

CSCI 1470

Eric Ewing

Friday,
2/7/25

Deep Learning

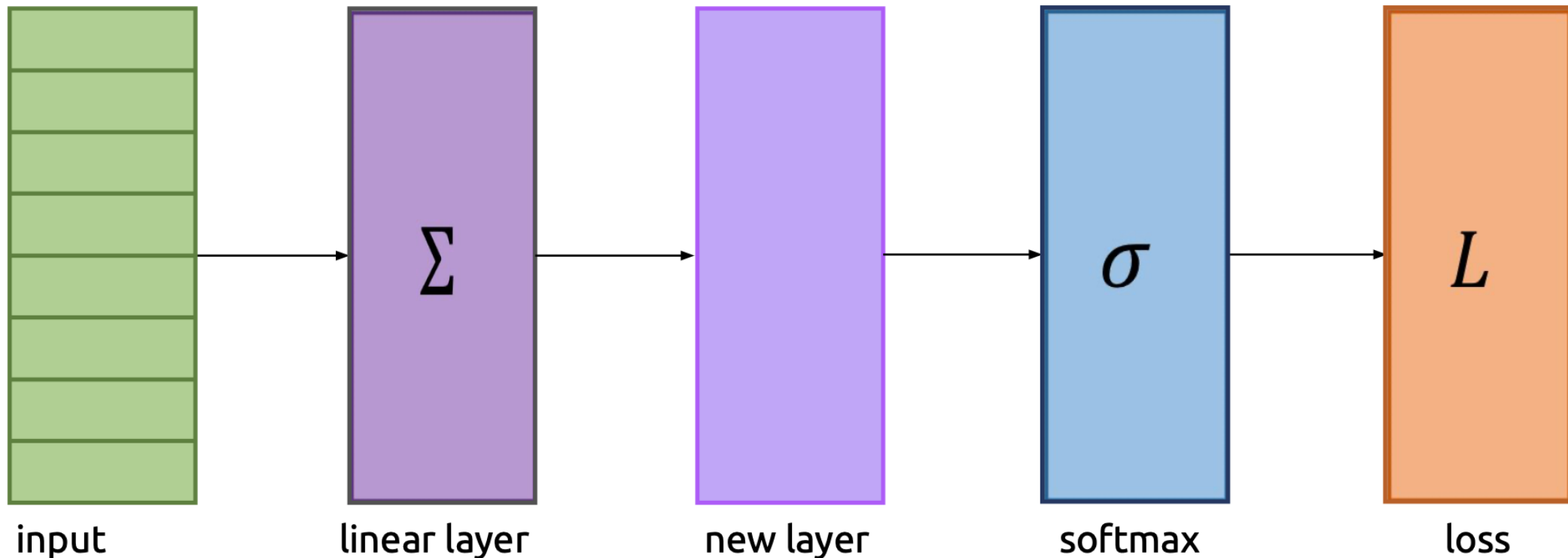
Day 8: Autograd and Hyperparameters

Generalizing Backpropagation

- What if we want to add another layer to our model?
- Calculating derivatives by hand *again* is a lot of work 😞

Can the computers do this for us?

Yes 😊



Computer-based Derivatives

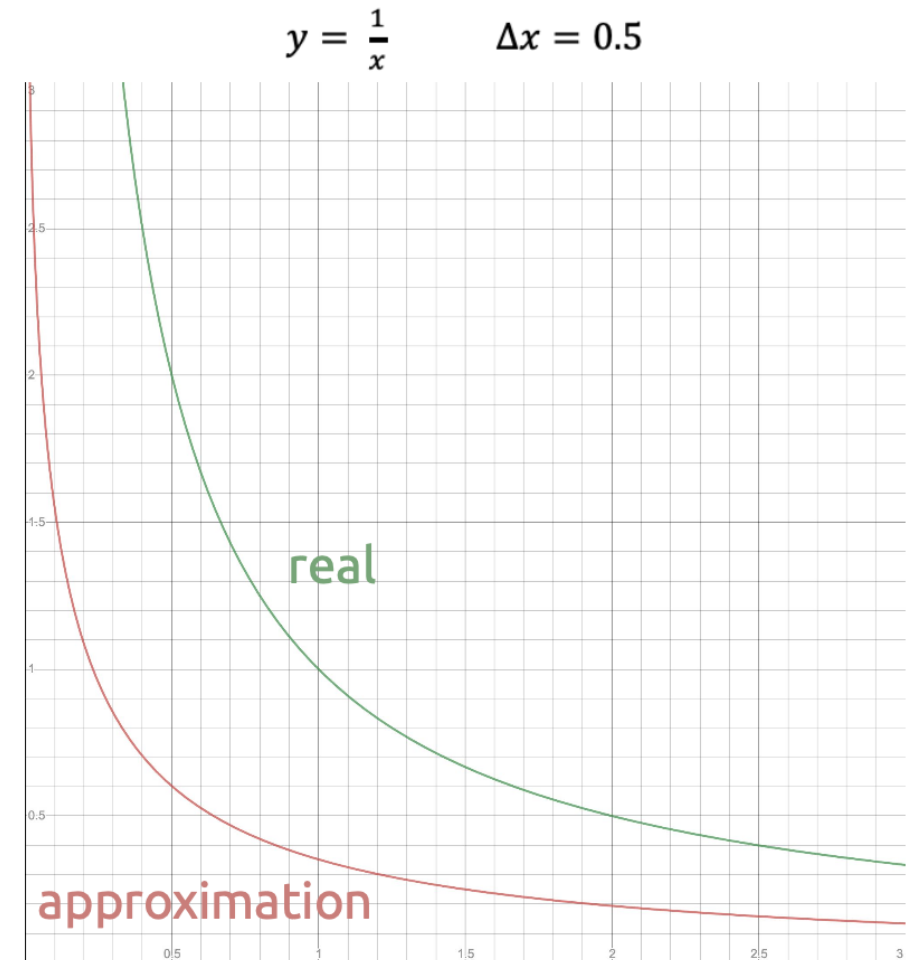
- **Numeric differentiation**

- $\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$
- Pick a small step size Δx
- Also called “finite differences”

Computer-based Derivatives

• Numeric differentiation

- $\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$
- Pick a small step size Δx
- Also called “finite differences”
- Easy to implement
- Arbitrarily inaccurate/unstable



Computer-based Derivatives

- Numeric differentiation
- **Symbolic differentiation**
 - Computer “does algebra” and simplifies expressions
 - What Wolfram Alpha does
<https://www.wolframalpha.com/>


$$d/dx (2x + 3x^2 + x(6 - 2))$$

 Extended Keyboard

 Upload

Derivative:

$$\frac{d}{dx} (2x + 3x^2 + x(6 - 2)) = 6(x + 1)$$

$$\frac{d}{dx} (6x + 3x^2)$$


Computer-based Derivatives

- Numeric differentiation
- **Symbolic differentiation**
 - Computer “does algebra” and simplifies expressions
 - What Wolfram Alpha does
 - **Exact (no approximation error)**
 - **Complex to implement**
 - **Only handles static expressions (what about e.g. loops?)**

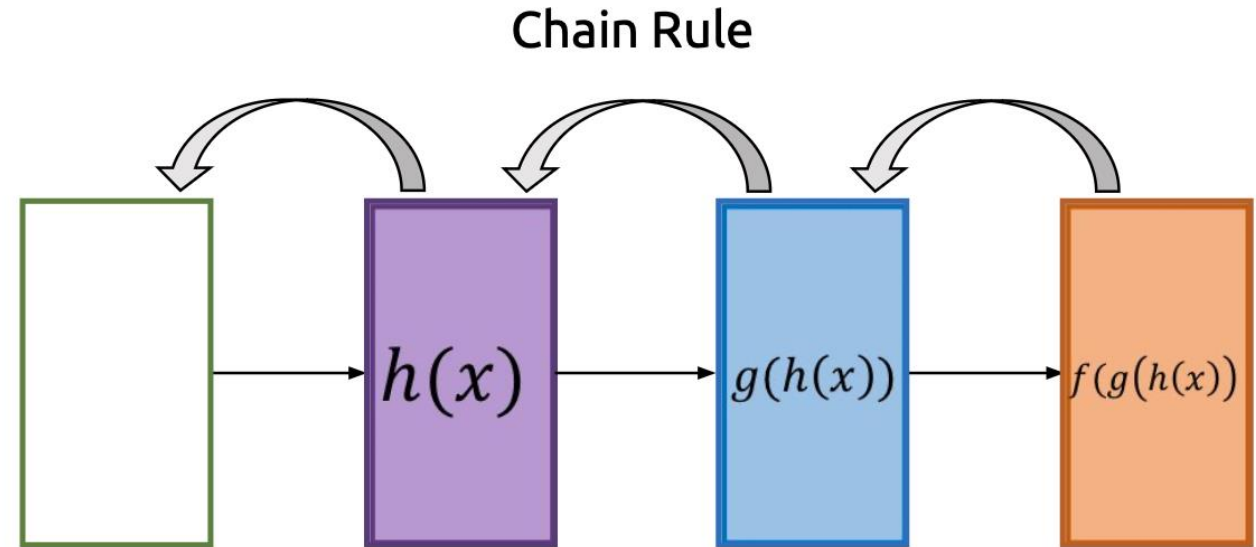
- Example:

```
while abs(x) > 5:  
    x = x / 2
```

