2nd Lecture, Optimization

Advanced Machine Learning

Miroslav Čepek, Zdeněk Buk, Rodrigo da Silva Alves, Vojtěch Rybář, **Petr Šimánek**

FIT CTU

28. 2. 2024

,

What do we need for successful training of neural networks?

- Pretty easy, right?

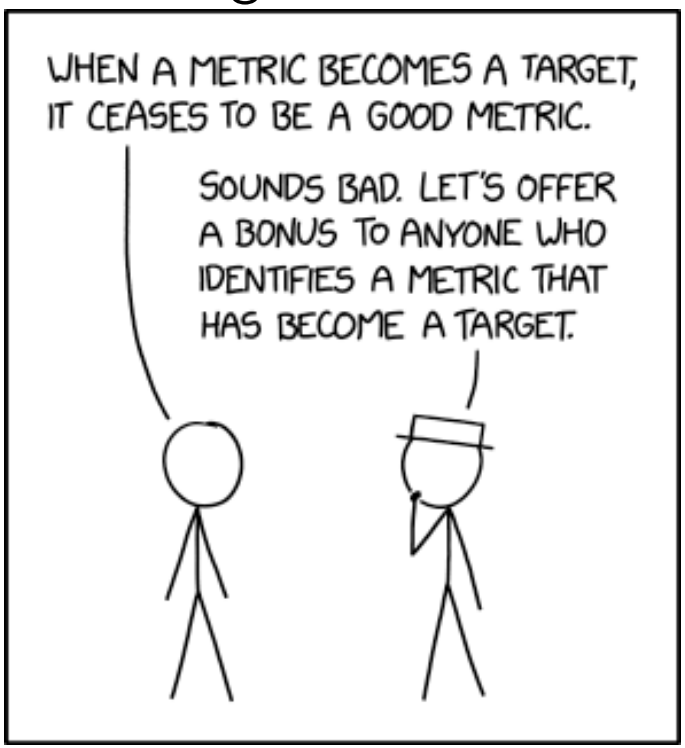
What do we do to get the best accuracy/MSE?

Not so easy anymore:

- Better data, more data, augmentations,
- Choose the best architecture, CNNs, transformers, NEAT, and infinite possibilities,
- Optimization! Which method and parameters? And why?
- Choosing the right hyper-parameters.

Goodhart's law

A general law, overfitting is one instance, how to fight it?



- Early stopping, regularization, great data and loss function, adding noise, small net or XXL net

Is training NNs that simple?

The boundary between convergence and divergence is fractal even for extremely small networks.
https://sohl-dickstein.github.io/2024/02/12/fractal.html

What do we want from the optimization algorithm:?

- converge sometimes/most of the time
- converge with most initial weights
- converge fast
- generalize well
- be robust to small perturbations to the system
- small memory requirements
- no need to tune hyperparameters

Stochastic Gradient Descent

Deep neural net $f : \mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$.

$$f(x) = g_N \circ g_{N-1} \circ \cdots \circ g_1(x).$$

Where

$$g_j(x) = \sigma(W_j x + b_j), W_j \in \mathbb{R}^{d_x \times d_{j-1}}, b_j \in \mathbb{R}^{d_j}.$$

σ is the activation function (component-wise).

Supervised learning with loss L (Empirical risk minimization)

Stochastic Gradient Descent

Most common algorithm: Initialization: $W_1^0, \ldots, W_N^0$ Iterate for k $k = 0, 1, 2, \ldots$

$$W_j^{k+1} = W_j^k - \eta_k \nabla_{W_j} L(W_1^k, \ldots, W_N^k), j = 1, \ldots, N.$$

With some step sizes $\eta_0, \eta_1, \ldots$

Stochastic Gradient Descent

What do we want from the optimization algorithm:

- converge most of the time
- converge with "most" initial weights
- converge fast
- generalize well
- be robust to small perturbations to the system
- small memory requirements compared to what?
- no need to tune hyper-parameters

Generalization of SGD

We don't know for sure why the SGD-trained NN generalize. The analysis is difficult for even simple networks.

