# Generalization, Deep Learning, and Representations

*Machine Learning Fundamentals for Economists*

Jesse Perla

*jesse.perla@ubc.ca*

*University of British Columbia*

## Table of contents

## Overview

## Summary

- Step back from optimization and fitting process to discuss the broader issues of statistics and function approximation
- Key topics:
  - → ERM, interpolation, and generalization error
  - → Features and Representations
  - → Neural Networks and broader hypothesis classes
- In the subsequent lecture we can briefly return to the optimization process itself to discuss
  - → Global vs. Local solutions
  - → Inductive bias and regularization

## Packages

```python
import jax
import jax.numpy as jnp
import numpy as np
from jax import random
import optax
import jax_dataloader as jdl
from jax_dataloader.loaders import DataLoaderJAX
from flax import nnx
import torch
import torch.nn as nn
import copy
```

## Reminder: ERM and Generalization

## Reminder: Population vs. Empirical Risk (see Intro Lecture)

### Population Risk

$$f^* = \arg\min_{f \in \mathcal{F}} \underbrace{\mathbb{E}_{(x,y)\sim\mu^*}\left[\ell(f,x,y)\right]}_{\equiv R(f,\mu^*)}$$

### Empirical Risk

$$\theta^* = \arg\min_{\theta \in \Theta} \underbrace{\frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \ell(f_\theta,x,y)}_{\equiv \hat{R}(\theta,\mathcal{D})}$$

- Two separate sources of approximation error here (the bias-variance tradeoff):
  1. **Approximation error** $\varepsilon_{\text{app}}$ (i.e. bias): $f_{\theta^*} \in \mathcal{H}(\Theta)$ may not be able to represent $f^*$
  2. **Generalization error** $\varepsilon_{\text{gen}}$ (i.e. variance): Finite sample size of $\mathcal{D} \overset{\text{iid}}{\sim} \mu^*$

## Reminder: Error Decomposition (see Intro Lecture)

- Abuse notation and decompose following Bottou and Bousquet (2007), Murphy (2022)

$$\mathbb{E}_{\mathcal{D} \overset{\text{iid}}{\sim} \mu^*}\left[\min_{\theta \in \Theta} \hat{R}(\theta, \mathcal{D}) - \min_{f \in \mathcal{F}} R(f, \mu^*)\right] = \underbrace{R(f_{\theta^*}, \mu^*) - R(f^*, \mu^*)}_{\equiv \varepsilon_{\text{app}}(f_{\theta^*})} + \underbrace{\mathbb{E}_{\mathcal{D} \overset{\text{iid}}{\sim} \mu^*}\left[\hat{R}(\theta^*, \mathcal{D}) - R(f_{\theta^*}, \mu^*)\right]}_{\equiv \varepsilon_{\text{gen}}(f_{\theta^*})}$$

- Modern ML shows we can often reduce both
  → Punchline: simplicity leads to better generalization, but # parameters is poor heuristic for simplicity
- With a very flexible $\mathcal{H}(\Theta)$ such as neural networks often $\varepsilon_{\text{app}}(f_{\theta^*}) \approx 0$
- Then error is $\varepsilon_{\text{gen}}(f_{\theta^*}) \approx \hat{R}(\theta^*, \mathcal{D}) - \hat{R}(\theta^*, \mathcal{D}_{\text{test}})$ given new samples $\mathcal{D}_{\text{test}} \overset{\text{iid}}{\sim} \mu^*$

## “Deep” Learning

- Here we will discuss the design of the hypothesis class $\mathcal{H}(\Theta)$
- Contrast learned vs. engineered features/representations

  → This, combined with features of the $\mu^*$, is key to the magic of how deep learning can work for such high-dimensional inputs.

- We will address concerns of overfitting in the generalization lecture

## Features and Representations

## Features

- The **features** of a problem start with the inputs that we use within the hypothesis class $\mathcal{H}(\Theta)$
- Economists are used to **shallow** approximations, e.g.,
  - → Linear functions, $f_\theta(x) = \theta \cdot x$
  - → Orthogonal polynomials, $f_\theta(x) = \sum_{i=1}^{d} \theta_i \phi_i(x)$ for basis $\phi_i(x)$
- Economists **feature engineer** to choose the appropriate $x \in \mathcal{X}$ form raw data, typically then used with shallow approximations
  - → e.g. log, dummies, polynomials, first-differences, means, etc.
  - → Embeds all sorts of priors, wisdom, and economic intuition
  - → Priors are inescapable - and a good thing as long as you are aware when you use them and how they affect inference

## Fixed Representations in ERM

- We can think of this manual process as taking the raw data $x$ and transforming it into a **representation** $z \in \mathcal{Z}$ with $\phi : \mathcal{X} \to \mathcal{Z}$
- Then, instead of finding a $f_\theta : \mathcal{X} \to \mathcal{Y}$, we can find a $h_\theta : \mathcal{Z} \to \mathcal{Y}$ and use $\ell(h_\theta \circ \phi, x, y)$. i.e., $f_{\theta^*} \equiv h_{\theta^*} \circ \phi$

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \ell(h_\theta \circ \phi, x, y)$$

→ At this point, we have deliberately left off parameters (i.e., $\phi$ not $\phi_\theta$)

## Reminder: Polynomial Basis are Shallow (see Intro Lecture)

- Suppose $x \in \mathbb{R}$ and we want to approximate $f(x)$ with a polynomial of degree $d$
- For some polynomial basis $T_1(x), \dots, T_d(x)$ (e.g., monomials, Chebyshev, Legendre, etc.)

$$\phi(x) = [1 \quad T_1(x) \quad T_2(x) \quad \cdots \quad T_d(x)]^\top$$

→ Note that there are no "learned" parameters!

