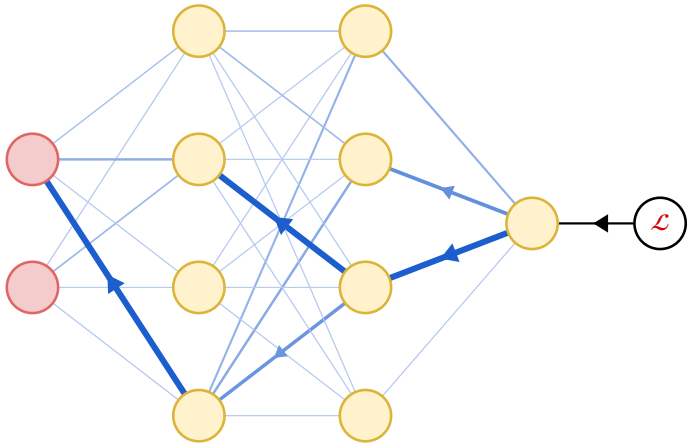


UNIT 03 · CHAPTER 1

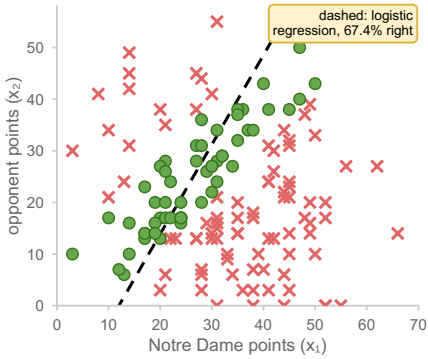
Backpropagation

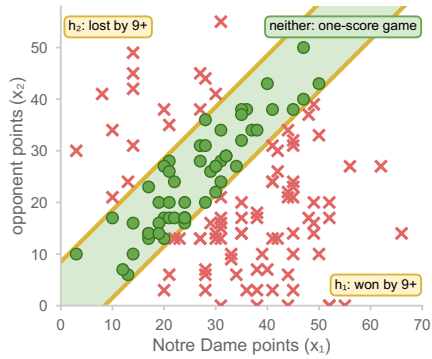
CSE 30124 · Introduction to Artificial Intelligence

William Theisen · University of Notre Dame

The XOR Problem

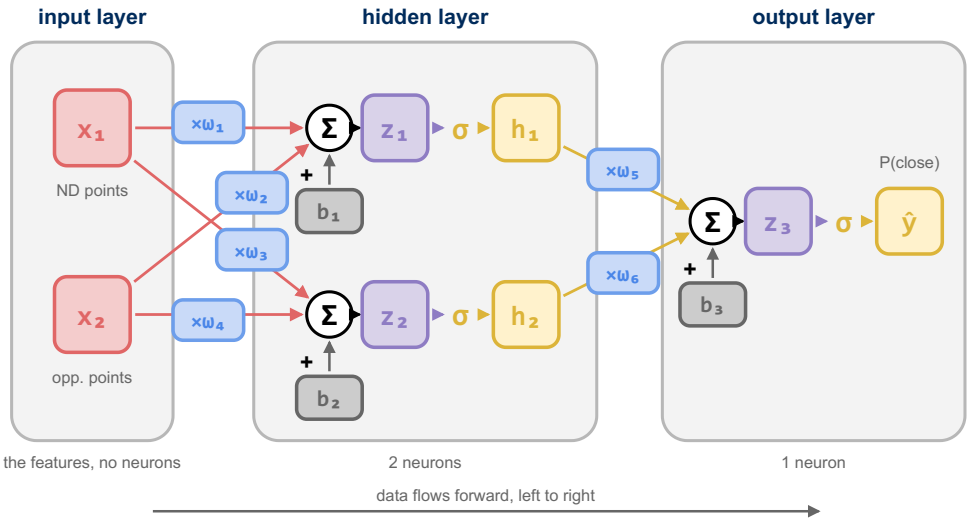
What if instead of asking **win** or **loss**, we wanted to **decide whether a game was close or not (within 8 points)**





Asking if a game was close or not results in data that is very similar to the **XOR problem** we saw with **perceptrons**, where our **data is not**

Neural Networks: Diagram



Combining our three **artificial neurons** gives us a simple **FFN**

The organization of neurons in a neural network is called its _____.

