CUHK ECON

AI×ECON

Summer School @ **CUHK**

AUGUST 7-8 2024

Artificial Neural Network

AI × ECON Summer School @ CUHK

Artificial Neural Network

- The most prominent A.I. systems today are powered by artificial neural networks (ANN).
- ANN is the pinnacle of model complexity

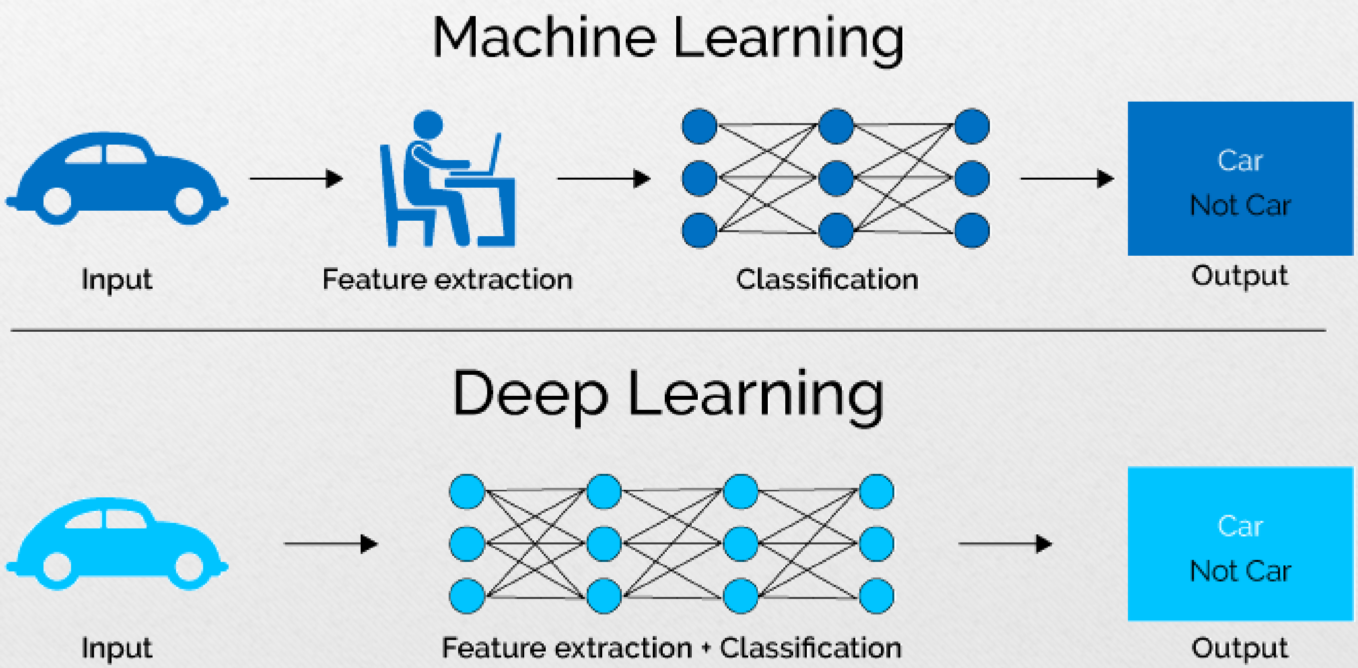


Image source: xenonstack

Artificial Neural Network

This tutorial:

- What is ANN?
- What are the common types of ANN?
- How to write an ANN in Python?

Image source: xenonstack

This is a Linear Regression

$$\overbrace{y}^{\text{Output}} = \alpha + \underset{\text{Data}}{\vec{x}}\vec{\beta}$$

Where $\vec{x} = [x_1 \quad \cdots \quad x_k]$ is one sample of features (a.k.a. one observation of independent variables)

$\vec{\beta} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_k \end{bmatrix}$ is a vector of weights (a.k.a. coefficients)

This is a Logistic Regression

$$\underbrace{P(y=1)}^{\text{Output}} = G\Big(\alpha + \underset{\text{Data}}{\vec{x}}\vec{\beta}\Big)$$

Where $G(z) = \frac{e^z}{1+e^z}$

This is a Neuron

$$\overbrace{h}^{\text{Output}} = G\left(\alpha + \underbrace{\vec{x}}_{\text{Input}}\vec{\beta}\right)$$

Where $G(z)$ is a non-linear function

$G(z)$ is called the **activation function** and h the neuron's **activation**

Artificial Neural Network

(A Very Simple One)

Output

Output Layer $h_2 = G_2(\alpha_2 + h_1\beta_2)$

Hidden Layer $h_1 = G_1(\alpha_1 + \vec{x}\vec{\beta}_1)$

Data

Artificial Neural Network

(With Two Hidden Neurons)

Output

Output Layer $h_2 = G\left(\alpha_2 + \vec{h}_1 \vec{\beta}_2\right)$

Hidden Layer $h_{1,1} = G\left(\alpha_{1,1} + \vec{x}\vec{\beta}_{1,1}\right)$ $h_{1,2} = G\left(\alpha_{1,2} + \vec{x}\vec{\beta}_{1,2}\right)$

Data

Artificial Neural Network

(With Multiple Hidden Neurons)

Output

Output Layer $h_2 = G_2(\alpha_2 + \vec{h}_1 \vec{\beta}_2)$

Hidden Layer $\vec{h}_1 = G_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$

Data

Artificial Neural Network

(With Two Hidden Neurons and Two Output Neurons)