- This loop could run once or 100 times, it's impossible to know

Computer-based Derivatives

- Numeric differentiation
- Symbolic differentiation
- **Automatic differentiation**
 - Use the chain rule at runtime

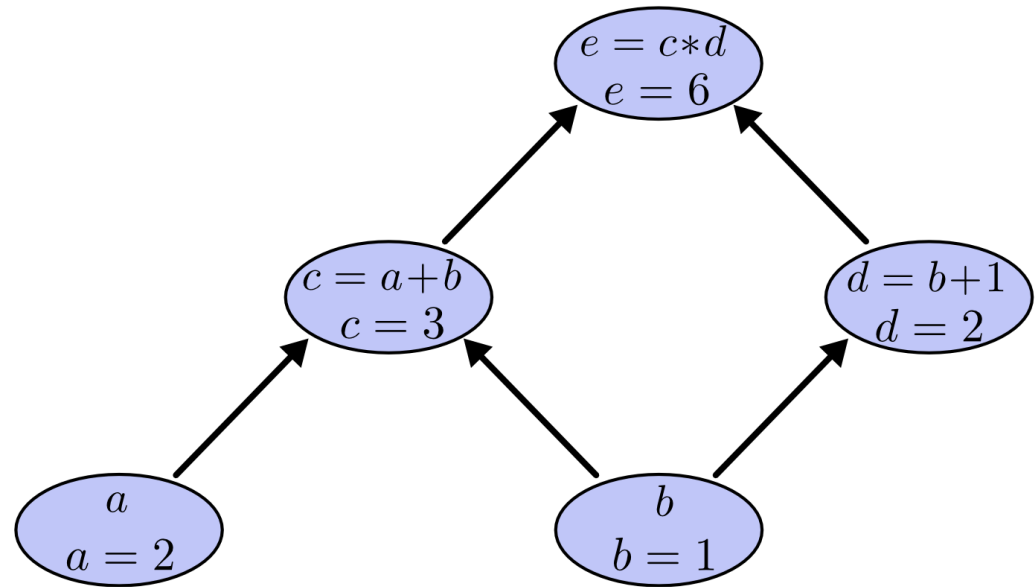


Computer-based Derivatives

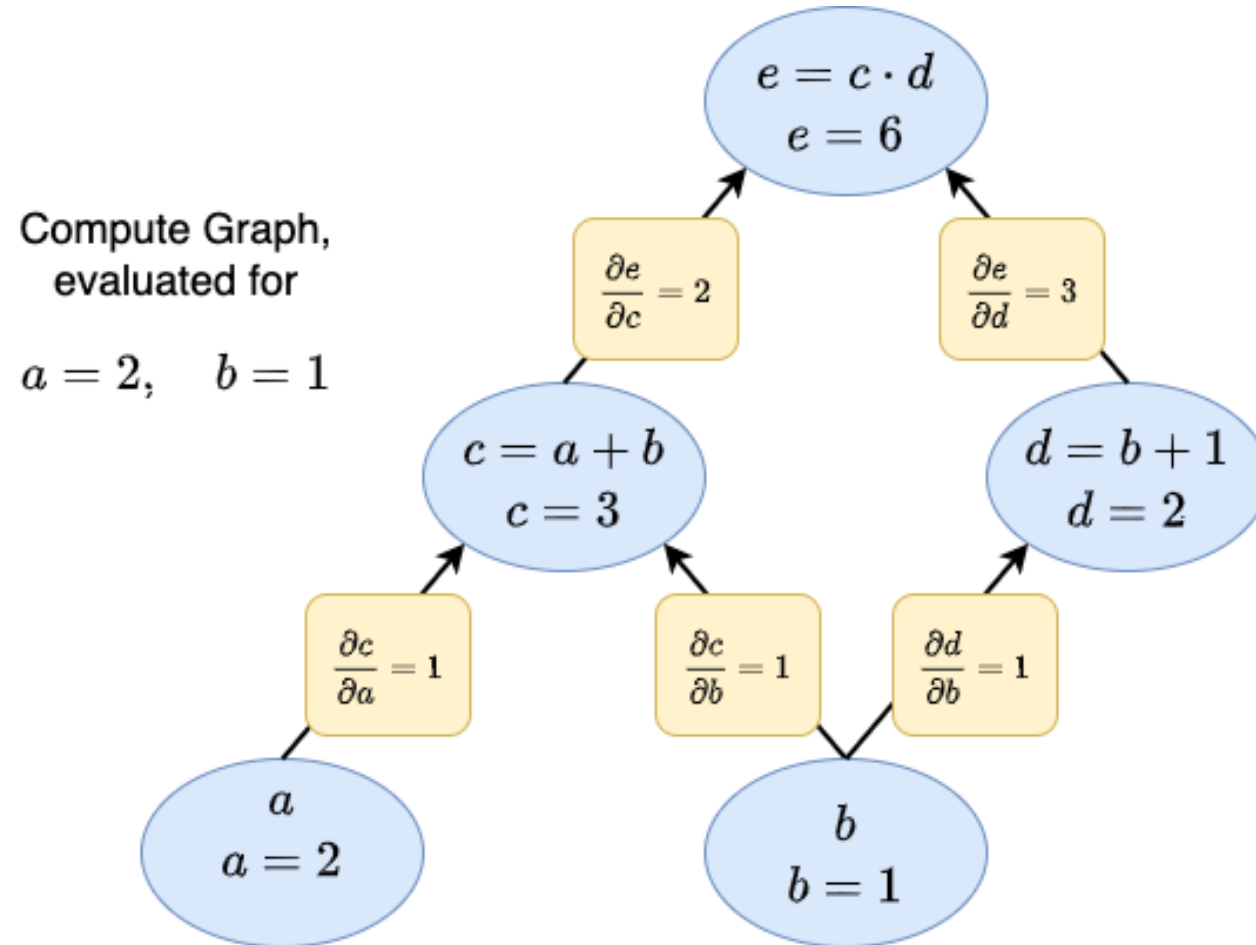
- Numeric differentiation
- Symbolic differentiation
- **Automatic differentiation**
 - Use the chain rule at runtime
 - Gives exact results
 - Handles dynamics (loops, etc.)
 - Easier to implement
 - Can't simplify expressions
- $\sin^2 x + \cos^2 x \Rightarrow 1$
- Automatic differentiation doesn't know this identity, will end up evaluating the entire expression on the left hand side

Computation Graph

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



Computation Graph and Derivatives



Tensorflow Gradient Tape

- Tensorflow will maintain a compute graph of operations performed within a Gradient Tape context
 - Can automatically differentiate operations on request
- This is the purpose and usefulness of Deep Learning Frameworks!
- For the most part, you only have to specify the forward operations and TF (or Torch/Jax) will take care of the rest.

Why should you care about compute graphs?


(This is much more of a common issue in pytorch than tensorflow)

```
def train_with_memory_leak():
    running_loss = 0.0
    for epoch in range(100):
        for i, (inputs, targets) in enumerate(loader):
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

            running_loss += loss

        if i % 10 == 9:
            print(f'Loss: {running_loss / 10}')
            running_loss = 0.0
```

Running loss' compute graph will contain the compute graph of loss!

