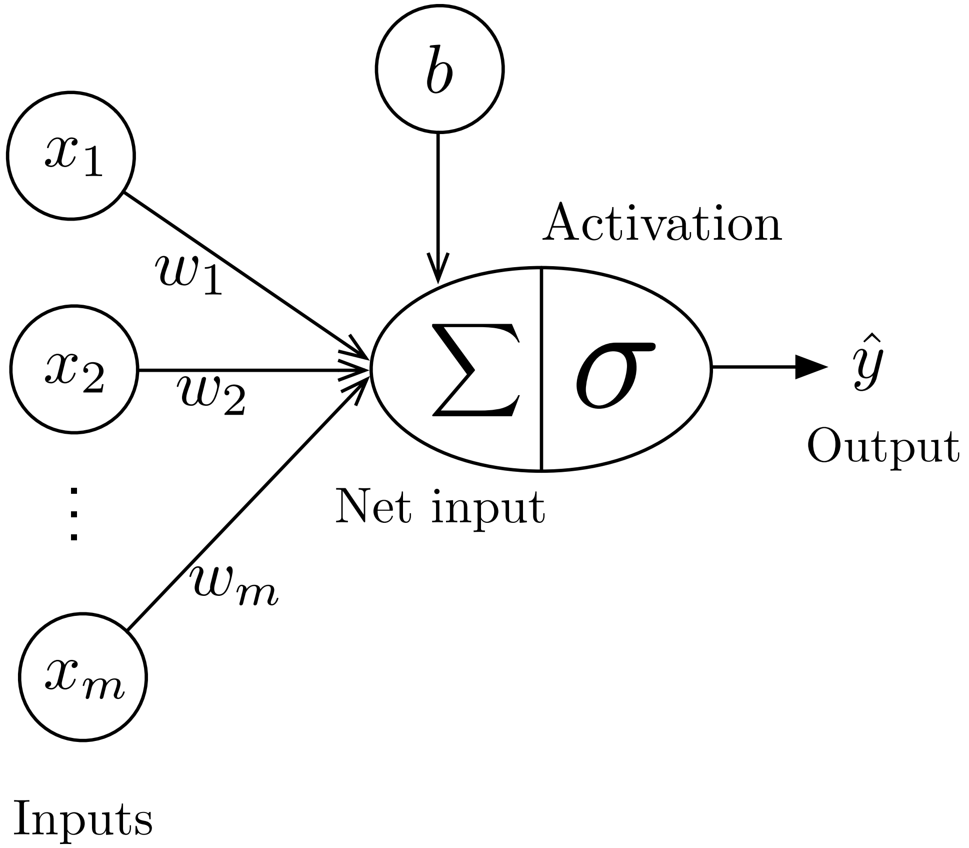
Second Order Derivatives

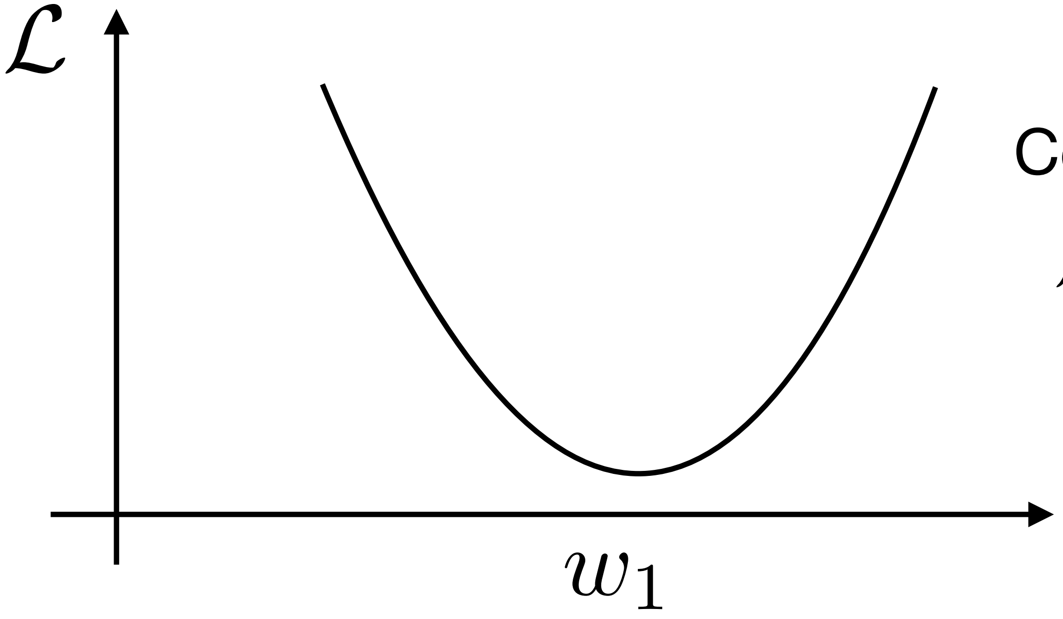
Lucky for you, we won't need second order derivatives in this class ;)

