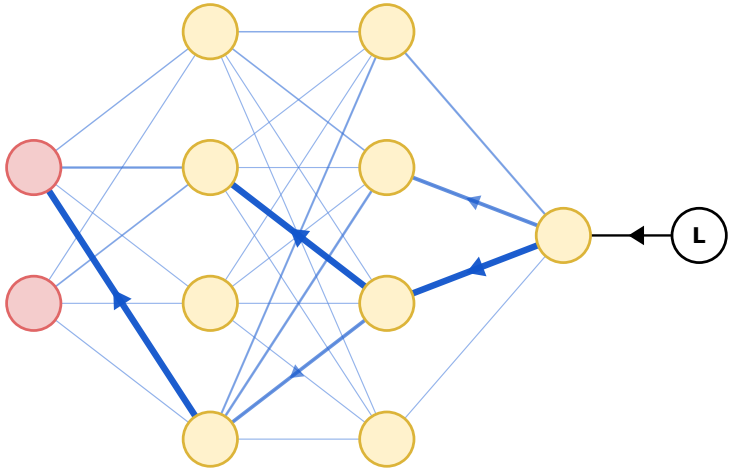
UNIT 03 · CHAPTER 1

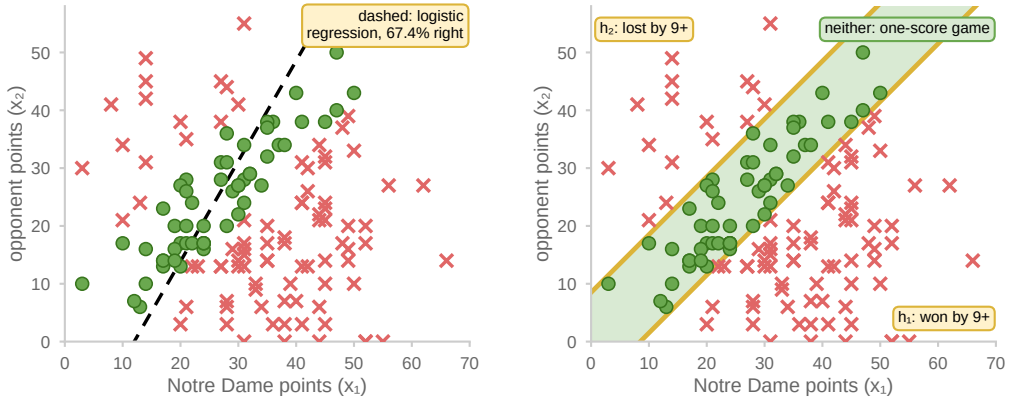
Backpropagation

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame



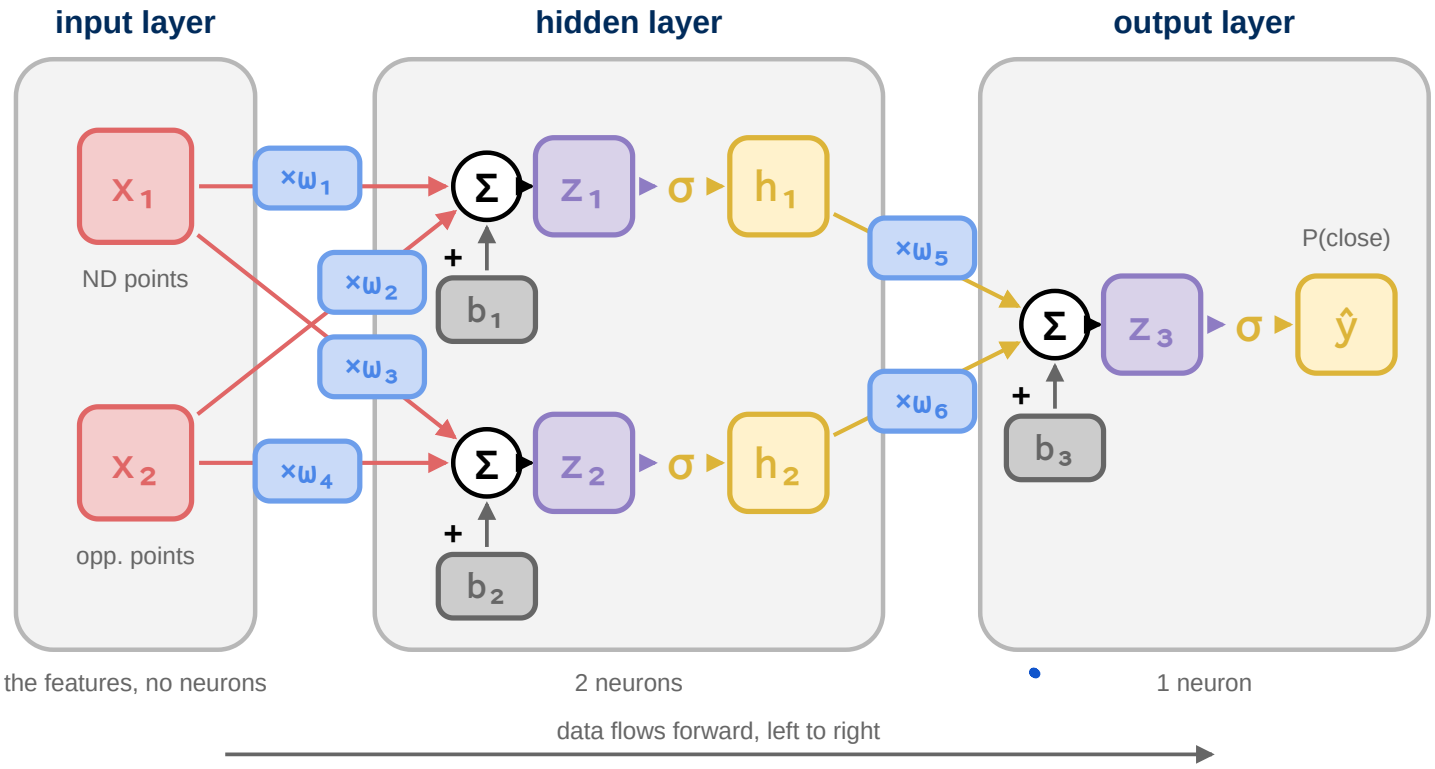
The XOR Problem

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** linearly separable

Neural Networks: Diagram



Combining our three **artificial neurons** gives us a simple **FFN**.
The organization of neurons in a neural network is called its architecture

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
```



$$z_1 = X w_1^T + b_1$$

$$A_1 = \sigma(z_1)$$

$$z_2 = A_1 w_2^T + b_2$$

$$L = BCE(\sigma(X w_1^T + b_1) w_2^T + b_2, y)$$

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.]]) # ND Score, Opp. Score
y = torch.tensor([[1.], [1.], [0.], [0.]])  # close = 1, not close = 0