Feed-Forward Networks - Old Exam Question

```python
import torch
import torch.nn as nn
torch.set_printoptions(precision=2)

class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(SimpleNN, self).__init__()

        self.fc1 = nn.Linear(input_size, hidden_size)
        self.sig = nn.Sigmoid()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = self.sig(z1)
        z2 = self.fc2(a1)

        return z2
```

a.) Draw the computation graph for the network. (4 pts.)

b.) Give the loss of the neural network (L) as a single statement of nested functions. (1 pt.)

Last Class: Feed-Forward Networks

```python
# Last lecture's 4 games: ND points, opp. points
X = torch.tensor([[50., 43.], [22., 24.], [38.,  0.], [14., 42.]]) #
y = torch.tensor([[1.], [1.], [0.], [0.]])  # close = 1, not close =

torch.manual_seed(0)
model = SimpleNN(2, 2, 1)   # fresh: random weights
for name, p in model.named_parameters(): print(name, p.detach())

y_hat = torch.sigmoid(model(X))   # P(close)
print("y_hat (predictions):", y_hat.detach().flatten())
print("mapped predictions (y_hat > 0.5):", (y_hat > 0.5).float().flat

criterion = nn.BCELoss()
loss = criterion(y_hat, y)
print("loss:", round(loss.item(), 3))
```

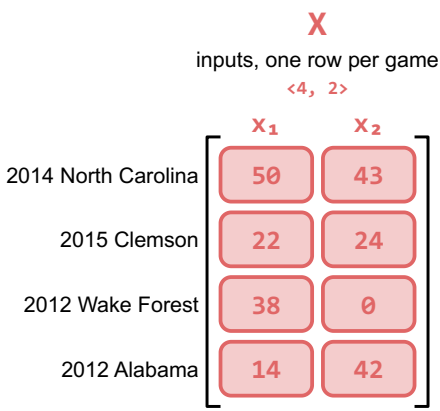
OUTPUT

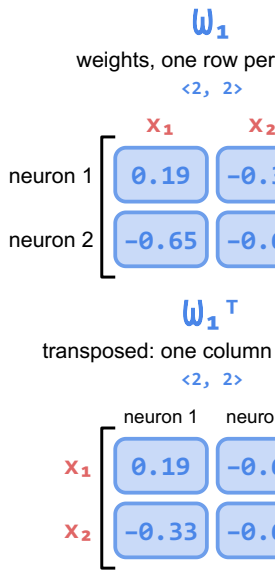
```
fc1.weight tensor([[ 0.19, -0.33],
        [-0.65, -0.68]])
fc1.bias tensor([0.44, 0.58])
fc2.weight tensor([[0.15, 0.32]])
fc2.bias tensor([0.06])
y_hat (predictions): tensor([0.52, 0.52, 0.55, 0.52])
mapped predictions (y_hat > 0.5): tensor([1., 1., 1., 1.])
loss: 0.713
```

Condensing the Hidden Layer

We can compactly represent the operations of the hidden layer as a single matrix multiplication.

$$Z_1 = XW_1^T + b_1$$
$$A_1 = \sigma(Z_1)$$





Weights as Matrices: Symbolic

four games times both hidden neurons' weights (transposed: one column per neuron), plus the bias (added to every row)

$$Z_1 = XW_1^\top + b_1 = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

game i, neuron j: every input times that neuron's weight on it, plus that neuron's bias

$$Z_1 = \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \\ z_{31} & z_{32} \\ z_{41} & z_{42} \end{bmatrix} = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{12} + b_1 & x_{11}w_{21} + x_{12}w_{22} + b_2 \\ x_{21}w_{11} + x_{22}w_{12} + b_1 & x_{21}w_{21} + x_{22}w_{22} + b_2 \\ x_{31}w_{11} + x_{32}w_{12} + b_1 & x_{31}w_{21} + x_{32}w_{22} + b_2 \\ x_{41}w_{11} + x_{42}w_{12} + b_1 & x_{41}w_{21} + x_{42}w_{22} + b_2 \end{bmatrix}$$

the sigmoid of every cell

$$A_1 = \sigma(Z_1) = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} = \begin{bmatrix} \sigma(z_{11}) & \sigma(z_{12}) \\ \sigma(z_{21}) & \sigma(z_{22}) \\ \sigma(z_{31}) & \sigma(z_{32}) \\ \sigma(z_{41}) & \sigma(z_{42}) \end{bmatrix}$$