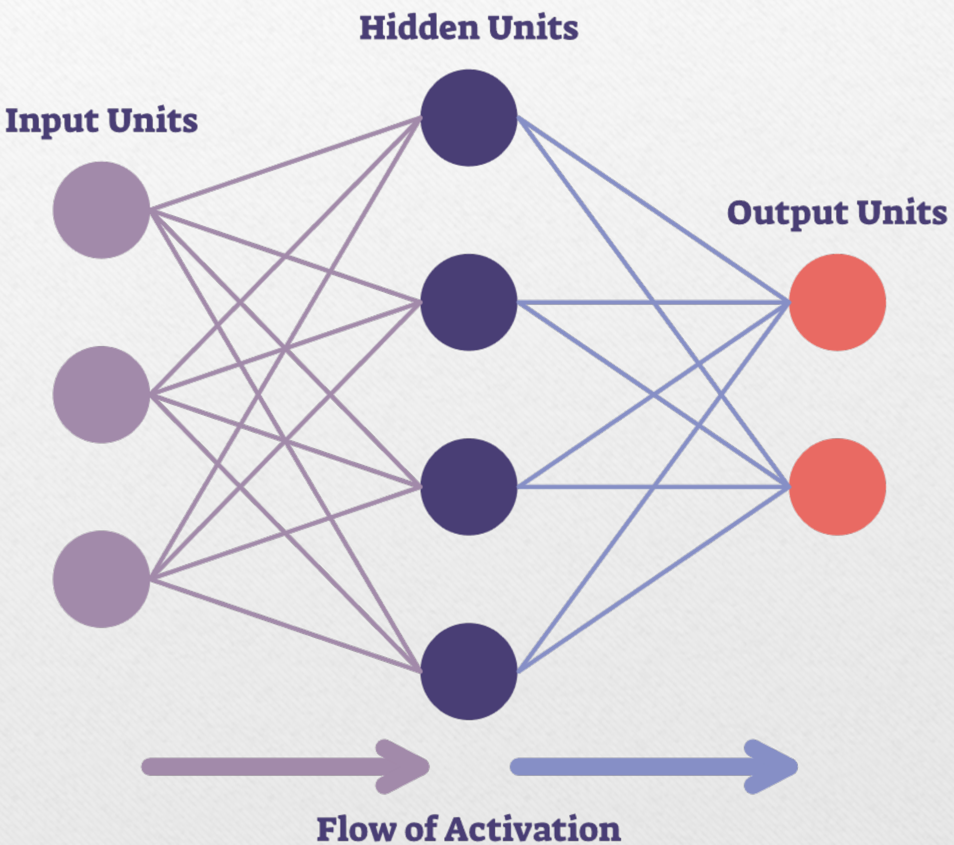
Output 1 | Output 2

Output Layer $h_{2,1} = G\left(\alpha_{2,1} + \vec{h}_1 \vec{\beta}_{2,1}\right)$ $h_{2,2} = G\left(\alpha_{2,2} + \vec{h}_1 \vec{\beta}_{2,2}\right)$

Hidden Layer $h_{1,1} = G\left(\alpha_{1,1} + \vec{x} \vec{\beta}_{1,1}\right)$ $h_{1,2} = G\left(\alpha_{1,2} + \vec{x} \vec{\beta}_{1,2}\right)$

Data

AI Neural Networks



Artificial Neural Network

(With Multiple Hidden Neurons and Outputs)

Output

Output Layer $\vec{h}_2 = G_2\left(\vec{\alpha}_2 + \vec{h}_1 \boldsymbol{\beta}_2\right)$

Hidden Layer $\vec{h}_1 = G_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$

Data

Classification Task

Softmax

(a.k.a. Multinomial Logit)

Output 1 Output 2

Output Layer $h_{2,1} = F\left(\alpha_{2,1} + \vec{h}_1\vec{\beta}_{2,1}\right)$ $h_{2,2} = F\left(\alpha_{2,2} + \vec{h}_1\vec{\beta}_{2,2}\right)$

Hidden Layer $h_{1,1} = F\left(\alpha_{1,1} + \vec{x}\vec{\beta}_{1,1}\right)$ $h_{1,2} = F\left(\alpha_{1,2} + \vec{x}\vec{\beta}_{1,2}\right)$

Data

Why Does Activation Have to be Non-Linear?

$$h_2 = G_2(\alpha_2 + \beta_2 h_1)$$

$$h_1 = G_1(\alpha_1 + \vec{\beta}_1 \vec{x})$$

If G_1 is linear, you get

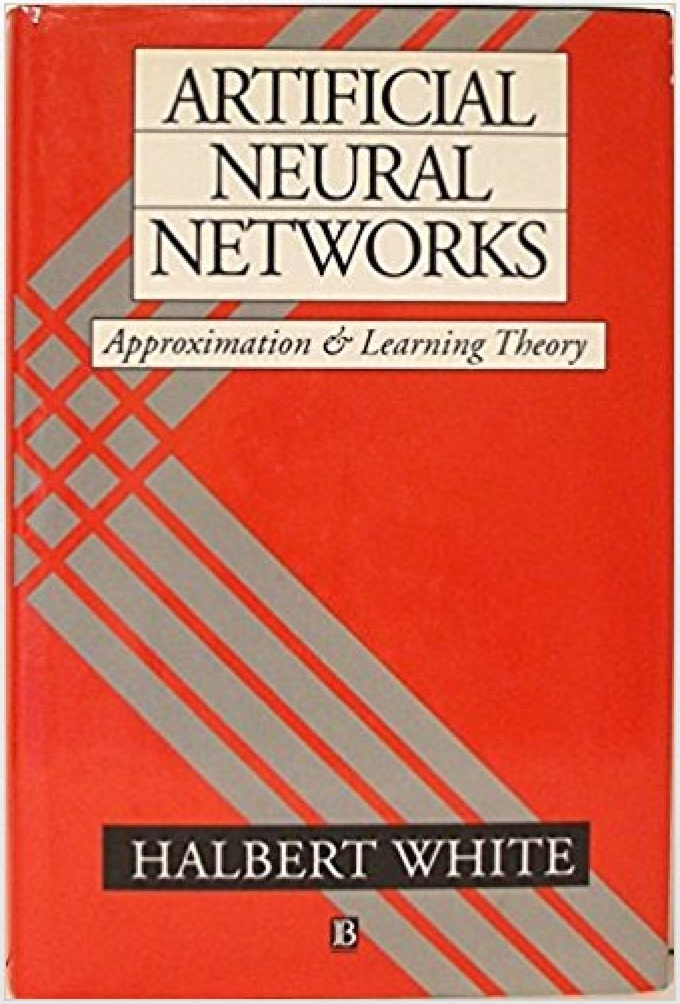
$$h_2 = G_2\left(\alpha_2 + \beta_2(\alpha_1 + \vec{\beta}_1 \vec{x})\right)$$

$$= G_2(\alpha_2' + \vec{\beta}_2' \vec{x})$$

So there is no point in having the hidden layer.