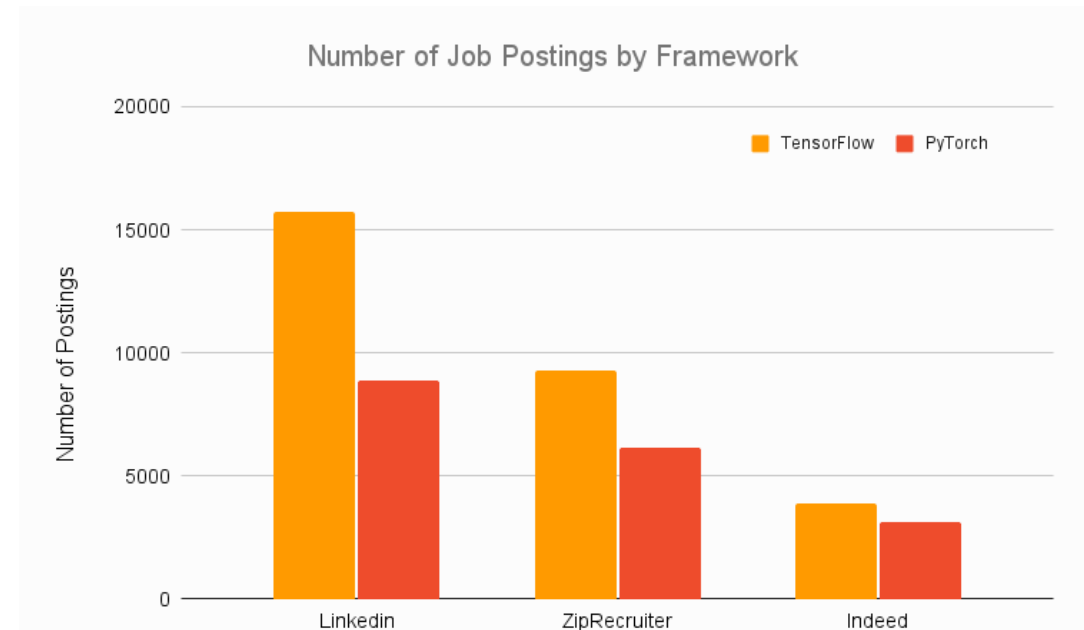
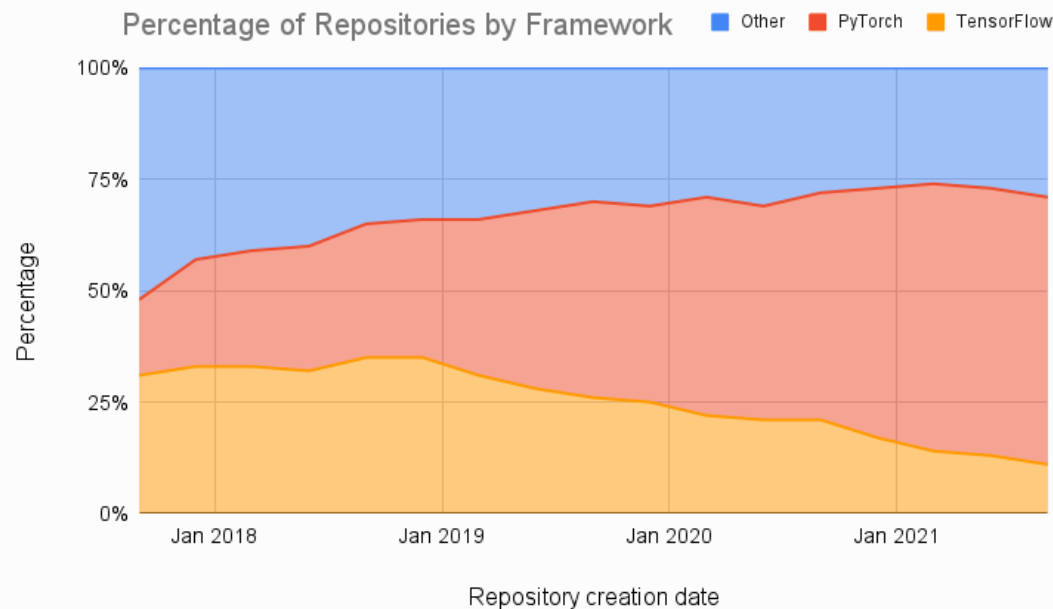


The memory required to store running_loss will only ever increase!

DL Frameworks



- Main current frameworks are Tensorflow, Pytorch, and Jax
- TF and torch are becoming increasingly similar in style and performance
- Jax is new and different



Tensorflow

Tensorflow

- Developed and maintained by Google

Tensorflow

- Developed and maintained by Google
- In addition to autodiff features it also provides:
 - Many common functions (i.e., Softmax, Sigmoid, Cross Entropy, etc.)
 - An easy way to train models (**Keras**)
 - Strong support for hardware acceleration (i.e., if you have a GPU, TF will figure out how to use it)

Tensorflow

- Developed and maintained by Google
- In addition to autodiff features it also provides:
 - Many common functions (i.e., Softmax, Sigmoid, Cross Entropy, etc.)
 - An easy way to train models (Keras)
 - Strong support for hardware acceleration (i.e., if you have a GPU, TF will figure out how to use it)
- “Easier to deploy to production” (has been the general consensus previously, but other frameworks have caught up)

Tensorflow

- Developed and maintained by Google
- In addition to autodiff features it also provides:
 - Many common functions (i.e., Softmax, Sigmoid, Cross Entropy, etc.)
 - An easy way to train models (Keras)
 - Strong support for hardware acceleration (i.e., if you have a GPU, TF will figure out how to use it)
- “Easier to deploy to production” (has been the general consensus previously, but other frameworks have caught up)
- TF lite for on device applications (e.g., phones)

Pytorch

Pytorch

- Developed by Facebook AI (now Meta)

Pytorch

- Developed by Facebook AI (now Meta)
- More common in the research and academic community

Pytorch

- Developed by Facebook AI (now Meta)
- More common in the research and academic community
- “More flexible” and easier to write custom backward passes

Pytorch

- Developed by Facebook AI (now Meta)
- More common in the research and academic community
- “More flexible” and easier to write custom backward passes
- No Gradient Tape, each tensor (matrix/vector) is “trainable” or not. If a tensor is trainable then all operations on it are tracked.

Pytorch

- Developed by Facebook AI (now Meta)
- More common in the research and academic community
- “More flexible” and easier to write custom backward passes
- No Gradient Tape, each tensor (matrix/vector) is “trainable” or not. If a tensor is trainable then all operations on it are tracked.
- Slightly more work to use GPUs or other hardware

Pytorch

- Developed by Facebook AI (now Meta)
- More common in the research and academic community
- “More flexible” and easier to write custom backward passes
- No Gradient Tape, each tensor (matrix/vector) is “trainable” or not. If a tensor is trainable then all operations on it are tracked.
- Slightly more work to use GPUs or other hardware
- Harder to track stats
 - (I still use TF’s tensorboard stat tracker when using Pytorch)

Pytorch

- Developed by Facebook AI (now Meta)
- More common in the research and academic community
- “More flexible” and easier to write custom backward passes
- No Gradient Tape, each tensor (matrix/vector) is “trainable” or not. If a tensor is trainable then all operations on it are tracked.
- Slightly more work to use GPUs or other hardware
- Harder to track stats
 - (I still use TF’s tensorboard stat tracker when using Pytorch)
- Easier to learn and use than tensorflow
 - Better error reporting, training code is harder to write but easier to debug

Jax

Jax

- Also developed by Google...

Jax

- Also developed by Google...
- Very new compared to Pytorch and Tensorflow

Jax

- Also developed by Google...
- Very new compared to Pytorch and Tensorflow
- ***Much Faster***

Jax

- Also developed by Google...
- Very new compared to Pytorch and Tensorflow
- ***Much Faster***
- Takes advantage of Just In Time (JIT) compiling to speed up execution

Jax

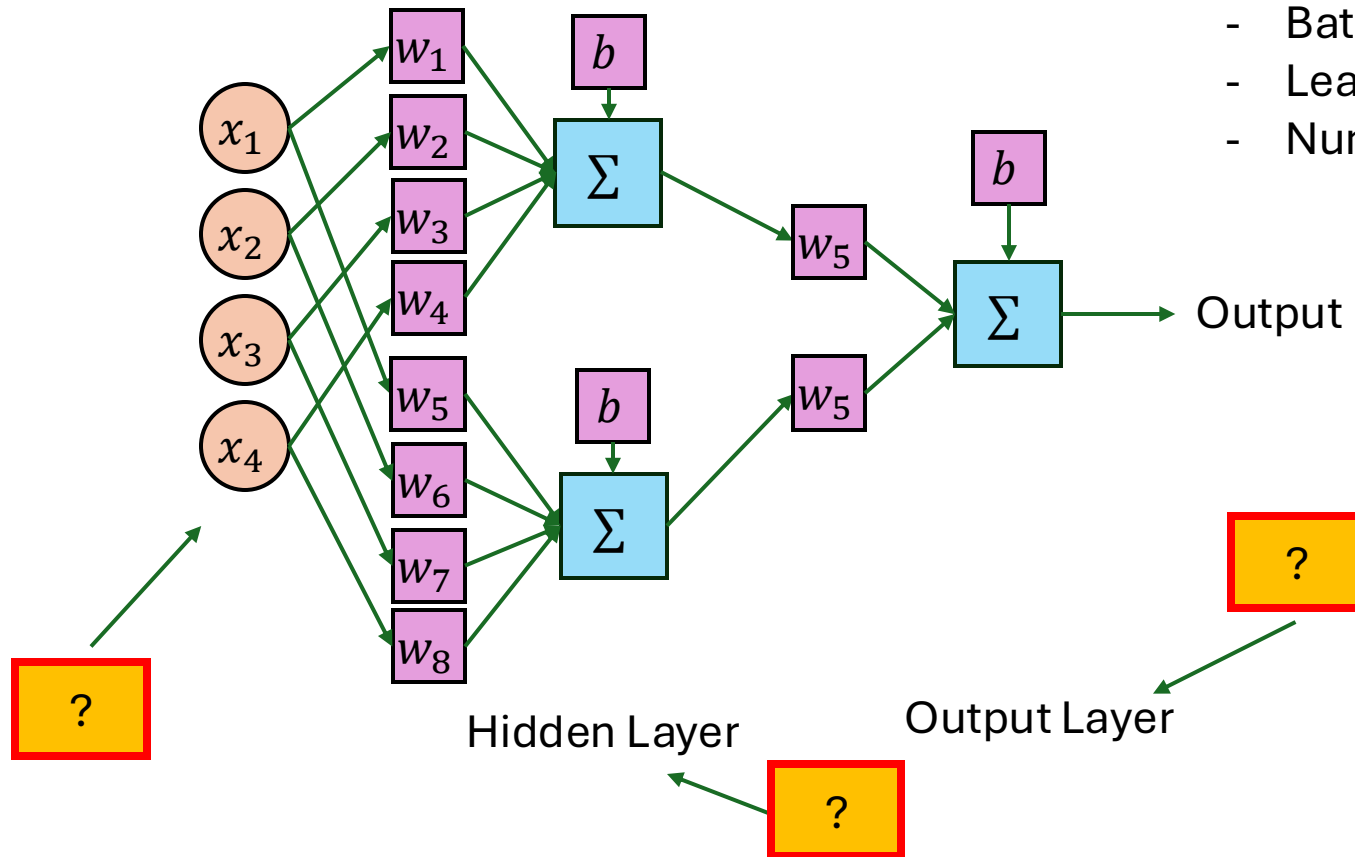
- Also developed by Google...
- Very new compared to Pytorch and Tensorflow
- ***Much Faster***
- Takes advantage of Just In Time (JIT) compiling to speed up execution
- Functional programming paradigm

Hyperparameter Tuning

Hyperparameters

What do you (the programmer) have control of when training neural networks?

