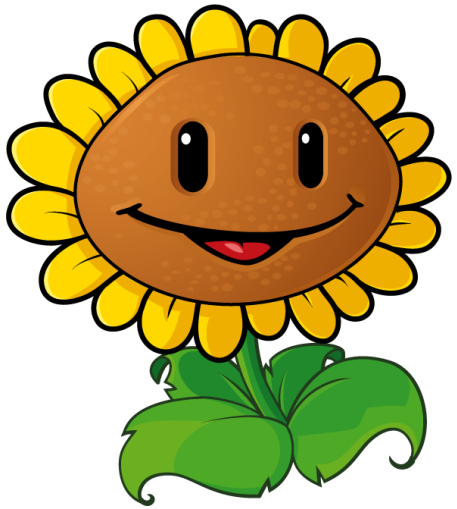
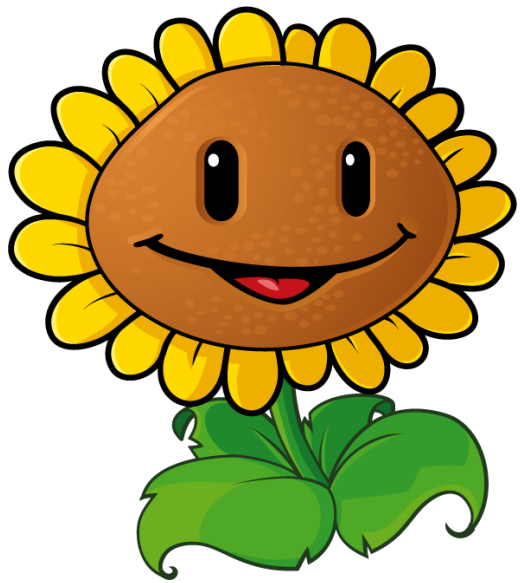
Neural Networks and Deep Learning

CSCI 410

Eric Ewing





Backpropagation

- We train our neural networks using Stochastic Gradient Descent (gradient descent with batches)
- Backpropagation is the algorithm for finding the gradient of our loss function (with respect to our learnable parameters

Deep Learning Libraries

Automatically compute gradients of neural network components and loss functions

How do they work?

Make programming neural networks easy!

Chain rule

If f and g are both differentiable and $F(x)$ is the composite function defined by $F(x) = f(g(x))$ then F is differentiable and F' is given by the product

$$F'(x) = f'(g(x))\, g'(x)$$

Differentiate outer function

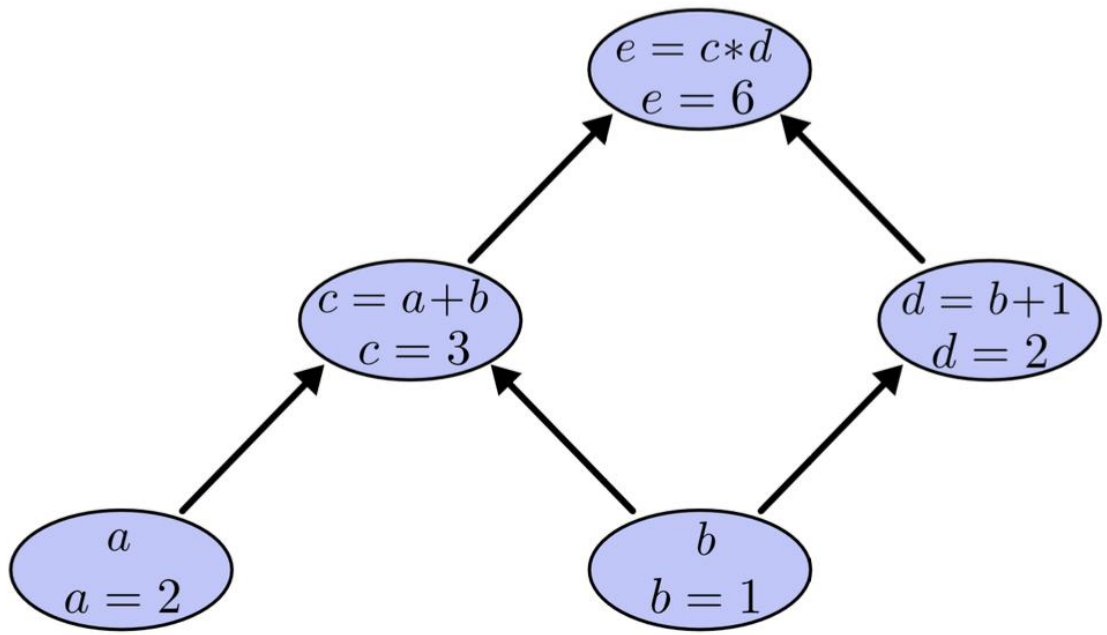
Differentiate inner function

Courtesy: https://www.onlinemathlearning.com/chain-rule.html

Computation Graph

$$e = (a + b) \cdot (b + 1)$$

For each node, track the operation and input variables

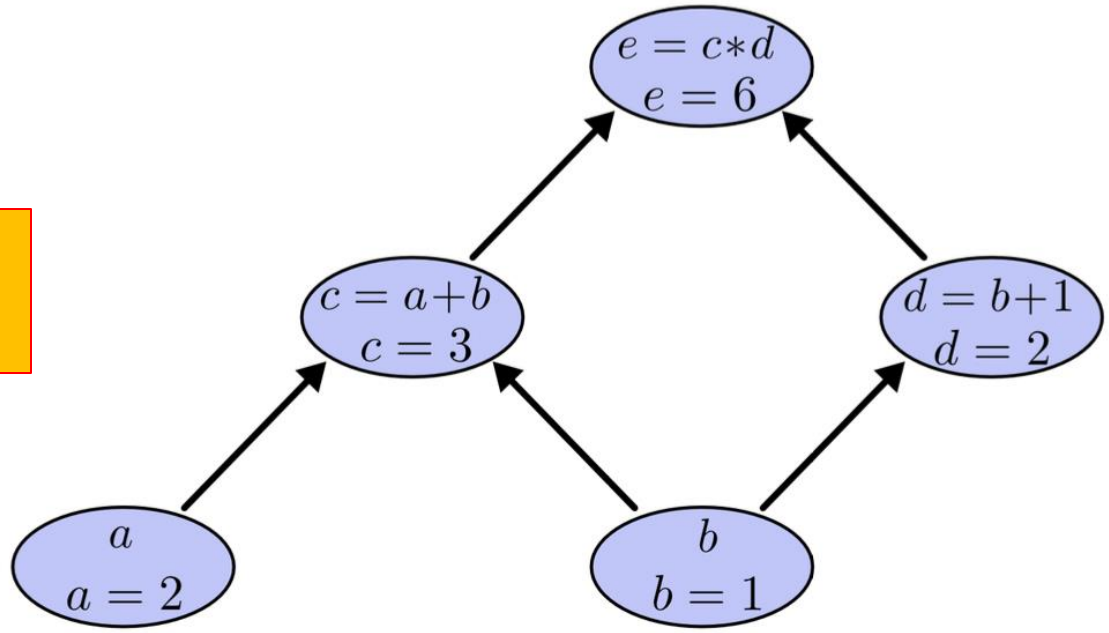


Computation Graph

$$e = (a + b) \cdot (b + 1)$$

For each node, track the operation and input variables

How to compute derivatives of *e* with respect to each input?
Want $\frac{de}{da}, \frac{de}{db}$