An econometrician's view of artificial neural network: a bunch of regressions stacked together

- Halbert White made significant contribution to the theoretical foundation of ANN in the 1980s
- Application was rare because the lack of computational power



Why is ANN Powerful?

Definition A function $\Psi: R \rightarrow [0,1]$ is a **squashing function** if it is non-decreasing, $\lim_{\lambda \to \infty} \Psi(\lambda) = 1$ and $\lim_{\lambda \to -\infty} \Psi(\lambda) = 0$.

Universal approximation theorem

(Hornik, Stinchcombe and White 1989 Theorem 2.4)

Let $\Sigma^r(G) = \left\{f: \mathbb{R}^r \rightarrow \mathbb{R}: f(x) = \sum_{i=1}^{q} \beta_i G(a_i + b_i x), a_i, b_i, \beta_i \in \mathbb{R}^r, q = 1,2, \ldots\right\}$

For every squashing function Ψ, every continuous function $g: \mathbb{R}^r \rightarrow \mathbb{R}$, every compact subset $K \subset \mathbb{R}^r$ and every $\varepsilon > 0$, there exists an $f \in \Sigma^r(\Psi)$ such that

$$\sup_{x \in K} |f(x) - g(x)| < \varepsilon$$

Why is ANN Powerful?

The universal approximation theorem says ANN can approximate any continuous function arbitrarily well.

The theorem can be extended to multiple outputs and non-continuous functions under additional assumptions.

Universal approximation theorem

(Hornik, Stinchcombe and White 1989 Theorem 2.4)

Let $\Sigma^r(G) = \{f: \mathbb{R}^r \to \mathbb{R}: f(x) = \sum_{i=1}^{q} \beta_i G(a_i + b_i x), a_i, b_i, \beta_i \in \mathbb{R}^r, q = 1,2, \dots\}$

For every squashing function Ψ, every continuous function $g: \mathbb{R}^r \to \mathbb{R}$, every compact subset $K \subset \mathbb{R}^r$ and every $\varepsilon > 0$, there exists an $f \in \Sigma^r(\Psi)$ such that

$$\sup_{x \in K} |f(x) - g(x)| < \varepsilon$$

Why is ANN Powerful?

Note that this is an existence theorem—it does not say how many neurons q is needed, which squashing/activation function Ψ works best, or how to get the biases and weights.

Universal approximation theorem

(Hornik, Stinchcombe and White 1989 Theorem 2.4)

Let $\Sigma^r(G) = \{f: \mathbb{R}^r \to \mathbb{R}: f(x) = \sum_{i=1}^{q} \beta_i G(a_i + b_i x), a_i, b_i, \beta_i \in \mathbb{R}^r, q = 1,2, \ldots\}$

For every squashing function Ψ, every continuous function $g: \mathbb{R}^r \to \mathbb{R}$, every compact subset $K \subset \mathbb{R}^r$ and every $\varepsilon > 0$, there exists an $f \in \Sigma^r(\Psi)$ such that

$$\sup_{x \in K} |f(x) - g(x)| < \varepsilon$$

Deep Learning

- Deep Learning refers to the stacking of multiple hidden layers
- Typical layer count is in the single digit but can go as high as a hundred.

Output

$$\vec{h}_n = G_2(\vec{\alpha}_2 + \vec{h}_{n-1}\boldsymbol{\beta}_n)$$

Hidden Layers

$$\vec{h}_{n-1} = G_1(\vec{\alpha}_1 + \vec{h}_1\boldsymbol{\beta}_{n-1})$$

$$\vec{h}_1 = G_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$$

Data

How Does an ANN Learn?

- **Gradient Descent**

Gradient Descent

- **Gradient descent** is a very general optimization algorithm, usable with pretty much all models.
- It uses the first derivative of the loss function with respect to a parameter to adjust the parameter.

Gradient Descent

- Gradient descent uses the <u>first derivative</u> of the loss function with respect to a parameter to adjust the parameter.
- E.g. suppose our model is

$$\hat{y} = \alpha + x$$

This is a (simplified) regression task, which we can solve by minimizing the squared error:

$$c = (y - \hat{y})^2$$

- For illustration purpose, let us assume the data is generated by $y = 5 + x$, so that if $x = 1, y = 6$.

Gradient Descent

Error $\epsilon \quad = y - \hat{y}$

Loss $\quad = \epsilon^2$

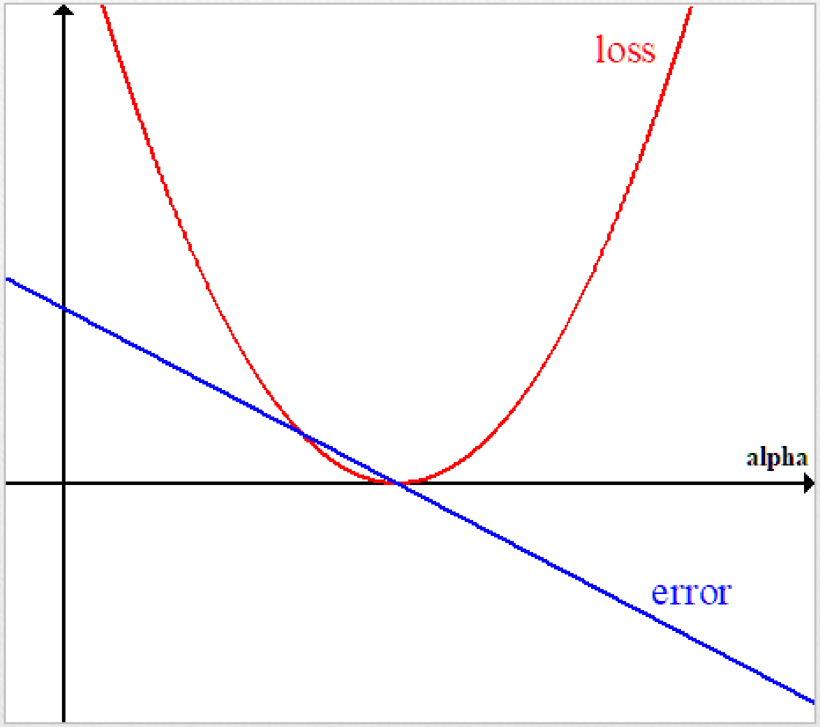
First derivative of loss function:

$$\frac{d}{d\alpha}\epsilon^2$$

$$= 2\epsilon \frac{d}{d\alpha}[y - (\alpha + x)]$$

$$= -2\epsilon$$

This is the marginal effect, or **gradient**, of α on loss.