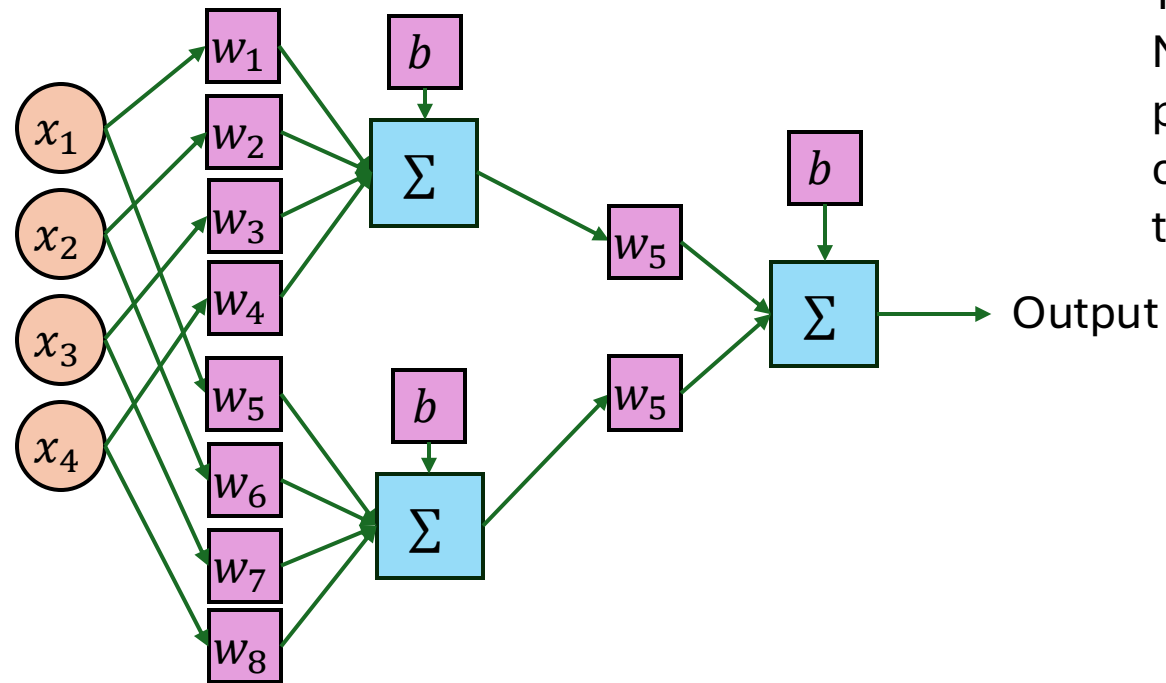
- Network Initialization
- Hidden Layer Size
- Number of hidden layers
- Activation Functions
- Optimizer (SGD, Adam, RMSProp)
- Batch Size
- Learning rate
- Number of Epochs



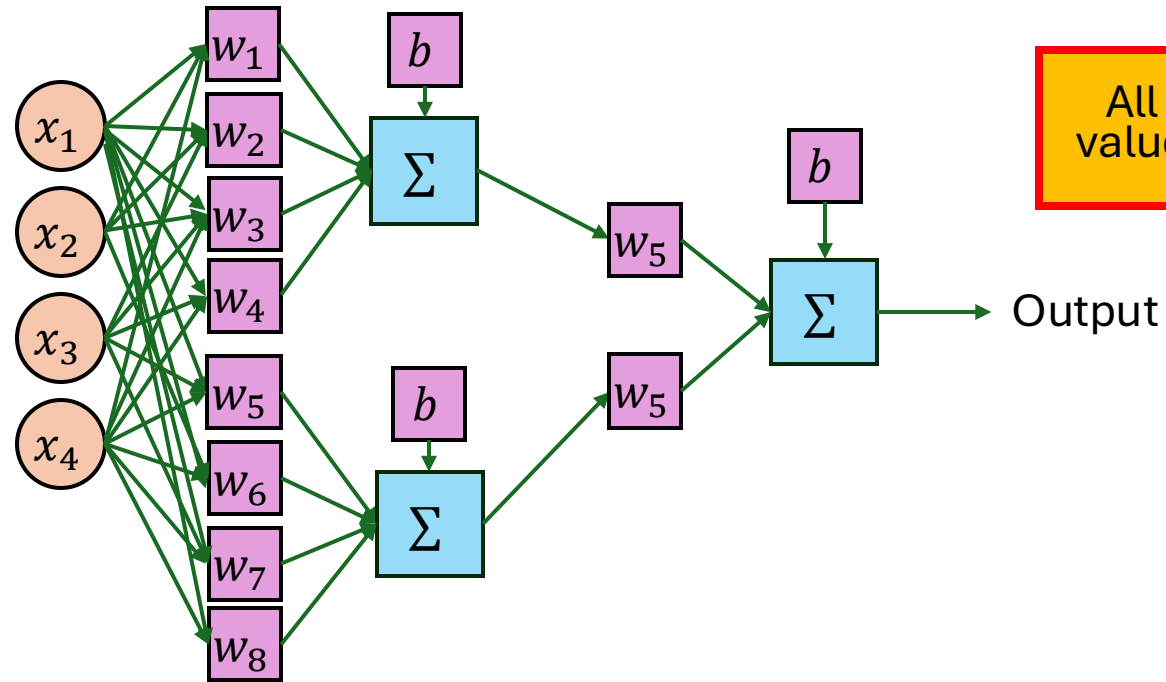
Hyperparameters

The **parameters** of a Neural Network are what is trained (e.g., weights and biases).

The **hyperparameters** of a Neural Network are the parameters that **you** have control of that control that training.



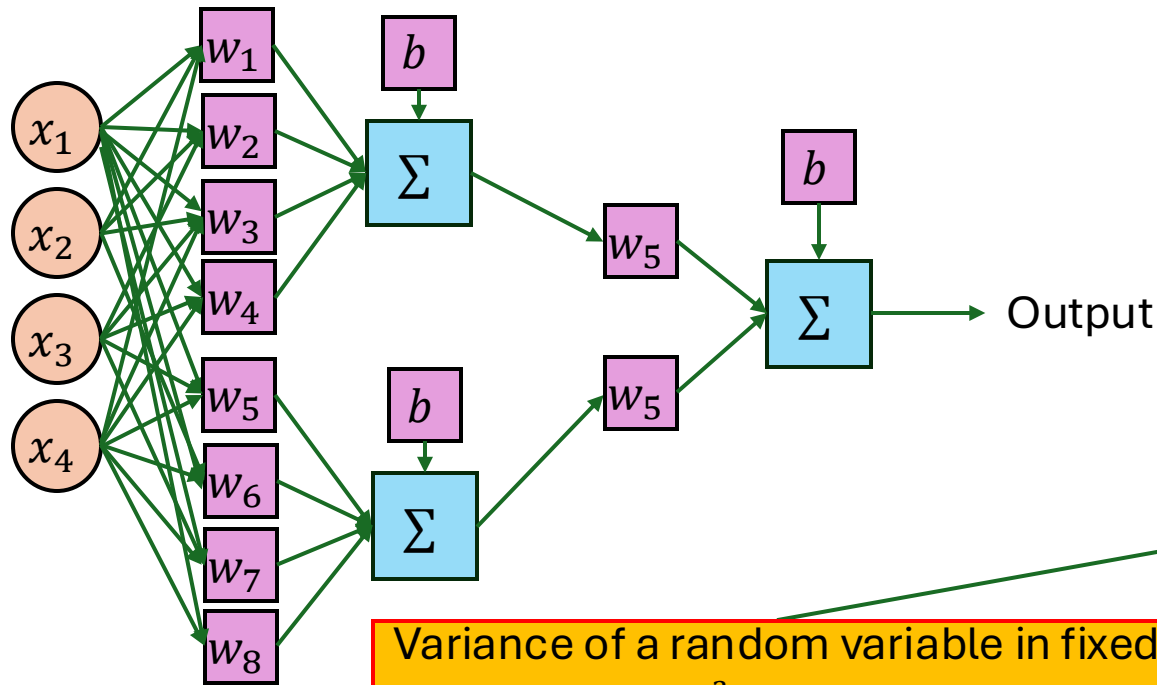
Network Initialization



What if we begin with all parameters set to 0?