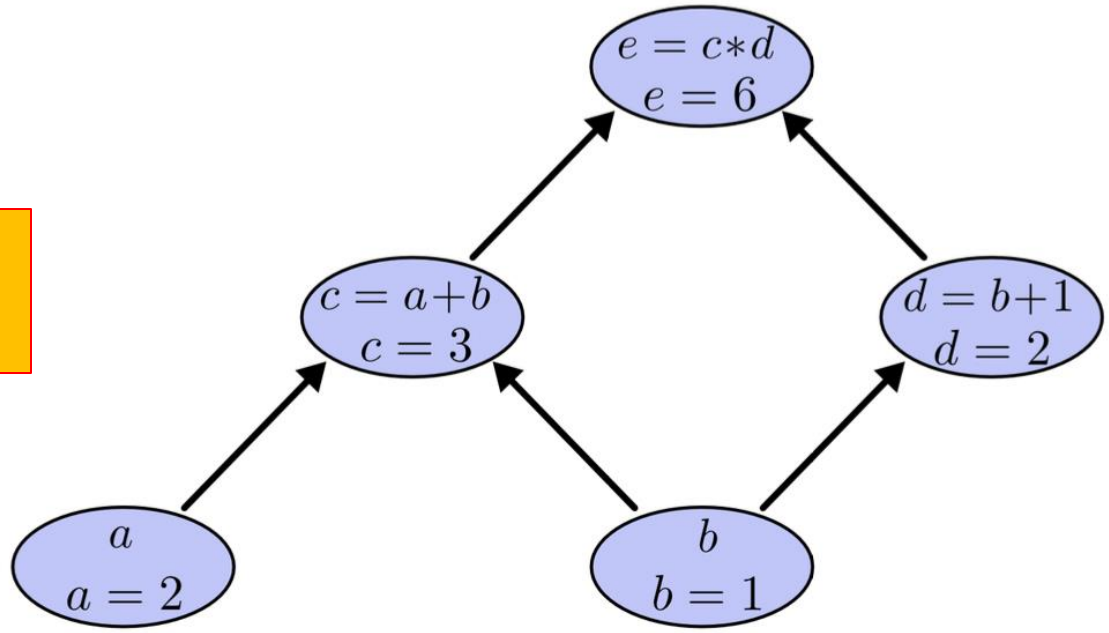


Computation Graph

$$e = (a + b) \cdot (b + 1)$$

For each node, track the operation and input variables

How to compute derivatives of *e* with respect to each input?
Want $\frac{de}{da}, \frac{de}{db}$



1. Run compute graph in "forward direction"
   Compute *e* by executing each operation
2. Run compute graph in "reverse direction"
   Compute derivatives at each node

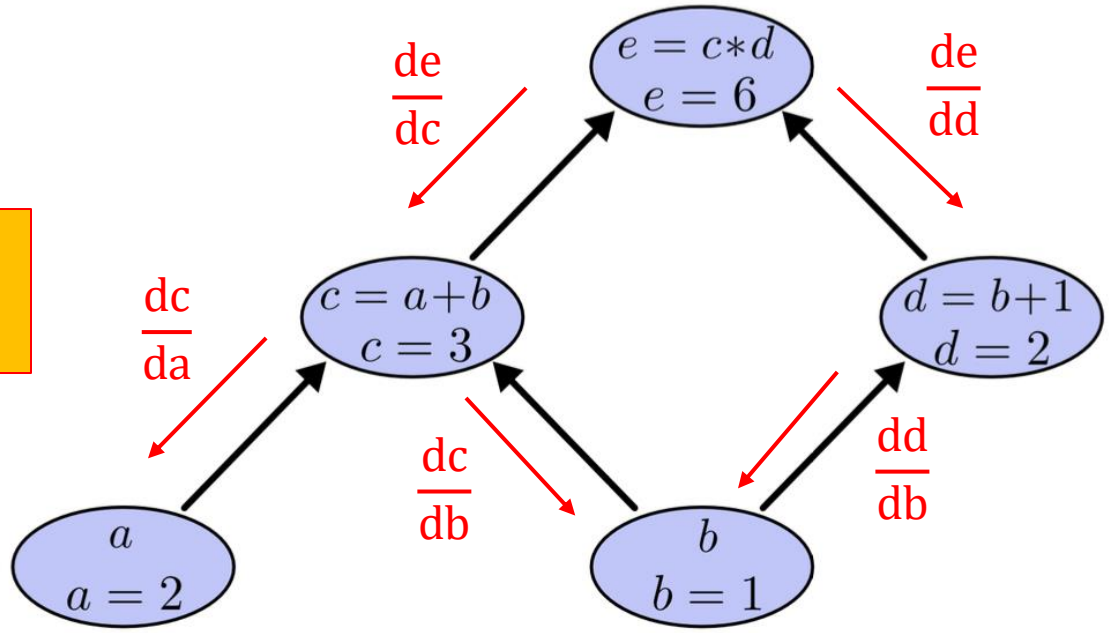
$$\frac{de}{da} = \frac{de}{dc}\frac{dc}{da} = d \qquad \frac{de}{db} = \frac{de}{dd}\frac{dd}{db} + \frac{de}{dc}\frac{dc}{db} = c + d$$

Computation Graph

$$e = (a + b) \cdot (b + 1)$$

For each node, track the operation and input variables

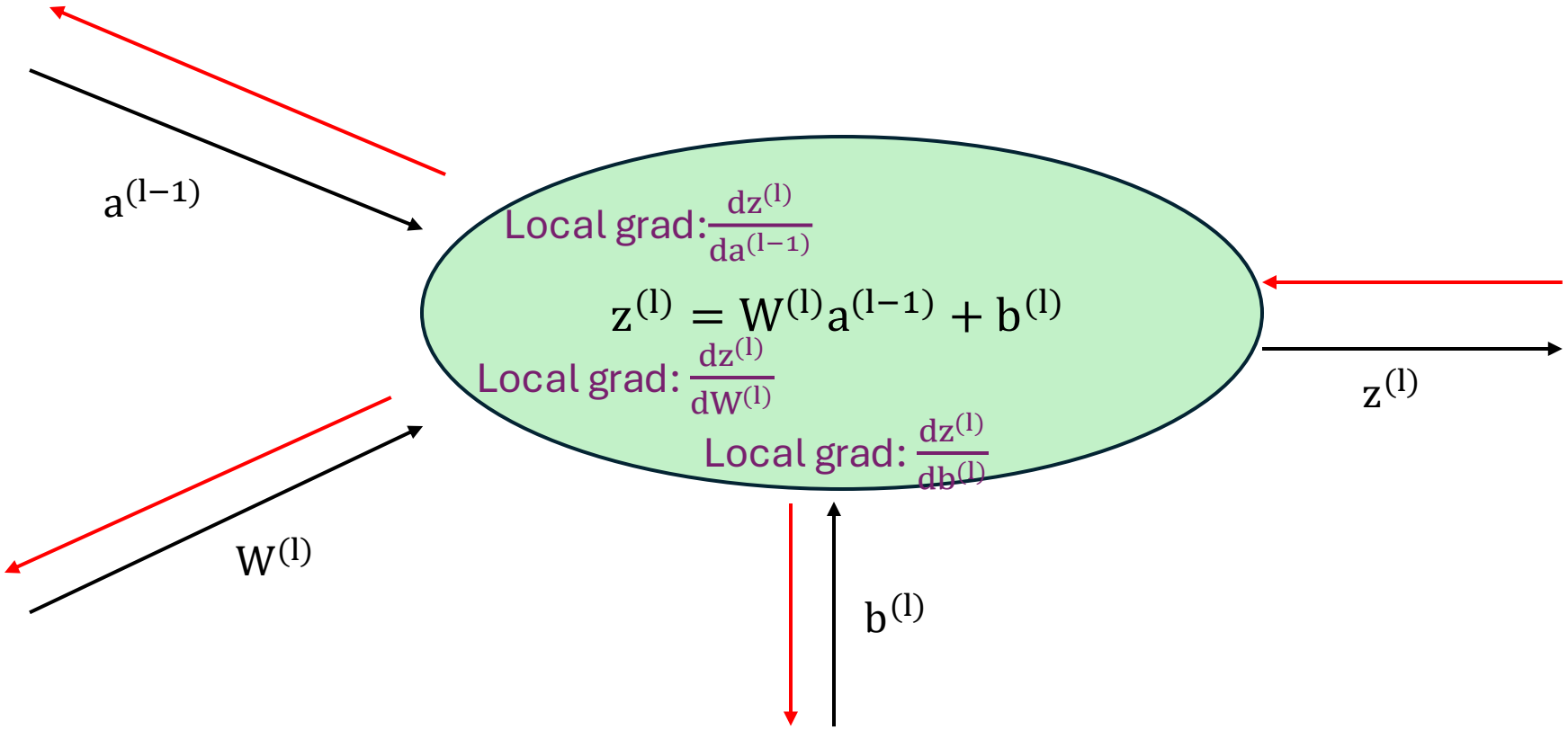
How to compute derivatives of *e* with respect to each input?
Want $\frac{de}{da}, \frac{de}{db}$