- Approximate $f_\theta(x)$ with $h_\theta(z) = W^\top z$, where $W \in \mathbb{R}^{d+1}$ and $W \in \theta$

$$f_\theta(x) \equiv W^\top \phi(x) = \sum_{i=0}^{d} W_i T_i(x)$$

## Finding Representations is an Art

- Choosing the right $\phi$ is crucial to success.
  - → It is the process of finding **latent variables** or summary statistics.
- What is our goal when choosing $\phi$?
  - → Drop irrelevant information
  - → Find $z$ that captures the relevant information in $x$ for the problem at hand (i.e., the particular $\ell, y, \mathcal{D}$)
  - → See ProbML Book 2 Section 5.6 for more on information theory
  - → Disentangled (i.e., the factors of variation are separated out in $z$)

## Problem Specific or Reusable?

- Are representations reusable between tasks? (i.e., $h^1_\theta \circ \phi_{\theta^\phi}$ and $h^2_\theta \circ \phi_{\theta^\phi}$ with the same $\theta^\phi$)?
  - → e.g., our wisdom of when to take logs or first-differences is often reusable
- Remember: representations are on $\mathcal{X}$, not $\mathcal{Y}$
  - → Encodes age old experience from working with the datasources
  - → We probably gained this by seeing $\mathcal{Y}$ from previous tasks
- What if $\mathcal{X}$ is complicated, or we are worried we may have chosen the wrong $z = \phi(x)$?
  - → Can we learn this automatically?

## Can Representations Be Learned?

- The short answer is: yes, but it is still an art (with finite data)
- **Representation learning** finds representations $\phi_\theta : \mathcal{X} \to \mathcal{Z}$ using $\mathcal{D}$
  - → Hopefully: works well for $x \sim \mu^*$, and for many $\ell(h_\theta \circ \phi_\theta, x, y)$
- This happens in subtle ways in many different methods. e.g.
  - → If we run "unsupervised" clustering or embeddings on our data to embed it into a lower dimensional space then run a regression
  - → "Learning the kernel". See ProbML Book 2 Section 18.6
  - → Autoencoders and variational autoencoders
  - → Deep learning approximations, which we will discuss in detail

# Benefits of Having Learned Representations

- If you know the correct features, there is no benefit besides maybe dimension reduction. But are you so sure they are correct?
- Can handle complicated data (e.g. text, networks, high-dimensions)
- Maybe the representations are reusable across problems, just like they were for our manual feature engineering
  - → This is part of the process of **transfer learning** and **fine-tuning**
- Using a good representation is more sample efficient because data ends up used in fitting $h_\theta$ instead of jointly finding $f_\theta = h_\theta \circ \phi_\theta$
- Maybe problems which are complicated and nonlinear in $\mathcal{X}$ are simpler in $\mathcal{Z}$ (i.e., linear regression in $\mathcal{Z}$)
- Above all: good representations overfit less and **generalize better**

## Jointly Learning Representations and Functions?

- Because $f_\theta \equiv h_\theta \circ \phi_\theta$, the simplest approach is just to jointly learn both
- Come up with some hypothesis class $\mathcal{H}(\Theta)$ that flexible enough to have both the representation and function of interest
- Extra flexibility could overfit (i.e., $\varepsilon_{\text{gen}}$ could increase even if $\varepsilon_{\text{app}} \searrow 0$)
  - → Hence the crucial need for regularization in various forms
- Notice the nested structure here of $h_\theta \circ \phi_\theta$
  - → Hints at why Neural Networks might work so well

## Is this a Mixture of Supervised and Unsupervised?

- Remember that we talked about finding representations as intrinsic to the data itself, and hence "unsupervised"
  - → Fitting $h_\theta \circ \phi_\theta$ jointly combines supervised and unsupervised
- Nested structure means we may be able to use this for new problems
  - → Isolate the parameters of $\phi_\theta$ from those of $h_\theta$
  - → Train $h_\theta \circ \phi_\theta$ for a new $h_\theta$ and fit jointly again or **freeze** $\phi_\theta$
  - → Can work shockingly well in practice!
  - → What do Neural Networks Learn When Trained on Random Labels?
- Best to start with existing representations and **fine-tune** (essential in LLMs)

## Neural Networks

## Neural Networks Include Almost Everything

- **Neural Networks** as just an especially flexible functional form
  - → Linear, polynomials, and all of our standard approximation fit in there
- However, when we say "Neural Network" you should have in mind approximations that are typically
  - → parameterized by some complicated: $\theta \equiv \{\theta_1, \theta_2, \cdots, \theta_L\}$
  - → nested with **layers**: $y = f_1(f_2(\ldots f_L(x;\theta_L);\theta_2);\theta_1) \equiv f(x;\theta)$
  - → **highly overparameterized**, such that $\theta$ is often massive, often far larger than the amount of data

## Reminder: MLP Architecture (see Differentiation Lecture)

- The term “neural network” is very broad and covers a variety of approximation classes, with a high degree of flexibility.
- While there is no theoretical requirement to use these patterns, in practice hardware is optimized to make them fast.
- For example, consider the function $f : \mathbb{R}^N \to \mathbb{R}^M$ defined as

$$y = f_\theta(x) = W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3$$

  → $W_1 \in \mathbb{R}^{K\times N}, W_2 \in \mathbb{R}^{K,\times K}, W_3 \in \mathbb{R}^{M\times K}, b_1 \in \mathbb{R}^K, b_2 \in \mathbb{R}^K, b_3 \in \mathbb{R}^M$

  → $\sigma(\cdot)$ is a scalar non-linear function applied componentwise

  → Combine into a set of parameters $\theta \equiv \{W_1, W_2, W_3, b_1, b_2, b_3\}$

## Neural Network Terminology

- Remember: there is no single design! Different architectures for different problems
- **Depth** $L$: number of layers (e.g., $L = 3$ for $W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3$)
- **Width** $K$: dimension of each hidden layer (size of $W_\ell$)
  - → Layers need not all have the same width, but often do for simplicity
- **Activation function** $\sigma(\cdot)$: nonlinear function applied componentwise
  - → Common choices: $\mathbf{tanh(\cdot)}$, $\mathbf{max(0, \cdot)}$ (ReLU), GELU
  - → See ProbML Book 1 Section 13.2.4 for numerical properties of gradients
- **Parameters** $\theta \equiv \{W_1, b_1, \ldots, W_L, b_L\}$: the collection of all learnable weights and biases
- Alternate linear/nonlinear transformations, ending with a linear layer (or one matching the output domain)