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().flatten())

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

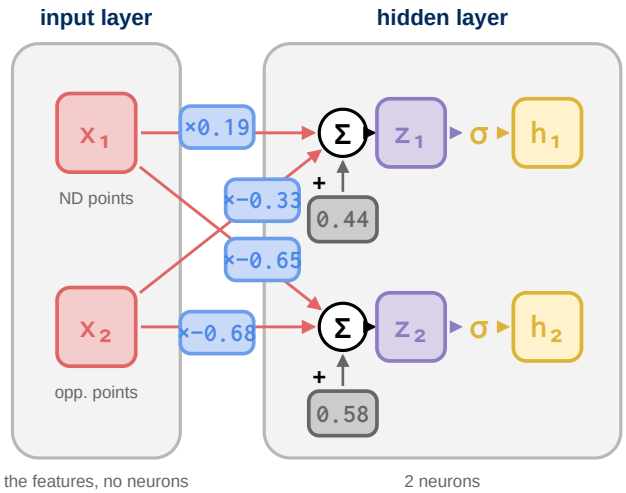
OUTPUT

```
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.

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



X

inputs, one row per game

<4, 2>

	x_1	x_2
2014 North Carolina	50	43
2015 Clemson	22	24
2012 Wake Forest	38	0
2012 Alabama	14	42

W_1

weights, one row per neuron

<2, 2>

	x_1	x_2
neuron 1	0.19	−0.33
neuron 2	−0.65	−0.68

b_1

biases, one per neuron

<2>

0.44
0.58

W_1^T

transposed: one column per neuron

<2, 2>

	neuron 1	neuron 2
x_1	0.19	−0.65
x_2	−0.33	−0.68

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	$Z = X\omega + b$	`Keras`/`TensorFlow`, many textbooks
Outputs × inputs	$Z = X\omega^T + b$	`PyTorch` `nn.Linear` and this course