Gradient Descent

We have an initial guess of what α is (usually just a random number.)

This gives us an initial prediction $\hat{y}$ and corresponding error and loss.

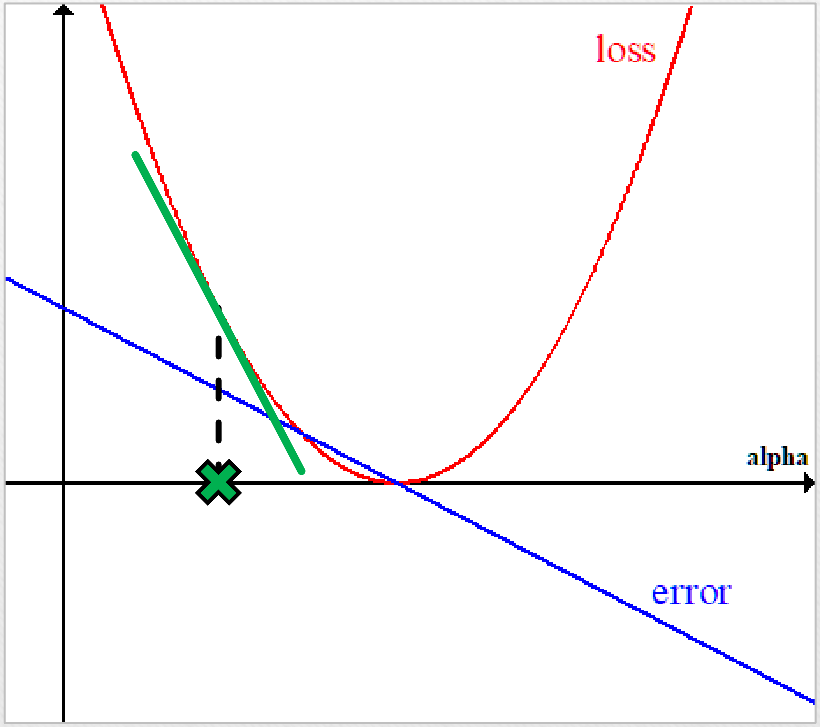
e.g.

$\hat{\alpha}_0 = 2$

$\hat{y}_0 = 2 + 1 = 3$

$\epsilon_0 = y - \hat{y} = 6 - 3 = 3$

$\frac{d}{d\alpha}\epsilon^2 = -2\epsilon = -6$



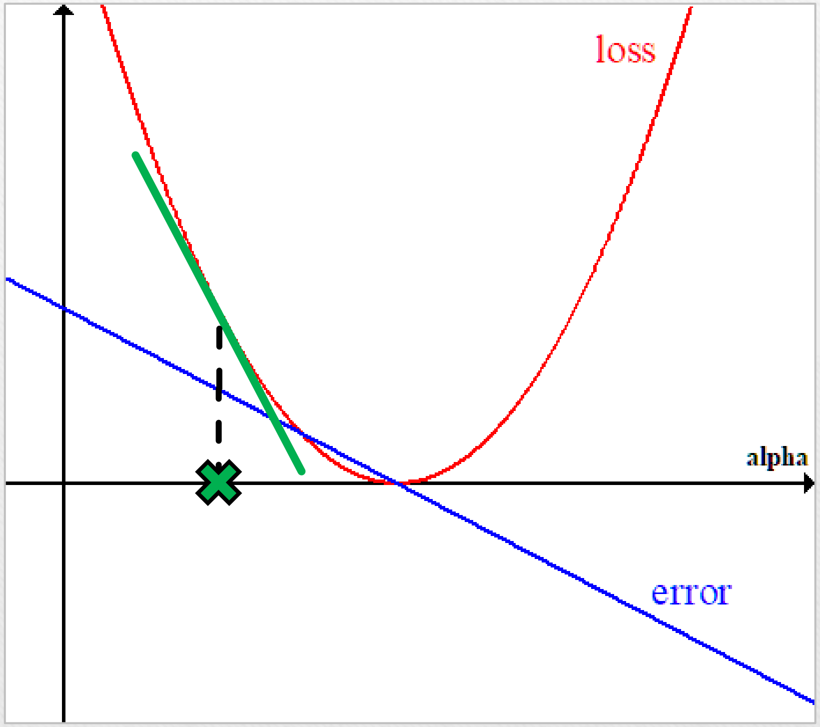
Gradient Descent

$\epsilon_0 = 3$

$\frac{d}{d\alpha}\epsilon^2 = -6 > 0$

In our example, ϵ is positive, so the gradient is negative.

This means loss is decreasing in alpha.

