10417/10617
Intermediate Deep Learning: Fall 2023

Yuanzhi Li / Russ Salakhutdinov

Machine Learning Department

Lecture 2: Deep learning basics: Perceptron

- In the previous lecture, we learned that a feed-forward neural network (FFN) is defined as:
- $F(x) = W_{L+1}\,\sigma(W_L\sigma(W_{L-1}\sigma \circ \cdots \circ \sigma(W_1 x + b_1) \ldots + b_{L-1}) + b_L) + b_{L+1}$
- This is called a L-hidden-layer FFN.
- One single unit: $n(x) = \sigma\left(w^T x + b\right)$ where w, x are d dimensional vectors, b is a scaler.

Perceptron

- One single unit: $n(x) = \sigma\left(w^T x + b\right)$ where w, x are d dimensional vectors, b is a scaler.
- When σ is the step function ($\sigma(x) = 0\ if\ x < 0, otherwise\ \sigma(x) = 1$), this unit is called the "perceptron".
- Perceptron was invented in 1943 by Warren McCulloch and Walter Pitts.
- Perceptron is known as the "simplest feed forward neural network".

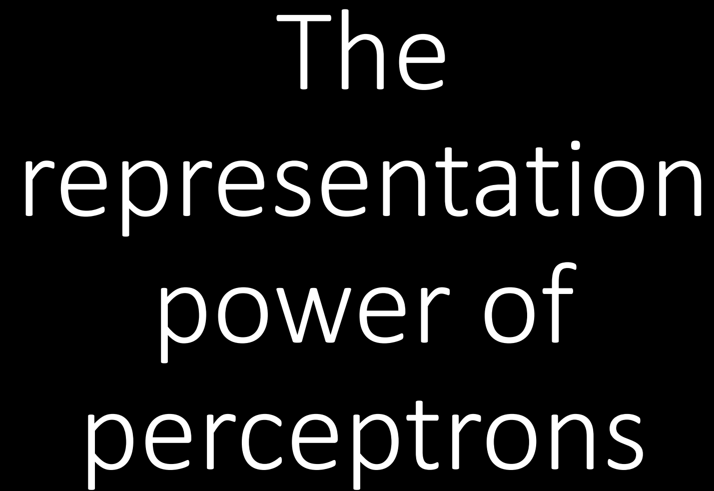
Using a perceptron as a learner.

- How do we train a perceptron learner?
- Perceptron is typically used in supervised learning, binary classification, where we have N training data $x^{(1)}, x^{(2)}, \dots, x^{(N)} \in R^d$, and labels $y^{(1)}, y^{(2)}, \dots, y^{(N)} \in \{0, 1\}$
- We typically train a perceptron learner using MSE loss (Mean Square Error loss), meaning we want to minimize
- $L(w, b) = \frac{1}{N} \sum_{i \in [N]} \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2$

The representation power of perceptron

What functions can perceptron approximate?

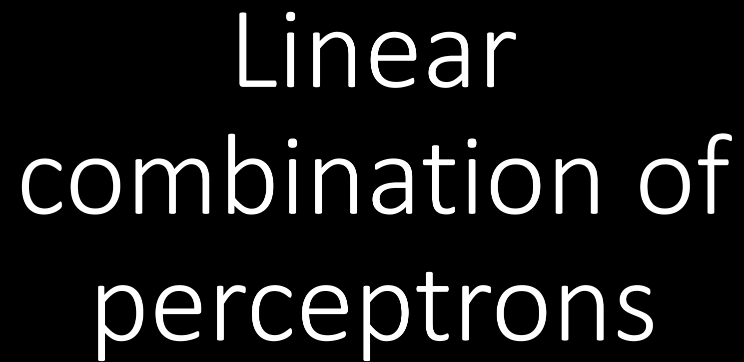
Perceptron can only approximate 0, 1 functions that are linearly separable.