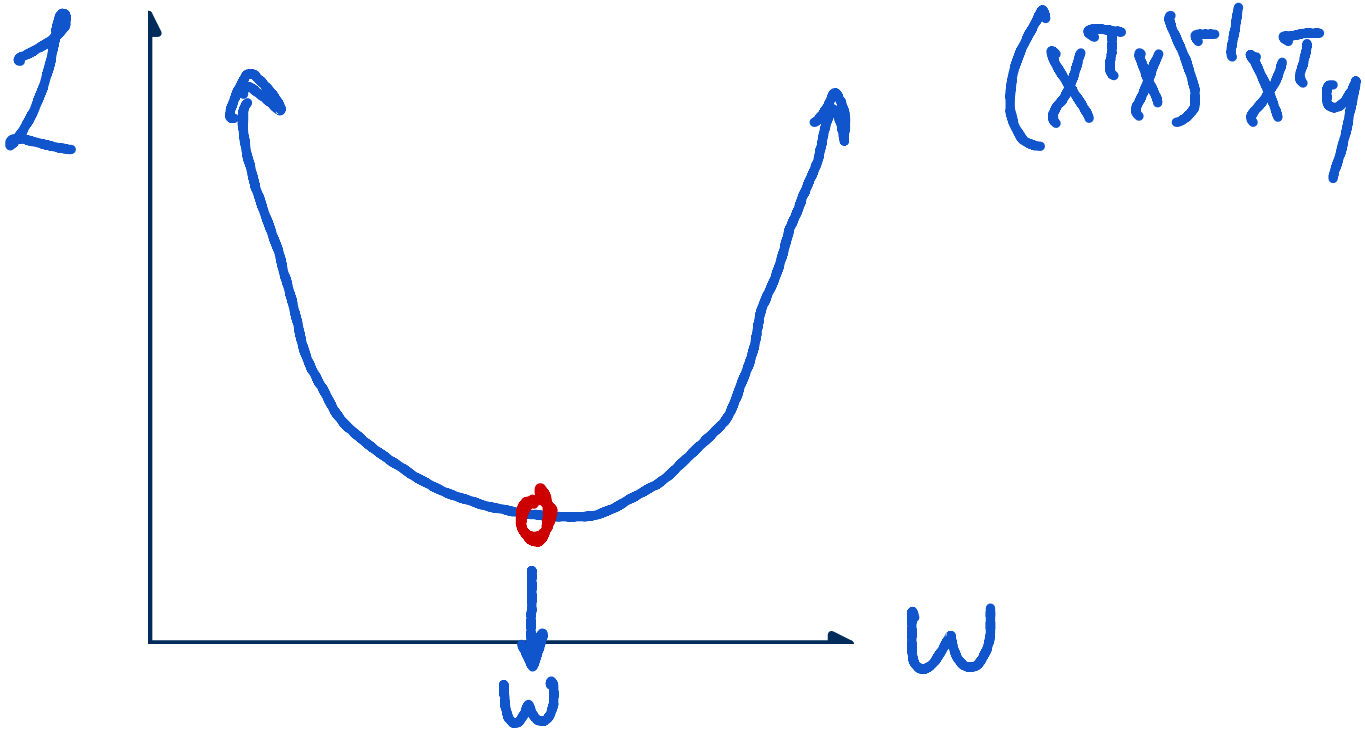
`PyTorch` stores **one row per neuron** so it reads like the math: $X\omega^T + b$

Improving the Weights

Linear Regressions and OLS

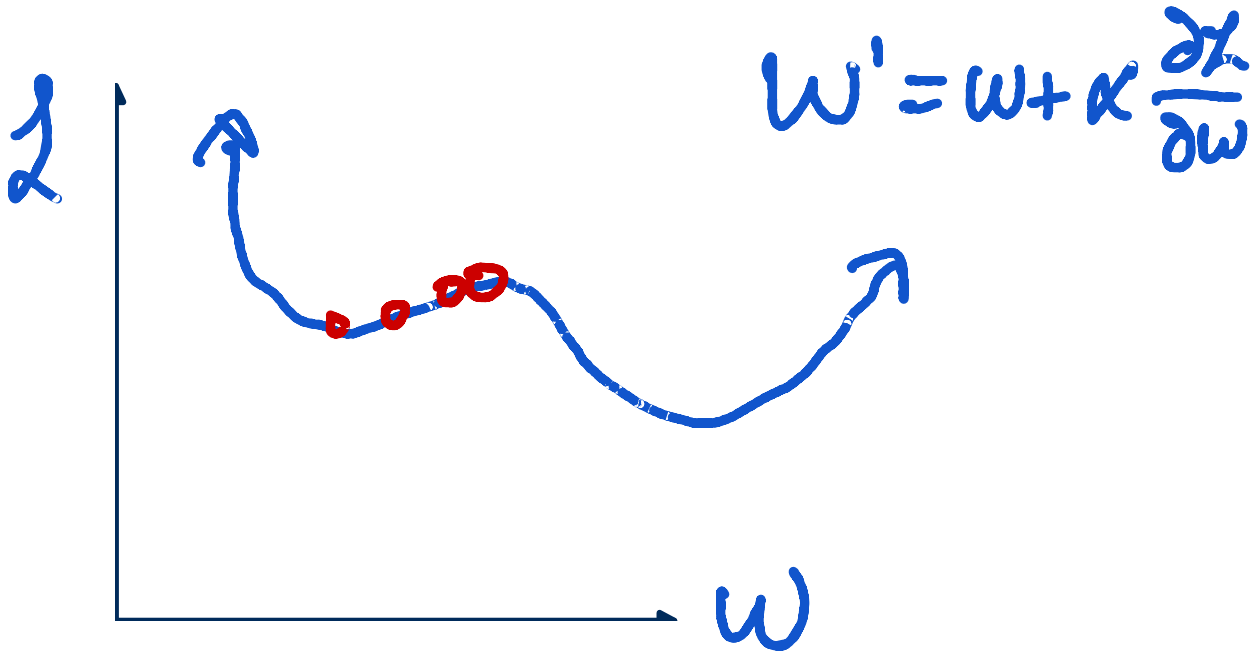
Due to the simplicity of **linear regression**, we could use a closed form sol. to **directly find the optimal set of weights**.