All neurons would have the same value, gradients would be the same.

Network Initialization



Variance of a random variable in fixed range $[-x, x]$ is $\frac{x^2}{3}$ (easy to derive from definition of variance)

Idea #1: Uniform random weights between -1 and 1 (works fine)

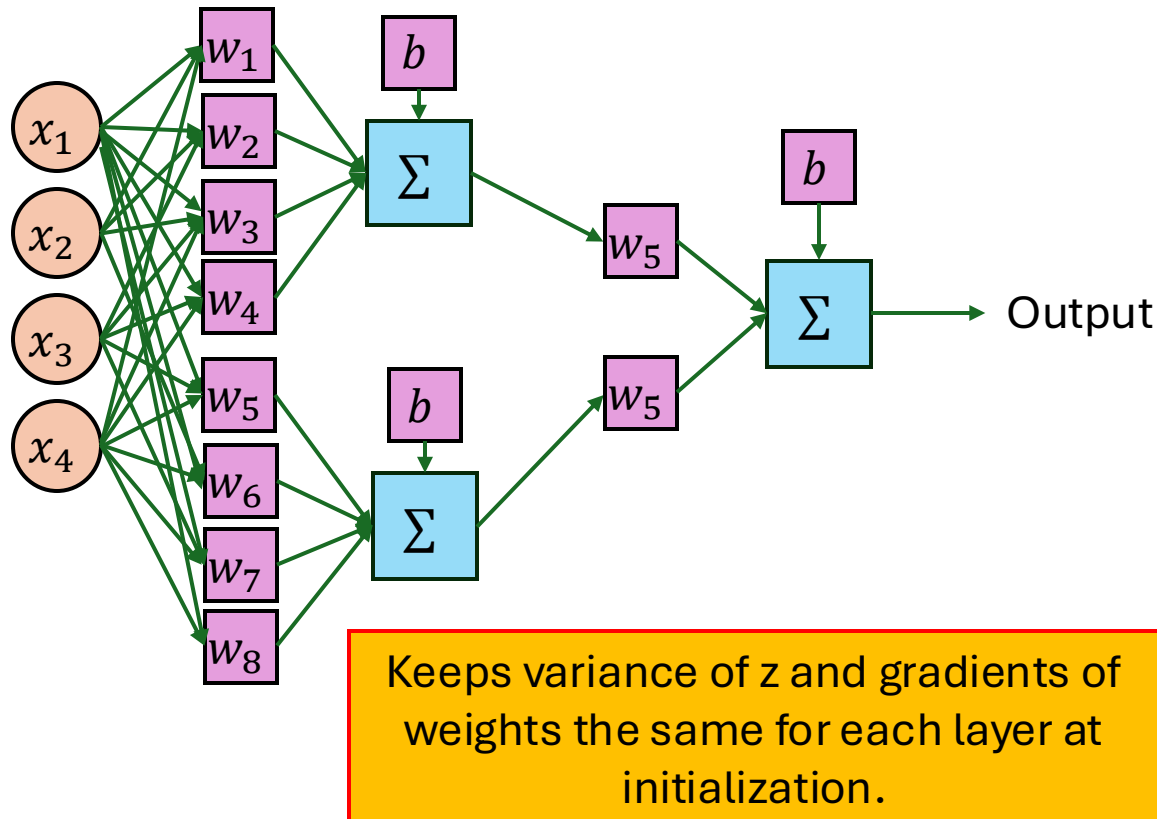
But... what if there are many many weights in a layer? The scale of the output can grow.

Idea #2: Xavier (Glorot) initialization:

Uniform: Initialize each weight uniformly at random in the range $[-x, x]$ with $x = \sqrt{\frac{6}{n_{in} + n_{out}}}$

Normal: Initialize each weight with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$

Network Initialization



Idea #1: Uniform random weights between -1 and 1 (works fine)

But... what if there are many many weights in a layer? The scale of the output can grow.

Idea #2: Xavier (Glorot) initialization:

Uniform: Initialize each weight uniformly at random in the range $[-x, x]$ with $x = \sqrt{\frac{6}{n_{in} + n_{out}}}$

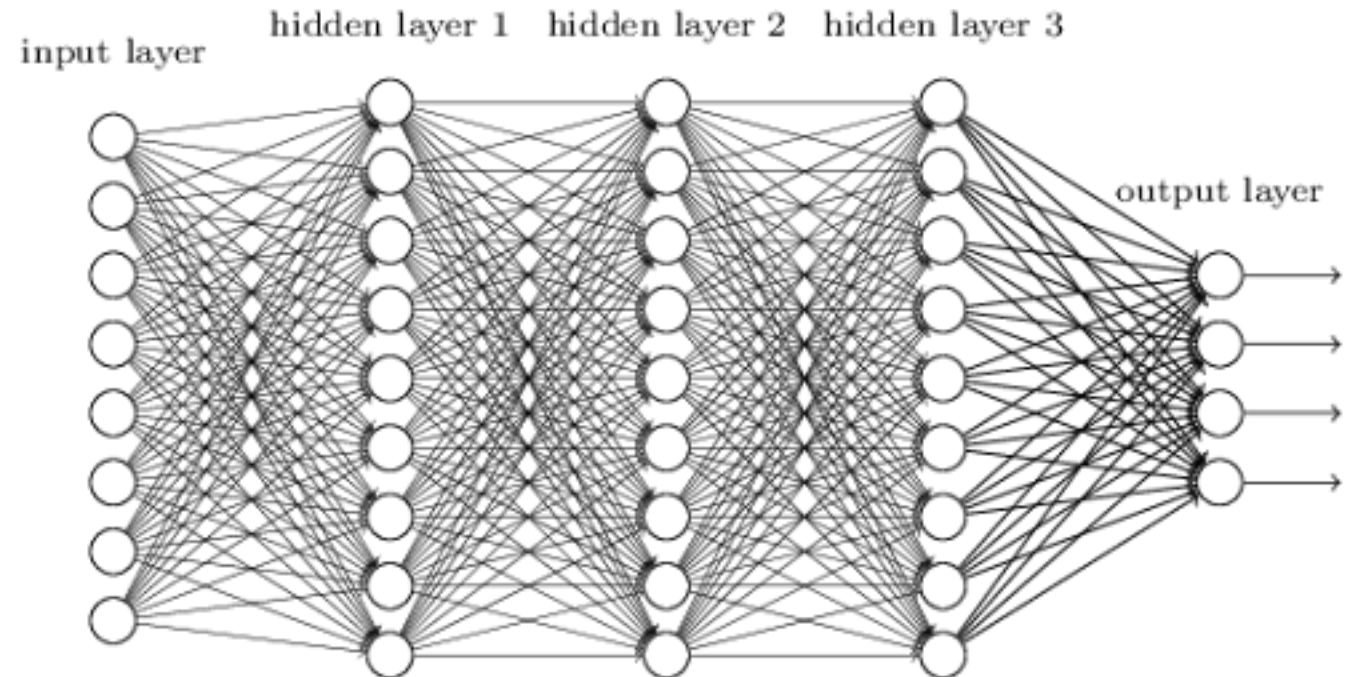
Normal: Initialize each weight with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$

Hidden Layers

- How deep (# hidden layers) should your network be?
- How wide (# neurons in a layer) should your network be?

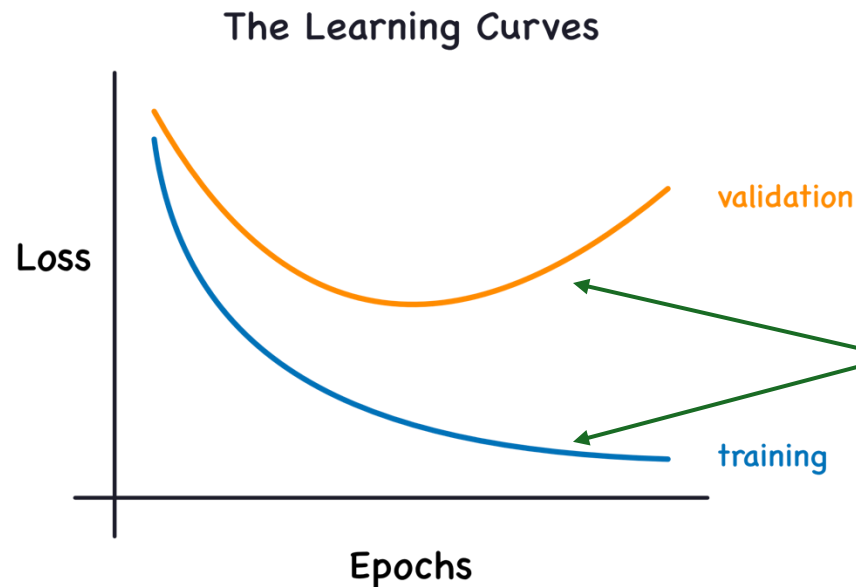
How complex is the problem you are trying to solve?