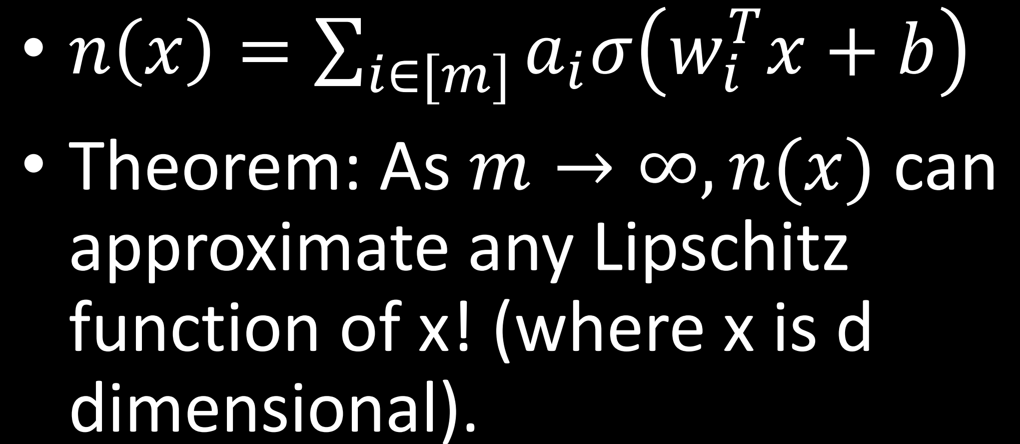
The representation power of perceptrons

- A single perceptron does not look good … It's almost a linear function.
- What about we use a linear combination of perceptrons?
- What about using $n(x) = \sum_{i \in [m]} a_i \sigma\left(w_i^T x + b\right)$?

Linear combination of perceptrons



- $n(x) = \sum_{i \in [m]} a_i \sigma\left(w_i^T x + b\right)$
- Theorem: As $m \to \infty, n(x)$ can approximate any Lipschitz function of x! (where x is d dimensional).



Linear combination of perceptrons

- Proof: For every Lipschitz function G in R (one dimensional), we can approximate G by writing it as:
- $G(x) = \sum_i I_{x_i}(x) \times G(x_i)$
- Where each $x_i = -C + \epsilon i$, for a sufficiently large C and sufficiently small $\epsilon > 0$, for $i \in \left[\frac{2C}{\epsilon}\right]$, and $I_{x_i}(x) = 1 \; if \; x \in \left[x_i - \frac{\epsilon}{2}, x_i + \frac{\epsilon}{2}\right), otherwise \; I_{x_i}(x) = 0.$

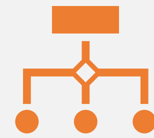
Linear combination of perceptrons

- $G(x) = \sum_i I_{x_i}(x) \times G(x_i)$
- For each $I_{x_i}(x)$, we can write it as:
- $I_{x_i}(x) = \sigma\left(x - (x_i - \frac{\epsilon}{2})\right) + \sigma\left(x_i + \frac{\epsilon}{2} - x\right) - 1$
- So $G(x) = \sum_i \left(\sigma\left(x - (x_i - \frac{\epsilon}{2})\right) + \sigma\left(x_i + \frac{\epsilon}{2} - x\right) - 1\right) \times G(x_i)$

Linear combination of perceptrons

- For high dimensional G, we can use the Fourier Transformation: There exists $F: R^d \rightarrow R$ such that:
- $G(x) = \int_{R^d} F(w) \cos(2\pi \times w^T x) \, dw$
- This means that we can look at each one dimensional function $F(w) \cos(2\pi \times w^T x)$ (along dimension w), and apply our previous approximation result.