This is due to the loss curve being **perfectly convex**



Gradient Descent

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

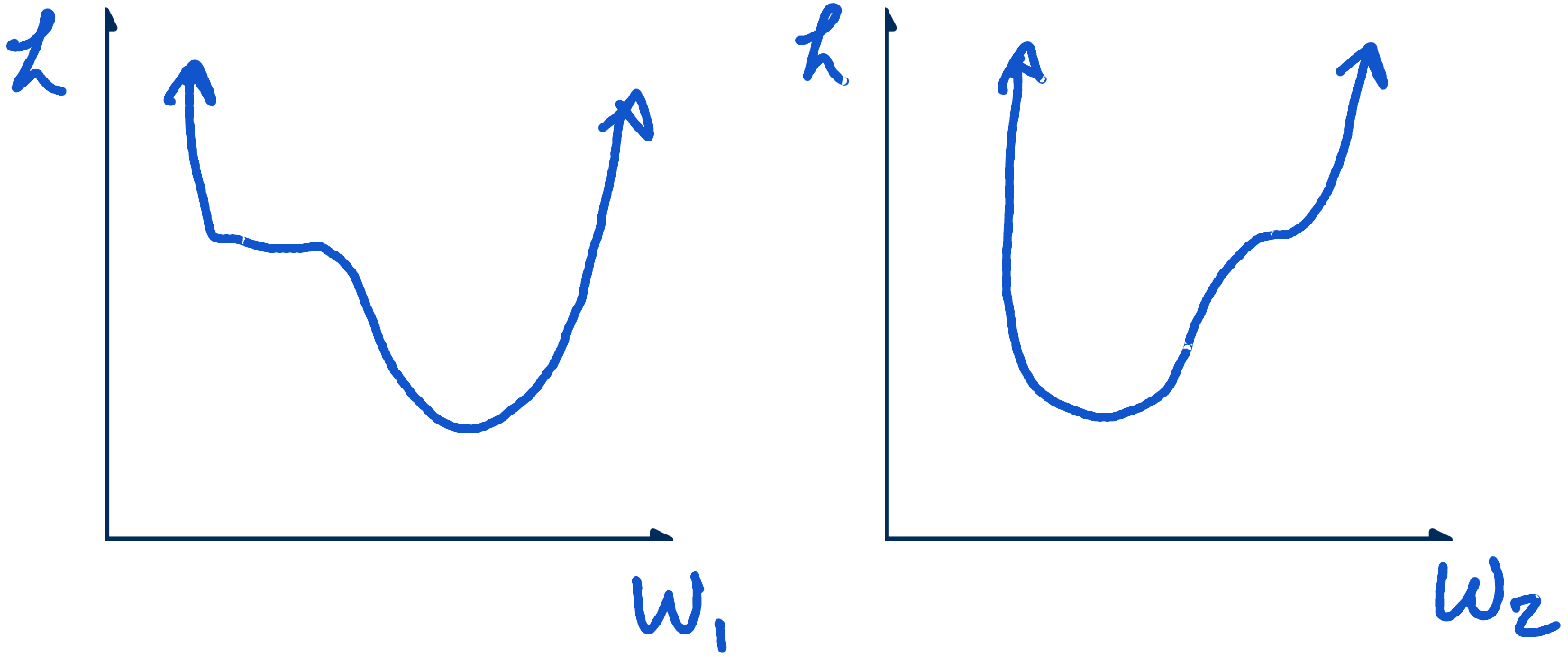


A **gradient** is the direction **at a given point in a specified direction**

In the case of a **univariate function**, it is simply **the** first deriv **at a selected point**.

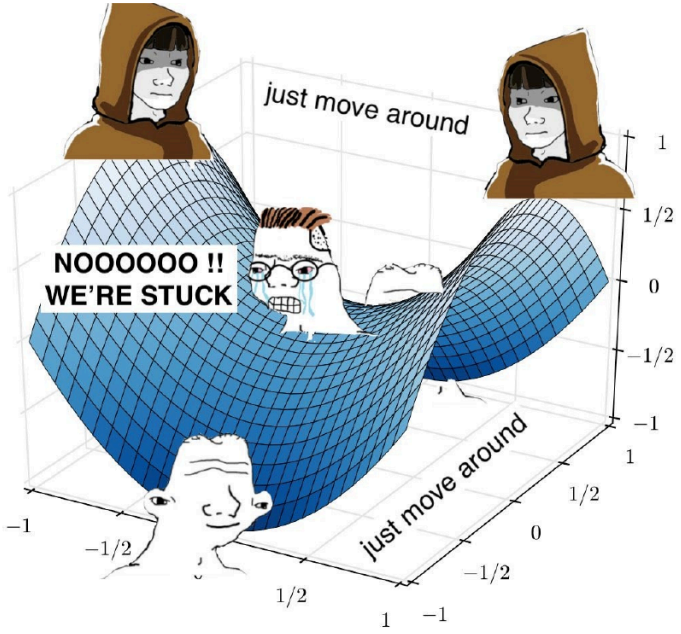
Multivariate Gradient Descent

In the case of a **multivariate function**, the gradient is a vector of partial derivatives