Training Linear Regression with Gradient Descent

1. Online, batch, and minibatch mode
2. Relation between perceptron and linear regression
3. An iterative training algorithm for linear regression
4. (Optional) Calculus refresher I: Derivatives
5. (Optional) Calculus refresher II: Gradients
6. **Understanding gradient descent**
7. Training an adaptive linear neuron (Adaline)

Back to Linear Regression



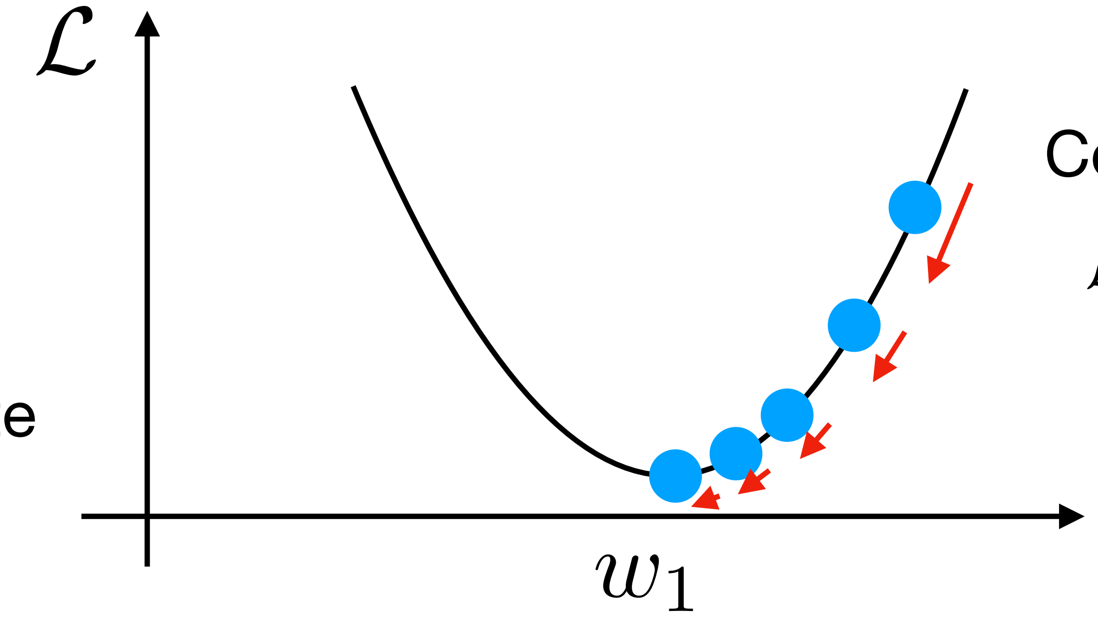


Convex loss function

$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Gradient Descent

Learning rate and steepness of the gradient determine how much we update

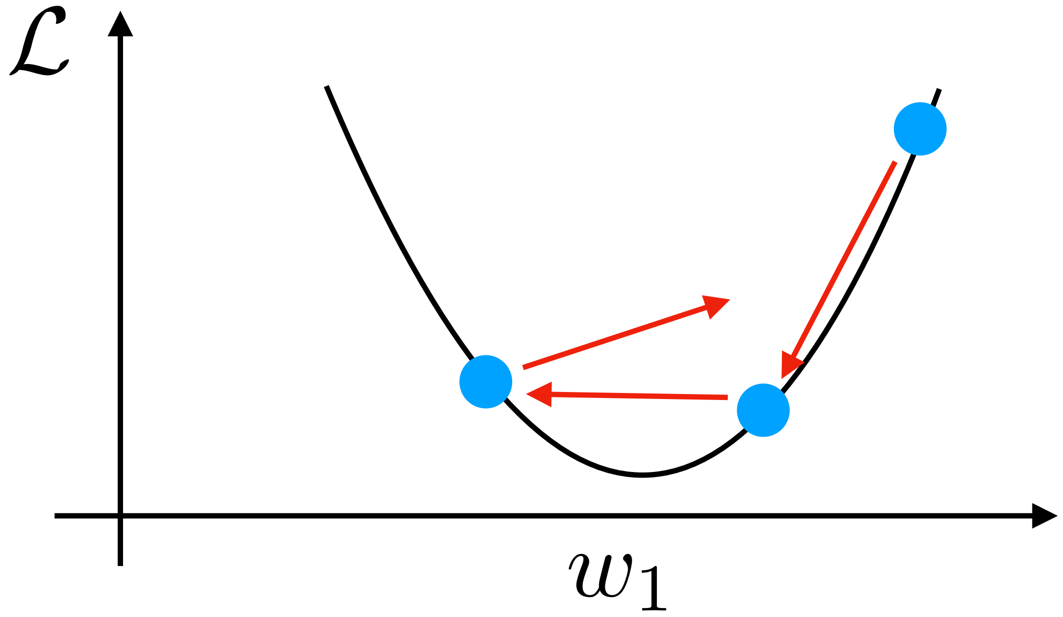


Convex loss function

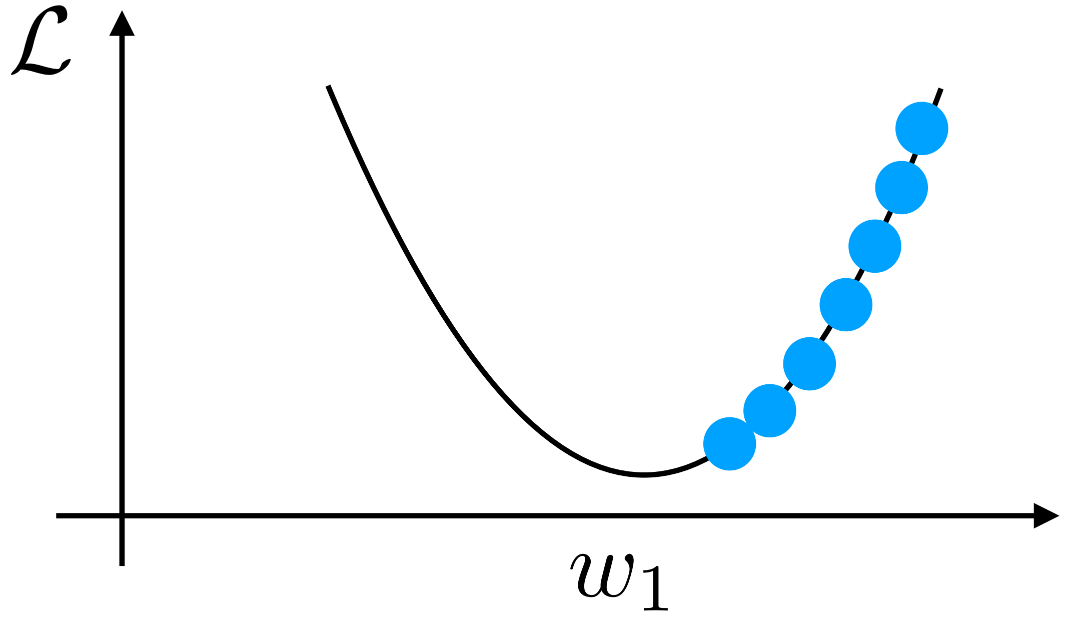
$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Gradient Descent

If the learning rate is too large, we can overshoot



If the learning rate is too small, convergence is very slow



(Least-Squares) Linear Regression

The update rule turns out to be this:

"On-line" mode

Perceptron learning rule

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w} + b)$
        (b) $\mathrm{err} := (y^{[i]} - \hat{y}^{[i]})$
        (c) $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$
            $b := b + err$
```

Stochastic gradient descent

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w} + b)$
        (b) $\nabla_{\mathbf{w}}\mathcal{L} = (y^{[i]} - \hat{y}^{[i]})\mathbf{x}^{[i]}$
            $\nabla_{b}\mathcal{L} = (y^{[i]} - \hat{y}^{[i]})$
        (c) $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}}\mathcal{L})$
            $b := b + \underset{\text{learning rate}}{\eta} \times \underbrace{(-\nabla_{b}\mathcal{L})}_{\text{negative gradient}}$
```

Linear Regression Loss Derivative

$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Sum Squared Error (SSE) loss

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\
&= \frac{\partial}{\partial w_j} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]})^2 \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}^{[i]} \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} x_j^{[i]} \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) x_j^{[i]}
\end{aligned}$$

(Note that the activation function is the identity function in linear regression)

Linear Regression Loss Derivative (alt.)