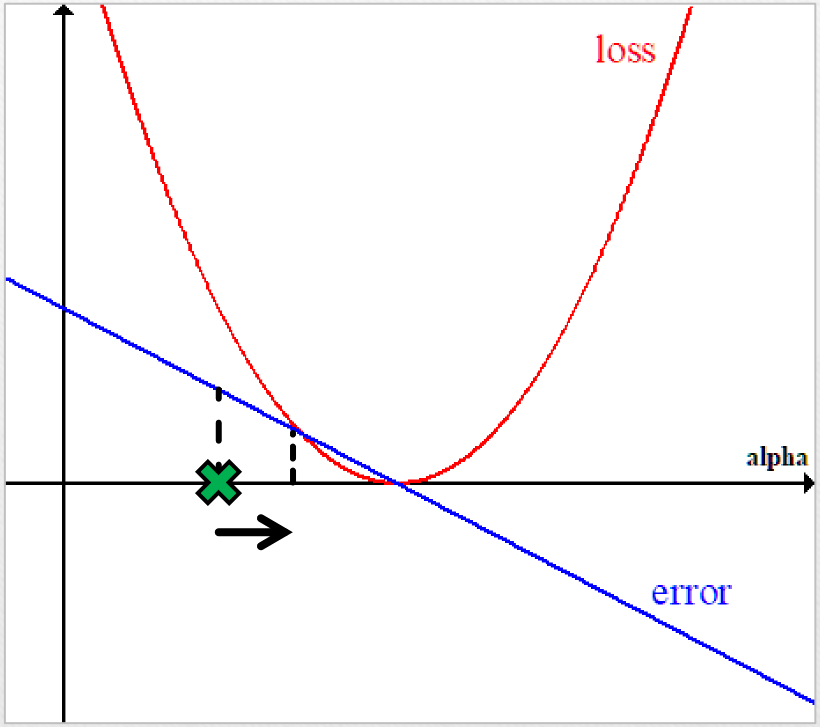


Gradient Descent

This makes sense, because if

$$y > \hat{y} = \alpha + x$$

increasing α will bring $\hat{y}$ closer to y.

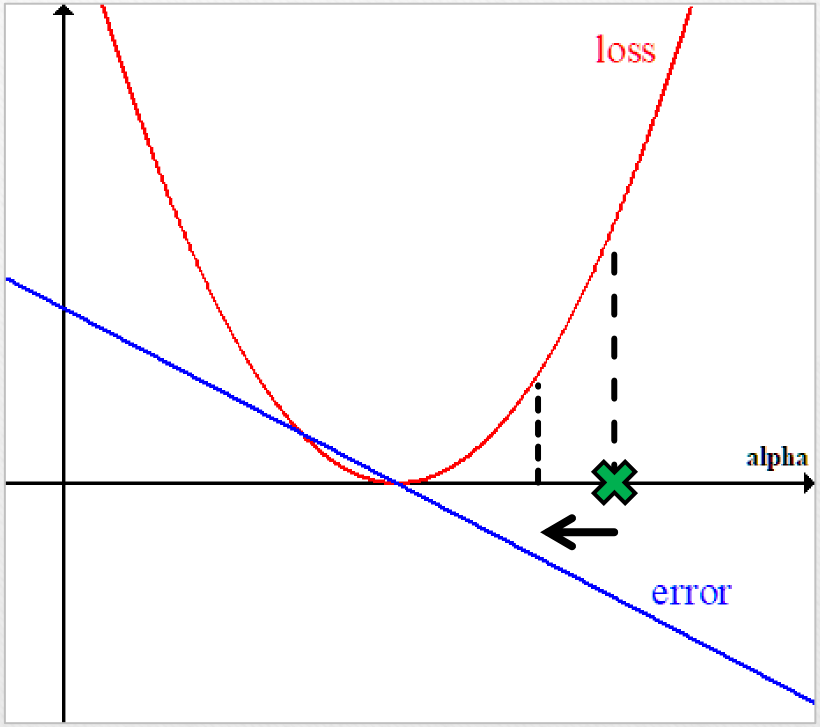


Gradient Descent

Conversely, if $\epsilon < 0$, then

$$y < \hat{y} = \alpha + x$$

decreasing α will bring $\hat{y}$ closer to y.



Gradient Descent

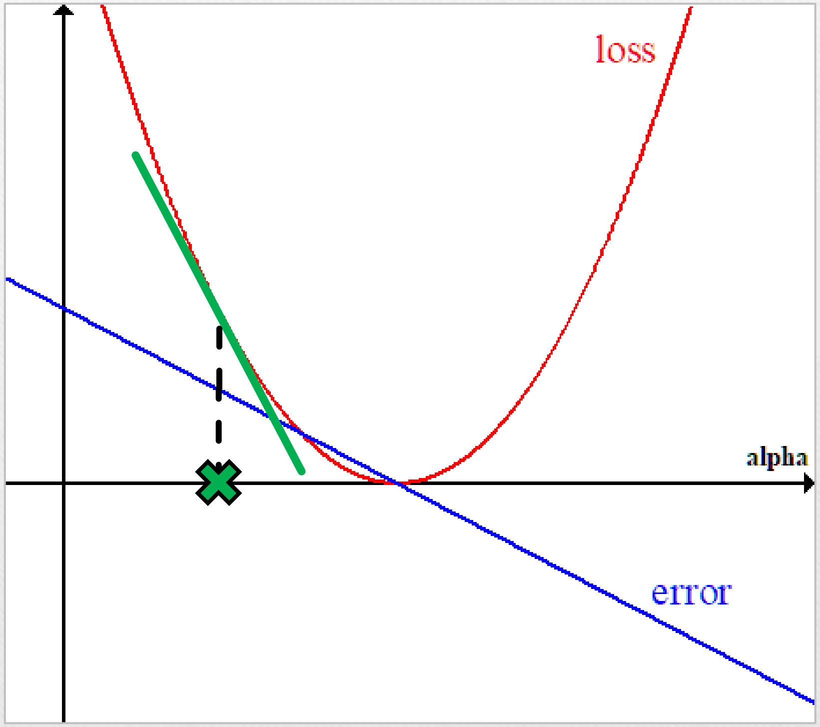
The gradient tells us the direction we need to adjustment our parameter.

The amount we need to adjust is, to a first-order approximation, inversely proportional to the gradient.

E.g.

$$\frac{d}{d\alpha}\epsilon^2 = -6$$

α needs to be bigger.