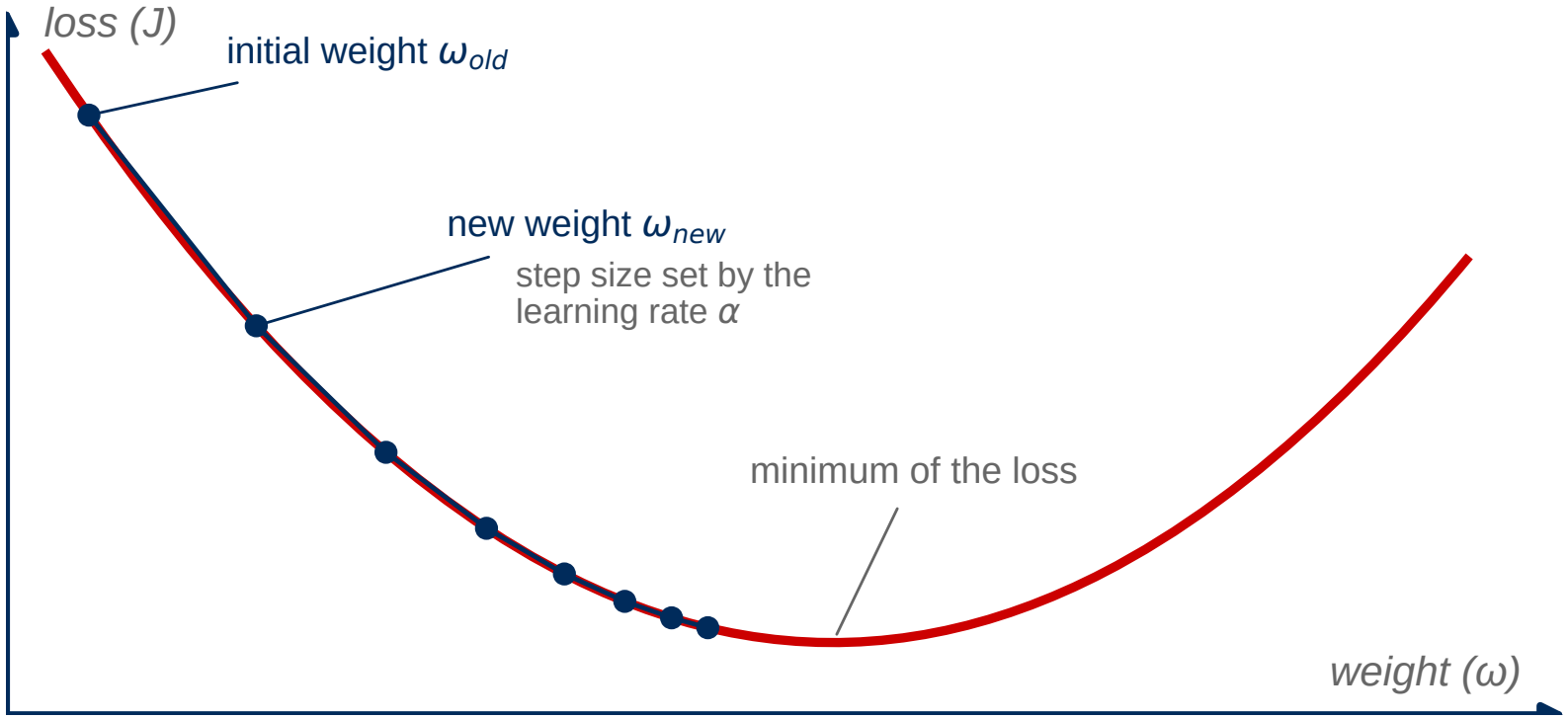


Quasi-Convex Functions

We can also use **quasi-convex functions** with **gradient descent**, but need to **watch out for** local minima



Gradient Descent Algorithm



[1.1] one step

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

Common Loss Functions

- Mean Squared Error MSE
- Mean abs. error MAE
- Binary Cross Entropy (BCE)
- Cross Entropy (CE)

