Neural Networks I: How Learning Works

Introduction

▶ Code

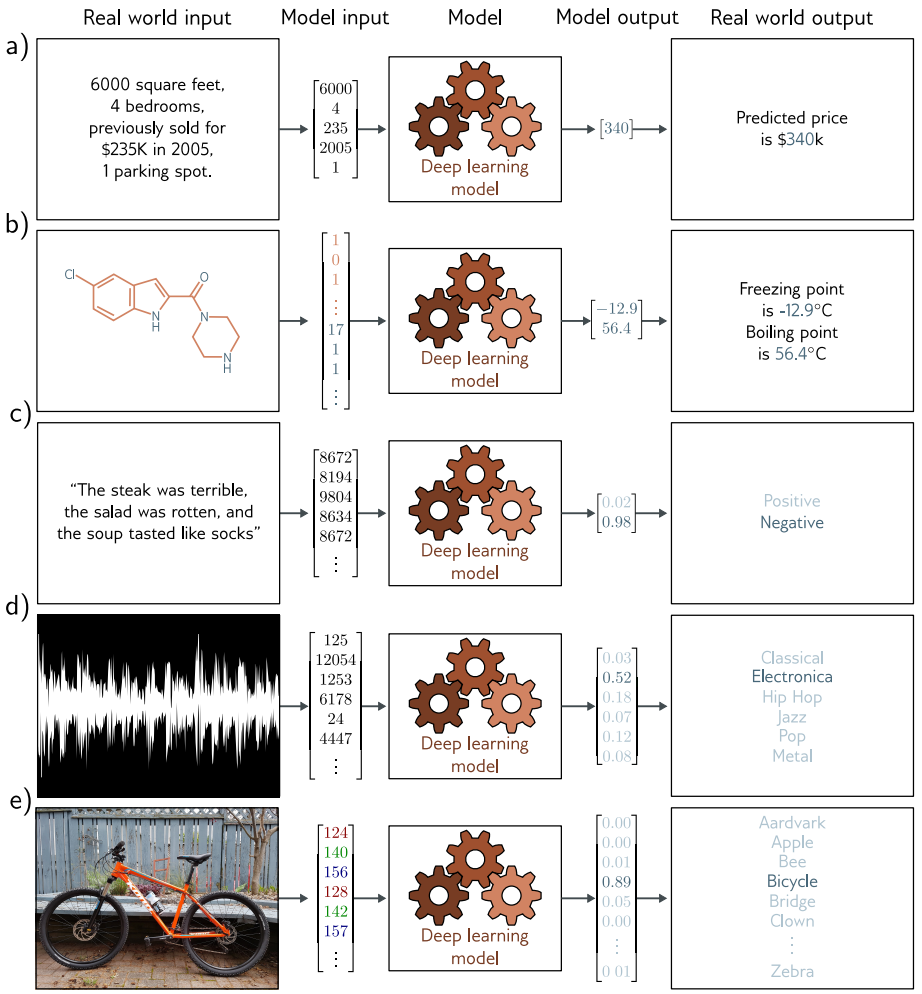
This is the first of two lectures on neural networks. Today is about **how learning works**: we build the model up from regression, define what “training” means, and then look carefully at the machinery that makes it possible.

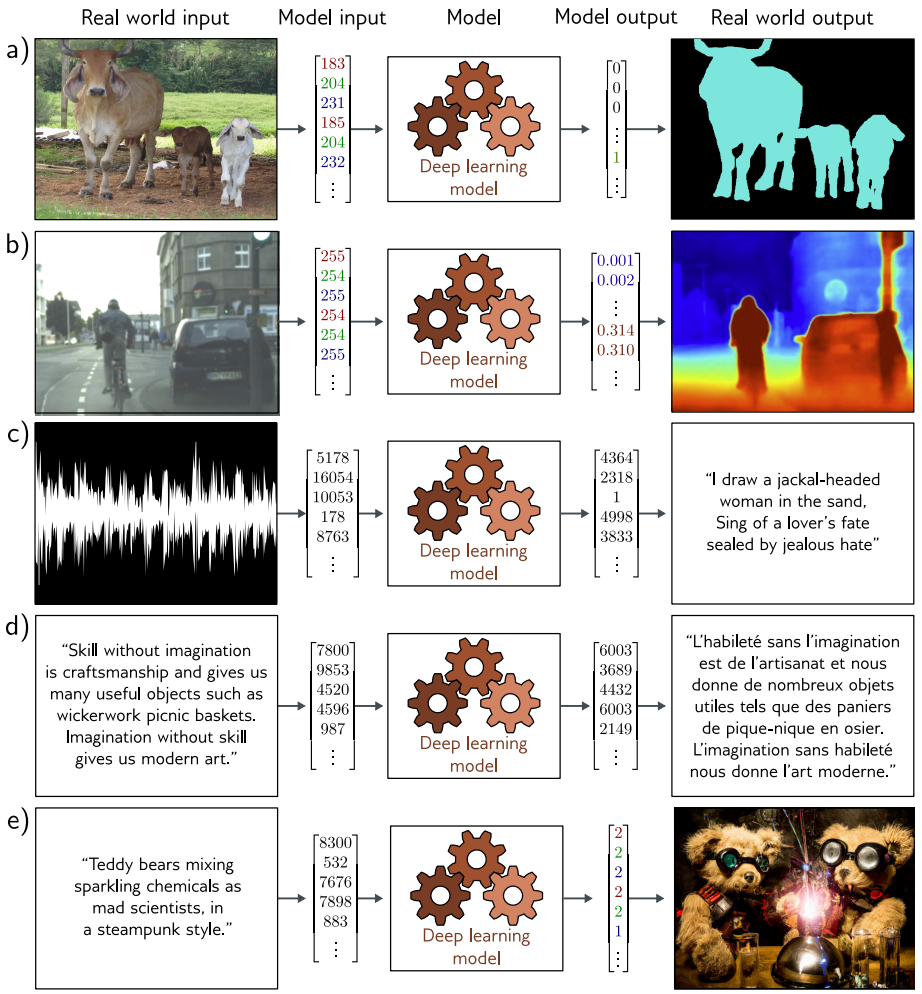
We’ll cover:

- How neural networks extend linear and logistic regression
- The Multi-Layer Perceptron (MLP) as a function of its parameters
- The loss function and gradient descent
- **Backpropagation**: the chain rule organized on a compute graph
- Why we use **stochastic** (mini-batch) gradient descent

Next lecture, Neural Networks II, is about making training work in practice with scikit-learn.