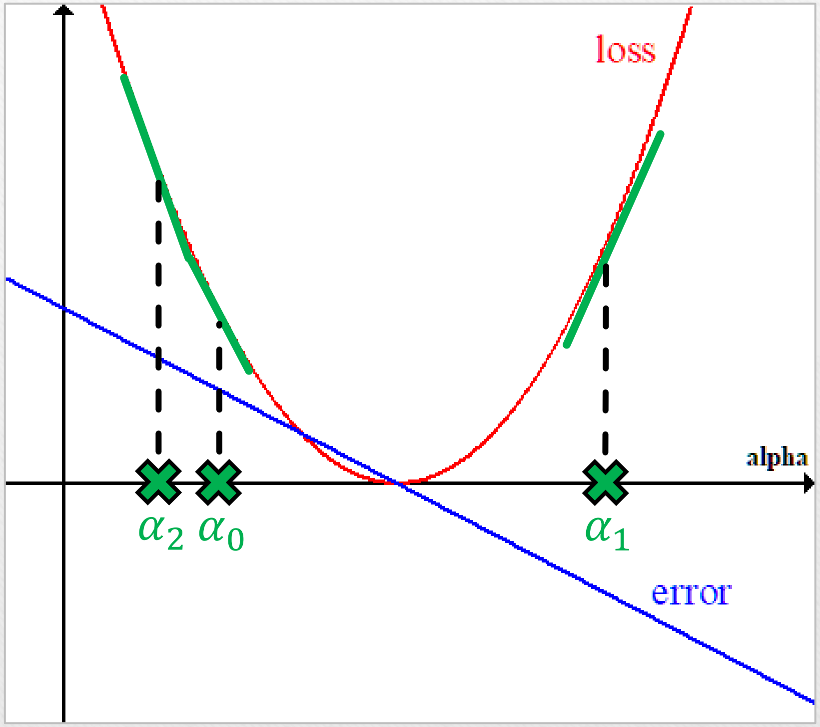


Gradient Descent

If we adjust the parameter by the exact amount of the gradient, we might overshoot:

$$\hat{\alpha}_1 = \hat{\alpha}_0 - \frac{d}{d\alpha}\epsilon^2$$
$$= 2 - (-6)$$
$$= 8 > 5 = \text{true } \alpha$$

Overshooting could result in our parameters bouncing around the true value, never to converge.



Gradient Descent

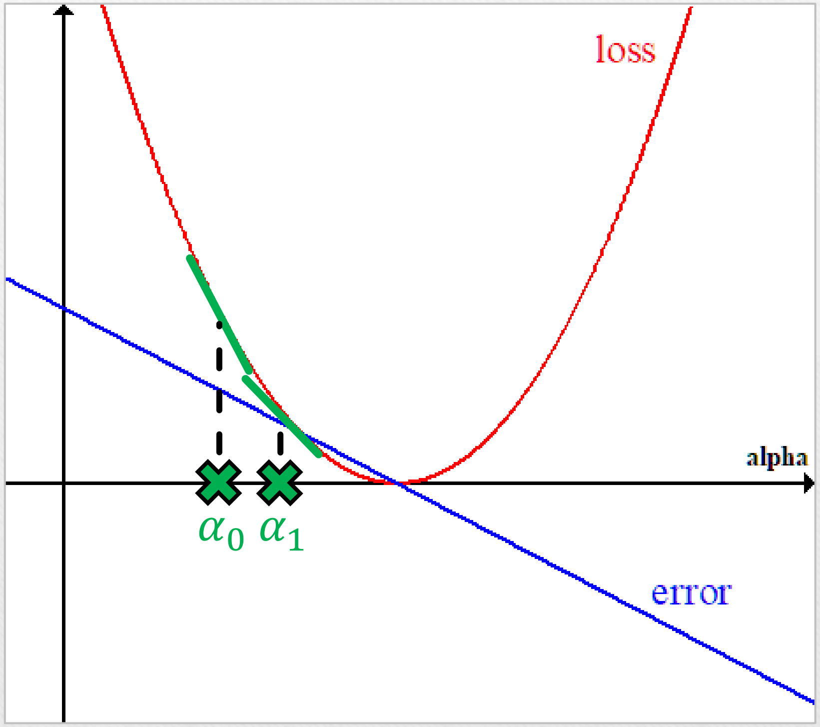
To prevent overshooting, we moderate the gradient by a *learning rate* γ:

$$\hat{\alpha}_1 = \hat{\alpha}_0 - \gamma \frac{d}{d\alpha} \epsilon^2$$

E.g. if $\gamma = 0.1$:

$$\begin{aligned} \hat{\alpha}_1 &= \hat{\alpha}_0 - \gamma \frac{d}{d\alpha} \epsilon^2 \\ &= 2 - 0.1(-6) \\ &= 2.6 < 5 = \text{true } \alpha \end{aligned}$$

Now we are not overshooting, but it will take us more iterations to get to the true α value.



Gradient Descent

- The gradient tells us the direction we need to adjustment our parameter. In the above example, the amount we need to adjust is, to a first-order approximation, proportional to $-dc/d\alpha$.
- Moderating the adjustment by a learning rate γ, we have the following update rule:

$$\alpha_t = \alpha_{t-1} - \gamma \left.\frac{dc}{d\alpha}\right|_{\alpha_{t-1}}$$

Or more typical in computer science:

$$\alpha \leftarrow \alpha - \gamma \frac{dc}{d\alpha}$$

- With more than one sample, we take the average.

Stochastic Gradient Descent

- Learning is quite slow if we only update model weights after we go through all data.
- We could instead update every time after we gone through a given number of samples. This is **Stochastic Gradient Descent (SGD).**
- Because we are not using all data, we could be updating towards the wrong direction sometimes, but on average the updates will be correct. Hence, *stochastic*.
- The stochastic nature is actually a good property because it helps the model escape from local optima.

How Does an ANN Learn?

- **Gradient Descent**
- **Back Propagation**

Back Propagation

- Because neural network have multiple layers, the gradient of lower layers needs be computed using the chain rule. This process is called **back propagation**.

Output

$$\vec{h}_2 = F_2\left(\vec{\alpha}_2 + \vec{h}_1 \boldsymbol{\beta}_2\right)$$

$$\vec{h}_1 = F_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$$

Data

Back Propagation

- Because neural network have multiple layers, the gradient of lower layers needs be computed using the chain rule. This process is called **back propagation**.