## (Asymptotic) Universal Approximation Theorems

- At this point you may expect a theorem that says that neural networks are **universal approximators**. e.g., Cybenko, Hornik, and others
  - → i.e., for any function $f^*(x)$, there exists a neural network $f_\theta(x)$ for some $\mathcal{H}(\Theta)$ that can approximate it arbitrarily well
  - → Takes limits of the number of parameters at each layer (e.g. $\theta_\ell \nearrow$), or sometimes the number of layers (e.g. $L \nearrow$)
- A low bar to pass that rarely gives useful guidance or bounds
  - → We do not use enormously "wide" approximations
  - → The theorems are too pessimistic, as NNs do much better in practice
  - → Important when doing core functional analysis and asymptotics

## How Can we Fit Huge Approximations?

- At this point you should not be scared off by a big $\theta$
- The ML and deep-learning revolution is built on ideas you have covered
  - → Scalar loss $\mathcal{L}(\theta)$, use reverse-mode AD to get gradients $\nabla_\theta \mathcal{L}(\theta)$.
  - → AD software like PyTorch or JAX makes it easy to collect the $\theta$ and run the AD on complicated $\mathcal{L}(\theta)$
  - → Hardware (e.g. GPUs) can make common operations like VJPs fast
  - → Optimization methods like SGD work great
  - → When gradient memory is an issue, for finite-sum objectives, can use $\nabla_\theta \mathcal{L}(\theta) \approx \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} \nabla_\theta \ell(\theta; x)$
  - → Regularization, in all its forms, helps when used appropriately
- But that doesn't explain why this can help generalization performance?

## Puzzling Empirical Success, Ahead of Theory

- Deep learning and AD are old, but only recently have the software and hardware been good enough to scale
- Bias-variance tradeoff: adding parameters can make things worse
  - → But ML practitioners often find the opposite **empirically**
  - → Frequent success with lots of layers and massive over-parameterization
  - → This seems counter to all sorts of basic statistics intuition
- ML theory is still catching up to the empirical success
  - → We will try to give some perspectives on **when/why deep learning works**
  - → First, we will have to be precise on what we mean by “works”

## Rough Intuition on The Success of Deep Learning

- The broadest intuition on why deep learning with flexible, overparameterized neural networks often works well is a combination of:
    1. The massive number of parameters makes the optimization process find more generalizable solutions (sometimes through regularization)
    2. The depth of approximations allowing for better representations
    3. The optimization process seems to be able to learn those representations
    4. The representations are reusable across problems
    5. Regularization (implicit and explicit) helps us avoid overfitting
- Art, not magic
    - → Need to design architectures ($\mathcal{H}$), optimization, and regularization

## Many Problem Specific Variations

- For examples on why multiple layers help see: Mark Schmidt's CPSC 440, CPSC340, ProbML Book 1 Section 13.2.1 on the XOR Problem and 13.2.5-13.2.7 for more
- Use economic intuition and problem specific knowledge to design $\mathcal{H}$
  - → Encode knowledge of good representations, easier learning
- For example, you can approximation function $f : \mathbb{R}^N \to \mathbb{R}$ which are symmetric in arguments (i.e. permutation invariance) with $\rho : \mathbb{R}^M \to \mathbb{R}$, $\phi : \mathbb{R} \to \mathbb{R}^M$

$$f(X) = \rho\left(\frac{1}{N}\sum_{x\in X}\phi(x)\right)$$

- See Probabilistic Symmetries and Invariant Neural Networks or Exploiting Symmetry in High Dimensional Dynamic Programming

## Transfer Learning and Few-Shot Learning

- If you have a good representation, you can often use it for new problems
- Take the $\theta$ as an initial condition for the optimizer and **fine-tune** on the new problem
- e.g. take $\phi \equiv f_1 \circ f_2 \circ \ldots f_{L-1}$ (e.g. the all but the last layer) and **freeze them** only changing the last $f_L$ layer with training
  - → Or only freeze some of them
  - → May work well even if the task was completely different. Many $y$ $are simply linear combinations of the disentangled representations - part of why kernel methods (which we will discuss in a future lecture) work well
- Can find sometimes “one-shot”, “few-shot”, or “zero-shot” learning
  - → e.g. Zero-Shot Learning for image classification

## More Perspectives on Representations

## The Tip of the Iceberg

- The ideas sketched out previously are just the beginning
- This section points out a few important ideas and directions for you to explore on your own

## Autoencoders and Representations

- One unsupervised approach is to consider **autoencoders**. See ProbML Book 1 Section 20.3, ProbML Book 2 Section 21, and CPSC 440 Notes
- Consider connection between representations and compression
- Let $\phi : \mathcal{X} \to \mathcal{Z}$ be an **encoder** and $h : \mathcal{Z} \to \mathcal{X}$ be a **decoder**, parameterized by some $\theta_d$ and $\theta_e$ then we want to find the empirical equivalent to

$$\min_{\theta_e,\theta_d} \mathbb{E}_{x\sim\mu^*}(h(\phi(x;\theta_e);\theta_d) - x)^2 + \text{regularizer}$$

- Whether the $\phi(x;\theta_e)$ is a good representation depends on whether "compression" of the information of $x$ is useful for downstream tasks

# Manifold Hypothesis

- Are there always simple, reusable representations?
- On perspective is called the **Manifold Hypothesis**, see **ProbML Book 1** Section 20.4.2
- The basic idea is that even the most complicated data sources in the real world ultimately are concentrated on a low-dimensional manifold
  - → including images, text, networks, high-dimensional time-series, etc.
  - → i.e., the world is much simpler than it appears if you find the right transformation
- See **ProbML Book 1** Section 20.4.2.4 describes Manifold Learning, which tries to learn the geometry of these manifolds through variations on nonlinear dimension reduction