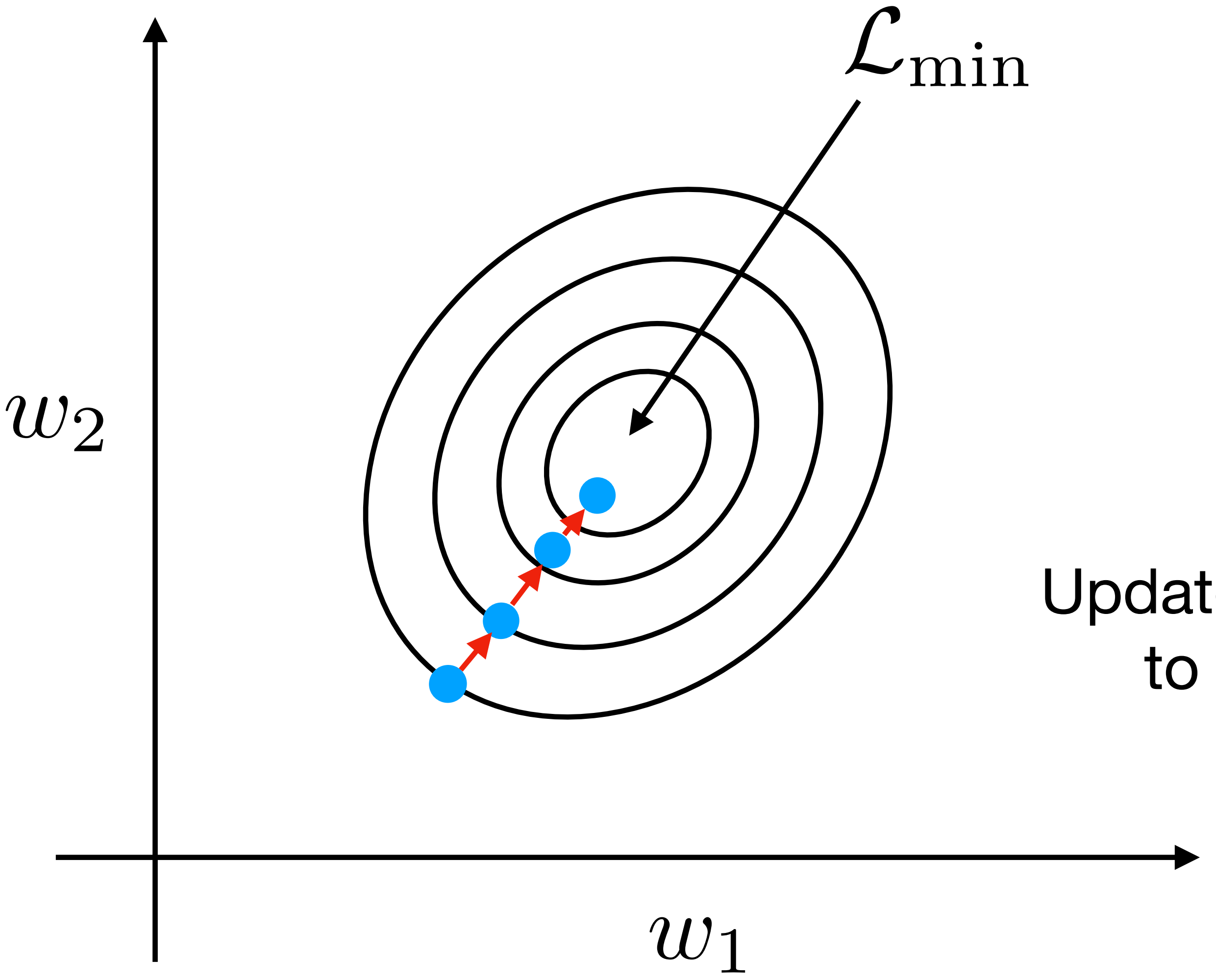
$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Mean Squared Error (MSE) loss often scaled by factor 1/2 for convenience

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\
&= \frac{\partial}{\partial w_j} \sum_i \frac{1}{2n} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]})^2 \\
&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}^{[i]} \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} x_j^{[i]} \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) x_j^{[i]}
\end{aligned}$$

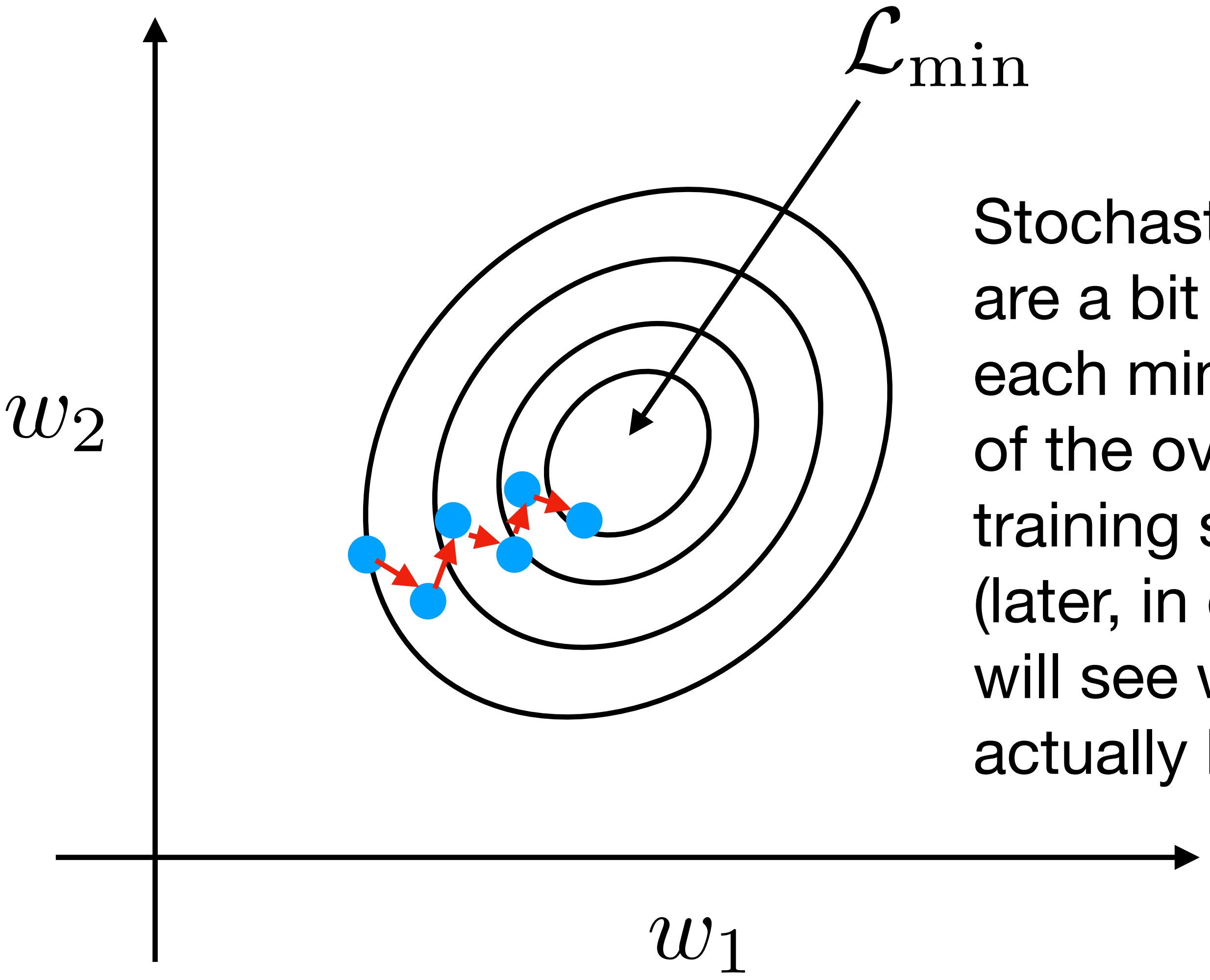
(Note that the activation function is the identity function in linear regression)