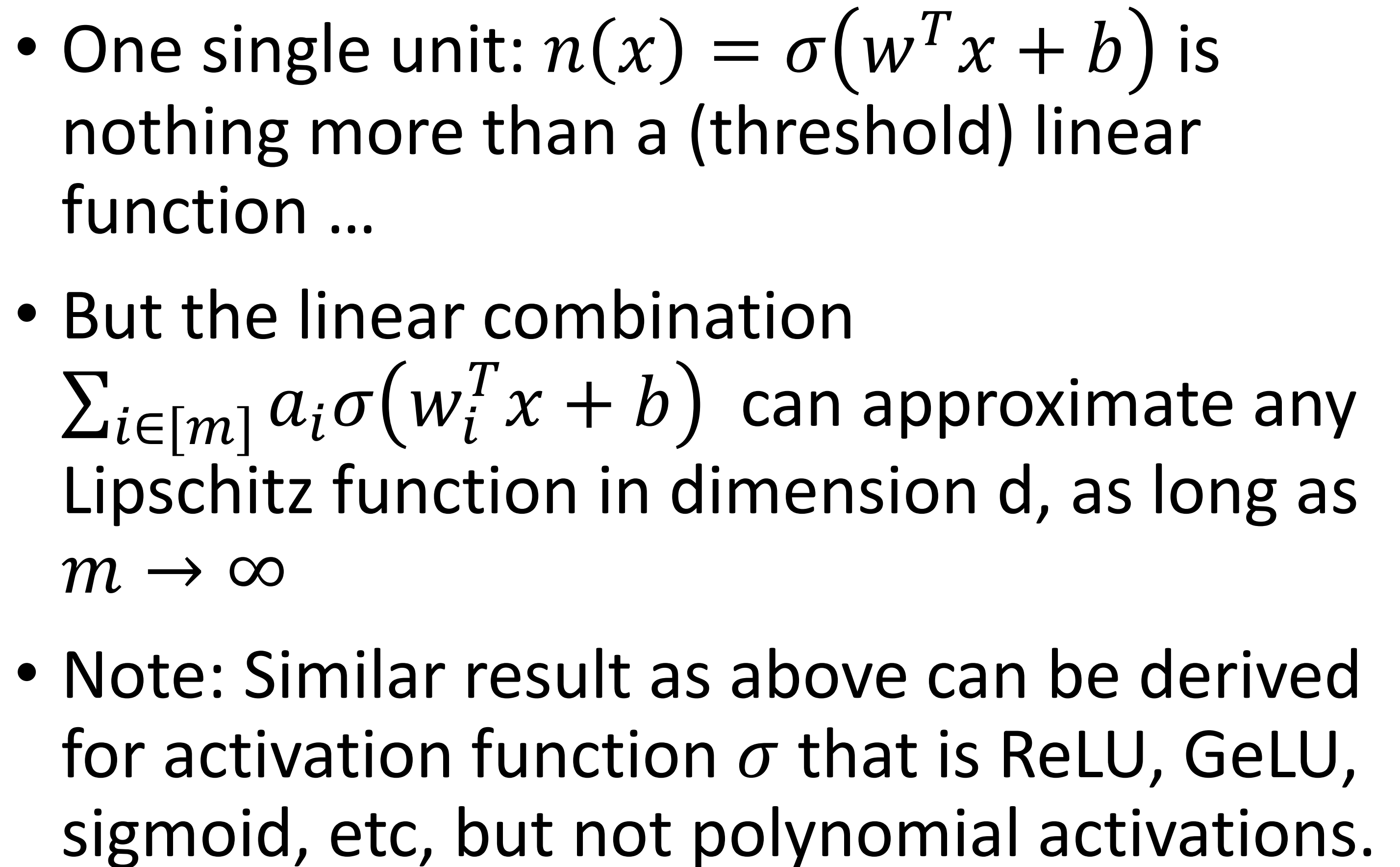
Linear combination of perceptrons



The same proof applies to non-linear activations such as sigmoid, ReLU.

Linear combinations of ReLU/sigmoid functions can approximate a step function in one dimension (R).

Linear combination of perceptrons



Depth versus Width

- Recall in the last lecture, we learnt that:
- $F(x) = W_{L+1}\,\sigma(W_L\sigma(W_{L-1}\sigma \circ \cdots \circ \sigma(W_1 x + b_1)\ \ldots + b_{L-1}) + b_L) + b_{L+1}$
- Can approximate any Lipschitz function, as long as σ is non linear and

$$\mathrm{L} \to \infty, d_{l+1} \geq 2d.$$

- In this lecture we learnt that even for L = 1, F(x) can approximate any Lipschitz function, as long as $d_2 \to \infty$, when the activation function σ that is ReLU, GeLU, sigmoid, etc.
- So no matter whether depth goes to infinity or width goes to infinity, neural networks are universal approximator.