Weights as Matrices: Numeric

four games times both hidden neurons' weights (transposed: one column per neuron), plus the bias (added to every row)

$$Z_1 = XW_1^\top + b_1 = \begin{bmatrix} 50 & 43 \\ 22 & 24 \\ 38 & 0 \\ 14 & 42 \end{bmatrix} \begin{bmatrix} 0.19 & -0.65 \\ -0.33 & -0.68 \end{bmatrix} + \begin{bmatrix} 0.44 & 0.58 \end{bmatrix}$$

each row is one game, each column one hidden neuron: every input times its weight, plus that neuron's bias

$$Z_1 = \begin{bmatrix} -4.25 & -61.16 \\ -3.30 & -30.04 \\ 7.66 & -24.12 \\ -10.76 & -37.08 \end{bmatrix} = \begin{bmatrix} 50 \cdot 0.19 + 43 \cdot (-0.33) + 0.44 & 50 \cdot (-0.65) + 43 \cdot (-0.68) + 0.58 \\ 22 \cdot 0.19 + 24 \cdot (-0.33) + 0.44 & 22 \cdot (-0.65) + 24 \cdot (-0.68) + 0.58 \\ 38 \cdot 0.19 + 0 \cdot (-0.33) + 0.44 & 38 \cdot (-0.65) + 0 \cdot (-0.68) + 0.58 \\ 14 \cdot 0.19 + 42 \cdot (-0.33) + 0.44 & 14 \cdot (-0.65) + 42 \cdot (-0.68) + 0.58 \end{bmatrix}$$

the sigmoid of every cell

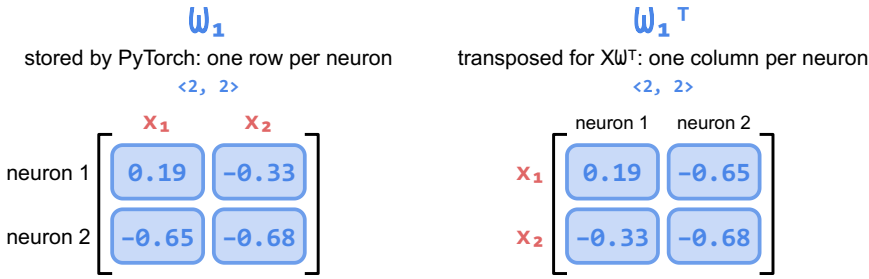
$$A_1 = \sigma(Z_1) = \begin{bmatrix} 0.01 & 0.00 \\ 0.04 & 0.00 \\ 1.00 & 0.00 \\ 0.00 & 0.00 \end{bmatrix} = \begin{bmatrix} \sigma(-4.25) & \sigma(-61.16) \\ \sigma(-3.30) & \sigma(-30.04) \\ \sigma(7.66) & \sigma(-24.12) \\ \sigma(-10.76) & \sigma(-37.08) \end{bmatrix}$$

Weights as Matrices: Equation Conventions

There are a number of different conventions for writing the **linear transformation** as used in a `linear_layer`. **I use a slightly non-standard one** that I believe maps better to `PyTorch` and the diagrams I show in class.

Convention	Equation	Where you'll see it
Bias folded in	$Z = X\omega$ (X gets a column of 1s)	**Linear regression**, **perceptrons** and **SVMs**
Inputs × outputs	$Z = XW + b$	`Keras`/`TensorFlow`, many textbooks
Outputs × inputs	$Z = XW^T + b$	`PyTorch` `nn.Linear` and this course

`PyTorch` stores **one row per neuron** so it reads like the math: $XW^T + b$



Improving the Weights

Linear Regressions and OLS

Due to the simplicity of **linear regression**, we could use a _____ to **directly find the optimal set of weights**.

This is due to the _____ being **perfectly convex**



Gradient Descent

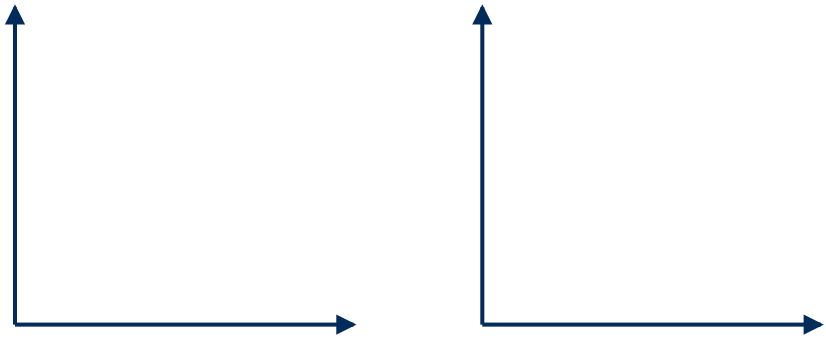
Gradient Descent is an **optimization method** that _____ tries to find a **set of values (weights)** that _____