Logistic Regression with Gradient Descent

(full derivation)

$$\mathcal{L} = BCE(\sigma(Xw^T + b), y)$$

$$\hat{y} = \sigma(Xw^T + b) \qquad w' = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$$

$$* \quad \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w}$$

$$z_1 = Xw^T + b$$

$$* \quad \frac{\partial \mathcal{L}}{\partial w} = (\hat{y} - y)\, X = \begin{bmatrix} \nabla w_{11} & \nabla w_{12} \\ \nabla w_{21} & \nabla w_{22} \end{bmatrix}$$

$$\hat{y} = \sigma(z_1)$$

$$L = BCE(\hat{y}, y)$$

Yes you should understand backprop



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

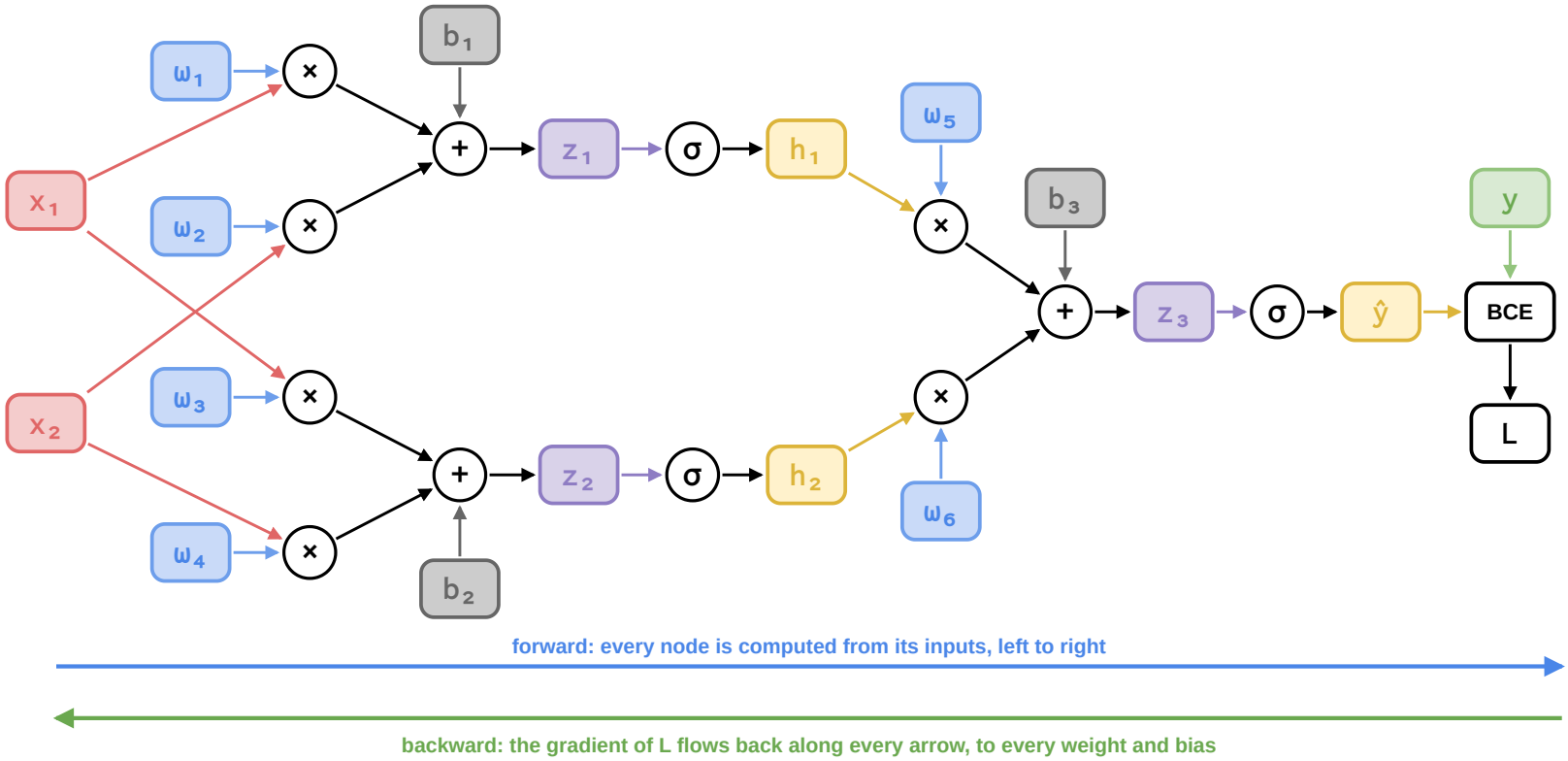
Backpropagation (Backprop)

Backpropagation is a technique in deep learning to compute the gradient w.r.t all params for a network.