Process of (informed) trial and error.
How do you know if one hyper-parameter setting is better than another?



How to use the Validation Set

(In theory)



Model starts overfitting

Early Stopping Algorithm

1. Track training loss and validation loss
2. If validation loss starts to increase, terminate

What if your validation loss is much higher than training loss?

Your model has overfit, try **reducing** its size

What if your validation loss and training loss are both high?

Your model has underfit, try **increasing** its size

Is adding more width or depth better?

◀ CSCI 1470

CSCI 1470

Deep Learning

Section S01, CRN 26629

Spring 2025

Depth-Width Tradeoffs in Approximating Natural Functions with Neural Networks

Itay Safran

Weizmann Institute of Science

`itay.safran@weizmann.ac.il`

Ohad Shamir

Weizmann Institute of Science

`ohad.shamir@weizmann.ac.il`

With the same number of total parameters, deep networks can learn more complex functions.

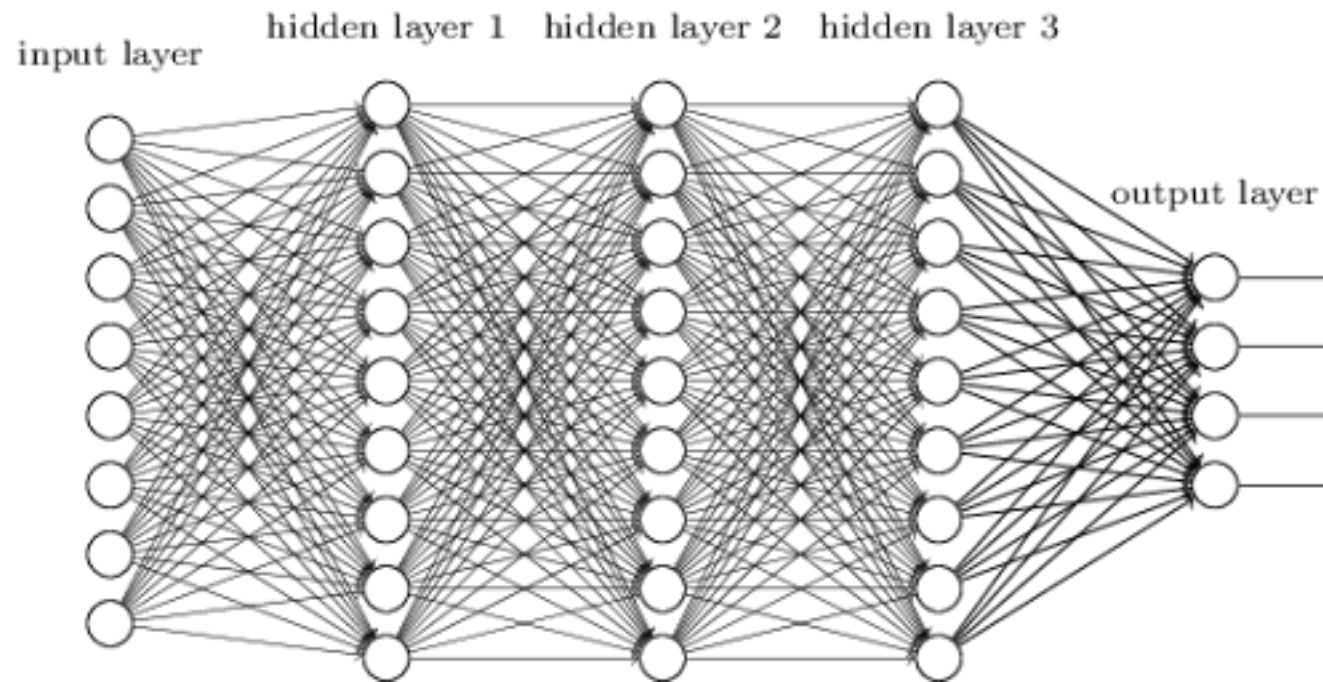
Recall that NNs are compositions of functions for which we are learning parameters:

$$f(g(h(i(j(x))))$$

It's better (in general) to have more functions composed than it is to have more complex functions

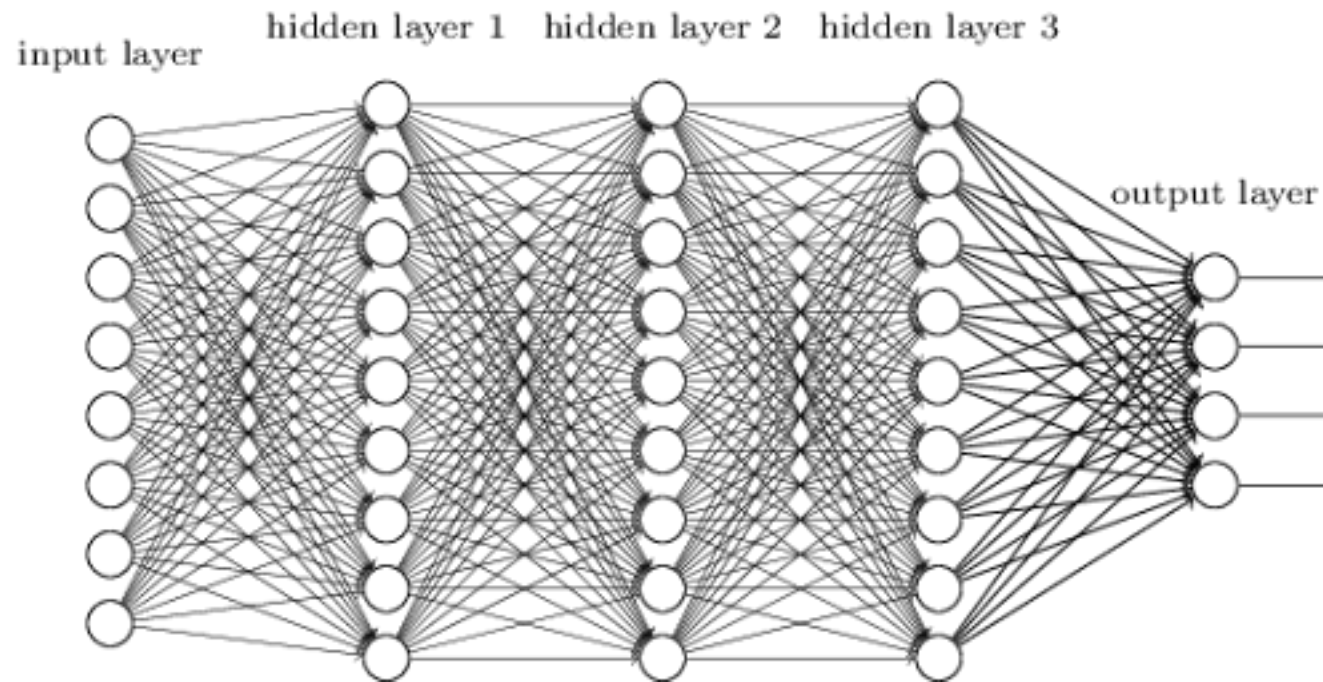
- If there are 10 inputs, 3 layers of 10 neurons, and 4 outputs, how many weights are there total?

$$\begin{aligned} W_1 &\in \mathbb{R}^{10 \times 10} \\ W_2 &\in \mathbb{R}^{10 \times 10} \\ W_3 &\in \mathbb{R}^{10 \times 10} \\ W_4 &\in \mathbb{R}^{10 \times 4} \\ \text{Total} &= 340 \end{aligned}$$