Why Neural Networks?





From Regression to Neural Networks

Linear Regression Revisited

Recall linear regression predicts a continuous output:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p = \mathbf{x}^T \boldsymbol{\beta}$$

Or in matrix form for multiple samples:

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}$$

- **Question:** What's the main limitation of linear regression?

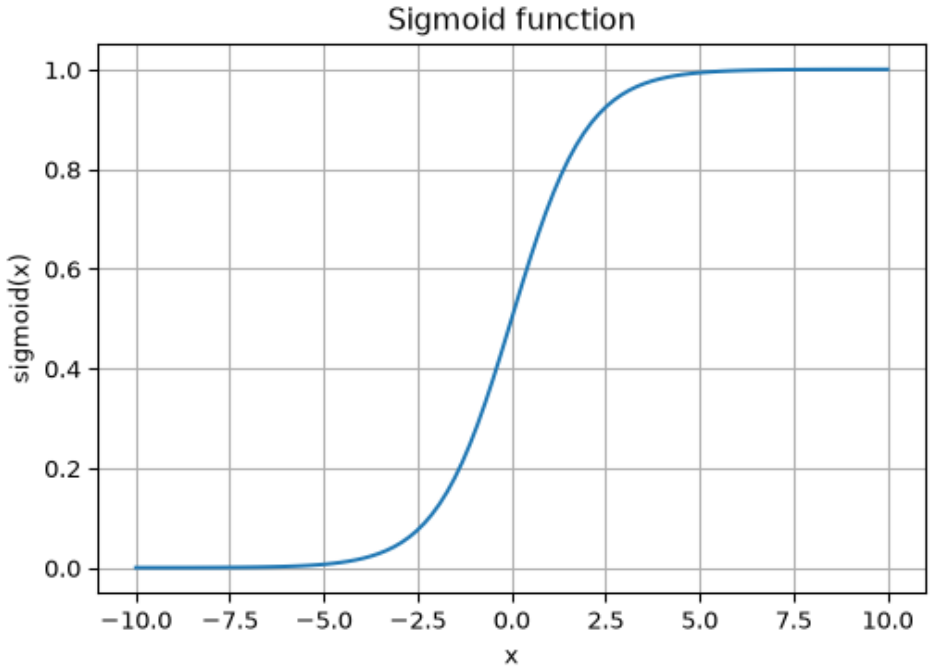
Logistic Regression

- Adds Non-linearity
- For binary classification, logistic regression applies a **sigmoid function**:

$$P(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^T\boldsymbol{\beta}) = \frac{1}{1 + e^{-\mathbf{x}^T\boldsymbol{\beta}}}$$

Logistic Regression, cont.

The sigmoid function introduces non-linearity:



The Key Insight

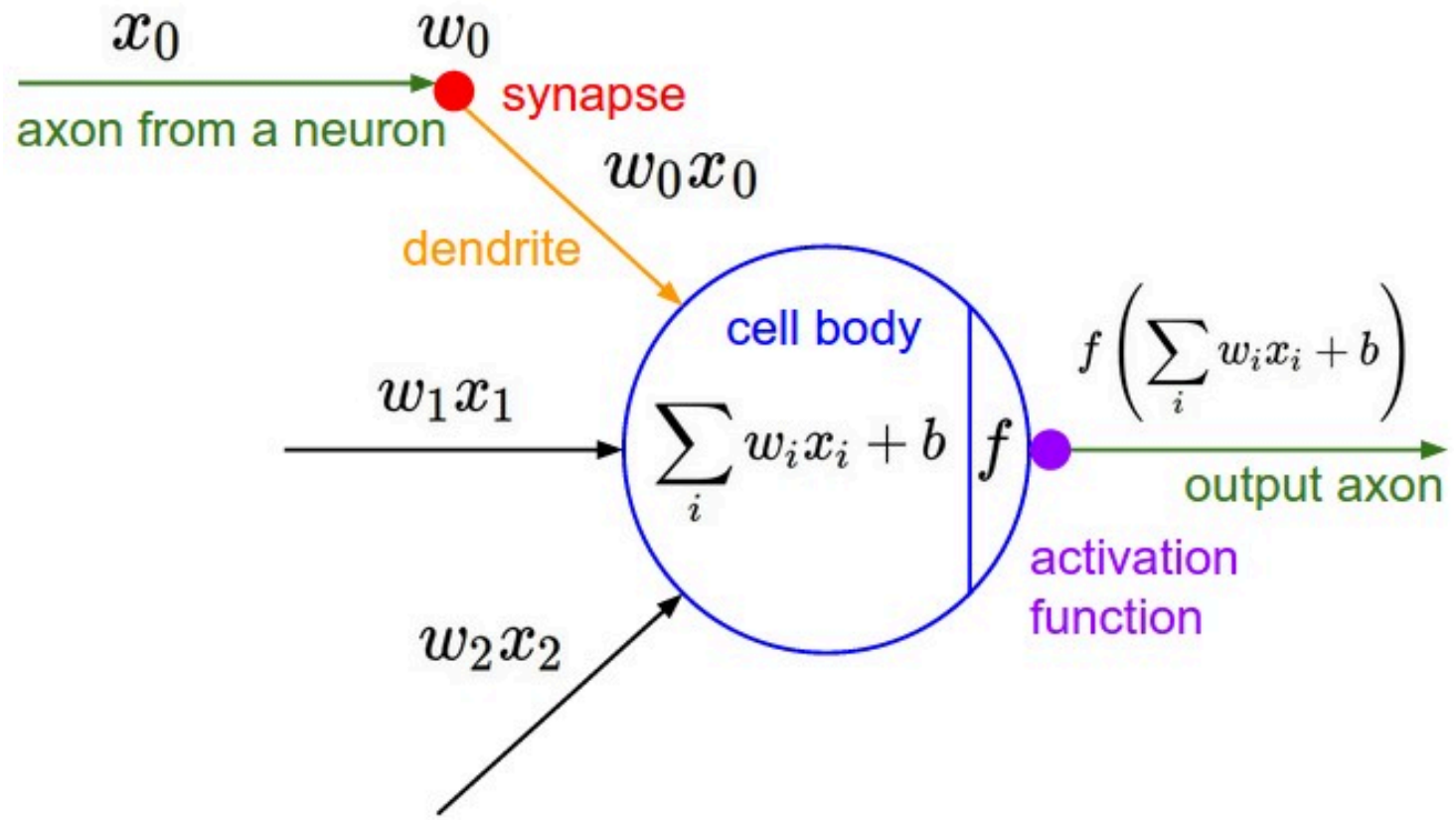
> A single neuron with a sigmoid activation is essentially logistic regression!