Error

$$\vec{h}_2 = F_2\left(\vec{\alpha}_2 + \vec{h}_1 \boldsymbol{\beta}_2\right)$$

$$\vec{h}_1 = F_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$$

Back Propagation

E.g. single target regression task, the gradient of the second weight of the first neuron is:

$$\frac{\partial}{\partial \beta_1^{(1,2)}}(y - h_2)^2$$

$$= -2(y - h_2)F_2'(\vec{\alpha}_2 + \vec{h}_1\vec{\beta}_2)\frac{\partial}{\partial \beta_1^{(1,2)}}(\vec{\alpha}_2 + \vec{h}_1\vec{\beta}_2)$$

$$= -2(y - h_2)F_2'(\vec{\alpha}_2 + \vec{h}_1\vec{\beta}_2)\beta_2^{(1)}$$

$$\cdot \frac{\partial}{\partial \beta_1^{(1,2)}}F_1\left(\alpha_1 + \vec{x}\vec{\beta}_1^{(1)}\right)$$

$$= -2(y - h_2)F_2'(\vec{\alpha}_2 + \vec{h}_1\vec{\beta}_2)\beta_2^{(1)}$$

$$\cdot F_1'\left(\vec{\alpha}_1 + \vec{x}\vec{\beta}_1^{(1)}\right)x_2$$

An important feature of any neural network library is to automate the computation of gradient.

Error

$$\vec{h}_2 = F_2(\vec{\alpha}_2 + \vec{h}_1\boldsymbol{\beta}_2)$$

$$\vec{h}_1 = F_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$$

Back Propagation

- As the number of layers go up, the process becomes quite brittle.
- Remember the chain rule involves a lot of multiplication.
- Small times small means very small, resulting in **vanishing gradient**.
- Big times big means very big, resulting in **exploding gradient**.
- Neither is good for learning.
- A lot of research goes into finding ways to combat the issue: different activation functions, different randomization distributions…

Error

$$\vec{h}_2 = F_2\left(\vec{\alpha}_2 + \vec{h}_1 \boldsymbol{\beta}_2\right)$$

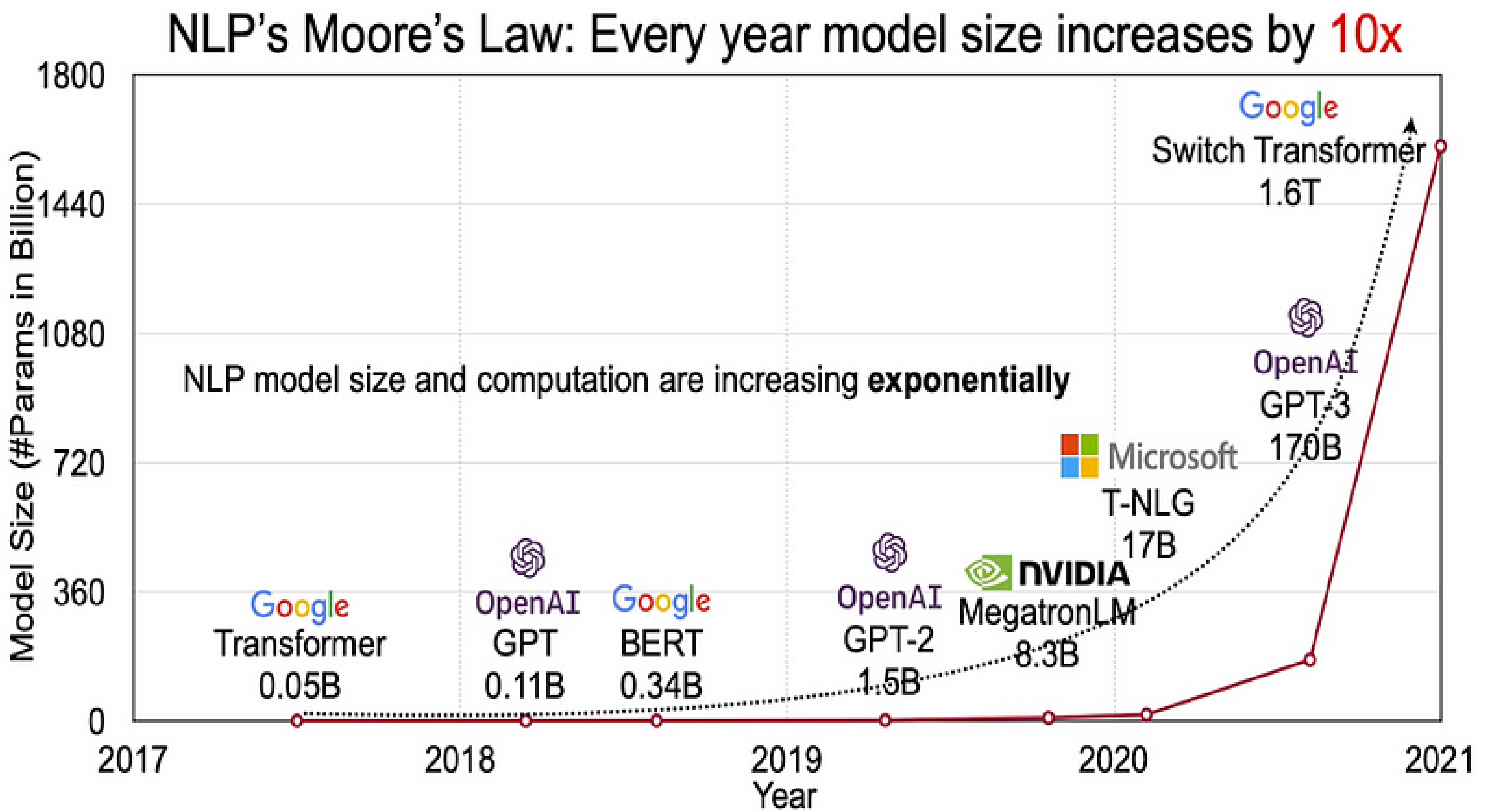
$$\vec{h}_1 = F_1(\vec{\alpha}_1 + \vec{x}\boldsymbol{\beta}_1)$$

Hands-on Demo

Computation

- The idea of artificial neural network can be traced back to as far back as the 1940s.
- Due to the large number of parameters and large data size involved, effective use of ANN was prohibitive until 2010s.

Model Size



Source: Harishdatalab@medium

It Takes a Lot to Train These Models!