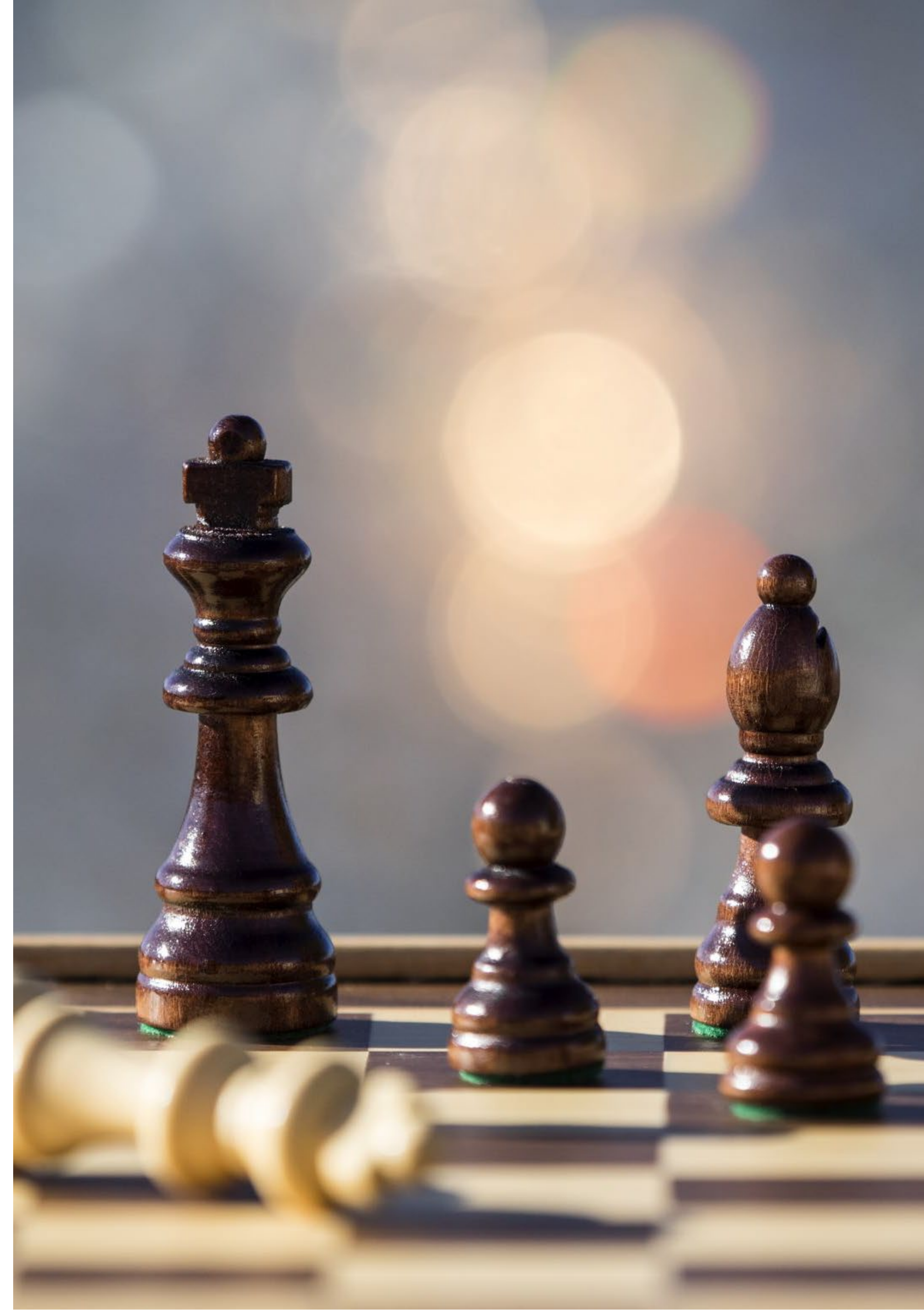
Depth versus Width

So no matter whether depth goes to infinity or width goes to infinity, neural networks are universal approximators.

Do we want a very wide shallow or not-so-wide deep network??