1. Run compute graph in "forward direction"
   Compute *e* by executing each operation
2. Run compute graph in "reverse direction"
   Compute derivatives at each node

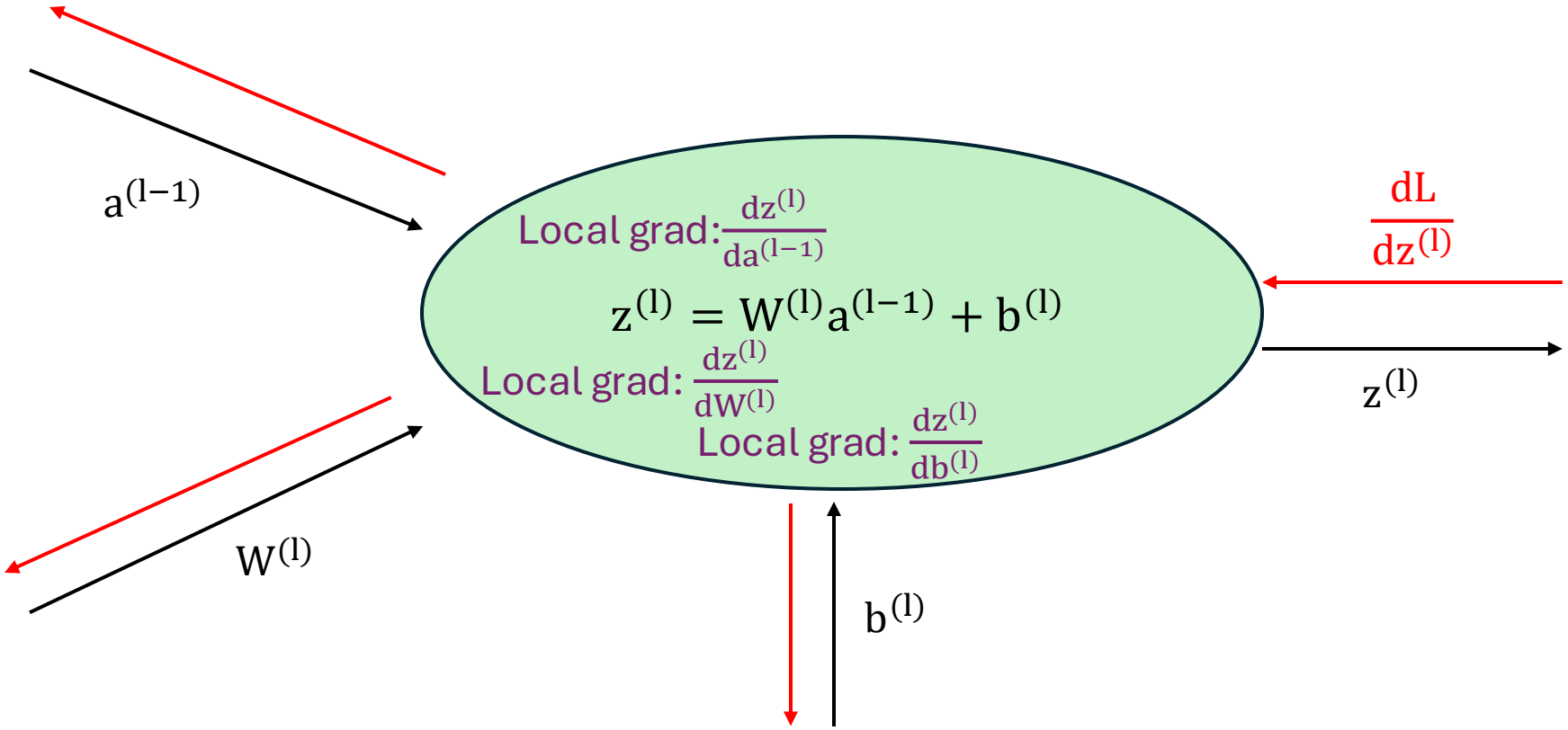
$$\frac{de}{da} = \frac{de}{dc}\frac{dc}{da} = d \qquad \frac{de}{db} = \frac{de}{dd}\frac{dd}{db} + \frac{de}{dc}\frac{dc}{db} = c + d$$

Compute Graph for a Single Layer



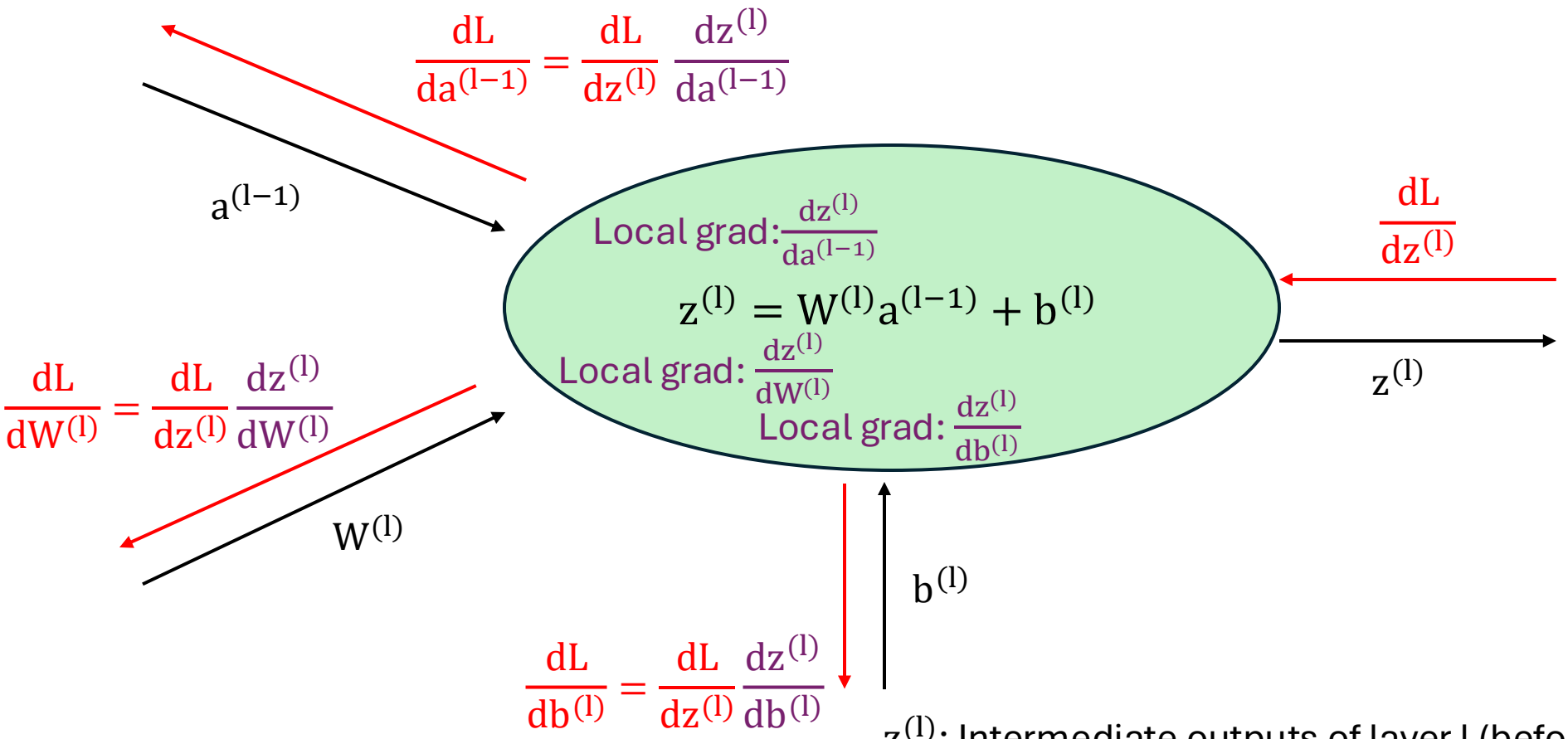
$z^{(l)}$: Intermediate outputs of layer l (before activation)
$a^{(l-1)}$: Value after activation function is applied at layer l-1
$W^{(l)}$: Weight matrix of layer l
$b^{(l)}$: Bias term at layer l