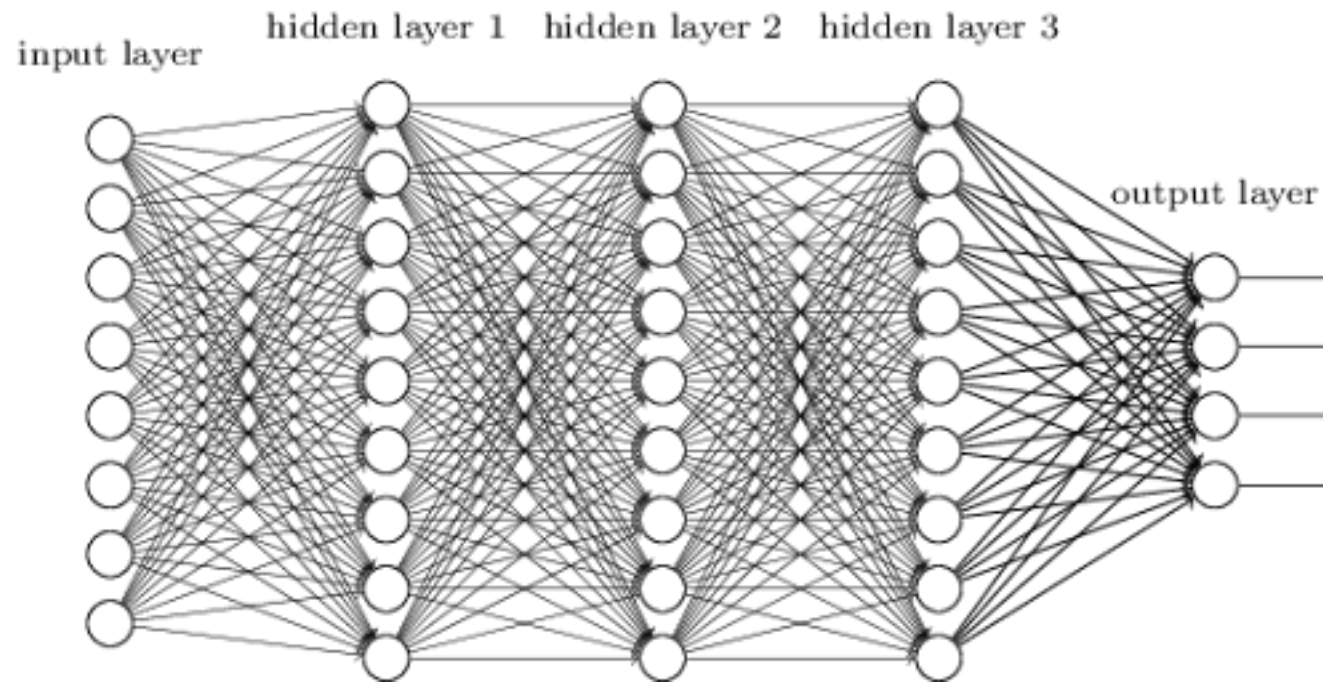


- If there are 10 inputs, 3 layers of 10 neurons, and 4 outputs, how many weights are there total?
- What if we double the width of each hidden layer?

$$\begin{aligned}W_1 &\in \mathbb{R}^{10 \times 20} \\W_2 &\in \mathbb{R}^{20 \times 20} \\W_3 &\in \mathbb{R}^{20 \times 20} \\W_4 &\in \mathbb{R}^{20 \times 4} \\ \text{Total} &= 1080\end{aligned}$$



- If there are 10 inputs, 3 layers of 10 neurons, and 4 outputs, how many weights are there total?
- What if we double the depth? of each hidden layer?



$$\begin{aligned} W_1 &\in \mathbb{R}^{10 \times 10} \\ W_2 &\in \mathbb{R}^{10 \times 10} \\ W_3 &\in \mathbb{R}^{10 \times 10} \\ W_4 &\in \mathbb{R}^{10 \times 10} \\ W_5 &\in \mathbb{R}^{10 \times 10} \\ W_6 &\in \mathbb{R}^{10 \times 10} \\ W_7 &\in \mathbb{R}^{10 \times 4} \\ \text{Total} &= 640 \end{aligned}$$

Overparameterization

Overparameterization: Using more parameters than necessary for a ML problem.

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou

Daan Wierstra Martin Riedmiller

DeepMind Technologies

`{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com`

~10,000 parameters in network

Overparameterization

Overparameterization: Using more parameters than necessary for a ML problem.

Most of the time, networks use many more parameters than *necessary*.

In general, it's impossible to know the fewest amount of parameters that could solve a problem.

PLAYING ATARI WITH SIX NEURONS

Giuseppe Cuccu

eXascale Infolab

Department of Computer Science
University of Fribourg, Switzerland
name.surname@unifr.ch

Julian Togelius

Game Innovation Lab

Tandon School of Engineering
New York University, NY, USA
julian@togelius.com

Philippe Cudré-Mauroux

eXascale Infolab

Department of Computer Science
University of Fribourg, Switzerland
name.surname@unifr.ch

ABSTRACT

Deep reinforcement learning, applied to vision-based problems like Atari games, maps pixels directly to actions; internally, the deep neural network bears the responsibility of both extracting useful information and making decisions based on it. By separating the image processing from decision-making, one could better understand the complexity of each task, as well as potentially find smaller policy representations that are easier for humans to understand and may generalize better. To this end, we propose a new method for learning policies and compact state representations separately but simultaneously for policy approximation in reinforcement learning. State representations are generated by an encoder based on two novel algorithms: Increasing Dictionary Vector Quantization makes the encoder capable of growing its dictionary size over time, to address new observations as they appear in an open-ended online-learning context; Direct Residuals Sparse Coding encodes observations by disregarding reconstruction error minimization, and aiming instead for highest information inclusion. The encoder autonomously selects observations online to train on, in order to maximize code sparsity. As the dictionary size increases, the encoder produces increasingly larger inputs for the neural network: this is addressed by a variation of the Exponential Natural Evolution Strategies algorithm which adapts its probability distribution dimensionality along the run. We test our system on a selection of Atari games using tiny neural networks of only 6 to 18 neurons (depending on the game's controls). These are still capable of achieving results comparable—and occasionally superior—to state-of-the-art techniques which use two orders of magnitude more neurons.

Overparameterization

PLAYING ATARI WITH SIX NEURONS

Giuseppe Cuccu

eXascale Infolab

Department of Computer Science
University of Fribourg, Switzerland
name.surname@unifr.ch

Julian Togelius

Game Innovation Lab

Tandon School of Engineering
New York University, NY, USA
julian@togelius.com

Philippe Cudré-Mauroux

eXascale Infolab

Department of Computer Science
University of Fribourg, Switzerland
name.surname@unifr.ch

(This paper doesn't use SGD or backprop, but another optimization method)

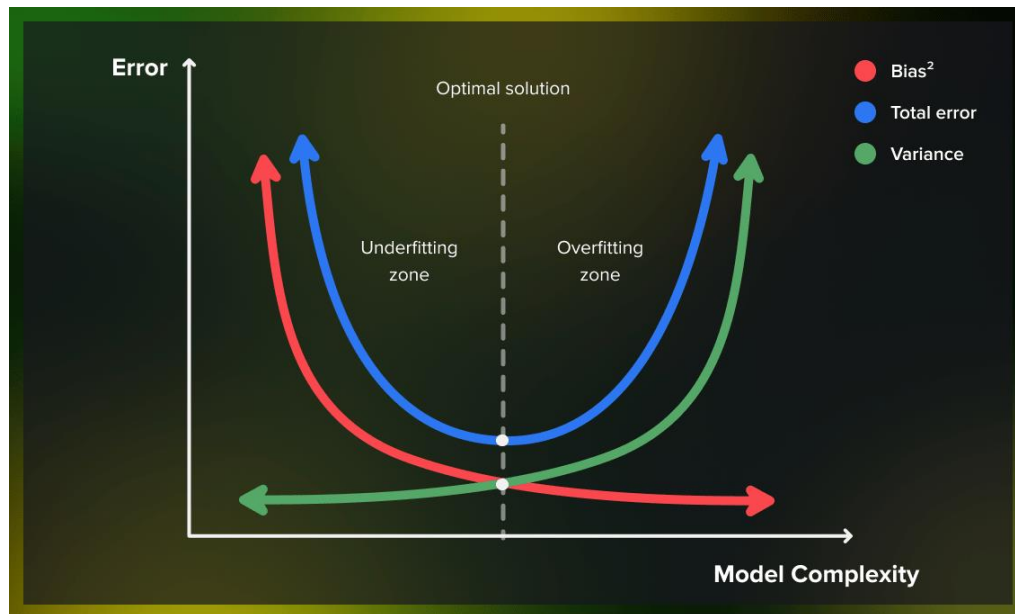
ABSTRACT

Deep reinforcement learning, applied to vision-based problems like Atari games, maps pixels directly to actions; internally, the deep neural network bears the responsibility of both extracting useful information and making decisions based on it. By separating the image processing from decision-making, one could better understand the complexity of each task, as well as potentially find smaller policy representations that are easier for humans to understand and may generalize better. To this end, we propose a new method for learning policies and compact state representations separately but simultaneously for policy approximation in reinforcement learning. State representations are generated by an encoder based on two novel algorithms: Increasing Dictionary Vector Quantization makes the encoder capable of growing its dictionary size over time, to address new observations as they appear in an open-ended online-learning context; Direct Residuals Sparse Coding encodes observations by disregarding reconstruction error minimization, and aiming instead for highest information inclusion. The encoder autonomously selects observations online to train on, in order to maximize code sparsity. As the dictionary size increases, the encoder produces increasingly larger inputs for the neural network: this is addressed by a variation of the Exponential Natural Evolution Strategies algorithm which adapts its probability distribution dimensionality along the run. We test our system on a selection of Atari games using tiny neural networks of only 6 to 18 neurons (depending on the game's controls). These are still capable of achieving results comparable—and occasionally superior—to state-of-the-art techniques which use two orders of magnitude more neurons.

(We will cover other techniques for managing overfitting next week)

Overparameterization

Bias-Variance Tradeoff (Traditional Understanding)



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.