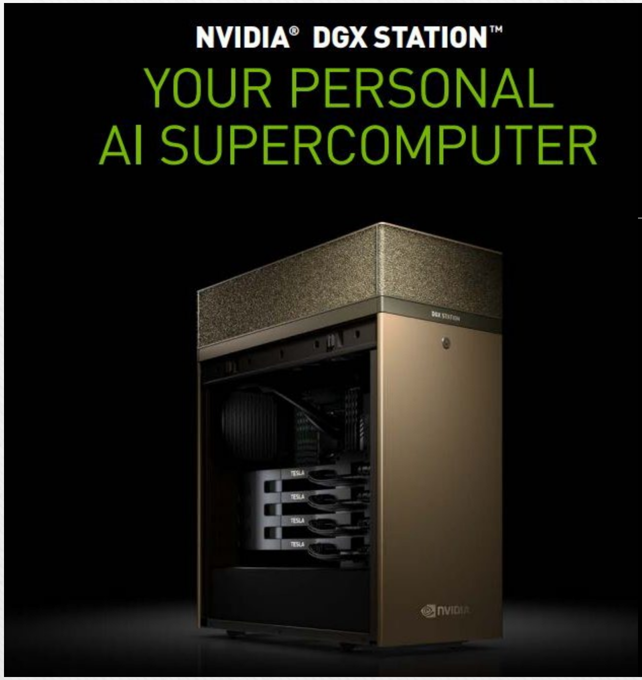
Model size	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Batch size	Achieved teraFlOPs per GPU	Percentage of theoretical peak FLOPs	Achieved aggregate petaFLOPs
1.7B	2304	24	1.7	1	32	512	137	44%	4.4
3.6B	3072	30	3.6	2	64	512	138	44%	8.8
7.5B	4096	36	7.5	4	128	512	142	46%	18.2
18B	6144	40	18.4	8	256	1024	135	43%	34.6
39B	8192	48	39.1	16	512	1536	138	44%	70.8
76B	10240	60	76.1	32	1024	1792	140	45%	143.8
145B	12288	80	145.6	64	1536	2304	148	47%	227.1
310B	16384	96	310.1	128	1920	2160	155	50%	297.4
530B	20480	105	529.6	280	2520	2520	163	52%	410.2
1T	25600	128	1008.0	512	3072	3072	163	52%	502.0

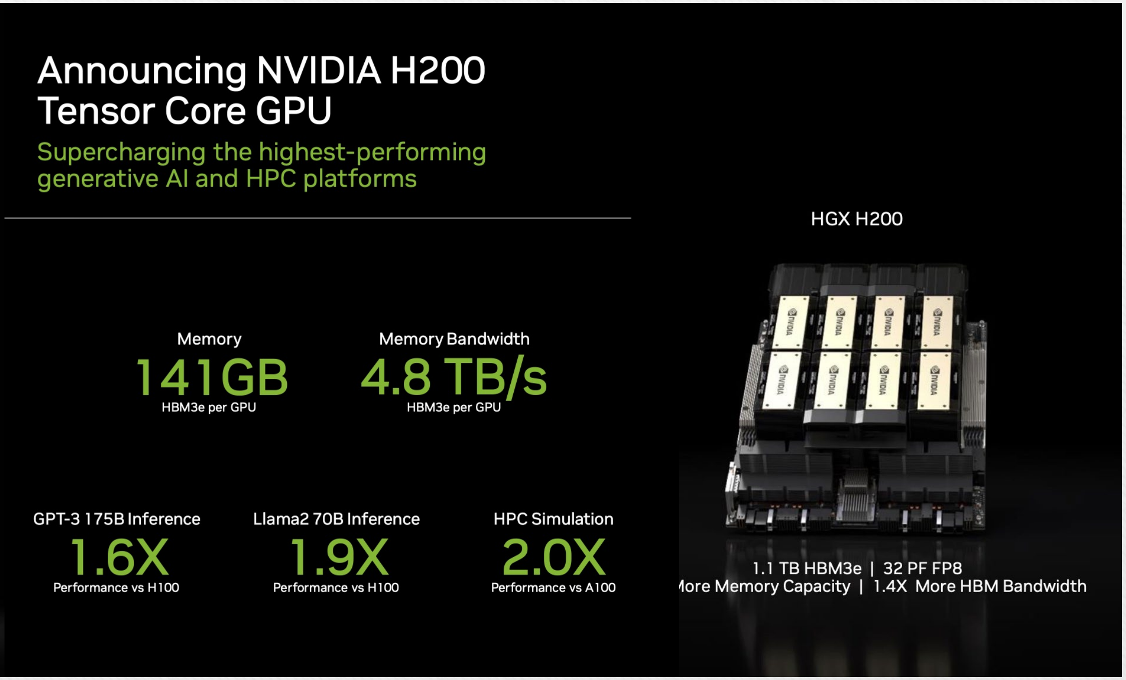
Source: Nvidia

SO WHO IS BUILDING THAT 100,000 GPU CLUSTER FOR XAI?

July 30, 2024 Timothy Prickett Morgan

ANN took off due to massive increase in computational capabilities, particularly in the use of **graphic processing unit (GPU)** for computation.





Announcing GB200 NVL72

Delivers New Unit of Compute

GB200 NVL72
- 36 GRACE CPUs
- 72 BLACKWELL GPUs
- Fully Connected NVLink Switch Rack

Training	720 PFLOPs
Inference	1,440 PFLOPs
NVL Model Size	27T params
Multi-Node Bandwidth	130 TB/s
Multi-Node All-Reduce	260 TB/s


NVIDIA
NVIDIA Corp
NASDAQ: NVDA
Overview
Financials
Compare
Market Summary > NVIDIA Corp
134.91 USD
+130.72 (3,119.81%) past 5 years
Closed: 10 Jul, 7:59 pm GMT-4 • Disclaimer
After hours 135.60 +0.69 (0.51%)
Follow
1D
5D
1M
6M
YTD
1Y
5Y
Max
150
100
50
0
2021
2022
2023
2024

Important ANN Architectures

Plus Important Specific Applications