Compute Graph for a Single Layer



$z^{(l)}$: Intermediate outputs of layer l (before activation)
$a^{(l-1)}$: Value after activation function is applied at layer l-1
$W^{(l)}$: Weight matrix of layer l
$b^{(l)}$: Bias term at layer l

Compute Graph for a Single Layer



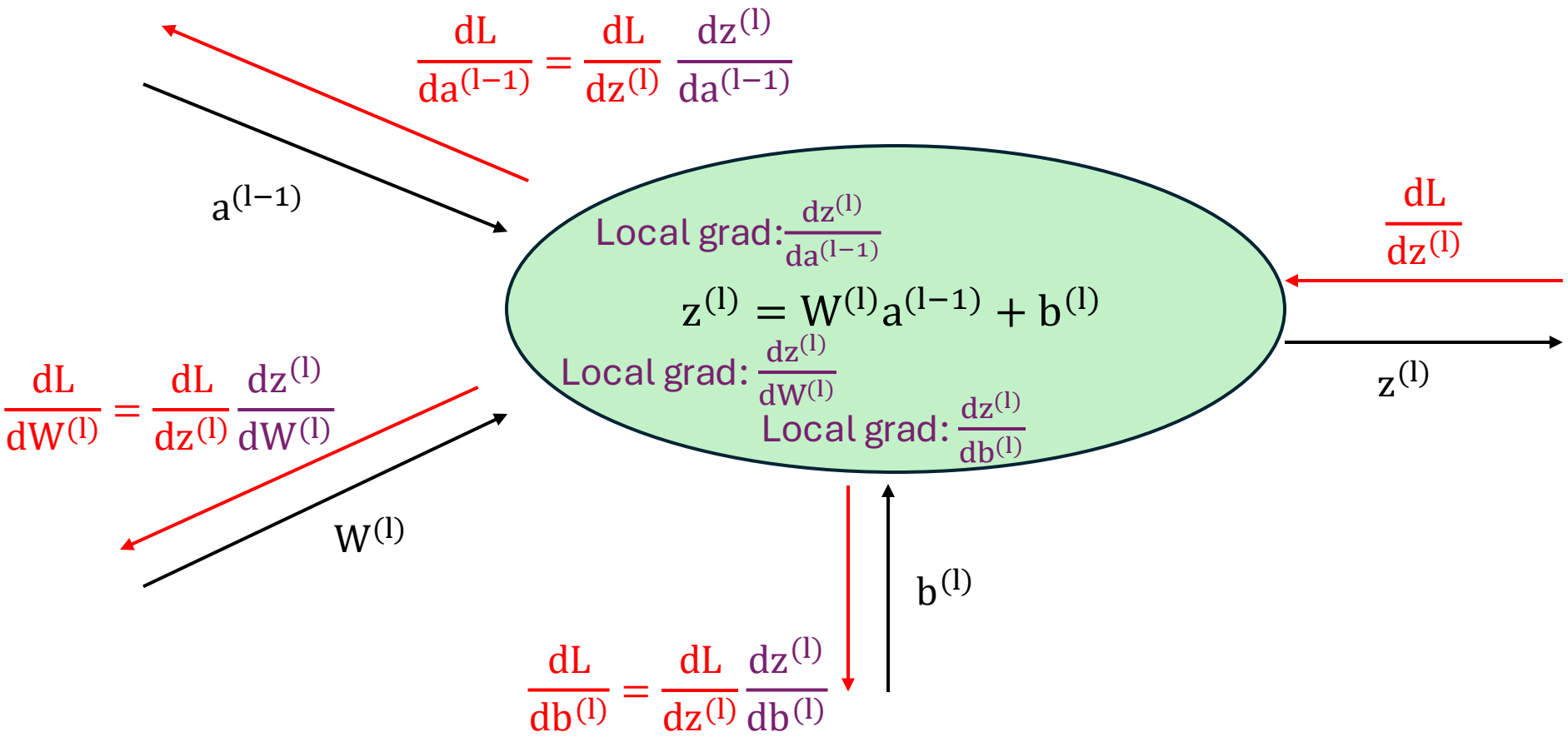
$z^{(l)}$: Intermediate outputs of layer l (before activation)

$a^{(l-1)}$: Value after activation function is applied at layer l-1

$W^{(l)}$: Weight matrix of layer l

$b^{(l)}$: Bias term at layer l

Compute Graph for a Single Layer



For each node:

Compose cumulative gradient $\frac{dL}{dz^{(l)}}$ with local gradients

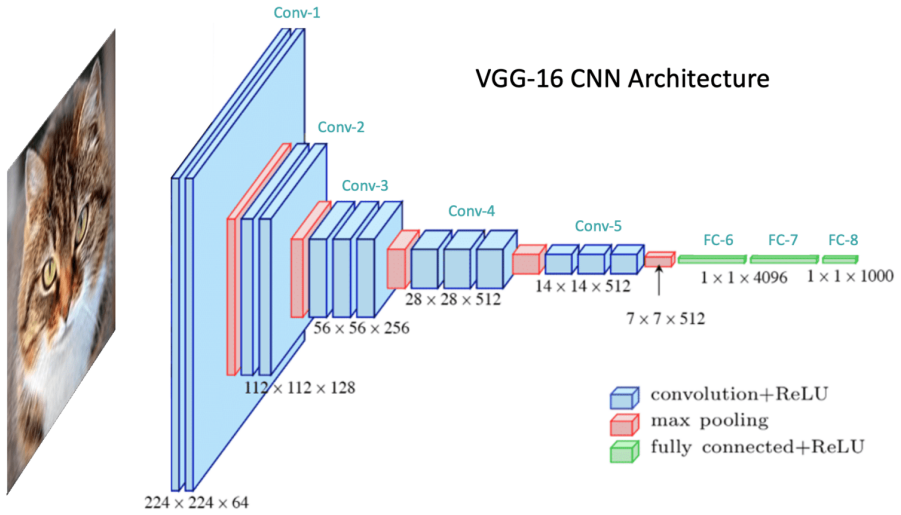
Pass new cumulative gradient to parent nodes, repeat

Deep Learning

MLPs use a single type of layer, called a dense, linear, or fully-connected layer

Neural Networks can use other type of function for a layer, so long as they are differentiable

Different layers will lead to different *Architectures*



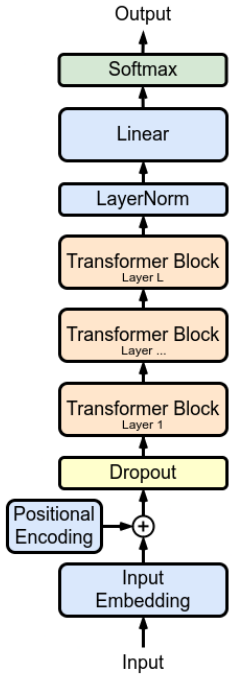
New Layer type: Convolution
Architecture: Convolutional Neural Network

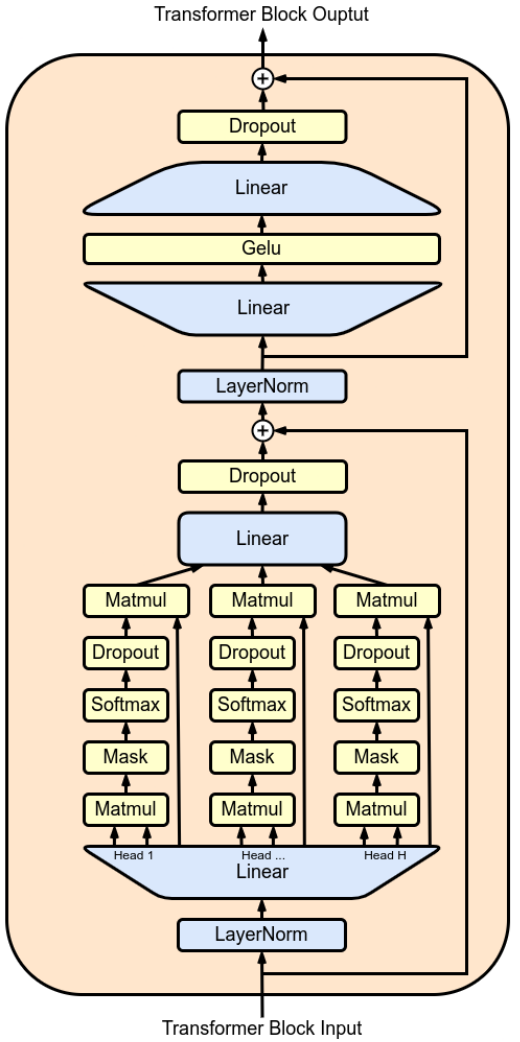
Deep Learning

MLPs use a single type of layer, called a dense, linear, or fully-connected layer