## Embeddings

- A related idea is to **embed** data into a different (sometimes lower-dimensional) space
- For example, text or images or networks, mapped into $\mathbb{R}^M$
- Whether it is lower or higher dimensional, the key is to preserve some geometric properties, typically a norm. i.e., $||x - y|| \approx ||\phi(x) - \phi(y)||$
- You can think of learned representations within the inside of neural networks as often doing embedding in some form, especially if you use an information bottleneck
- See ProbML Book 1 Section 20.5 and 23 for examples with text and networks

## Lottery Tickets and Representations

- The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks
- The idea is that inside of every huge, overparameterized approximation with random initialization and layers is a small sparse approximation
  - → You can show this by pruning most of the parameters and training it
- Perhaps huge, overparameterized functions with many layers are more likely to contain the lottery ticket
  - → Then the optimization methods are just especially good at finding them
  - → Ex-post you could prune, but ex-ante you don't know the sparsity structure

## Disentangling Representations

- Representations that separate out different factors of variation into separate variables are often easier to use for downstream tasks and are more transferrable
- The keyword in a literature review is to look for is **disentangled representations**
- It seems like this is hard to do **without supervision**
- But **semi-supervised** approaches seem to be a good approach

## Out of Distribution Learning

- We have been discussing a $\mathcal{D}$ from some idealized, constant $\mu^*$
- What if the distribution changes, or our samples are not IID. e.g., in control and reinforcement learning applications where the distribution depends on $f$?
- See ProbML Book 2 Section 19.1 for more
- This is called "robustness to distribution shift", covariate shift, etc. in different settings
- There are many methods, but one common element is:
  - → Models with better "generalization" tends to perform much better under distribution shift
  - → Good representations are much more robust
  - → With transfer learning you may be able to adapt very easily

## Differentiating Representations

## Recursive Parameterizations

- $\mathcal{H}$ design is crucial, and we can use economic intuition to guide
- If we are using ``deep" learning - where the representations themselves must be learned, then there will recursively defined, nested approximations
  - → Each of these approximations has its own parameters
  - → When a gradient is taken in the optimizing process, it is taken with respect to that recursive set of parameters
- ML frameworks like Pytorch, Keras, and Flax NNX help keep track of those nested parameters, update gradients, etc.

## Flax NNX Example

- We previously saw how to create our own `MyLinear` case for a linear function without an affine term (i.e., no “bias”)
- The “learnable” (i.e, differentiable) parameters are tagged with `nnx.Param`

```python
class MyLinear(nnx.Module):
    def __init__(self, in_size, out_size, rngs):
        self.out_size = out_size
        self.in_size = in_size
        self.kernel = nnx.Param(jax.random.normal(rngs(), (self.out_size, self.in_size)))
    # Similar to Pytorch's forward
    def __call__(self, x):
        return self.kernel @ x
```

## Differentiation With Respect to `nnx.Module`

- Differentiating `MyLinear` it finds the `nnx.Param`.
- The return type of `nnx.grad` does not perturb `out_size`, etc.

```python
def f_gen(m, x, b):
    return jnp.squeeze(m(x) + b)
rngs = nnx.Rngs(42)
N = 3
m = MyLinear(N, 1, rngs = rngs)
x = jax.random.normal(rngs(), (N,))
b = 1.0
f = lambda m: f_gen(m, x, b)
nnx.grad(f)(m)
```

```
State({
  'kernel': Param( # 3 (12 B)
    value=Array([[ 0.60576403,  0.7990441 , -0.908927  ]],
dtype=float32)
  )
})
```

## Nesting

- This recursively goes through any `nnx.Module`
- Using the built-in `nnx.Linear` instead, construct a simple NN

```
class MyMLP(nnx.Module):
    def __init__(self, din, dout, width: int, *, rngs: nnx.Rngs):
        self.width = width
        self.linear1 = nnx.Linear(din, width, use_bias = False, rngs=rngs)
        self.linear2 = nnx.Linear(width, dout, use_bias = True, rngs=rngs)

    def __call__(self, x: jax.Array):
        x = self.linear1(x)
        x = nnx.relu(x)
        x = self.linear2(x)
        return x
m = MyMLP(N, 1, 2, rngs = rngs)
```

## Reverse-mode AD

- Consider the scalar function $f(m)$ where $f : \mathcal{H} \to \mathbb{R}$
- Recall notation from the **Differentiation Lecture**
- For reverse mode, we propagate cotangents:

$$\bar{m} = \partial f(m)^\top[\bar{f}] = \bar{f} \cdot \nabla f(m)$$

- For scalar $f$, the natural cotangent seed is $\bar{f} = 1$, giving $\bar{m} = \nabla f(m)$
  - → This is what `jax.grad` does for scalar functions
- `nnx.grad` does this recursively through each `nnx.Module` and its `nnx.Param` values

## Splitting into Differentiable Parts

```python
graphdef, state = nnx.split(m)
print(state)
```

```
State({
  'linear1': {
    'kernel': Param( # 6 (24 B)
      value=Array([[ 0.2700633 ,  0.36592388],
             [-0.6980436 , -0.23368652],
             [-0.1246624 , -1.0047528 ]], dtype=float32)
    )
  },
  'linear2': {
    'bias': Param( # 1 (4 B)
      value=Array([0.], dtype=float32)
    ),
    'kernel': Param( # 2 (8 B)
      value=Array([[-0.21603853],
             [-1.0095432 ]], dtype=float32)
    )
  }
})
```

## `graphdef` Contains Fixed Values and Metadata

```
GraphDef(nodes=[NodeDef(
  type='MyMLP',
  index=0,
  outer_index=None,
  num_attributes=5,
  metadata=MyMLP
), NodeDef(
  type='GenericPytree',
  index=None,
  outer_index=None,
  num_attributes=0,
  metadata=({}, PyTreeDef(CustomNode(PytreeState[(False, False)], [])))
), NodeDef(
  type='Linear',
  index=1,
  outer_index=None,
  num_attributes=13,
  metadata=Linear
), NodeDef(
  type='GenericPytree',
  index=None,
  outer_index=None,
  num_attributes=0,
  metadata=({}, PyTreeDef(CustomNode(PytreeState[(False, False)], [])))
), NodeDef(
```