Backprop is a app. of chain rule that allows us to **implement Gradient Descent** for multi-layer models

Computational Graphs

Every **network** can be represented by a Comp. graph, or a **graph of the operations happening to the input** as it passes through the network



The Chain Rule

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

[1.2] the chain rule

$$\frac{d}{dx}\left[f(g(x))\right] = f'(g(x))\,g'(x)$$

Python Backprop by Hand

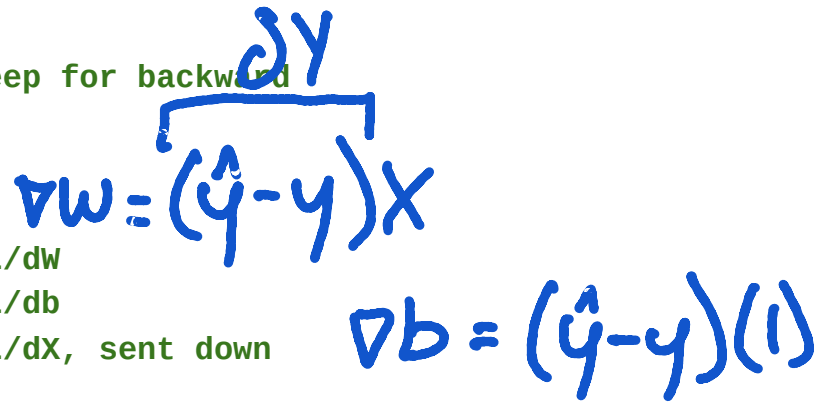
```python
import torch

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 X @ self.W.T + self.b

    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 down

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



∇W ∇b ∇X

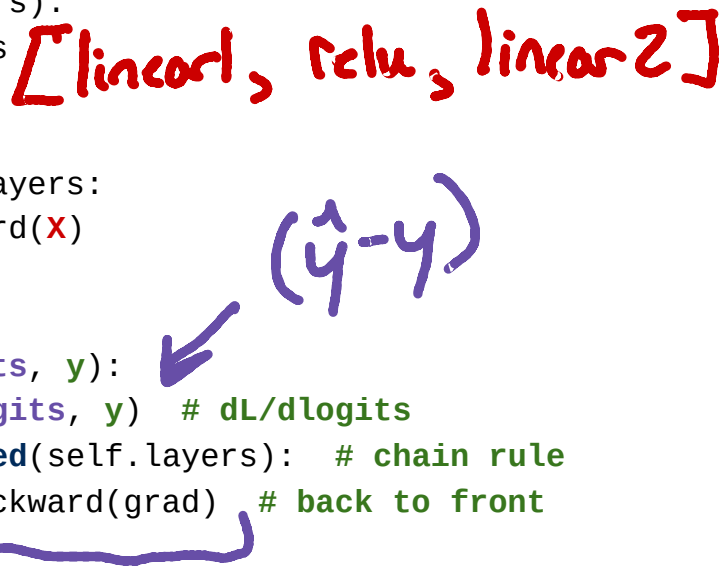
Python Backprop by Hand: The Network

```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, logits, y):
        grad = loss_grad(logits, y)  # dL/dlogits
        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)
```



Backprop Exercise (old final exam question)

4.) Given the pytorch code below, answer the following questions.