Neural networks extend this by:

Artificial Neurons

The Artificial Neuron

An artificial neuron is loosely modeled on biological neurons:



From cs231n

Neuron Components

A neuron performs the following operation:

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

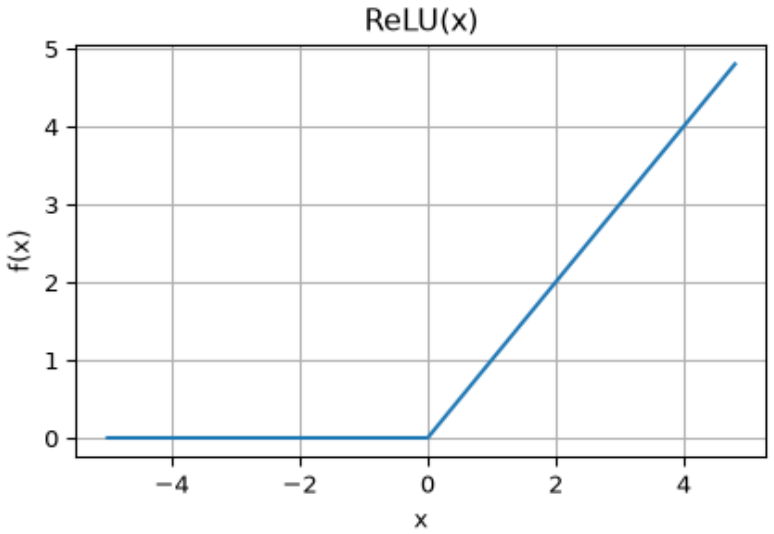
Where:

- x_i are the **inputs**
- w_i are the **weights** (parameters to learn)
- b is the **bias** (another parameter)
- f is the **activation function** (introduces non-linearity)

Activation Functions

ReLU (Rectified Linear Unit) - most popular today:

$$\text{ReLU}(x) = \max(0, x)$$

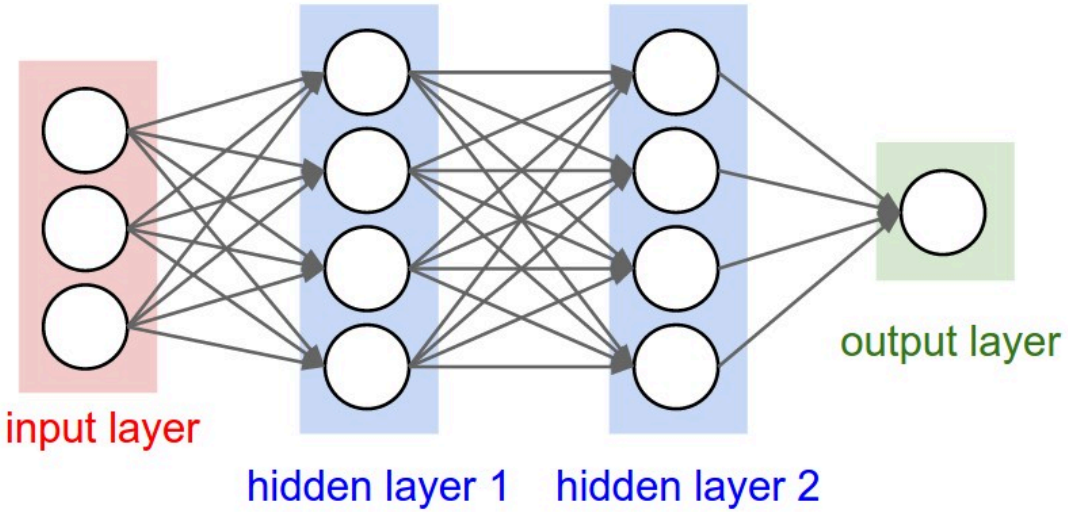


- Why activation functions?

Multi-Layer Perceptron (MLP)

MLP Architecture

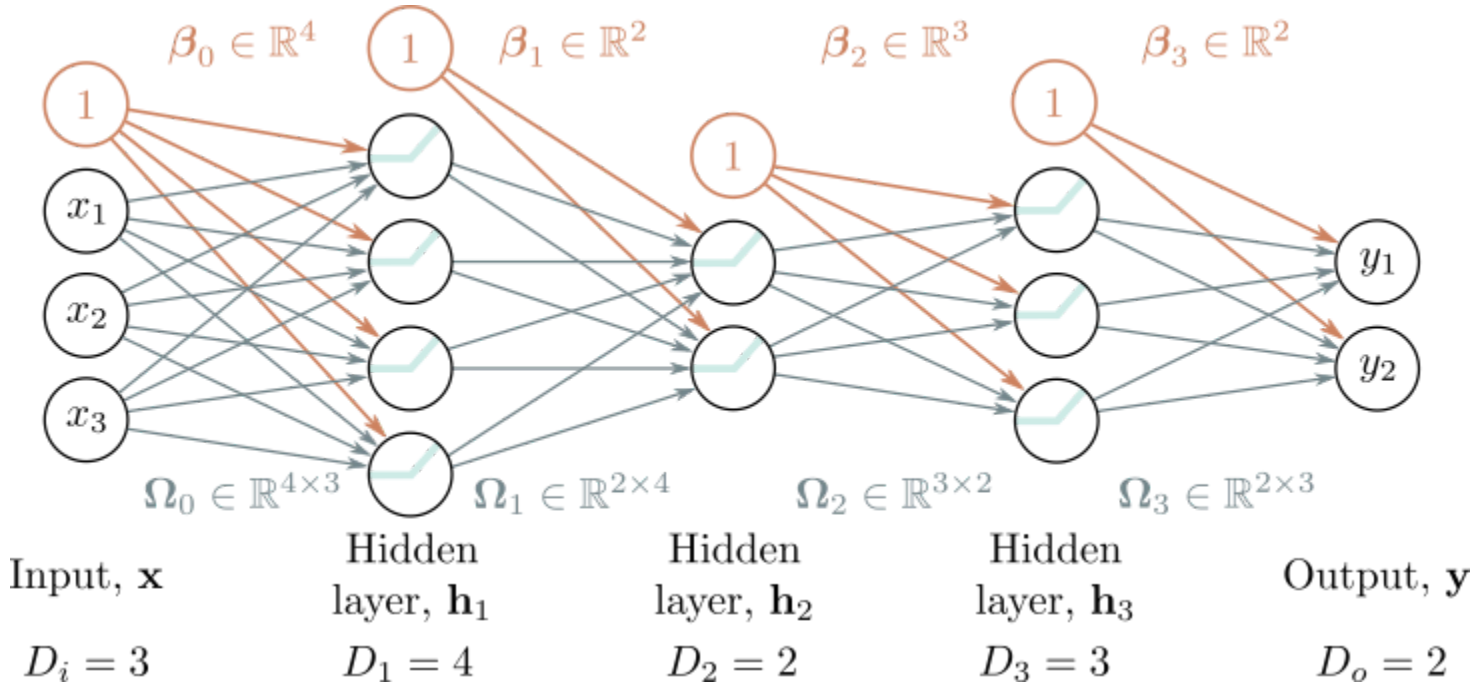
A Multi-Layer Perceptron stacks multiple layers of neurons:



From cs231n

- **Input layer**: Raw features
- **Hidden layers**: Learn intermediate representations
- **Output layer**: Final prediction

Matrix Formulation



FCN from UDL.

Key property: Every neuron in layer i connects to every neuron in layer $i + 1$.

This is also called a **Fully Connected Network (FCN)** or **Dense Network**.

MLP Mathematical Formulation

For a network with K hidden layers:

$$
\begin{aligned}
\mathbf{h}_1 &= f(\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}) \\
\mathbf{h}_2 &= f(\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1) \\
&\vdots \\
\mathbf{h}_K &= f(\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathbf{h}_{K-1}) \\
\hat{\mathbf{y}} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{h}_K
\end{aligned}
$$

Where:

- $\mathbf{h}_k$ = hidden layer activations
- $\boldsymbol{\Omega}_k$ = weight matrices
- $\boldsymbol{\beta}_k$ = bias vectors
- f = activation function (e.g., ReLU)

Training Neural Networks

The Loss Function

Training means finding weights that minimize a **loss function**:

For regression (e.g., predicting house prices):

$$L = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \quad \text{(Mean Squared Error)}$$

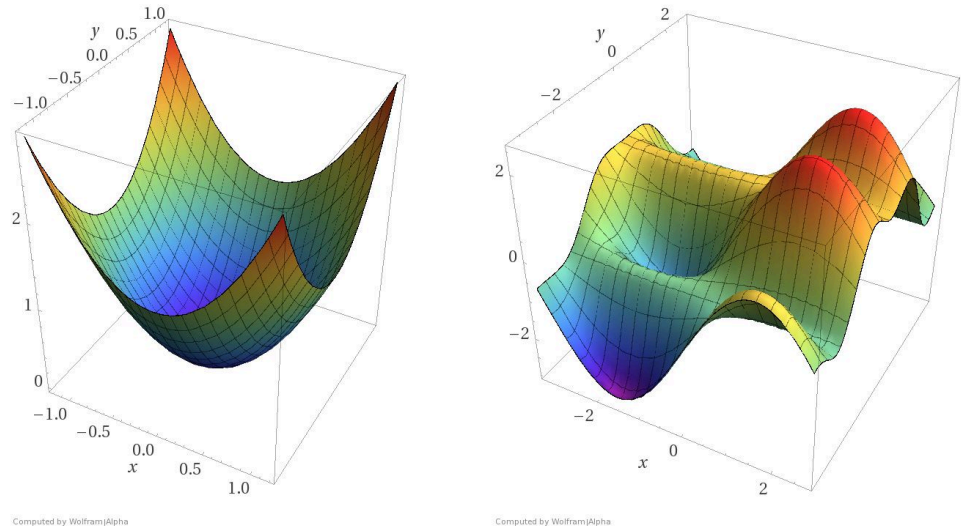
For classification (e.g., digit recognition):

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \quad \text{(Cross-Entropy)}$$

Goal: Find parameters $\theta = \{\boldsymbol{\Omega}_k, \boldsymbol{\beta}_k\}$ that minimize L.

Visualizing the Loss Surface

The loss function creates a surface over the parameter space:



- Left: **Convex** loss surface (e.g., linear regression)
- Right: **Non-convex** loss surface (e.g., neural networks)

For neural networks, we can’t solve analytically—we need **gradient descent**!

Gradient Descent

The Gradient Descent Intuition

Imagine you're lost in foggy mountains and want to reach the valley:

What would you do?

1. Look around 360 degrees
2. Find the direction sloping **downward most steeply**
3. Take a few steps in that direction
4. Repeat until the ground is level

This is **gradient descent**!

The Gradient

For a function $L(\mathbf{w})$ where $\mathbf{w} = (w_1, \dots, w_n)$, the **gradient** is:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

- The gradient points in the direction of **steepest increase**
- The negative gradient points toward **steepest decrease**

Gradient Descent Algorithm

Start with random weights $\mathbf{w}^{(0)}$, then iterate:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} L(\mathbf{w}^{(t)})$$

Where:

- η is the **learning rate** (step size)
- $\nabla_{\mathbf{w}} L$ is the **gradient** of the loss

Stop when:

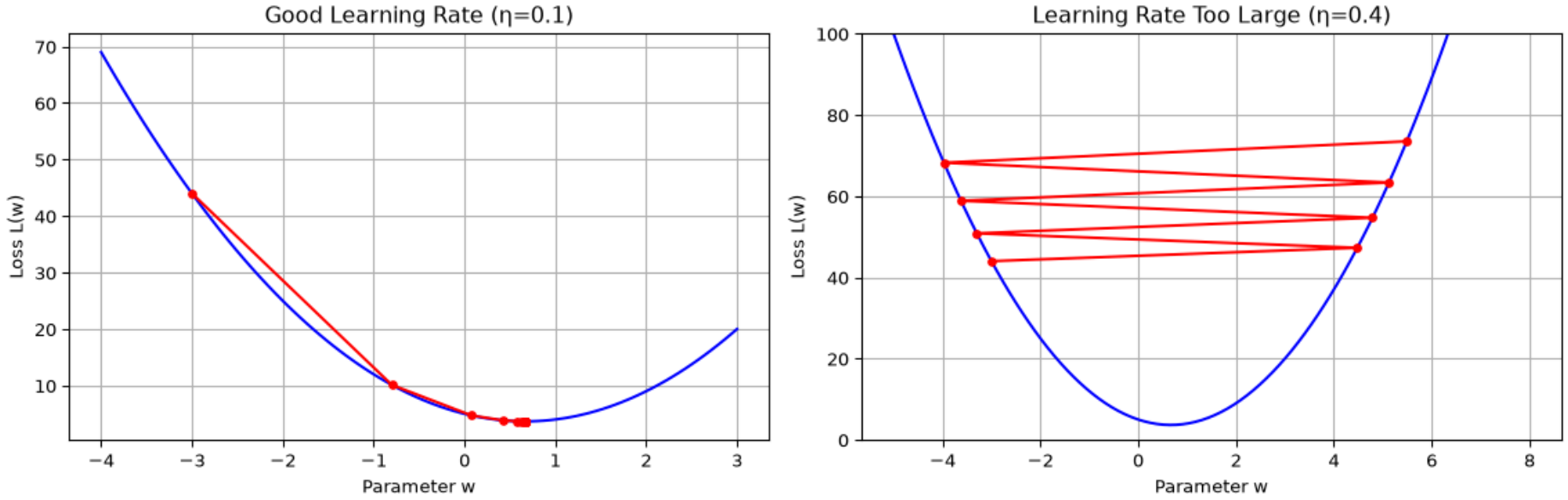
- Loss stops decreasing (convergence)
- Maximum iterations reached

Learning Rate Matters

The learning rate η is crucial:

Too small: Slow convergence

Too large: May fail to converge or even diverge!



Backpropagation

Where Do the Gradients Come From?

Gradient descent needs $\nabla_\theta L$ – the partial derivative of the loss with respect to **every** weight and bias in the network.

But L is a deeply nested composition:

$$L = \ell\Big(\boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \, f(\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \, f(\cdots f(\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}) \cdots)) \, , \; y \Big)$$

Computation Graph

The way we are going to differentiate more complex functions is to first build a "computation graph."

We'll see that we can "propagate backwards" through the graph to calculate the gradients of the loss function with respect to the parameters.

It's a scalable approach employed by TensorFlow and PyTorch, and in fact we'll follow the PyTorch interface definition.

> **Note**
>
> This section is a condensed version of NN II – Compute Graph and Backpropagation, which builds a full training framework. Read that for the complete treatment.

Building the `Value` Class

To do that we will

- build a data wrapper as a `class` called `Value`,
- record every arithmetic operation performed on it, so the graph builds itself, and
- store a gradient on every node so we can propagate backwards.

This is similar to how PyTorch defines its `Tensor` class.

Building the `Value` Class

First, the class has only a simple initialization method and a representation method.

```python
# Value version 1
class Value:

    def __init__(self, data):
        self.data = data

    def __repr__(self):
        """Return a string representation of the object for display"""
        return f"Value(data={self.data})"
```

Which we can instantiate and evaluate as follows.

```python
a = Value(4.0)
a
```

```
Value(data=4.0)
```

If you are not familiar with python classes, there are a few things to note here.

1. The property `self` is just a pointer to the object itself.
2. The `__init__` method is called when you initialize a class object.
3. The `__repr__` method is how you represent the class object.

Implementing Operations

The `Value` object doesn't do much yet. When python tries to add two objects `a` and `b`, internally it will call `a.__add__(b)`, so we add `__add__()`, `__mul__()` and a `relu()` method.

```python
# Value version 2
class Value:

    def __init__(self, data):
        self.data = data

    def __repr__(self):
        """Return a string representation of the object for display"""
        return f"Value(data={self.data})"

    def __add__(self, other): # self + other
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data)
        return out

    def __mul__(self, other): # self * other
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data)
        return out

    def relu(self):
        out = Value(np.maximum(0, self.data))
        return out
```

Implementing Operations

Now we can use the operations.

```
1 a = Value(4.0)
2 b = Value(-3.0)
3 c = Value(8.0)
4
5 d = a*b+c
6 d
```

```
Value(data=-4.0)
```

Internally, python calls `__mul__` on `a`, then `__add__` on the temporary product object.

Recording the Graph

In order to calculate the gradients, we will need to capture the computation graph.

To do that, each output stores pointers to its operands as a tuple of **child nodes**, plus the **operator** that produced it. We'll also add labels for convenience.

```python
# Value version 3
class Value:
                        #    vvvvvvvvvvvv  vvvvvvv  vvvvvvvv
    def __init__(self, data, _children=(), _op='', label=''):
        self.data = data
        self._prev = set(_children) # the operand nodes
        self._op = _op              # the operation that created this node
        self.label = label          # label for the node

    def __repr__(self):
        """Return a string representation of the object for display"""
        return f"Value(data={self.data})"

    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+') # store children and operator
        return out                                          #  ^^^^^^^^^^^^^  ^^^

    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other), '*')
        return out

```

Recording the Graph

Let's instantiate a few `Value` objects and do some operations with them.

```
a = Value(4.0, label='a')
b = Value(-3.0, label='b')
c = Value(8.0, label='c')

d = a*b ; d.label = 'd'
e = d + c ; e.label = 'e'
```

We can now inspect the operands and the operation that created each node.

```
e._prev, e._op, e.label
```

```
({Value(data=-12.0), Value(data=8.0)}, '+', 'e')
```

The name `_prev` will make more sense when we view these operations as a graph.