## Differentiation with Nested `nnx.Module`

```python
m = MyMLP(N, 1, 2, rngs = rngs)
delta_m = nnx.grad(f)(m)
delta_m
```

```
State({
  'linear1': {
    'kernel': Param( # 6 (24 B)
      value=Array([[ 0.,  0.],
             [ 0.,  0.],
             [-0., -0.]], dtype=float32)
    )
  },
  'linear2': {
    'bias': Param( # 1 (4 B)
      value=Array([1.], dtype=float32)
    ),
    'kernel': Param( # 2 (8 B)
      value=Array([[0.],
             [0.]], dtype=float32)
    )
  }
})
```

## Interpreting the “Adjoint”

- This gradient is a recursive structure for a change in the object linearized around the value $m$
- Note that this is a $\delta m$ for the state, and ignores the `graphdef`
- Crucially, we need to do the operation like $m - \eta\,\delta m$ to update the parameters, keeping fixed all of the non-differentiable parts
- The optimization process with find the update wrt the state, apply it, and then reconstruct the new `m` by applying the graphdef

## Split and Merge

- In that case, we can update the `state` from before, and make a new type using the `graphdef`

```python
eta = 0.01 # e.g., a gradient descent update

# jax.tree.map recursively goes through the model
# Updates the underlying nnx.Param given the delta_m grad
new_state = jax.tree.map(
  lambda p, g: p - eta*g,
  state, delta_m) # new_state = state - eta * delta_m
m_new = nnx.merge(graphdef, new_state)
f(m_new)
```

```
Array(0.0327667, dtype=float32)
```

## Manually Splitting and Merging

- The `nnx.jit, nnx.vmap, nnx.grad` will automatically split and merge for you (i.e., filtering) on `nnx.Module` types as arguments, then call underlying JAX functions
  - → Otherwise the `nnx.grad` etc. would not work without modification since the NN combine differentiable and non-differentiable parts
- There is some overhead. You can manually split into the `state` and `graphdef` and then merge them back together
- Next we will show that process with a different wrapper for the `f_gen` function

## Functions of the `state`

```python
graphdef, state = nnx.split(m)
@jax.jit # note jax.jit instead of nnx.jit
def f_split(state): # closure on graphdef
    m = nnx.merge(graphdef, state)
    return f_gen(m, x, b)
# Can use jax.grad, rather than nnx.grad
state_diff = jax.grad(f_split)(state)
print(state_diff)
new_state = jax.tree.map(
            lambda p, g: p - eta*g,
            state, delta_m)
m_new = nnx.merge(graphdef, new_state)
f(m_new)
```

```
State({
  'linear1': {
    'kernel': Param( # 6 (24 B)
      value=Array([[ 0.,  0.],
             [ 0.,  0.],
             [-0., -0.]], dtype=float32)
    )
  },
  'linear2': {
    'bias': Param( # 1 (4 B)
      value=Array([1.], dtype=float32)
    ),
    'kernel': Param( # 2 (8 B)
      value=Array([[0.],
             [0.]], dtype=float32)
```

## Pytorch (and Functorch)

- The gradients of structures in JAX are more explicit, as the functions are generally pure (i.e., side-effect free)
- Pytorch, on the other hand, implicitly does these same operations in the background.
  - → This can lead to concise code, but can also lead to confusion
- torch.func is a Torch library intended for more JAX-style function composition

## Pytorch Custom Types

- The equivalent to `nnx.Param` in Pytorch is `torch.nn.Parameter`

```python
class MyLinearTorch(nn.Module):
    def __init__(self, in_size, out_size):
        super(MyLinearTorch, self).__init__()
        self.out_size = out_size
        self.in_size = in_size
        self.kernel = nn.Parameter(torch.randn(out_size, in_size))
        # Similar to PyTorch's forward
    def forward(self, x):
        return self.kernel @ x
def f_gen_torch(m, x, b):
    return torch.squeeze(m(x) + b)
```

## Differentiating Modules in Pytorch

```python
m = MyLinearTorch(N, 1)
x_torch = torch.randn(N)
b_torch = torch.tensor(1.0)
def f_torch(m):
  return f_gen_torch(m,
         x_torch, b_torch)

# Clone the model
m_new = copy.deepcopy(m)
m_new.zero_grad()
output = f_torch(m_new)
output.backward()
print(m_new.kernel)
```

```
Parameter containing:
tensor([[0.5495, 1.6847, 0.1938]], requires_grad=True)
```

## Differentiating Custom Types

```python
class MyMLPTorch(nn.Module):
    def __init__(self, din, dout, width):
        super(MyMLPTorch, self).__init__()
        self.width = width
        self.linear1 = nn.Linear(din, width, bias=False)
        self.linear2 = nn.Linear(width, dout, bias=True)

    def forward(self, x):
        x = self.linear1(x)
        x = torch.relu(x)
        x = self.linear2(x)
        return x
```

## Reverse-mode AD Updates in Pytorch

```python
m = MyMLPTorch(N, 1, 2)
m.zero_grad()
output = f_torch(m)

# Start with d output = [1.0]
output.backward()
# Now `m` has the gradients

# Manually update parameters recursively
# Done in-place, as torch optimizers will do
with torch.no_grad():
    # Recursively
    for param in m.parameters():
        param -= eta * param.grad
    for name, param in m.named_parameters():
        print(f"{name}: {param.numpy()}")
```

```
linear1.weight: [[-0.15995035 -0.04328426  0.27121732]
 [-0.01335035 -0.17065024  0.4135141 ]]
linear2.weight: [[-0.4177399  -0.45026302]]
linear2.bias: [-0.39413318]
```

## Hyperparameter Optimization with Deep Learning

## Optimization is Challenging

- When using complicated representations, we often have many fixed values (e.g. the `width`) of the representations, as well as tweaks to the optimizer and algorithms
- Collectively we call the values not found through differentiation of the representations themselves **hyperparameters**
- When possible, you should use optimizers to find these in a systematic way (i.e., differentiating the solution to the optimizer itself)