Neural Networks can use other type of function for a layer, so long as they are differentiable

Different layers will lead to different *Architectures*





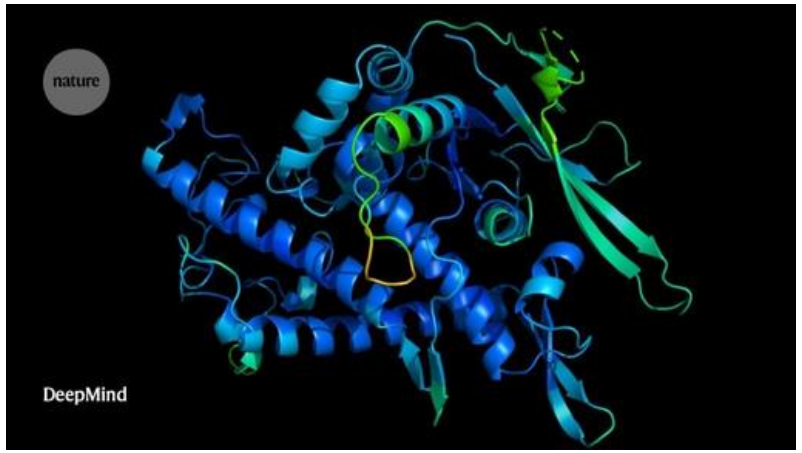
New Layer: *Self-Attention*
Architecture: Transformer

Architectures

- We construct different architectures to handle different types of data and tasks
- Fully connected layers make sense for data with features (e.g., the iris dataset)
- But fully connected layers are not particularly well suited for images, or sequential data, or many other types of data we encounter in the real world

Neural Networks

Why do they work so well?



Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.
  - For any function, there exists a neural network of fixed depth that can approximate within some ϵ of error.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.
  - For any function, there exists a neural network of fixed depth that can approximate within some ϵ of error.
  - If $\epsilon = 0$, i.e., we want a perfect approximation, we may need an infinitely wide network.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.
  - For any function, there exists a neural network of fixed depth that can approximate within some ϵ of error.
  - If $\epsilon = 0$, i.e., we want a perfect approximation, we may need an infinitely wide network.
  - This is an **existence** theorem, meaning it tells you that a neural network exists with these properties. It does not tell you how to find the weights of this network.

Intuition

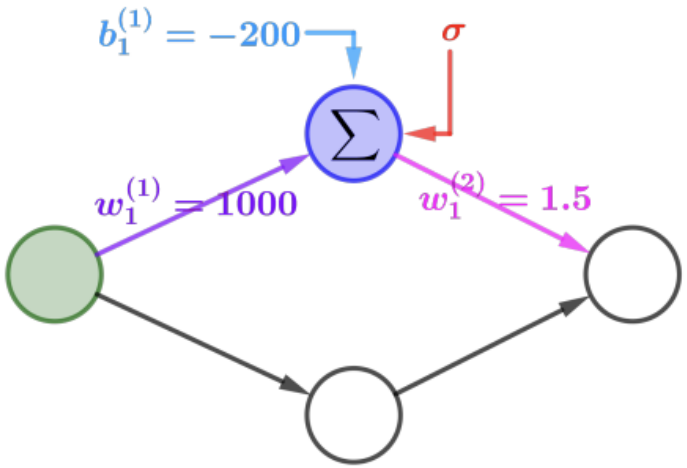


Figure 13: Addition of w_1^2

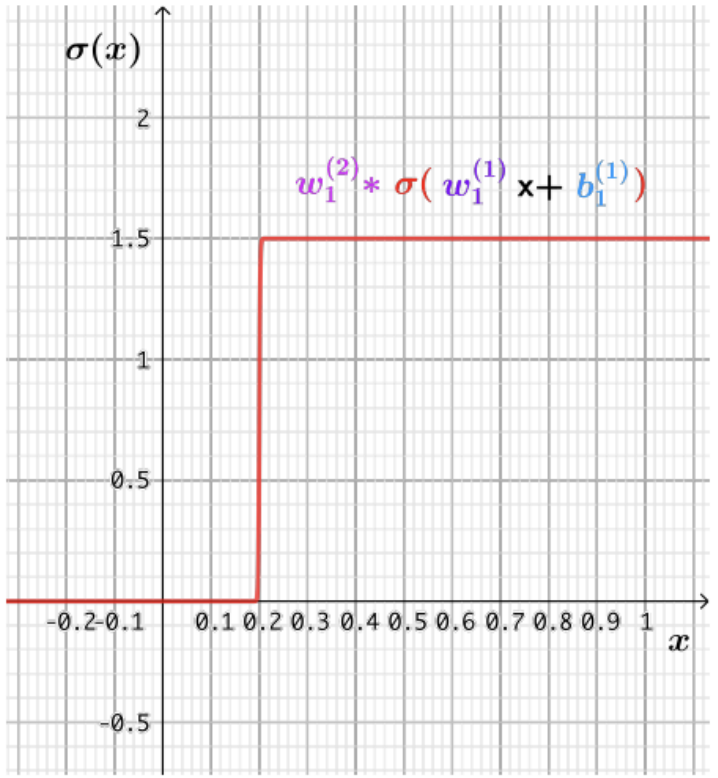


Figure 14: Scaled step-function

With large weights, sigmoid activation functions look like step functions

Source: Cybenko's Theorem and the capability of a neural network as function approximator

Approximating functions with step functions

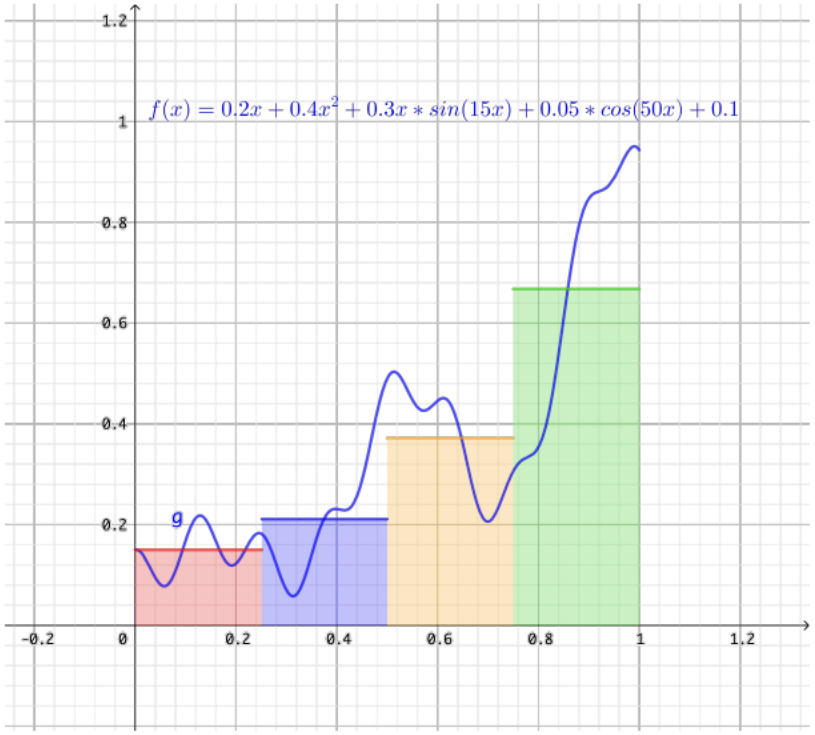


Figure 18: Approximation (N=4)