A **gradient** is the _____ **at a given point in a specified direction**

Multivariate Gradient Descent

In the case of a **multivariate function**, the gradient is a _____

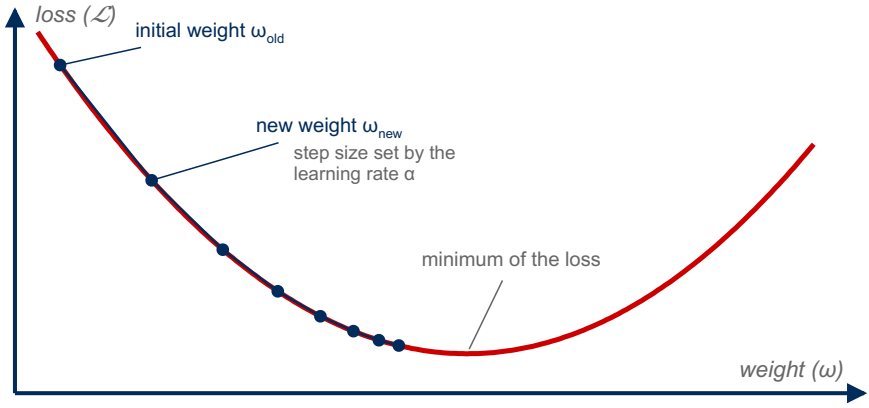


$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \omega_1} \\ \frac{\partial \mathcal{L}}{\partial \omega_2} \end{bmatrix}$$

slope along ω_1

slope along ω_2

Gradient Descent Algorithm



[1.1] one step

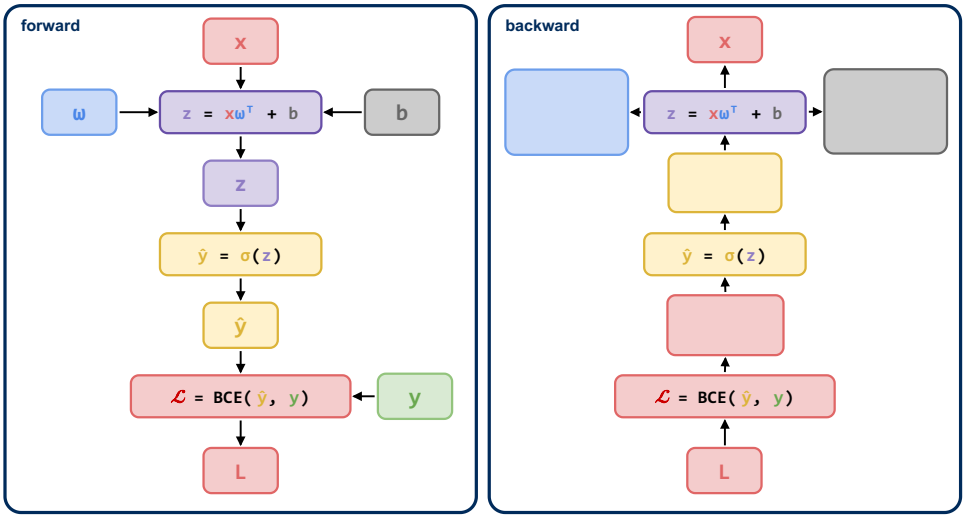
$$\omega_{new} = \omega_{old} - \alpha \frac{\partial \mathcal{L}}{\partial \omega}$$