  → But often you need to do this manually through a combination of brute force, intuition, and experience

## Commandline Arguments

- Given that you may want to solve your problem with a variety of different hyperparameters, possibly running in parallel, you need a convenient way to pass the values and see the results
- One model, framework, OS independent way to do this is to use commandline arguments
- For example, if you have a python file called `mlp_regression_jax_nnx_logging.py` that accepts arguments for the width and learning rate, you may call it with

```
python mlp_regression_jax_nnx_logging.py --width=128 --lr=0.01
```

## CLI Packages

- Many python frameworks exist to help you take CLI and convert to calling python functions. One convenient tool isjsonargparse
- Advantage: simply annotate a function with defaults, and it will generate the CLI

```python
import jsonargparse
def fit_model(width: int = 128, lr: float = 0.01): # etc.
    # ... your code...

if __name__ == "__main__":
    jsonargparse.CLI(fit_model)
```

- Then can call `python mlp_regression_jax_nnx_logging.py`, `python mlp_regression_jax_nnx_logging.py --width=64` etc.

## Logging

- While the CLI file could save output for later interpretation, a common approach in ML is to log results to visualize the optimization process, compare results, etc.
- One package for this is **Weights and Biases**
- This will log into a website calculations, usually organized by a project name, and let you sort hyperparameters, etc.
- To use this, setup an account and then add code to initialiize in your python file, then log intermediate results

```python
def fit_model(width: int = 128, lr: float = 0.01): # etc.
    wandb.init(project="mlp_test", mode="online")
    # ... your code...
    wandb.log({"test_loss": test_loss}) # etc.  Maybe per epoch
```

## HPO "Sweeps" with Weights and Biases

- Putting together the logging and the CLI, you can setup a process to run the code with a variety of parameters in a "sweep" (e.g., with sweep file)

```
wandb sweep mlp_regression_jax_nnx_sweep.yaml
```

→ Returns a sweep ID, which you can then run with `wandb agent <sweep_id>`

→ Which calls function given instructions (i.e., polling architecture) `python mlp_regression_jax_nnx_logging.py --width=128 --lr=0.0015` etc.

→ Benefit: run with same `<sweep_id>` on multiple computers/processes/etc.

## Example Sweep File

- Sets up a Bayesian optimization process on `lr` and `width` to minimize `test_loss` (if logged with `wandb.log({"test_loss": test_loss})`, etc.)

```yaml
program: lectures/examples/mlp_regression_jax_nnx_logging.py
name: Sweep Example
method: bayes
metric:
  name: test_loss
  goal: minimize
parameters:
  num_epochs:
    value: 300 # fixed for all calls
  lr: # uniformly distributed
    min: 0.0001
    max: 0.01
  width: # discrete values to optimize over
    values: [64, 128, 256]
```

## MLP Regression Example with Sweep

## MLP Regression

- See examples/mlp_regression_jax_nnx_logging.py for an example of a simple neural network optimized to fit a linear DGP
- The $\mathcal{H}$ is a simple nesting of `nnx.Linear` and `nnx.relu` layers

```python
class MyMLP(nnx.Module):
    def __init__(self, din: int, dout: int, width: int, *, rngs: nnx.Rngs):
        self.linear1 = nnx.Linear(din, width, rngs=rngs)
        self.linear2 = nnx.Linear(width, width, rngs=rngs)
        self.linear3 = nnx.Linear(width, dout, rngs=rngs)

    def __call__(self, x: jax.Array):
        x = self.linear1(x)
        x = nnx.relu(x)
        x = self.linear2(x)
        x = nnx.relu(x)
        x = self.linear3(x)
        return x
```

## Data Generating Process and Loss

- This randomly generates a $\theta$ and then generates data with

```python
theta = random.normal(rngs(), (M,))
X = random.normal(rngs(), (N, M))
Y = X @ theta + sigma * random.normal(rngs(), (N,))  # Adding noise
```

- ERM: with $m \in \mathcal{H}$, minimize the residuals for batch `(X, Y)`

```python
def residual(m, x, y):
    y_hat = m(x)
    return (y_hat - y) ** 2
def residuals_loss(m, X, Y):
    return jnp.mean(jax.vmap(residual, in_axes=(None, 0, 0))(m, X, Y))
```

- Optimizer uses loss differentiated wrt $m$ as discussed

## CLI

- In order to use `jsonargparse`, this creates a function signature with defaults

```python
def fit_model(
    N: int = 500,
    M: int = 2,
    sigma: float = 0.0001,
    width: int = 128,
    lr: float = 0.001,
    num_epochs: int = 2000,
    batch_size: int = 512,
    seed: int = 42,
    wandb_project: str = "grad_econ_ML",
    wandb_mode: str = "offline",  # "online", "disabled
):
# ... generate data, fit model, save test_loss
```

## Sweep File

```
program: lectures/examples/mlp_regression_jax_nnx_logging.py
name: Sweep Example
method: bayes
metric:
  name: test_loss
  goal: minimize
parameters:
  wandb_mode:
    value: online # otherwise won't log
  num_epochs:
    value: 300
  lr:
    min: 0.0001
    max: 0.01
  width:
    values: [64, 128, 256]
```

## Run and Visualize the Sweep

- To run this sweep, you can run the following command (checking the relative location file)