Batch Gradient Descent as Surface Plot



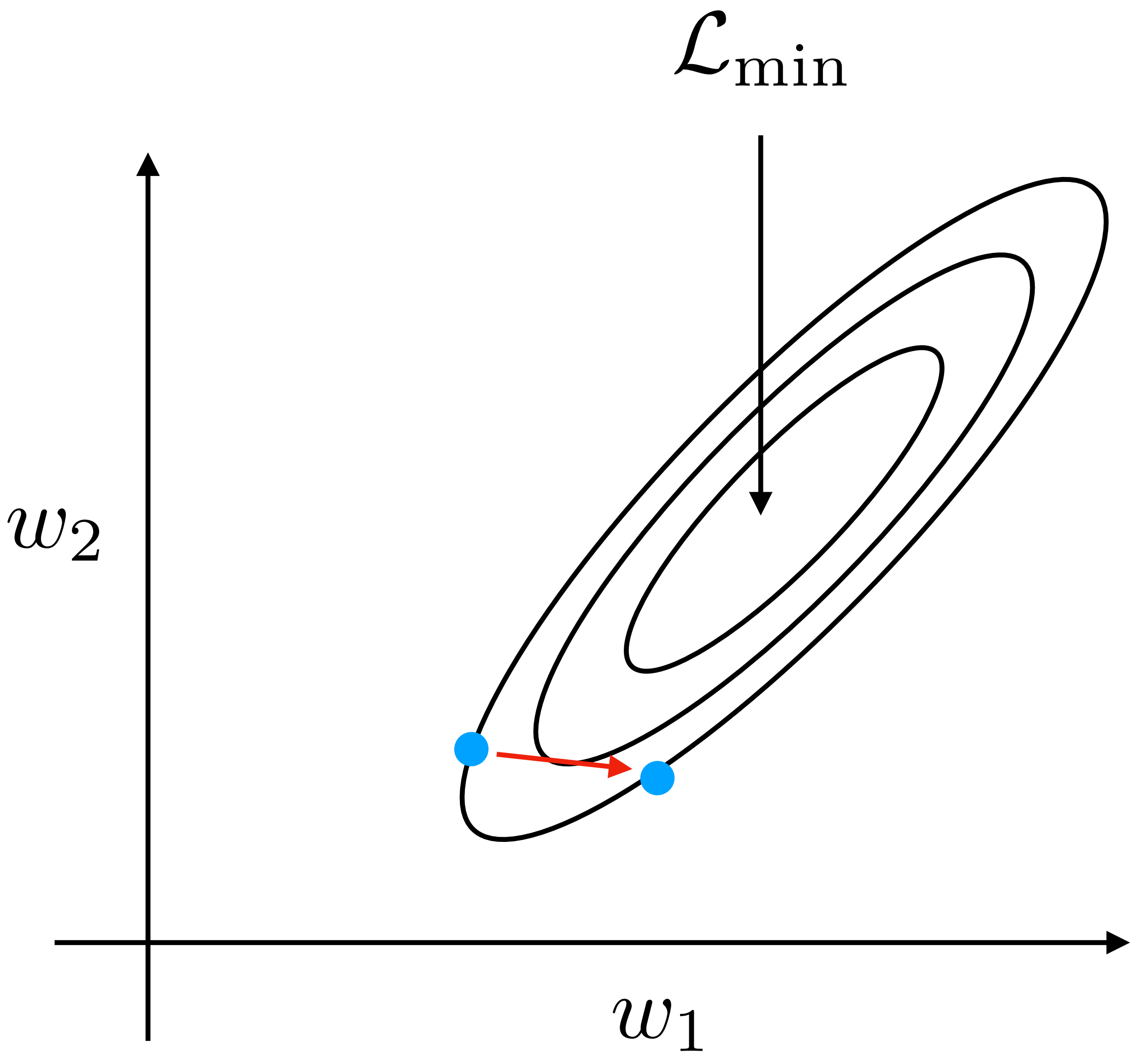
Updates perpendicular
to contour lines

Stochastic Gradient Descent as Surface Plot



Stochastic updates
are a bit noisier, because
each minibatch is an approximation
of the overall loss on the
training set
(later, in deep neural nets, we
will see why noisier updates are
actually helpful)

Batch Gradient Descent as Surface Plot



If inputs are on very different scales some weights will update more than others ... and it will also harm convergence

(always normalize inputs!)