Common Loss Functions

- _____
- _____
- _____
- _____

Logistic Regression with Gradient Descent

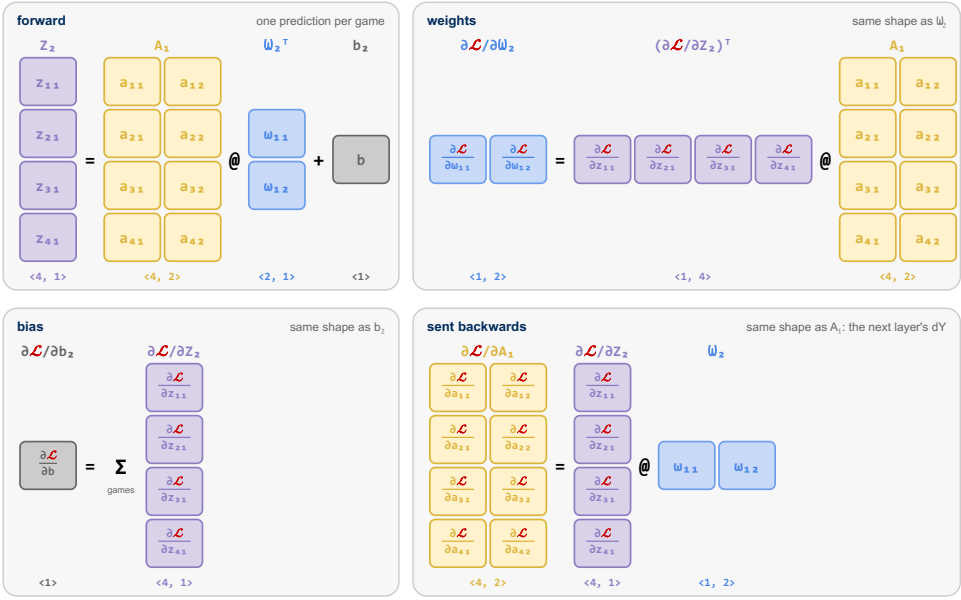
The **chain rule** enables us to find derivatives of _____!



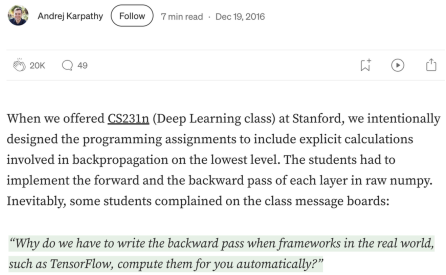
$$\frac{\partial \mathcal{L}}{\partial \omega_1} = \qquad \frac{\partial \mathcal{L}}{\partial b_1} =$$

Note: (full derivation) of **BCE** and **σ**

Dimensionality of Gradients



Yes, you should probably understand backpropagation



Andrej Karpathy Follow 7 min read · Dec 19, 2016

20K 49

When we offered CS231n (Deep Learning class) at Stanford, we intentionally designed the programming assignments to include explicit calculations involved in backpropagation on the lowest level. The students had to implement the forward and the backward pass of each layer in raw numpy. Inevitably, some students complained on the class message boards:

"Why do we have to write the backward pass when frameworks in the real world, such as TensorFlow, compute them for you automatically?"

karpathy.medium.com/yes-you-should-understand-backprop

Backpropagation (Backprop)

Backpropagation is a technique in deep learning to

for a network. Using the **chain rule** allows us to **implement Gradient Descent** for multi-layer models

Python Backprop by Hand - Linear Layer

```python
import torch

class LinearLayer:
    def __init__(self, n_in, n_out):
        self.W = torch.randn(n_out, n_in) * (2/n_in)**0.5
        self.b = torch.zeros(n_out)

    def forward(self, X):
        self.X = X                  # keep for backward
        return 

    def backward(self, dY):
        self.dW = dY.T @ self.X     # dL/dW
        self.db = dY.sum(dim=0)     # dL/db
        return dY @ self.W          # dL/dX, sent back

    def update(self, lr):
        self.W -= lr * self.dW
        self.b -= lr * self.db
```

Python Backprop by Hand - Sigmoid and ReLU

```python
class Sigmoid:
    def forward(self, Z):
        self.A = 1 / (1 + torch.exp(-Z))  # saved
        return self.A

    def backward(self, dA):
        return dA * self.A * (1 - self.A)  # dL/dZ

    def update(self, lr):
        pass  # no weights to learn
```

```python
class ReLU:
    def forward(self, Z):
        self.Z = Z  # saved
        return Z.clamp(min=0)

    def backward(self, dA):
        return dA * (self.Z > 0).float()  # dL/dZ

    def update(self, lr):
        pass  # no weights to learn
```

Python Backprop by Hand - Neural Network

```python
class Network:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X  # logits

    def backward(self, y_hat, y):
        grad = y_hat - y  # dL/dy_hat (hard-coded here)
        for layer in reversed(self.layers):  # chain rule
            grad = layer.backward(grad)  # back to front

    def update(self, lr):
        for layer in self.layers:
            layer.update(lr)
```

Our `network class` is really just more of a wrapper than anything else, most of the work is done in the `layer classes`