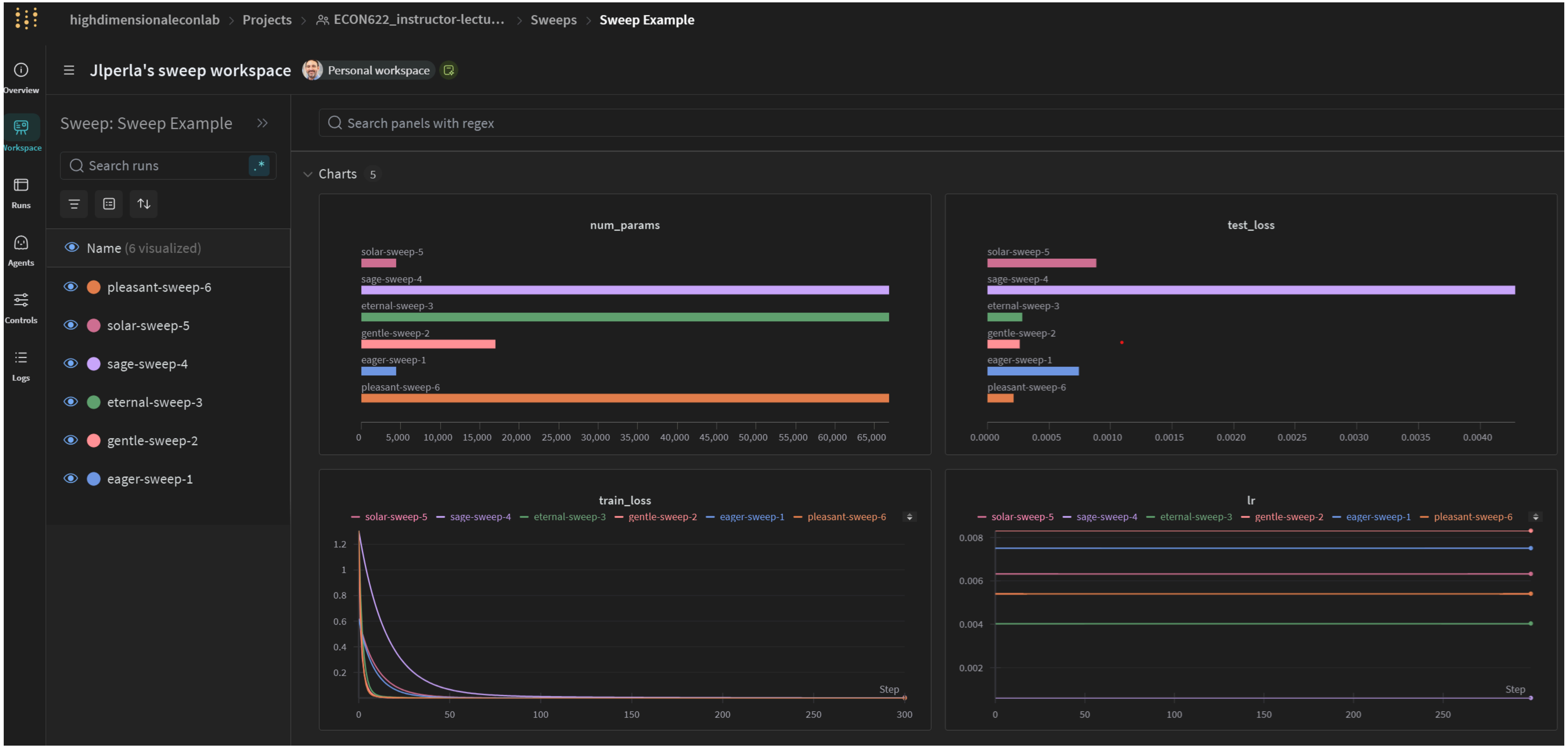
```
wandb sweep lectures/examples/mlp_regression_jax_nnx_sweep.yaml
```

- The output should be something along the lines of

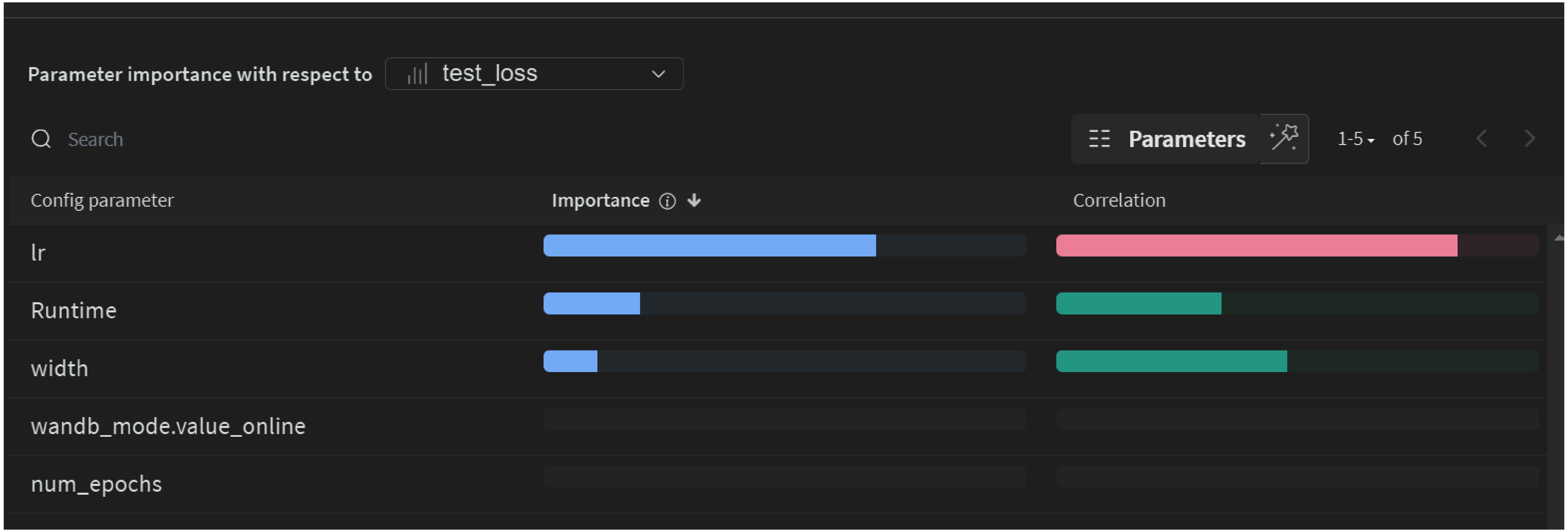
```
(grad-econ-ml) /Users/username/GitHub/grad_econ_ML>wandb sweep lectures/examples/mlp_regression_jax_nnx_sweep.yaml
wandb: Creating sweep from: lectures/examples/mlp_regression_jax_nnx_sweep.yaml
wandb: Creating sweep with ID: virfdcn6
wandb: View sweep at: https://wandb.ai/highdimensionaleconlab/grad_econ_ML-lectures_examples/sweeps/virfdcn6
wandb: Run sweep agent with: wandb agent highdimensionaleconlab/grad_econ_ML-lectures_examples/virfdcn6
```