Adding a Gradient Slot

Finally we add a member variable, `grad`, to store the partial derivative of the **output** node with respect to this node. It defaults to zero.

```python
# Value version 4
class Value:

    def __init__(self, data, _children=(), _op='', label=''):
        self.data = data
        self.grad = 0.0 # default to 0  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        self._prev = set(_children)
        self._op = _op # store the operation that created this node
        self.label = label # label for the node

    def __repr__(self):
        """Return a string representation of the object for display"""
        return f"Value(data={self.data})"

    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+')
        return out

    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other), '*')
        return out

    def relu(self):
```

Drawing the Compute Graph

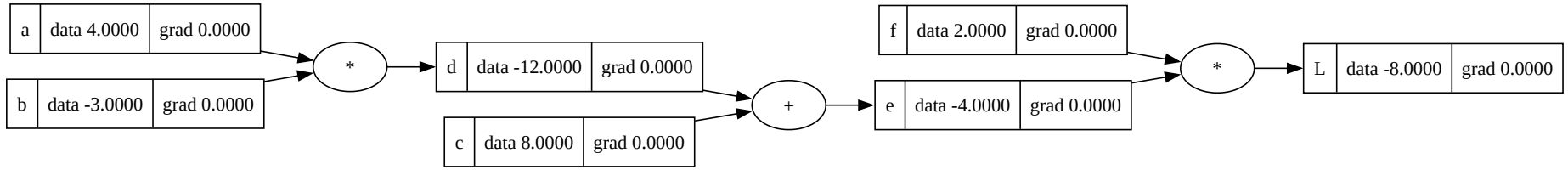
We now have enough information stored to visualize the graph. These two functions walk the graph to collect all nodes and edges (`trace`) and draw them as a directed graph (`draw_dot`).

▶ Show the code for `trace()` and `draw_dot()`

Drawing the Compute Graph

Let's build a small three-stage graph, ending in a node we'll call `L` (think: loss).

```
a = Value(4.0, label='a')
b = Value(-3.0, label='b')
c = Value(8.0, label='c')

d = a*b; d.label = 'd'
e = d + c; e.label = 'e'
f = Value(2.0, label='f')

L = e*f; L.label = 'L'

draw_dot(L)
```



Every `Value` becomes a node; the operators are drawn as small nodes too. Computing the `data` values left-to-right is the **forward pass**.

We have placeholders for the gradients, but they are currently all zero.

Manual Gradient Calculation

Before we automate backpropagation, let's calculate the gradients by hand to understand the procedure.

For the output node L, we trivially have $\frac{dL}{dL} = 1$:

$$\frac{dL}{dL} = \lim_{h \to 0} \frac{(L + h) - L}{h} = \frac{h}{h} = 1$$

```
1 L.grad = 1.0
```

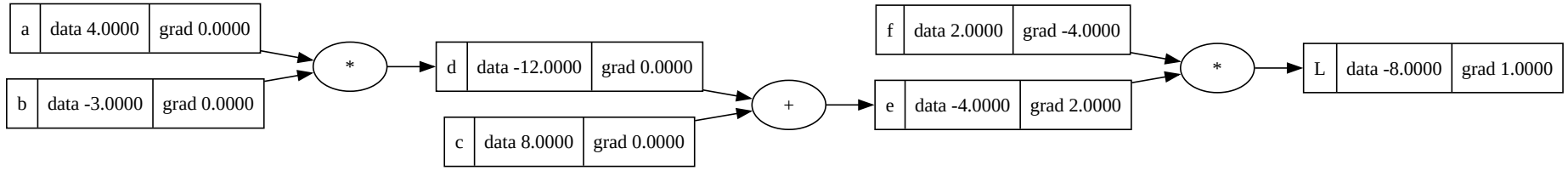
Manual Gradient Calculation

Going backwards one step, $L = e \times f$, so

$$\frac{\partial L}{\partial e} = \frac{\partial}{\partial e}(e \times f) = f, \qquad \frac{\partial L}{\partial f} = \frac{\partial}{\partial f}(e \times f) = e.$$

So we just assign the gradient to the value of the *other* operand.

```
1 e.grad = f.data
2 f.grad = e.data
3 draw_dot(L)
```



Tip

For **products**, the partial derivative w.r.t. one operand is simply the *other operand*.

Important

Manual Gradient Calculation

Sanity check: `f.grad` says L should change by `e.data` $= -4$ for a unit change in f. Let's wiggle f by h and see.

```
def wiggle(h = 0.0):
    a = Value(4.0, label='a')
    b = Value(-3.0, label='b')
    c = Value(8.0, label='c')

    d = a*b; d.label = 'd'
    e = d + c; e.label = 'e'
    f = Value(2.0, label='f')
    f += h

    L = e*f; L.label = 'L'
    print(L)

wiggle(0.0)
wiggle(1.0)
```

```
Value(data=-8.0)
Value(data=-12.0)
```

Propagating Back

Now we want $\frac{\partial L}{\partial c}$ – how much L varies if we vary c.

Looking at the graph, c influences e and e influences L:

$$c \to e \to L.$$

We have $e = d + c$, so the **local** derivative is

$$\frac{\partial e}{\partial c} = \frac{\partial}{\partial c}(d + c) = 1.$$

> **Tip**
>
> For **addition**, the partial derivative w.r.t. either operand is 1.

The Chain Rule

If a variable L depends on the variable e, which itself depends on the variable c, then L depends on c as well, via the intermediate variable e, and

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial c}.$$

More precisely, noting *where* each derivative is evaluated:

$$\left.\frac{\partial L}{\partial c}\right|_{c} = \left.\frac{\partial L}{\partial e}\right|_{e(c)} \cdot \left.\frac{\partial e}{\partial c}\right|_{c}.$$

> We evaluate the derivatives at the specific values of the variables that we calculated in the forward pass.

The Chain Rule

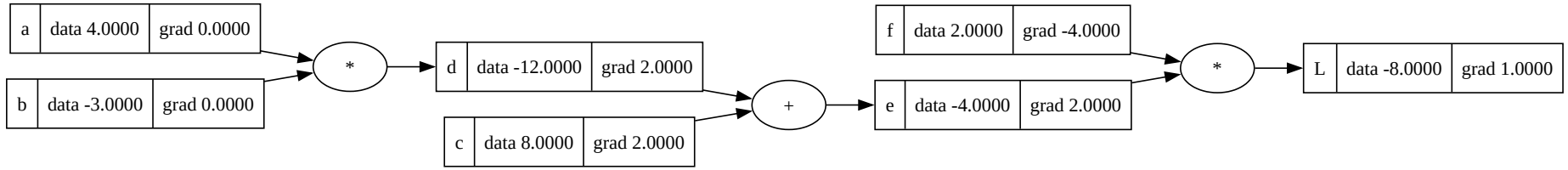
Since $\partial e / \partial c = 1$,

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial c} = \frac{\partial L}{\partial e} \cdot 1,$$

and identically for d.