Training a Neural Network by Hand

```
# Last lecture's 4 games: ND points, opp. points
X = torch.tensor([[50., 43.], [22., 24.], [38.,  0.], [14., 42.]]) #
y = torch.tensor([[1.], [1.], [0.], [0.]])  # close = 1, not close =

torch.manual_seed(0)
model = Network([LinearLayer(2, 2), Sigmoid(), LinearLayer(2, 1)])
bce = torch.nn.BCELoss()
y_hat = torch.sigmoid(model.forward(X))  # P(close)
print('y_hat at epoch 0:', y_hat.flatten(), 'loss:', bce(y_hat, y))

def train(model, X, lr=0.001, epochs=1000):
    for epoch in range(epochs):
        y_hat = torch.sigmoid(model.forward(X))
        model.backward(y_hat, y)
        model.update(lr)
    return torch.sigmoid(model.forward(X))

y_hat_trained = train(model, X)

print('y_hat at epoch 1000:', y_hat_trained.flatten(), 'loss:', bce(y
```

OUTPUT

```
y_hat at epoch 0: tensor([0.5038, 0.5367, 0.4577, 0.5866]) loss: tensor(0
y_hat at epoch 1000: tensor([0.5537, 0.5523, 0.3362, 0.5538]) loss: tenso
```

Some Issues with Backprop and Activation Functions

There are **some issues** that appear when you start using **backprop**, depending on the **activation functions** you use.

Saturated Sigmoids:

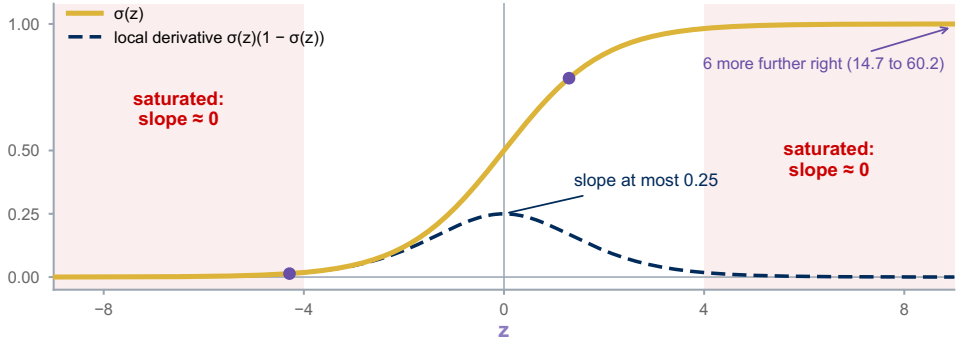
When a neuron's z is far from 0, the **sigmoid** _____, so its local derivative is effectively 0 and almost **no gradient gets back** through it. **The neuron is saturated.**

Dying ReLUs:

A **ReLU's** local derivative is _____, so a **neuron** that's negative on every input **passes no gradient back** and never updates again. **The neuron is dead.**

Saturated Sigmoids

Not dissimilar to the issues with the **step function**, if we are too far outside of the curve of the **sigmoid**, we get effectively **no useful gradient information** from it.



Purple dots: this section's network's **8** values of Z_1 (**4** games × **2** neurons) on raw scores, most of them far out on the flat top.

Saturated Neurons: Demo

```
torch.manual_seed(11) # ensure our random init is always the same

layers = [LinearLayer(2, 2), Sigmoid(), LinearLayer(2, 1)]
model = Network(layers)

y_hat = train(model, X, 0.5)

print("A1:", model.layers[1].A)
print("largest dL/dW1:", model.layers[0].dW.abs().max())
print("y_hat:", y_hat.flatten(), "loss:", bce(y_hat, y))
```