Training a single-layer neural network with gradient descent

1. Online, batch, and minibatch mode
2. Relation between perceptron and linear regression
3. An iterative training algorithm for linear regression
4. (Optional) Calculus refresher I: Derivatives
5. (Optional) Calculus refresher II: Gradients
6. Understanding gradient descent
7. **Training an adaptive linear neuron (Adaline)**

ADALINE

Widrow and Hoff's ADALINE (1960)

A nicely differentiable neuron model

Widrow, B., & Hoff, M. E. (1960). *Adaptive switching circuits* (No. TR-1553-1). Stanford Univ Ca Stanford Electronics Labs.

Widrow, B. (1960). *Adaptive" adaline" Neuron Using Chemical" memistors."*.

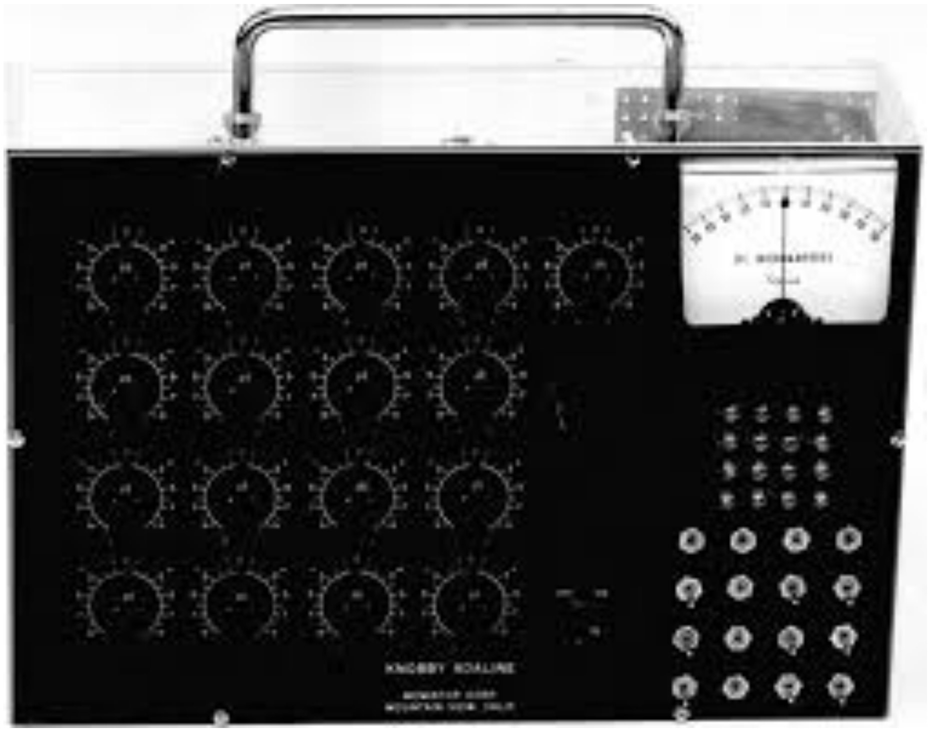
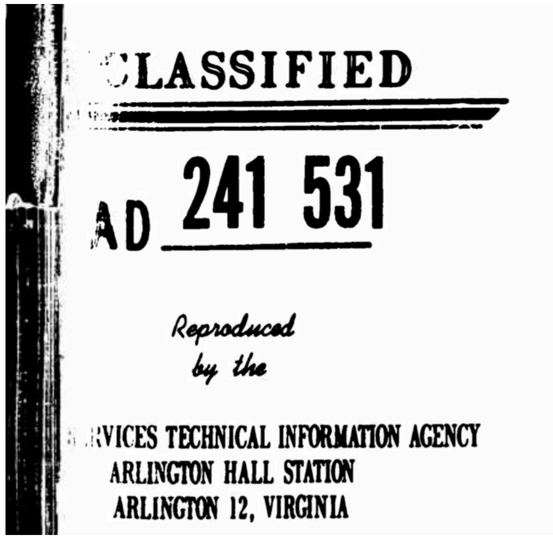


Image source: https://www.researchgate.net/profile/Alexander_Magoun2/publication/265789430/figure/fig2/AS:392335251787780@1470551421849/ADALINE-An-adaptive-linear-neuron-Manually-adapted-synapses-Designed-and-built-by-Ted.png