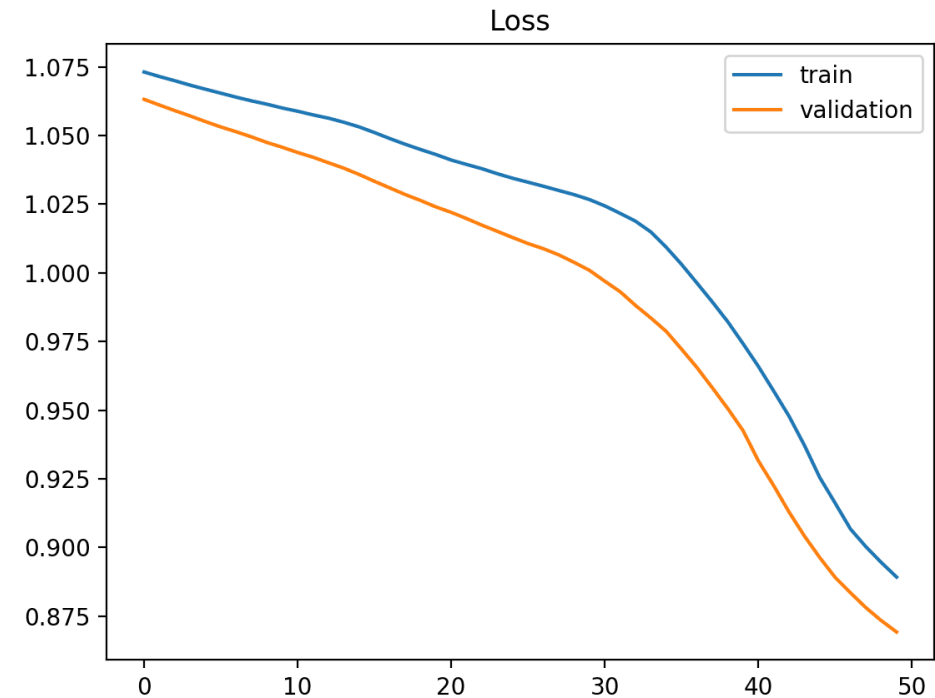
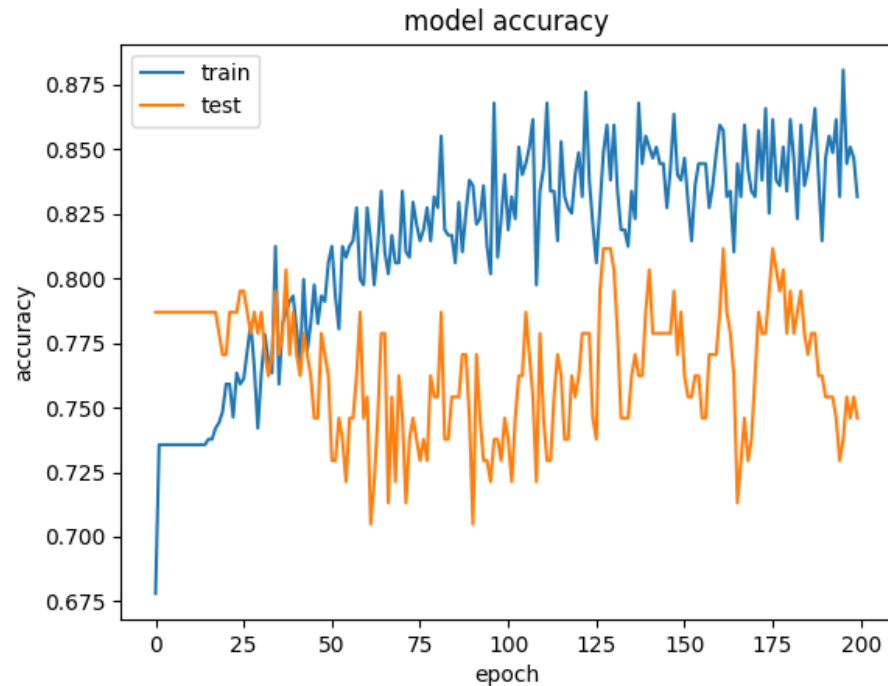
Understanding learning in this overparameterized regime requires new theory and foundational empirical studies, even for the simplest case of the linear model. The underpinnings of this understanding have been laid in very recent analyses of overparameterized linear regression and related statistical 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.

Optimizers

- SGD, SGD + Momentum, SGD + Adaptive Momentum (Adam), RMSProp,... the list is ever growing
- How do you choose between them?
- Just use Adam.
 - The only downside is that it might work so well that you end up overfitting.
 - Suggested initial learning rate of $3e-4$

Batch Size and Learning Rate

Having too small a batch or too high a learning rate can cause variance in training/validation loss – symptoms often look similar



General Tips

- Don't change too much at once.
- Keep track of parameters you've tested and track their performance
- Don't just randomly guess parameters, apply critical thinking, come up with a hypothesis and test your hypothesis.

(Use the scientific method)

Andrej Karpathy: A recipe for training neural networks

<https://karpathy.github.io/2019/04/25/recipe/>

Recap

DL Frameworks are ever evolving, but can handle gradient computations automatically with compute graphs.

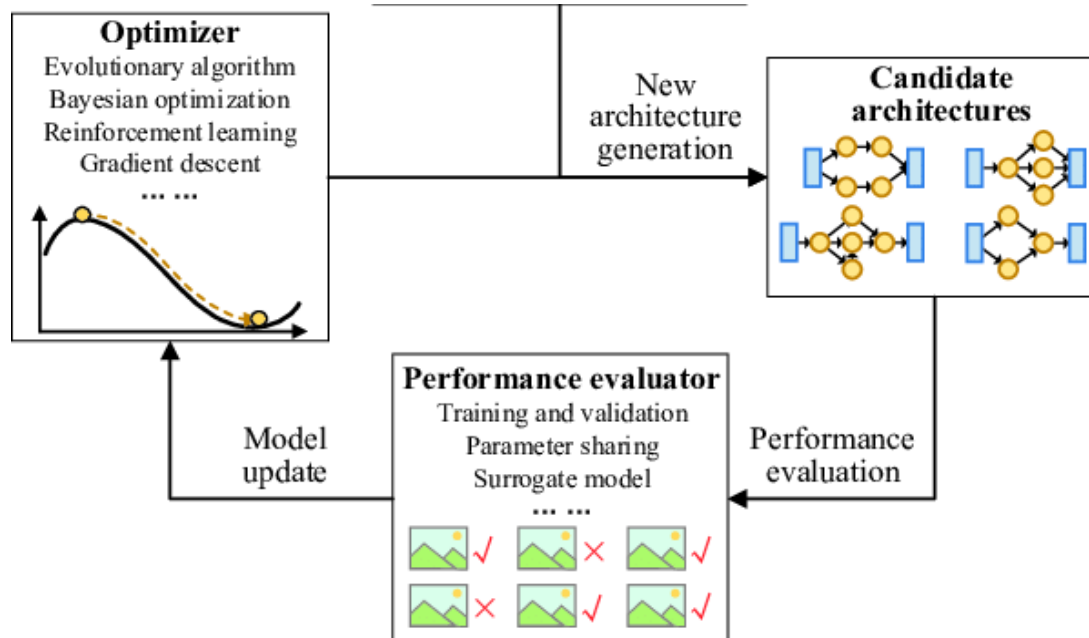
Hyperparameter tuning is an experimental process, use the scientific method!

You need to have a fundamental understanding of what you *expect* to have happen when you change each hyperparameter.

AutoML (If time)

Neural Architecture Search (NAS)

Changing hyperparameters results in different performance, can we run an optimization algorithm on our hyperparameters?



Pros:

- No longer need human input
- May find better hyperparameters than humans

Cons:

- Takes a very long time...
- Hyperparameters are discrete and highly dependent (e.g., width/depth), it's a really hard optimization problem...

AutoML (If time)

Option #1: Grid search

Define a set of possible parameters (i.e., learning rates, width, depth, etc)

Try every combination of hyperparameters possible, pick setting with best validation set performance.

What are some downsides of grid search?

AutoML (If time)

Option #2: Bayesian Optimization

We believe the performance of hyperparameters that are close together, should have similar results.

Define a set of possible parameters (i.e., learning rates, width, depth, etc)

Try sets of possible hyperparameters, each with some probability.

- The probability that you try a specific hyperparameter setting depends on the performance of nearby hyperparameter settings.
- Also track uncertainty of hyperparameters (i.e., settings you have not tried something close to before)

AutoML (If time)

Keras tuner is compatible with Tensorflow, Pytorch, and Jax and has various automatic hyperparameter tuning methods

KerasTuner



KerasTuner is an easy-to-use, scalable hyperparameter optimization framework that solves the pain points of hyperparameter search. Easily configure your search space with a define-by-run syntax, then leverage one of the available search algorithms to find the best hyperparameter values for your models. KerasTuner comes with Bayesian Optimization, Hyperband, and Random Search algorithms built-in, and is also designed to be easy for researchers to extend in order to experiment with new search algorithms.

Looking Forward

Up until this point, we've covered Neural Networks generally referred to as MLPs, feed forward networks, or networks made up of "Linear Layers"

Up next: Convolutional Neural Networks

What happens if our input image is shifted?