```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
```

$$z_1 = X w_1^T + b_1$$

$$A_1 = \text{ReLU}(z_1)$$

$$z_2 = A_1 w_2^T + b_2$$

$$\hat{y} = \text{softmax}(z_2)$$

$$L = CE(\hat{y}, y)$$

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

$$L = CE(\text{softmax}(\text{relu}(X w_1^T + b_1) w_2^T + b_2), y)$$

Backprop Exercise (old final exam question)

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

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

d.) Given the gradients, what's the next step we should take? (1 pt.)

e.) What task could this network be used for? (Hint: It's something I used as an example a lot) (1 pt.)

$$W_1' = W_1 - \alpha \frac{\partial L}{\partial W_1}$$

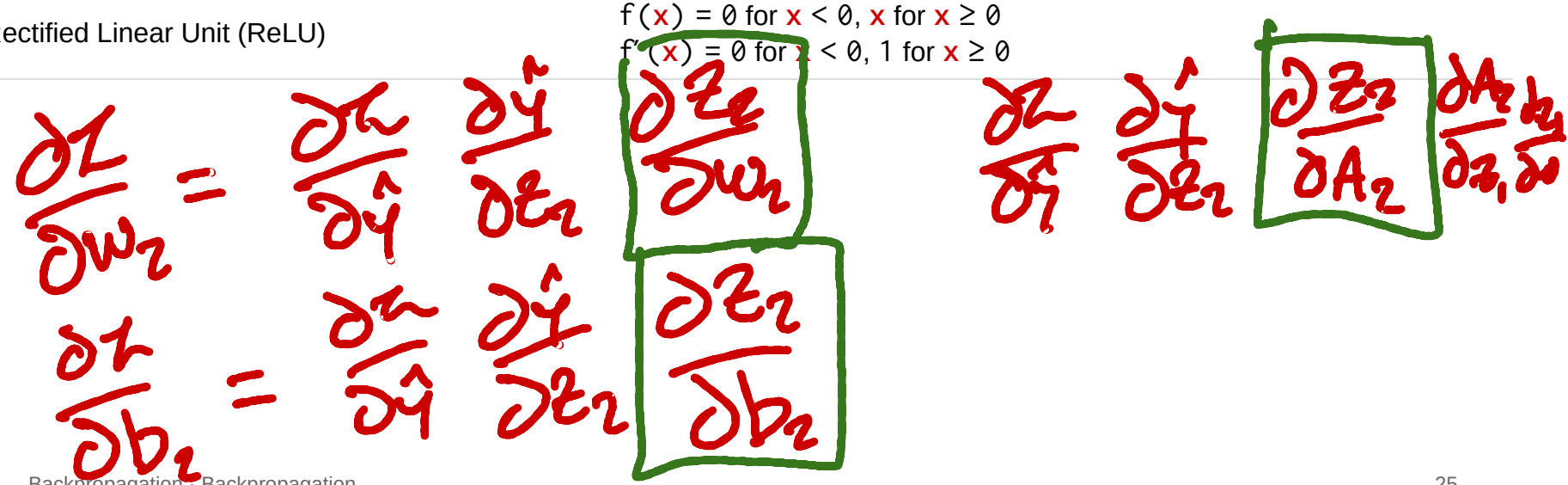
$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial W_2}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial A_1} \frac{\partial A_1}{\partial z_1} \frac{\partial z_1}{\partial W_1}$$

$$\frac{\partial L}{\partial W_1} = (\hat{y} - y)(W_2^T) \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} (X)$$

On the exam…

Name	Equation and derivative
Identity	$f(x) = x$ $f'(x) = 1$
Binary step	$f(x) = 0$ for $x < 0$, 1 for $x \geq 0$ $f'(x) = 0$ for $x \neq 0$, ? for $x = 0$
Logistic (a.k.a. Soft step)	$f(x) = 1 / (1 + e^{-x})$ $f'(x) = f(x)(1 - f(x))$
TanH	$f(x) = \tanh(x) = 2 / (1 + e^{-2x}) - 1$ $f'(x) = 1 - f(x)^2$
ArcTan	$f(x) = \tan^{-1}(x)$ $f'(x) = 1 / (x^2 + 1)$
Rectified Linear Unit (ReLU)	$f(x) = 0$ for $x < 0$, x for $x \geq 0$ $f'(x) = 0$ for $x < 0$, 1 for $x \geq 0$



On the exam…

Name	Equation and derivative
Mean Squared Error (MSE)	$L = (1/n)\ \Sigma\ (y - \hat{y})^2$ $\partial L/\partial \hat{y} = -2(y - \hat{y})$
Mean Absolute Error (MAE)	$L = (1/n)\ \Sigma\ \lvert y - \hat{y}\rvert$ $\partial L/\partial \hat{y} = -\mathrm{sign}(y - \hat{y})$
Huber Loss	$L = 0.5(y - \hat{y})^2$ if $\lvert y - \hat{y}\rvert \le \delta$, else $\delta(\lvert y - \hat{y}\rvert - 0.5\delta)$ piecewise gradient
Binary Cross Entropy (BCE)	$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$ $\partial L/\partial \hat{y} = (\hat{y} - y)\ /\ (\hat{y}(1 - \hat{y}))$

On the exam…

Name	Equation and derivative
BCE with Sigmoid	$L = -[y \log(\sigma(z)) + (1 - y) \log(1 - \sigma(z))]$ $\partial L / \partial z = \sigma(z) - y$
Categorical Cross Entropy	$L = -\Sigma_i \, y_i \log(\hat{y}_i)$ $\partial L / \partial z_j = \hat{y}_j - y_j$ (with $\hat{y} = \mathrm{softmax}(z)$)
KL Divergence	$L = \Sigma \, y \log(y \,/\, \hat{y})$ $\partial L / \partial \hat{y} = -y \,/\, \hat{y}$
Poisson Loss	$L = \hat{y} - y \log(\hat{y})$ $\partial L / \partial \hat{y} = 1 - y \,/\, \hat{y}$

Some Issues with Backprop

There are **some issues** that appear when you start **using backprop**:

- _____
- _____

Vanishing Gradient

Since you are multiplying increasingly _____ values with each other during backprop, the gradient eventually trends to _, meaning **updates will stop happening**.

Vanishing Gradient



Exploding Gradient

Almost the opposite of the vanishing gradient, the exploding gradient is when the gradients become _____

Local Minima

A common way to try and **break out of local minima** and **converge on solutions faster** is implementing _____

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 100)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        return self.fc2(F.relu(self.fc1(x)))

# Initialize network, optimizer and device (GPU)
device = torch.device("cuda" if torch.cuda.is_available()
                      else "cpu")
network = Net().to(device)
optimizer = optim.SGD(network.parameters(), lr=0.01,
                      momentum=0.5)
```

Modern Momentum: ADAM

The **most common gradient descent algorithm** (I've) seen today is **Adaptive Moment Estimation (ADAM)**. ADAM **combines the strengths of both momentum-based algorithms and adagrad**