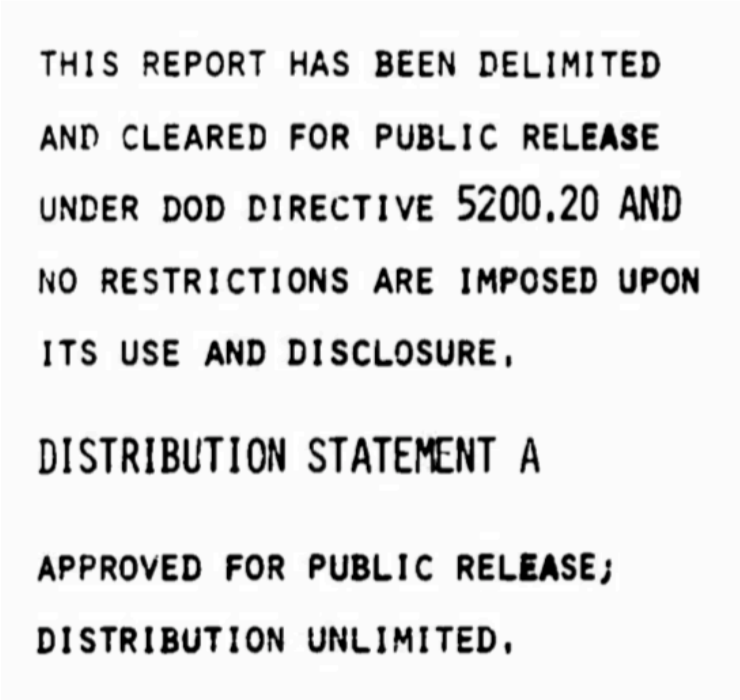


CLASSIFIED

AD 241 531

Reproduced by the

RVICES TECHNICAL INFORMATION AGENCY
ARLINGTON HALL STATION
ARLINGTON 12, VIRGINIA



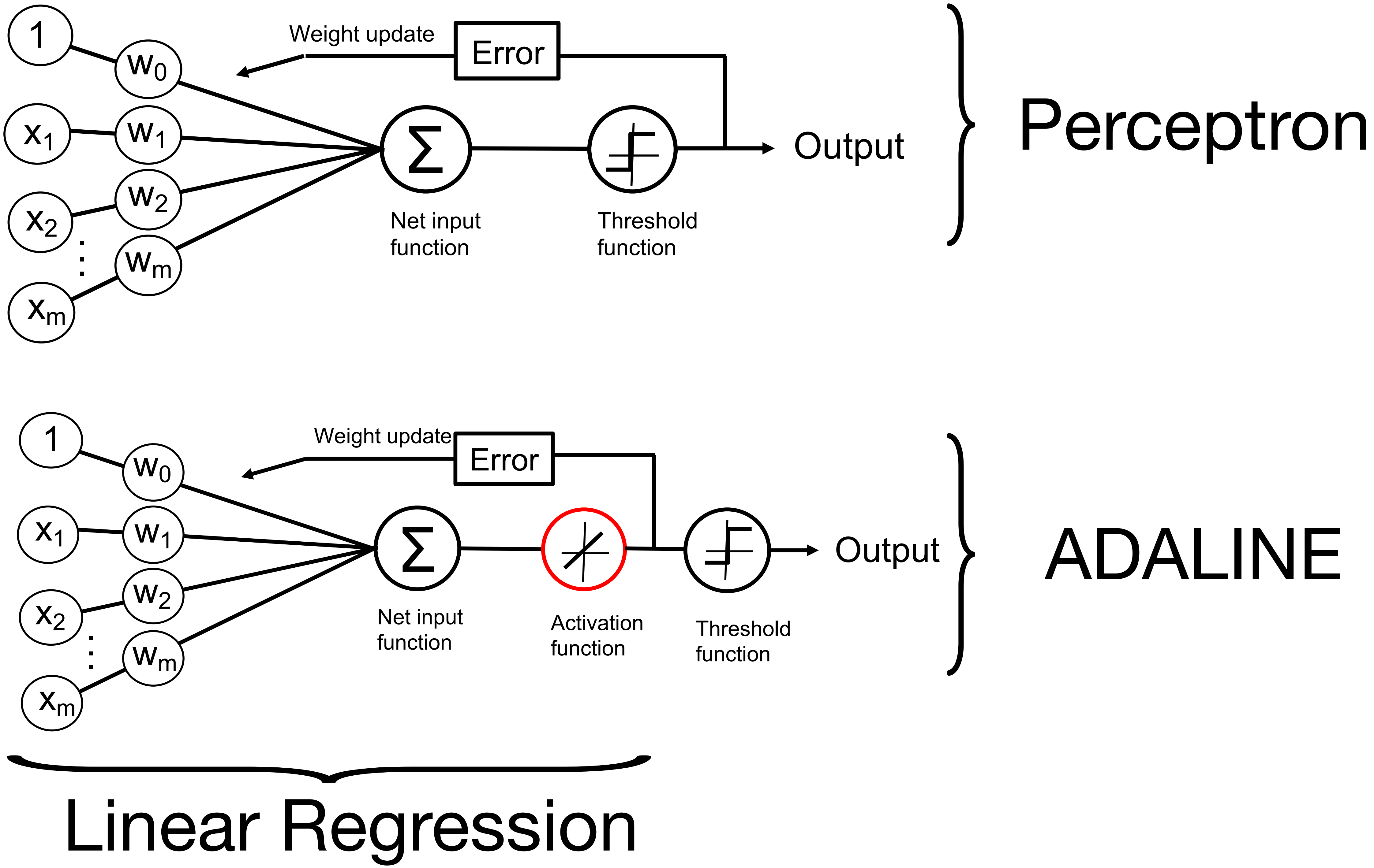
THIS REPORT HAS BEEN DELIMITED AND CLEARED FOR PUBLIC RELEASE UNDER DOD DIRECTIVE 5200.20 AND NO RESTRICTIONS ARE IMPOSED UPON ITS USE AND DISCLOSURE.