Depth versus Width: Depth win

- Main Theorem:
- For every width m, 1-hidden-layer FFN, there exists a depth-m FFN with width 2d that computes the same function.
- Theorem: There exists a 2-hidden-layer FFN of width 2d, such that
  - any 1-hidden-layer FFN (with any activation functions) that approximates it requires width $\geq \exp(d)$.
- Depth can approximate width efficiently, but width can not approximate depth efficiently.



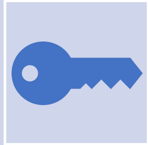
Intuition where depth matters

- Think about the function $f(x) = \left(x_1^2 + x_2^2 + \ \dots + x_d^2\right)^d$
- Using two hidden layer network, the first hidden layer can approximate the function $n(x) = x_1^2 + x_2^2 + \ \dots + x_d^2$ easily, with quadratic activation (can also be approximated efficiently with ReLU activation).
- The second layer can approximate the function $n(z) = z^d$ easily, with degree d activation (can also be approximated efficiently with ReLU activation).
- However, for a one-hidden layer network to approximate this function, we need to write $f(x)$ as a sum of simple functions. Expanding f as a sum has exponentially many terms.

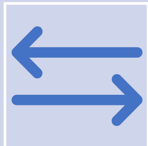
Multi-layer perceptron (MLP)

- $F(x) = W_{L+1}\,\sigma(W_L\sigma(W_{L-1}\sigma \circ \cdots \circ \sigma(W_1 x + b_1) \; \ldots + b_{L-1}) + b_L) + b_{L+1}$
- With L > 1, is also known as the Multi-layer perceptron (MLP).
  - MLP is used as one of the building block for transformer (even in GPT4).
- **Important**: Multi-layer perceptron does not require the activation to be the step function, any non-linear activation is fine.
- A perceptron (single layer perceptron) requires the activation function to be the step function.

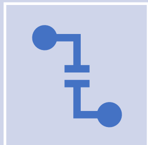
The choice of activation function



Another key question for MLPs is: How do we choose the activation functions?



We know that any non-linear activation function would make MLPs universal approximators.



The key difference in the activation functions lies in the “optimization difficulty” difference.

Optimizing MLPs

- Let's just consider a single unit $n(x) = \sigma\left(w^T x + b\right)$.
- And we want to minimize the MSE loss:
- $L(w, b) = \frac{1}{N} \sum_{i \in [N]} \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2$
- How do we fine the w, b that minimizes the MSE loss?

Gradient Descent

- A general method of minimizing a function is gradient descent.
- Starting with (arbitrary) $w^{(0)}, b^{(0)}$, at every iteration t, we update:
  - $(w^{(t+1)}, b^{(t+1)}) = (w^{(t)}, b^{(t)}) - \eta \nabla L\left(w^{(t)}, b^{(t)}\right)$
- Here, $\nabla L(w, b) = \nabla \frac{1}{N} \sum_{i \in [N]} \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2$
- $\nabla_{\mathrm{w}} \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2 = \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right) \times \sigma'\left(w^T x^{(i)} + b\right) \times x^{(i)}$
- $\nabla_{\mathrm{b}} \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2 = \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right) \times \sigma'\left(w^T x^{(i)} + b\right)$

Gradient Descent

- $\nabla_{\mathrm{w}}\left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2 = \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right) \times \sigma'\left(w^T x^{(i)} + b\right) \times x^{(i)}$
- $\nabla_{\mathrm{b}}\left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2 = \left(\sigma\left(w^T x^{(i)} + b\right) - y^{(i)}\right) \times \sigma'\left(w^T x^{(i)} + b\right)$
- When σ is the step function, $\sigma'(z) = 0$ for every z.
- This means that the weights of the neural network won't be updated by gradient descent …
- Key observation: We need the activation functions to have "non-zero gradient" and "uniform gradients" (not very large and very small at different places).