- Run `wandb agent highdimensionaleconlab/grad_econ_ML-lectures_examples/virfdcn6` and go to web to see results in progress

# General Sweep Results



# Correlations of Parameters to HPO Objective



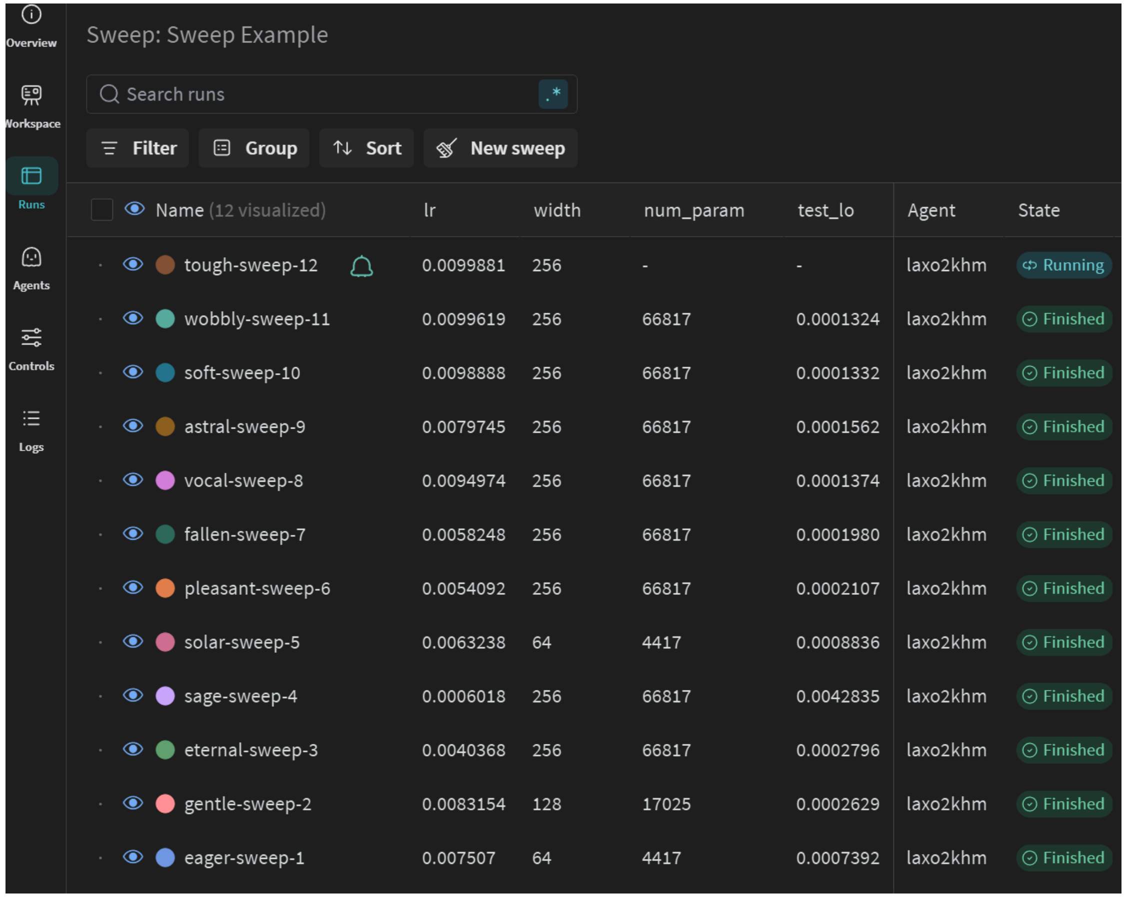
Parameter importance with respect to test_loss

Search

Parameters 1-5 of 5

| Config parameter | Importance | Correlation |
|---|---|---|
| lr | | |
| Runtime | | |
| width | | |
| wandb_mode.value_online | | |
| num_epochs | | |

# Sortable Table of Results



Sweep: Sweep Example

| Name (12 visualized) | lr | width | num_param | test_lo | Agent | State |
|---|---|---|---|---|---|---|
| tough-sweep-12 | 0.0099881 | 256 | - | - | laxo2khm | Running |
| wobbly-sweep-11 | 0.0099619 | 256 | 66817 | 0.0001324 | laxo2khm | Finished |
| soft-sweep-10 | 0.0098888 | 256 | 66817 | 0.0001332 | laxo2khm | Finished |
| astral-sweep-9 | 0.0079745 | 256 | 66817 | 0.0001562 | laxo2khm | Finished |
| vocal-sweep-8 | 0.0094974 | 256 | 66817 | 0.0001374 | laxo2khm | Finished |
| fallen-sweep-7 | 0.0058248 | 256 | 66817 | 0.0001980 | laxo2khm | Finished |
| pleasant-sweep-6 | 0.0054092 | 256 | 66817 | 0.0002107 | laxo2khm | Finished |
| solar-sweep-5 | 0.0063238 | 64 | 4417 | 0.0008836 | laxo2khm | Finished |
| sage-sweep-4 | 0.0006018 | 256 | 66817 | 0.0042835 | laxo2khm | Finished |
| eternal-sweep-3 | 0.0040368 | 256 | 66817 | 0.0002796 | laxo2khm | Finished |
| gentle-sweep-2 | 0.0083154 | 128 | 17025 | 0.0002629 | laxo2khm | Finished |
| eager-sweep-1 | 0.007507 | 64 | 4417 | 0.0007392 | laxo2khm | Finished |

Bottou, Léon, and Olivier Bousquet. 2007. "The Tradeoffs of Large Scale Learning." *Advances in Neural Information Processing Systems* 20.

Murphy, Kevin P. 2022. *Probabilistic Machine Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press. https://github.com/probml/pml-book.