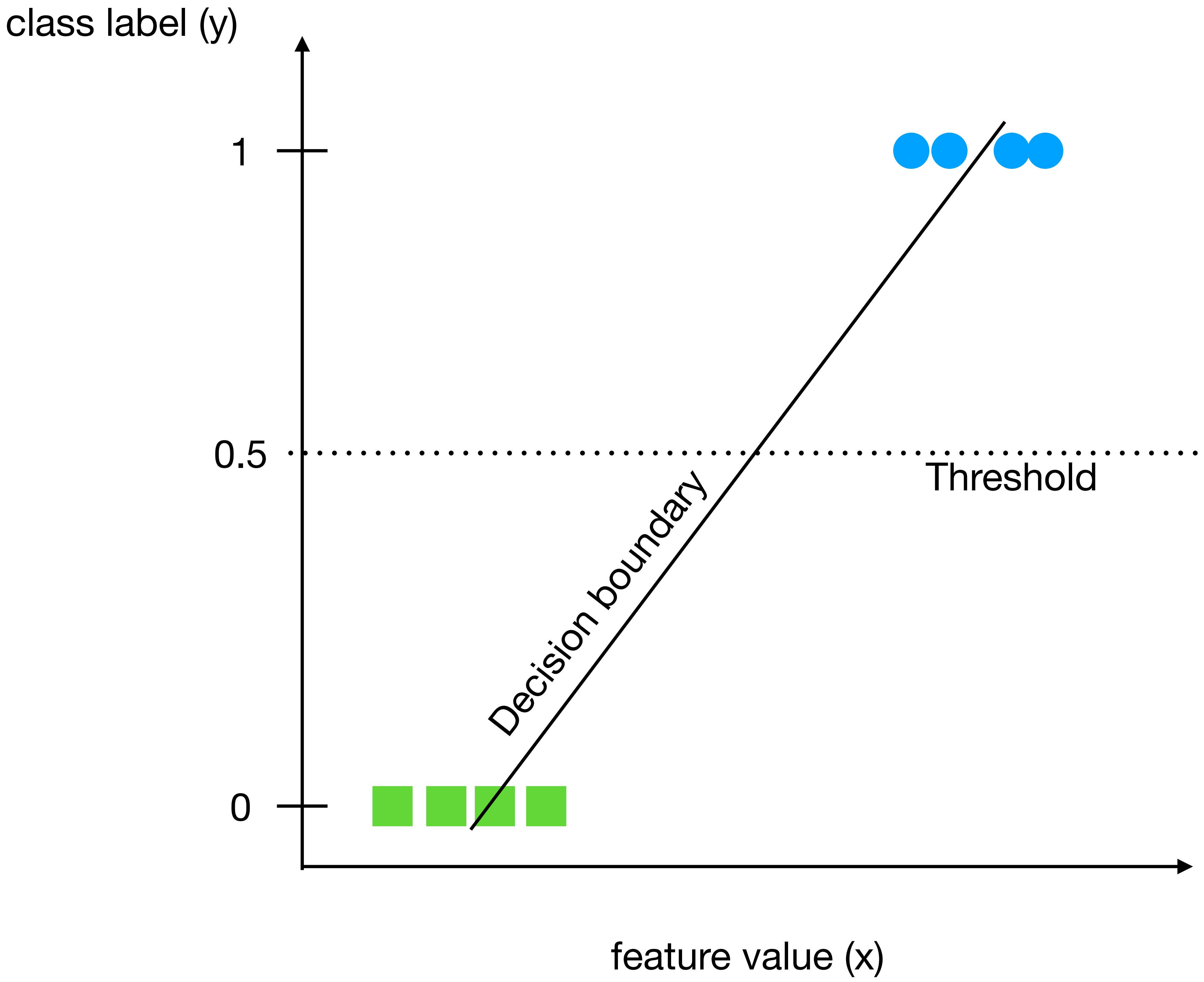
DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ADALINE

ADAptive LInear NEuron




class label (y)
1
0.5
0
Threshold
Decision boundary
feature value (x)

Code Examples

https://github.com/rasbt/stat453-deep-learning-ss21/tree/master/L05/code

Next Lecture:

Neurons with non-linear activation functions