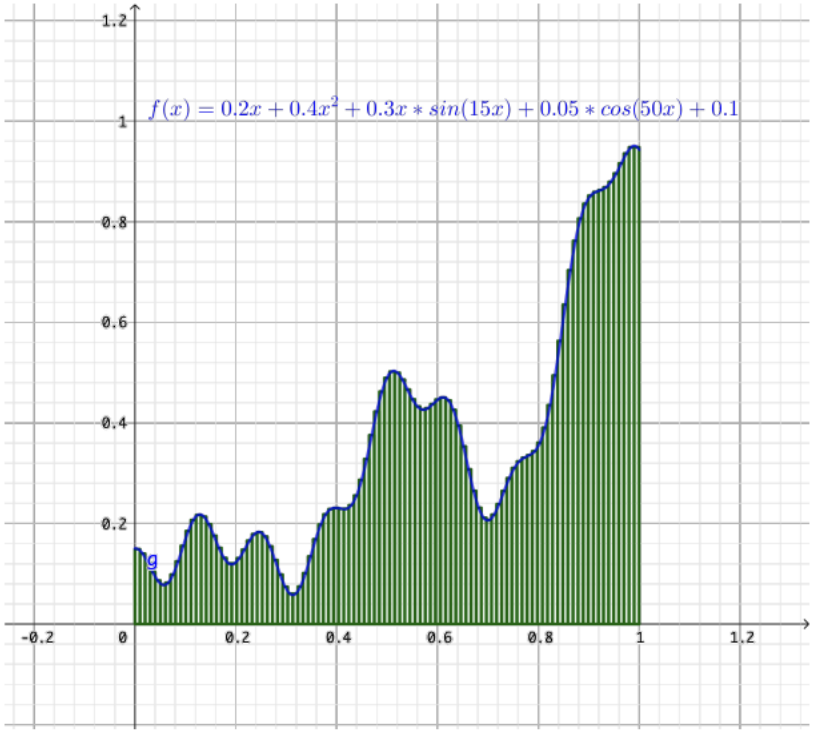


Figure 19: Approximation (N=100)

We can use step functions to approximate arbitrary functions

Neural Networks

- So that’s it, right? That’s why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks

- So that’s it, right? That’s why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Piecewise polynomials are universal function approximators (think Taylor expansions)

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Piecewise polynomials are universal function approximators (think Taylor expansions)

Wavelets (i.e., small pieces of sine and cosine) are universal function approximators

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Piecewise polynomials are universal function approximators (think Taylor expansions)

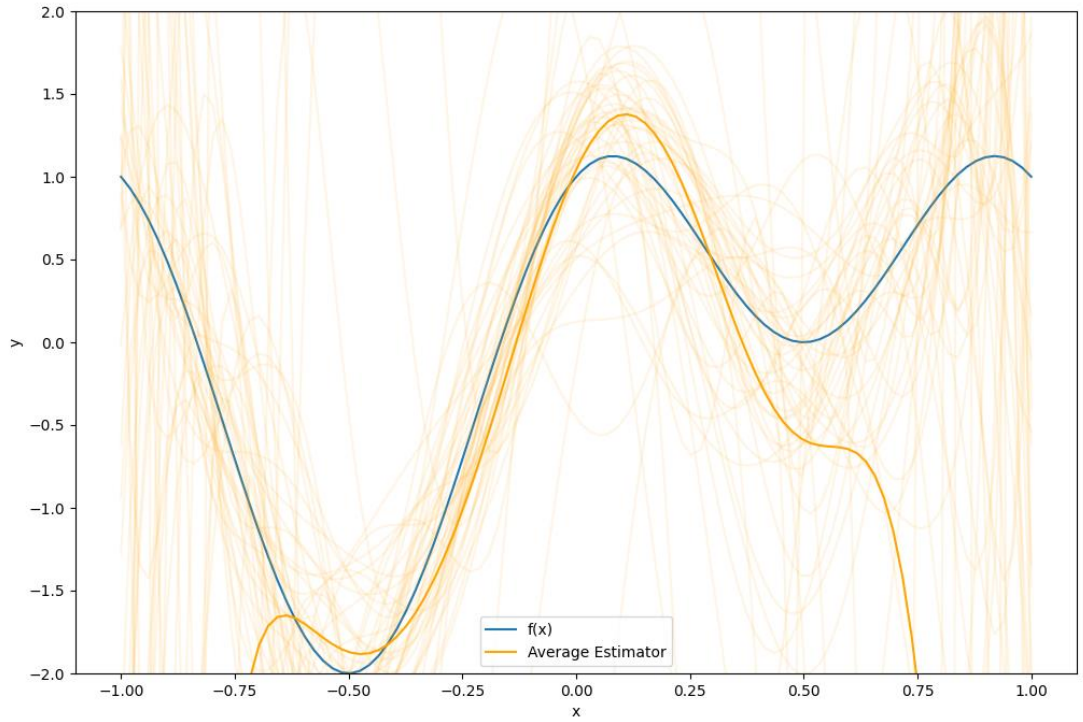
Wavelets (i.e., small pieces of sine and cosine) are universal function approximators

This theorem explains why neural networks are good at fitting the **training** dataset, not why they perform well on the **test** dataset.

Generalization

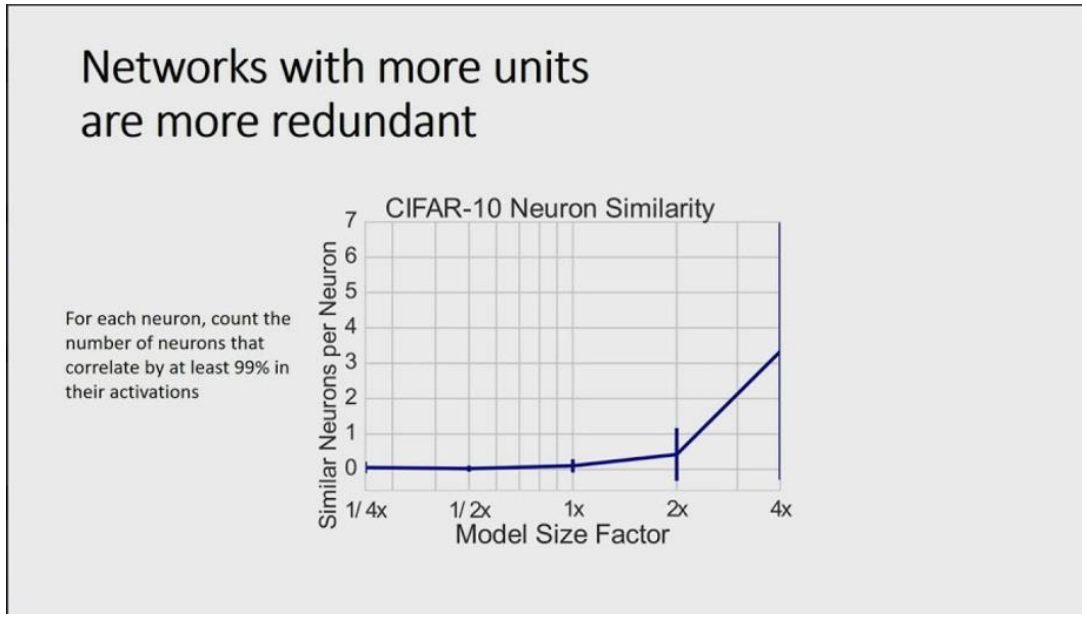
Neural Networks tend to *generalize* to unseen data better than other classes of models

Polynomial Regression: As the number of parameters approached the number of data points, our test error increased