> **Important**
>
> With the **addition operator**, we just *route the parent gradient to the child*.

```
1 d.grad = e.grad
2 c.grad = e.grad
3 draw_dot(L)
```



Propagating Back Again

One more step. We have $\frac{\partial L}{\partial d}$ and want $\frac{\partial L}{\partial a}$ and $\frac{\partial L}{\partial b}$.

Since $d = a \cdot b$, the local derivatives are $\partial d/\partial b = a$ and $\partial d/\partial a = b$, so by the chain rule

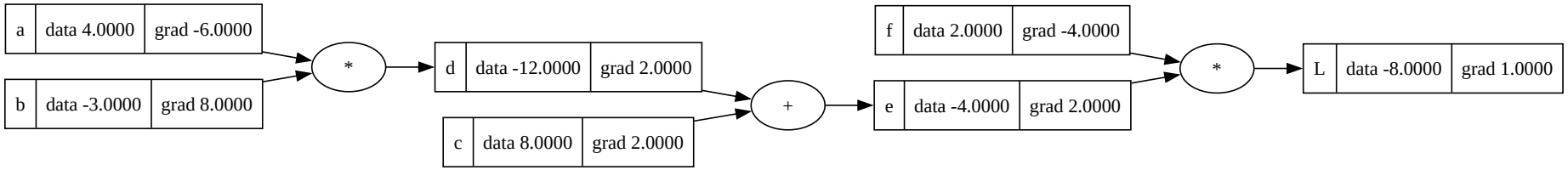
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial b} = \frac{\partial L}{\partial d} \cdot a, \qquad \frac{\partial L}{\partial a} = \frac{\partial L}{\partial d} \cdot b.$$

Fully expanded, this is a chain of local derivatives all the way from L back to b:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b}.$$

Propagating Back Again

```
1 b.grad = a.data * d.grad
2 a.grad = b.data * d.grad
3 draw_dot(L)
```



We've traversed all the way back to the inputs and calculated all the partial derivatives.

The Backpropagation Recipe

What we just did, stated as an algorithm:

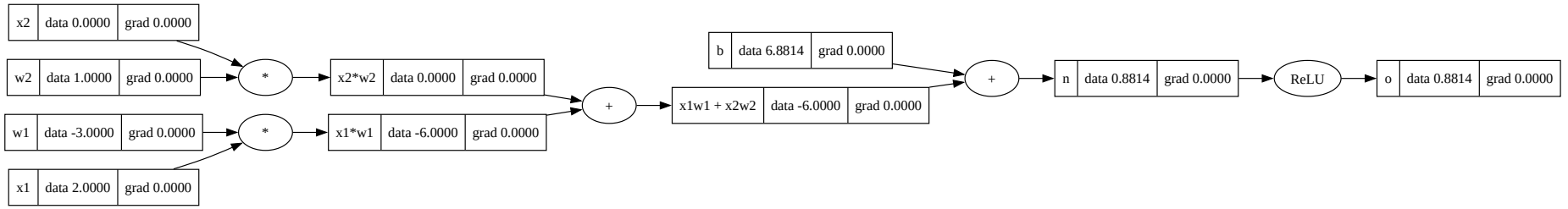
Backprop on a Neuron

The same recipe applies to a real neuron. Here is one with two inputs, two weights, a bias, and a ReLU:

```
# inputs x1, x2
x1 = Value(2.0, label='x1')
x2 = Value(0.0, label='x2')

# weights w1, w2
w1 = Value(-3.0, label='w1')
w2 = Value(1.0, label='w2')

# bias of the neuron
b = Value(6.8813735870195432, label='b')

x1w1 = x1*w1; x1w1.label = 'x1*w1'
x2w2 = x2*w2; x2w2.label = 'x2*w2'

x1w1x2w2 = x1w1 + x2w2; x1w1x2w2.label = 'x1w1 + x2w2'
n = x1w1x2w2 + b; n.label = 'n'
o = n.relu(); o.label = 'o'
```

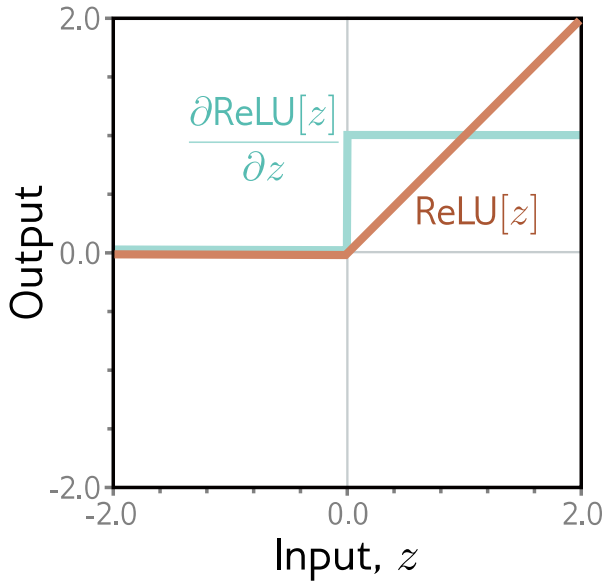
Backprop on a Neuron

```
1 draw_dot(o)
```



Backprop on a Neuron

The only new operation is the ReLU. It is technically not differentiable at 0, but in practice we take the derivative to be 0 when the input is ≤ 0 and 1 when it is > 0.




```
o.grad = 1.0
n.grad = (o.data > 0) * o.grad   # ReLU: pass through if the input was positive
x1w1x2w2.grad = n.grad           # '+' routes the gradient
b.grad = n.grad
x1w1.grad = x1w1x2w2.grad
x2w2.grad = x1w1x2w2.grad
w1.grad = x1.data * x1w1.grad    # '*' multiplies by the other operand
```

From Gradients to a Step

Once every parameter has its `grad`, a gradient descent step is just

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

applied to each parameter leaf node – exactly the update rule from the previous section.

> **Note**
>
> In M12 we finish the job: add a `backward()` method to `Value` that walks the graph automatically, handle nodes used more than once (gradients *accumulate*), assemble neurons into layers and an MLP, and write the training loop. In Neural Networks II we let scikit-learn do all of that for us.

Stochastic Gradient Descent

Full Batch vs Stochastic GD

Backprop gives us the gradient of the loss for **one** sample, $\nabla_{\mathbf{w}} \ell_i(\mathbf{w})$. How many samples should we run it on before taking a step?