Gradient Descent

- Key observation: We need the activation functions to have "non-zero gradient" and "uniform gradients" (not very large and very small at different places).
- Step function has zero-gradient, not good.
- Sigmoid function: The gradient vanishes (almost zero) for z being relatively small or large ($|z| > 10$), not good.
- Quadratic activation: The gradient explodes when the norm of z is too large, not good.
- Good activation: ReLU activation, the gradient is either 0 ($z < 0$) or 1 ($z >= 0$).
- Better activation: Leaky-ReLU activation, the gradient is either –a ($z < 0$) or 1 ($z > 0$).
- Great activation: GeLU activation, $\sigma(z) = \frac{z}{2} \times \left(1 + \operatorname{erf}\left(\frac{z}{2^{0.5}}\right)\right), \sigma'(z) \in [-1, 1]$ and GeLU is smooth.

Different activation functions

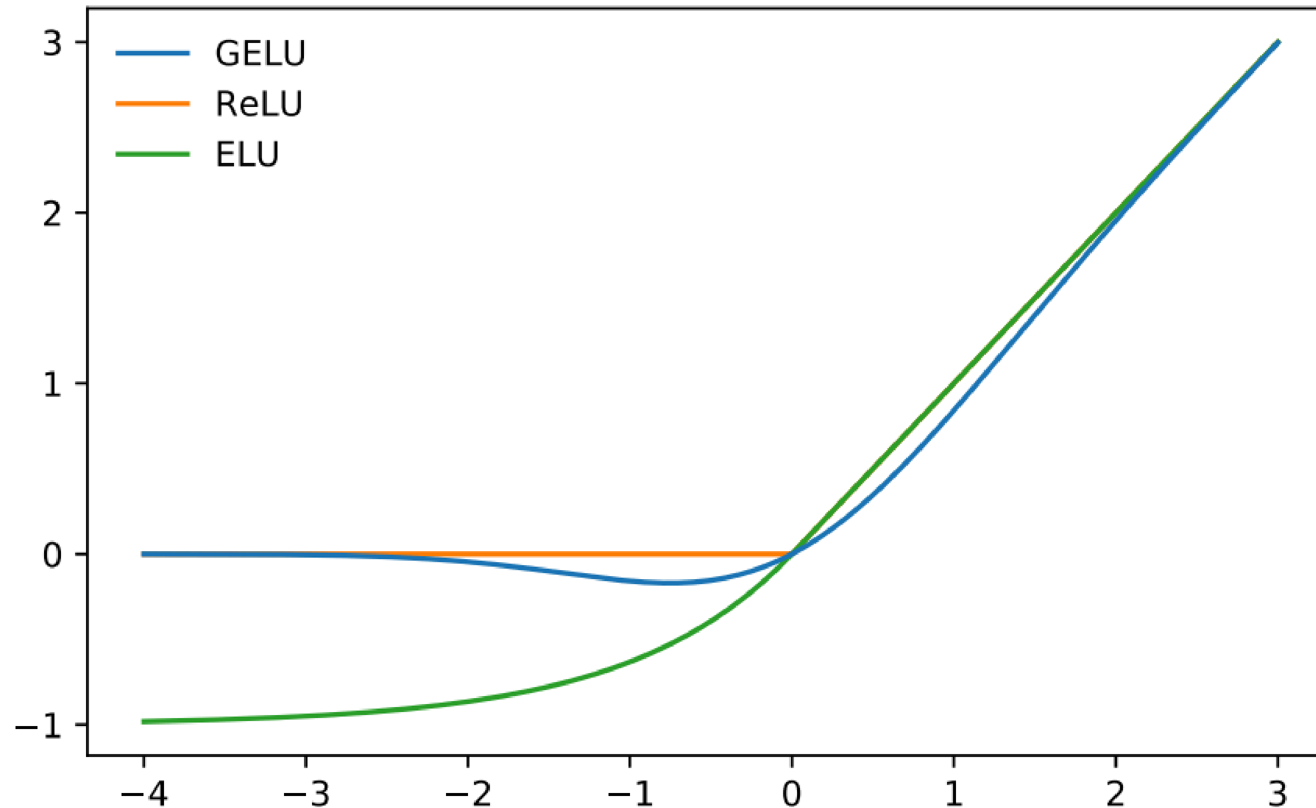


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

Optimization Landscape

But it just works in deep learning (with the help of over-parameterization).