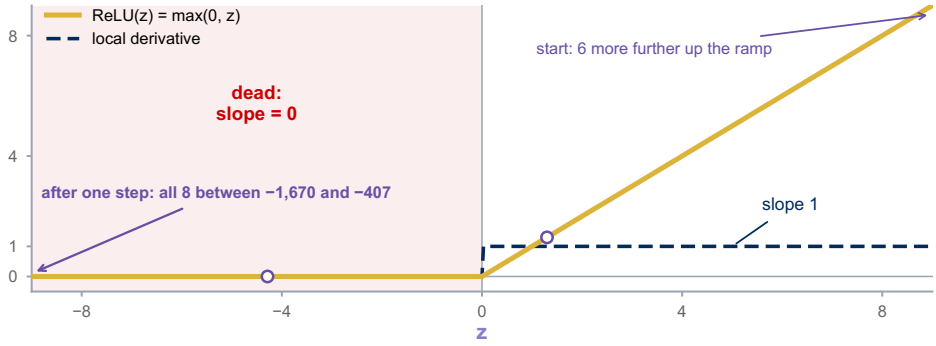
OUTPUT

```
A1: tensor([[1.0000e+00, 1.0000e+00],
        [1.0000e+00, 1.0000e+00],
        [1.0000e+00, 1.0000e+00],
        [1.0000e+00, 1.9767e-04]])
largest dL/dW1: tensor(8.6757e-05)
y_hat: tensor([0.6660, 0.6660, 0.6660, 0.0031]) loss: tensor(0.4782)
```

Same $2 \to 2 \to 1$ shape as last class (seed 11, $\alpha = 0.5$), and we keep it for the rest of the section. On raw scores the hidden layer is pinned at 1, so **three of the games look identical** to the output layer and almost no gradient reaches w_1.

Dying ReLUs

Similarly to the edges of a **sigmoid**, a **ReLU** gives **no useful gradient information** if the **Z** value is less than `0`.



Hollow dots: the same **8** values of Z_1 at the start, nearly all on the ramp. One gradient step later, all **8** are in the dead zone.

Dying ReLUs: Demo

```
torch.manual_seed(11) # ensure our random init is always the same

layers = [LinearLayer(2, 2), ReLU(), LinearLayer(2, 1)]
model = Network(layers)

y_hat = train(model, X, 0.5)

A1 = model.layers[1].forward(model.layers[0].forward(X))
dead = (A1.sum(dim=0) == 0).sum().item()

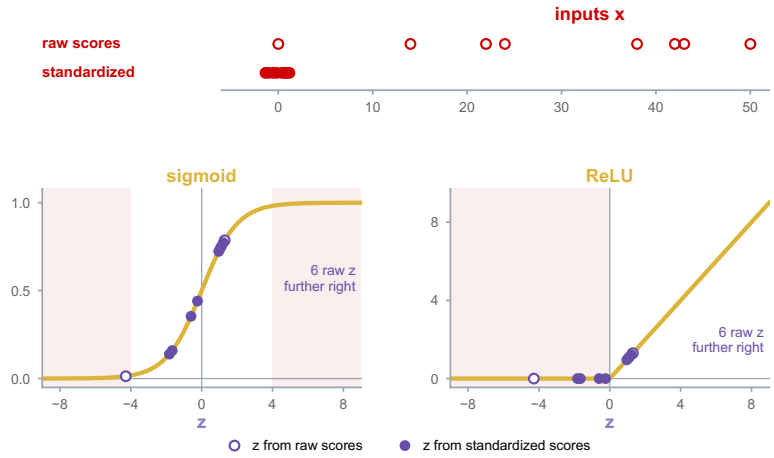
print("A1:", A1)
print("dead neurons:", dead, "of 2")
print("y_hat:", y_hat.flatten(), "loss:", bce(y_hat, y))
```

OUTPUT

```
A1: tensor([[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]])
dead neurons: 2 of 2
y_hat: tensor([0.5000, 0.5000, 0.5000, 0.5000]) loss: tensor(0.6931)
```

Same network, same starting weights, only the activation changed. Both neurons start alive, but the first step, with gradients up to 38 from the raw scores, pushes every z below 0 and **kills both**. After that every gradient is exactly 0.

Fixing Both: Standardize the Inputs



Standardized inputs sit on both sides of `0`, so every **z** starts near `0`: on the **steep part of the sigmoid**, and **positive for some games, negative for others** in each ReLU neuron.

Fixing Both: Standardize the Inputs

```
X_s = (X - X.mean(dim=0)) / X.std(dim=0)  # mean 0, std 1

torch.manual_seed(11)
layers = [LinearLayer(2, 2), Sigmoid(), LinearLayer(2, 1)]
y_hat = train(Network(layers), X_s, 0.5)
print("sigmoid y_hat:", y_hat.flatten())
print("sigmoid loss:", bce(y_hat, y))

torch.manual_seed(11)
layers = [LinearLayer(2, 2), ReLU(), LinearLayer(2, 1)]
y_hat = train(Network(layers), X_s, 0.5)
print("ReLU y_hat:", y_hat.flatten())
print("ReLU loss:", bce(y_hat, y))
```