Full Batch Gradient Descent: Compute gradient using ALL training samples:

$$\nabla_{\mathbf{w}} L = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{w}} \ell_i(\mathbf{w})$$

Problems:

- Slow for large datasets (millions of samples!)
- Memory intensive
- Can get stuck in local minima

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent: Historically meant using ONE random sample at a time:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \ell_i(\mathbf{w}^{(t)})$$

Advantages:

- Much faster per iteration
- Can escape local minima (due to noise)
- Enables online learning

Disadvantage:

- *Extremely* noisy gradient estimates
- May not converge exactly to minimum

Mini-Batch Gradient Descent

Mini-Batch GD: Best of both worlds—use a small batch of samples:

$$\nabla_{\mathbf{w}} L \approx \frac{1}{B} \sum_{i \in \text{batch}} \nabla_{\mathbf{w}} \ell_i(\mathbf{w})$$

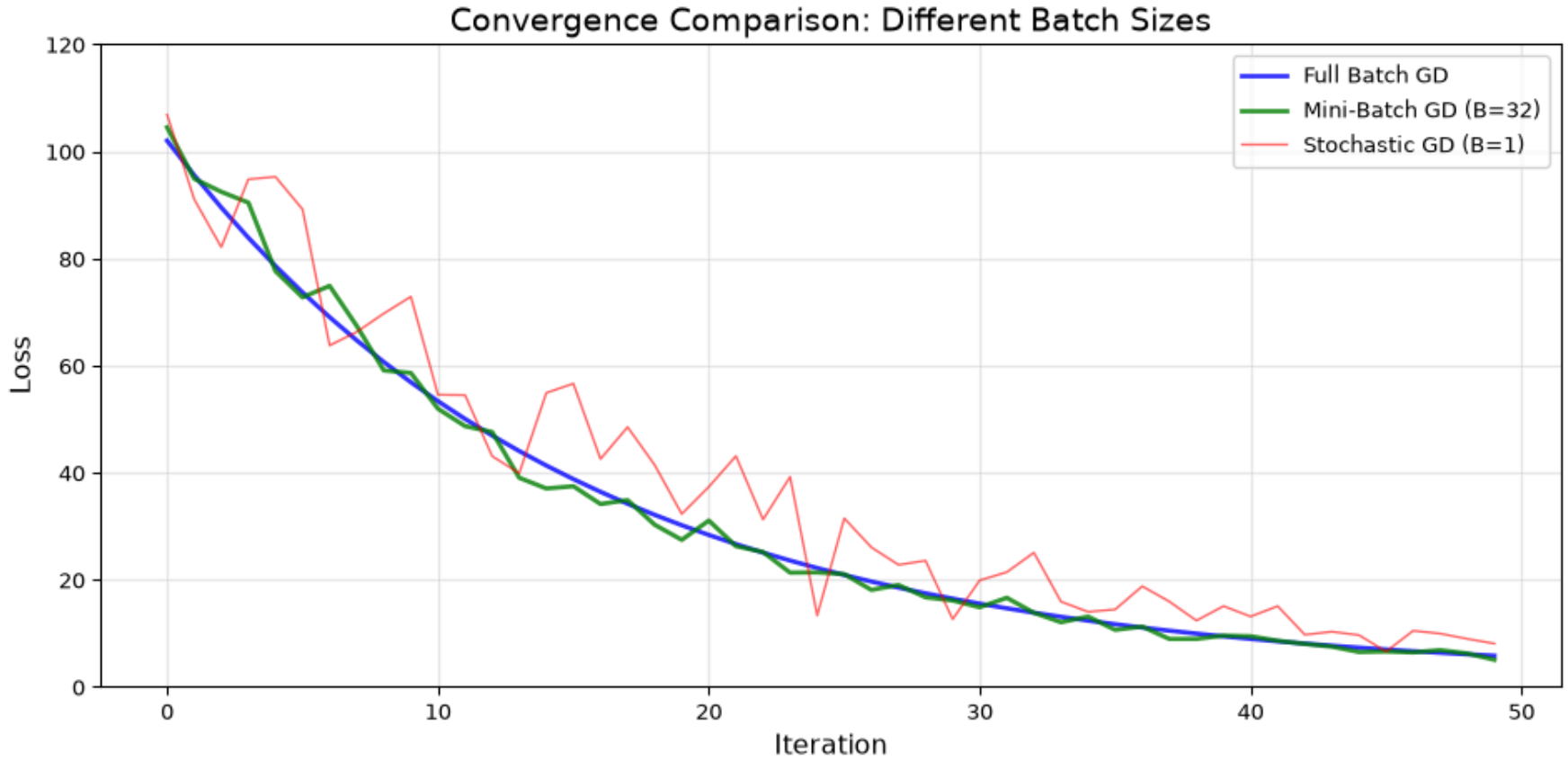
Typical batch sizes: 32, 64, 128, 256

Advantages:

- Balances speed and stability
- Efficient GPU parallelization
- Better gradient estimates than pure SGD

This is what most modern neural network training uses!

Visualizing Batch Strategies



(For illustration purposes only – not a real training curve.)

Summary

Summary

- A neuron is a weighted sum followed by a non-linear activation; a single sigmoid neuron *is* logistic regression
- An MLP stacks layers of neurons – a differentiable function $\hat{y} = g(\mathbf{x}; \theta)$ of its weights and biases
- Training = minimizing a loss $L(\theta)$ (MSE or cross-entropy) over the training set
- **Gradient descent** steps downhill: $\theta \leftarrow \theta - \eta \nabla_\theta L$
- **Backpropagation** computes $\nabla_\theta L$ by running the chain rule backwards over the compute graph: each node multiplies the incoming gradient by its *local* derivative
- **Stochastic / mini-batch GD** estimates the gradient from a small batch – cheaper per step, and the noise helps

Next: Making Training Work

In Neural Networks II we switch to practice: `MLPClassifier` and `MLPRegressor` in scikit-learn, MNIST and California housing, hyperparameters, preprocessing, and how to recognize and fix training that isn't working.

To Dig Deeper

Full treatments of today's topics in the course notes:

- NN I – Gradient Descent: gradient descent in more depth
- NN II – Compute Graph and Backpropagation: the full `Value` framework, automatic `backward()`, gradient accumulation, and a complete training loop

Additional resources:

- Understanding Deep Learning, Simon J.D. Prince, MIT Press, 2023, Chapters 3-7
- DS542, Deep Learning for Data Science