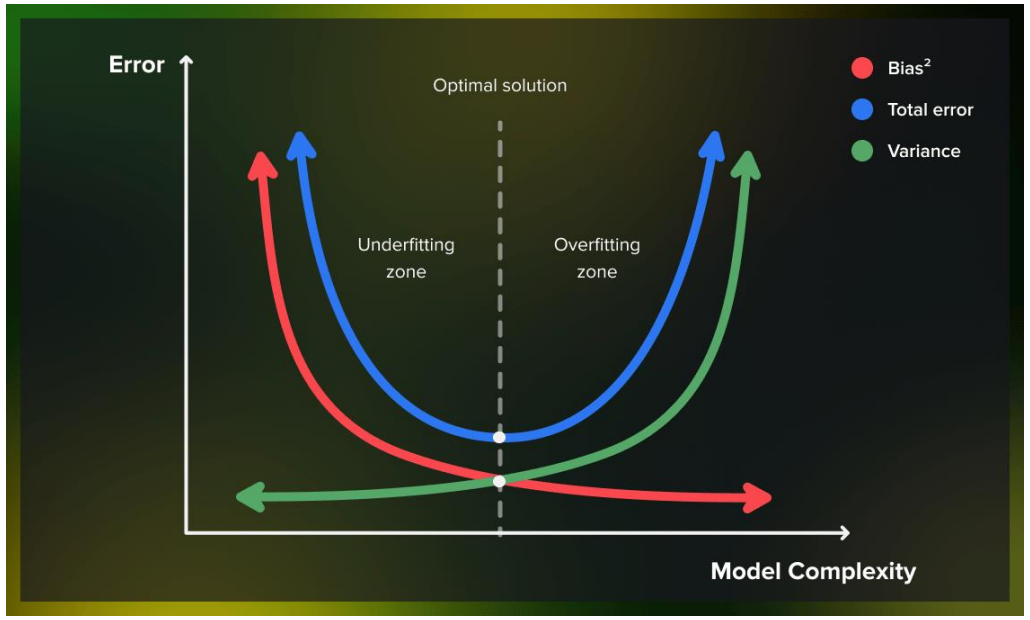
Overparameterization

- Neural Networks tend to have many more parameters than we have training examples
- The additional parameters add *redundancy*
- Larger neural networks tend to have many neurons that “do the same thing”
- The *effective dimension* of a neural network may be much smaller than the number of actual parameters



Overparameterization

Bias-Variance Tradeoff
(Traditional Understanding)



https://serokell.io/blog/bias-variance-tradeoff

A Farewell to the Bias-Variance Tradeoff?
An Overview of the Theory of Overparameterized Machine Learning

Yehuda Dar* **Vidya Muthukumar†** **Richard G. Baraniuk‡**

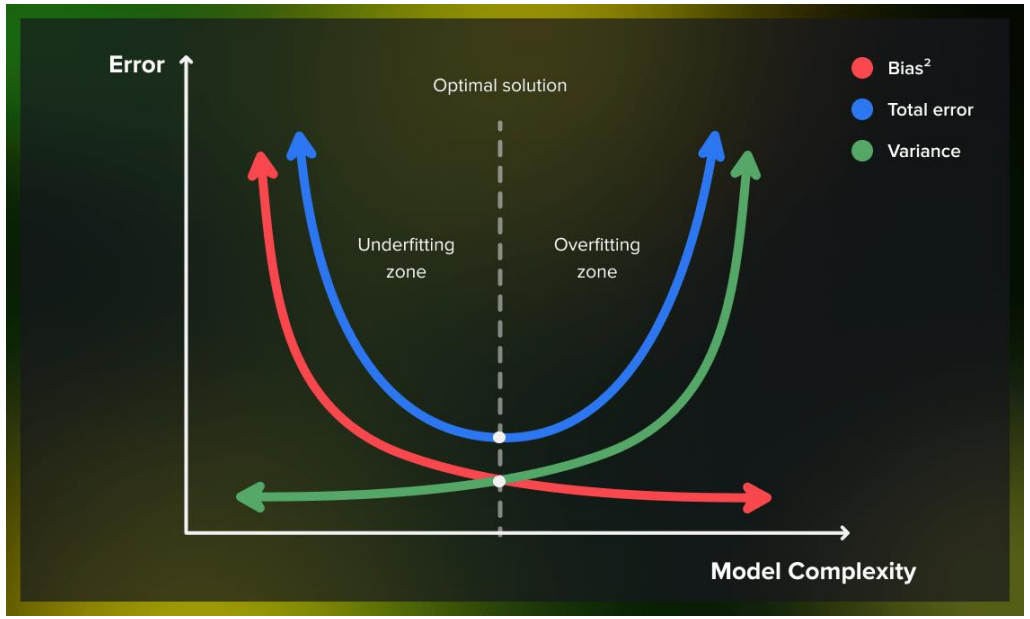
Abstract

The rapid recent progress in machine learning (ML) has raised a number of scientific questions that challenge the longstanding dogma of the field. One of the most important riddles is the good empirical generalization of *overparameterized* models. Overparameterized models are excessively complex with respect to the size of the training dataset, which results in them perfectly fitting (i.e., *interpolating*) the training data, which is usually noisy. Such interpolation of noisy data is traditionally associated with detrimental overfitting, and yet a wide range of interpolating models – from simple linear models to deep neural networks – have recently been observed to generalize extremely well on fresh test data. Indeed, the recently discovered *double descent* phenomenon has revealed that highly overparameterized models often improve over the best underparameterized model in test performance.

Understanding learning in this overparameterized regime requires new theory and foundational empirical studies, even for the simplest case of the linear model. The underpinnings of this understanding have been laid in very recent analyses of overparameterized linear regression and related statistical learning tasks, which resulted in precise analytic characterizations of double descent. This paper provides a succinct overview of this emerging *theory of overparameterized ML* (henceforth abbreviated as TOPML) that explains these recent findings through a statistical signal processing perspective. We emphasize the unique aspects that define the TOPML research area as a subfield of modern ML theory and outline interesting open questions that remain.

Overparameterization

Bias-Variance Tradeoff
(Traditional Understanding)



(Traditional View) If you are overfitting, reduce model complexity (smaller width/fewer layers). If underfitting, add more model complexity.

https://serokell.io/blog/bias-variance-tradeoff

A Farewell to the Bias-Variance Tradeoff?
An Overview of the Theory of Overparameterized Machine Learning