OUTPUT

```
sigmoid y_hat: tensor([0.9953, 0.9929, 0.0056, 0.0091])
sigmoid loss: tensor(0.0067)
ReLU y_hat: tensor([9.9886e-01, 9.9886e-01, 8.8724e-05, 1.3661e-04])
ReLU loss: tensor(0.0006)
```

Backprop Exercise

(old final exam question)

```python
import torch
import torch.nn as nn
class Model4_1(nn.Module):
    def __init__(self):
        super(Model4_1, self).__init__()
        self.lin1 = nn.Linear(784, 100)
        self.relu = nn.ReLU()
        self.lin2 = nn.Linear(100, 10)

    def forward(self, x):
        out = self.lin1(x)
        out = self.relu(out)
        out = self.lin2(out)
        return out
```

Note: You can assume `softmax` and `cross entropy loss` are used after the final layer, the two yielding a combined derivative of: $\hat{y} - y$

a.) Draw the computation graph for the network. (4 pts.)

Backprop Exercise (old final exam question)

b.) Give the loss as a single nested equation. (4 pts.)

$\mathcal{L} =$

c.) Give the derivative chain for calculating the gradient of the bias in the first layer. (3 pts.)

$$\frac{\partial \mathcal{L}}{\partial b_1} =$$

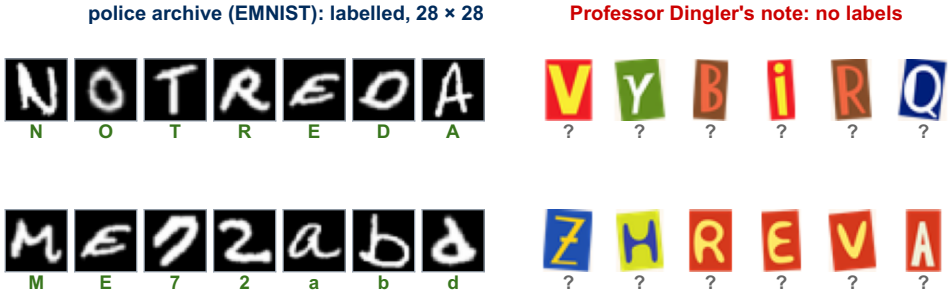
d.) Write the actual equation you'd use to calculate the gradient for the bias in the first layer. (1 pt.)

$$\frac{\partial \mathcal{L}}{\partial b_1} =$$

The MNIST Dataset

MNIST (LeCun and others, 1998) is the classic first dataset for neural networks: 70,000 **handwritten digits** (60,000 to train, 10,000 to test), each a 28 × 28 **grayscale image**. Flattened, that's the 784 **inputs** of the exam network we just did.

EMNIST (Cohen and others, 2017) adds letters in the same format. Its balanced split has 47 classes: 10 digits, 26 capitals and the 11 lowercase letters that don't look like their capitals.



In **Homework 06 and 07** we train on 9,400 of these from the police archive, then point the network at the **cut-out letters in Professor Dingler's note**.

Modern Momentum: ADAM

Plain **gradient descent** gives every weight the

α. The exam network has `79,510` weights and biases for MNIST, so weights with **big, noisy gradients overshoot** while weights with **tiny gradients barely move**. **Adam** (Adaptive Moment Estimation, Kingma and Ba, `2014`) is the optimizer I see most today, and it fixes the step with **two running averages**:

momentum: a running average of the gradient, so we keep rolling the way we've been going

$$m \leftarrow \beta_1 m + (1 - \beta_1) \frac{\partial \mathcal{L}}{\partial w}$$

a running average of the squared gradient: how big this weight's gradients usually are

$$v \leftarrow \beta_2 v + (1 - \beta_2) \left(\frac{\partial \mathcal{L}}{\partial w} \right)^2$$

every weight gets its own step size: big gradients take smaller steps, tiny ones take bigger steps

$$w \leftarrow w - \alpha \frac{m}{\sqrt{v} + \epsilon}$$

In `PyTorch` it's one line, `torch.optim.Adam(model.parameters(), lr=0.001)`, the optimizer **Homework 07's** **CNN** trains with.

NOTE: The defaults are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and Adam also corrects m and v for starting at `0`.

ADAM on MNIST

Kingma and Ba trained a `784` → `1000` → `1000` → `10` **ReLU** network with **dropout** on `MNIST`, the same kind of network as the exam question, just wider. **Adam falls fastest and ends lowest.**