Yehuda Dar* **Vidya Muthukumar†** **Richard G. Baraniuk‡**

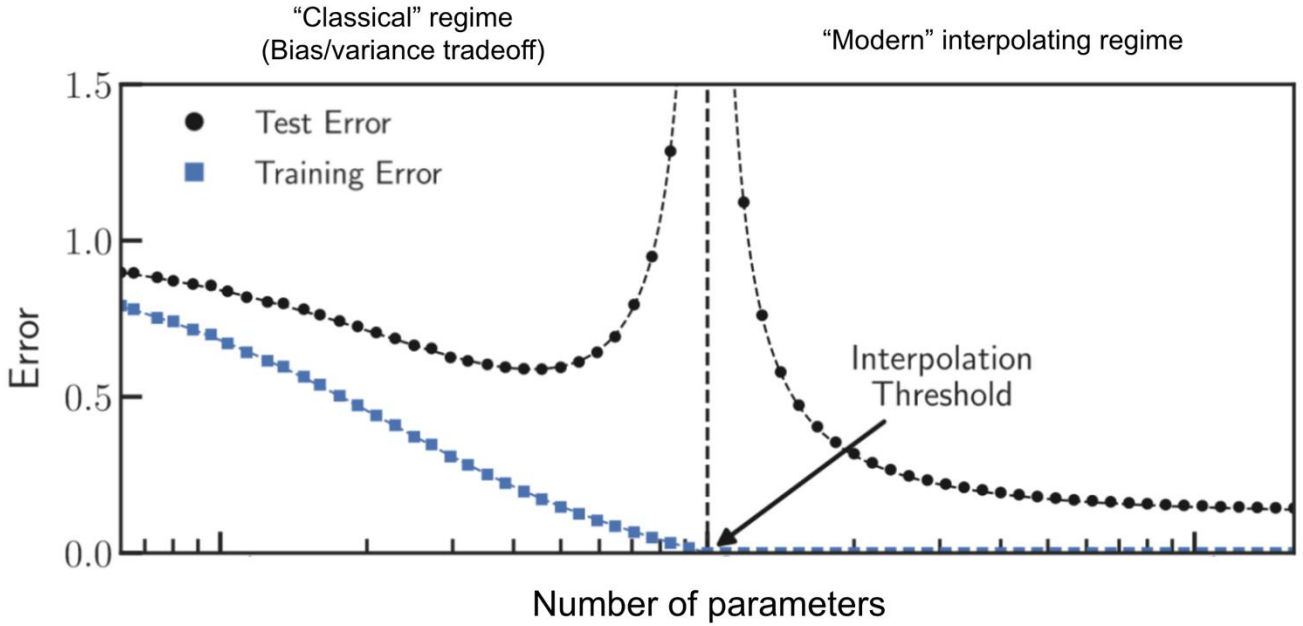
Abstract

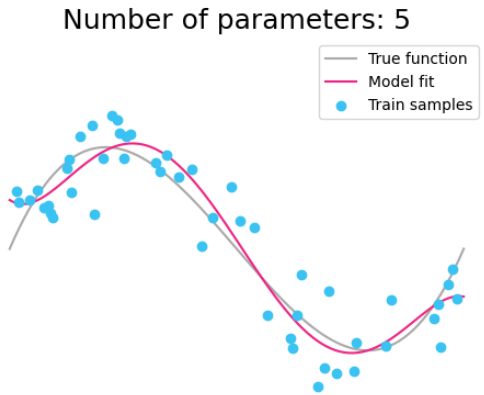
The rapid recent progress in machine learning (ML) has raised a number of scientific questions that challenge the longstanding dogma of the field. One of the most important riddles is the good empirical generalization of *overparameterized* models. Overparameterized models are excessively complex with respect to the size of the training dataset, which results in them perfectly fitting (i.e., *interpolating*) the training data, which is usually noisy. Such interpolation of noisy data is traditionally associated with detrimental overfitting, and yet a wide range of interpolating models – from simple linear models to deep neural networks – have recently been observed to generalize extremely well on fresh test data. Indeed, the recently discovered *double descent* phenomenon has revealed that highly overparameterized models often improve over the best underparameterized model in test performance.

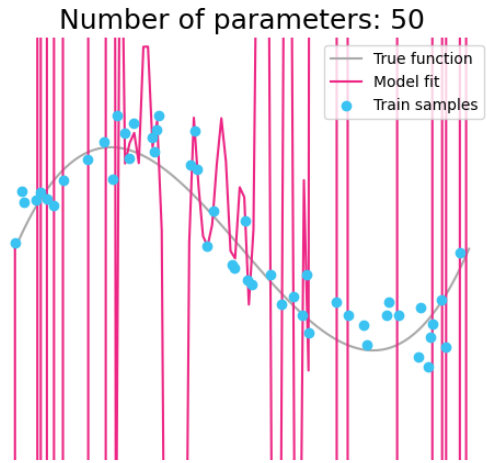
Understanding learning in this overparameterized regime requires new theory and foundational empirical studies, even for the simplest case of the linear model. The underpinnings of this understanding have been laid in very recent analyses of overparameterized linear regression and related statistical learning tasks, which resulted in precise analytic characterizations of double descent. This paper provides a succinct overview of this emerging *theory of overparameterized ML* (henceforth abbreviated as TOPML) that explains these recent findings through a statistical signal processing perspective. We emphasize the unique aspects that define the TOPML research area as a subfield of modern ML theory and outline interesting open questions that remain.

Overparameterization

Double Descent: a phenomenon observed with models with large numbers of parameters where after an initial spike in test error, increasing the number of parameters actually reduces overfitting







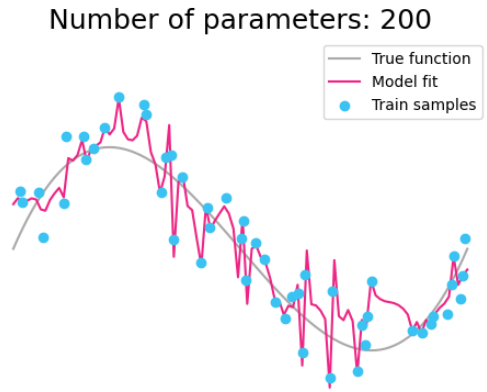
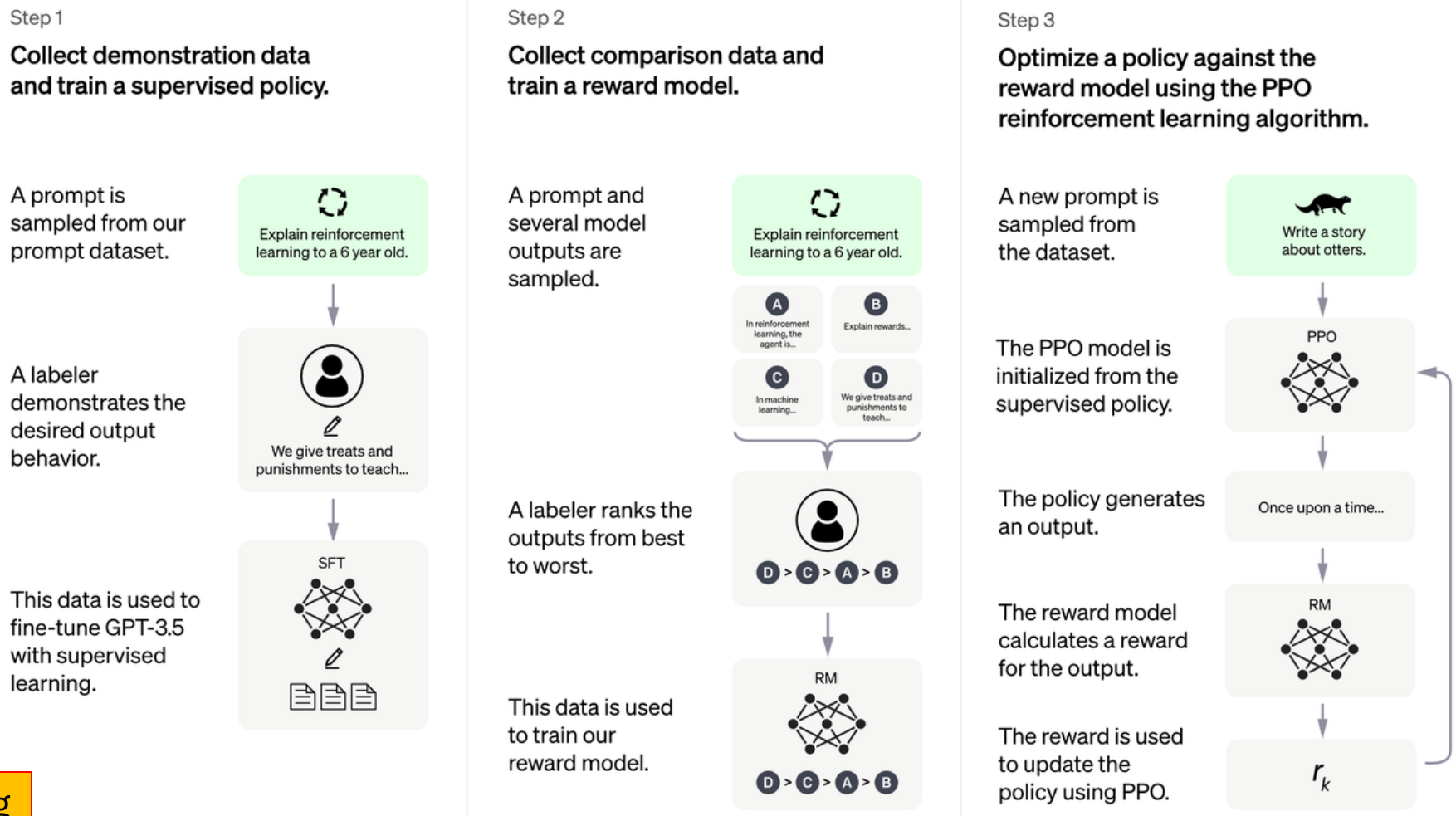


Image Source: Chris Said, https://chris-said.io/2023/04/21/double-descent-in-human-learning/

How to train GPT 3

Step 1: train a neural network to do language modeling using supervised learning (i.e., predict next token/word)

Step 2/3: Use Reinforcement learning to “align” model to human preferences.



Up next: Reinforcement Learning