- Implicit bias
- Implicit network simplification

Implicit Regularization of SGD

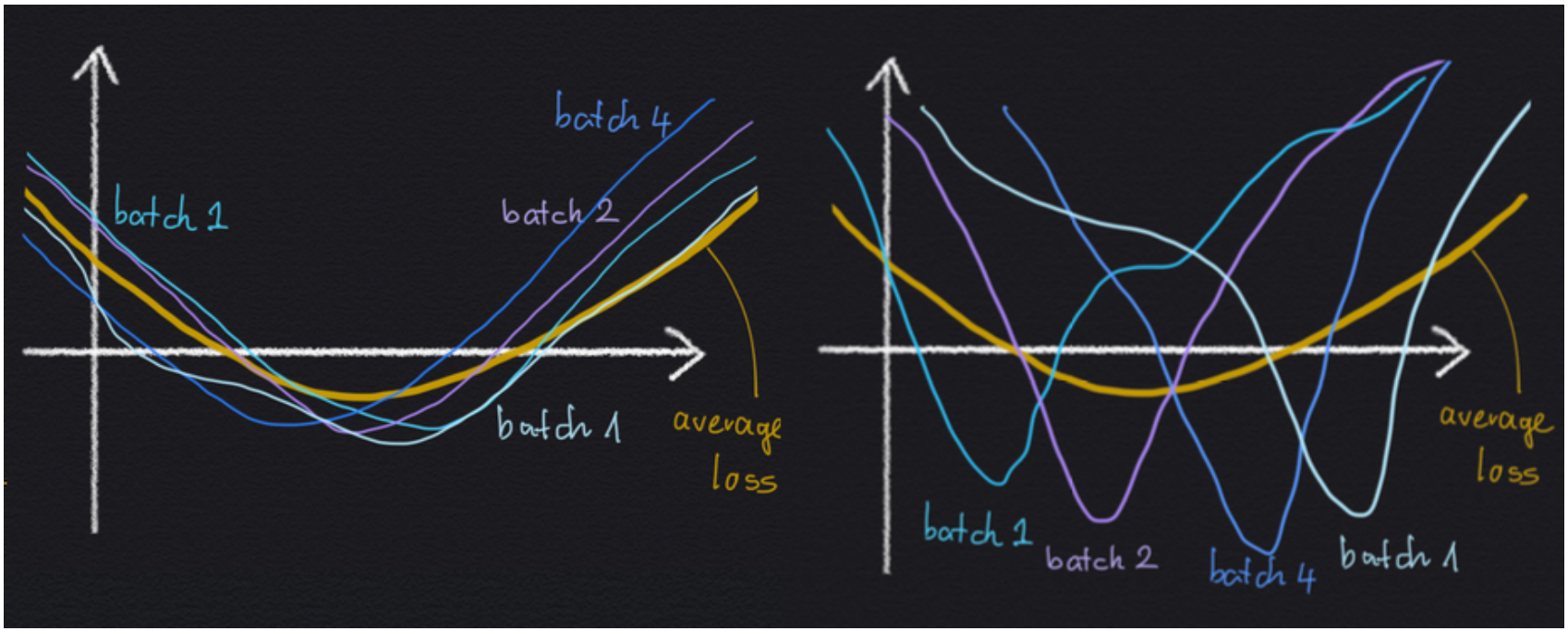
- Implicit gradient norm regularization, (Barrett and Dherin, 2020)
- Noise anisotropy leads to minima not affected by this noise, not true for Gaussian (Kunin, 2020, HaoChen)
- Prefering flat minima (Li et al.)
- Escape from sharp minima by smoothing (Zhu)

- Gradient descent finds a minimum that also minimizes not only the loss function but also its gradient.
- GD optimizes a slightly different function than we expected.
- SGD minimizes also the "variance" of mini-batch gradients.

Smith, 2021, On the Origin of Implicit Regularization in Stochastic Gradient Descent, link.

Implicit Regularization of SGD

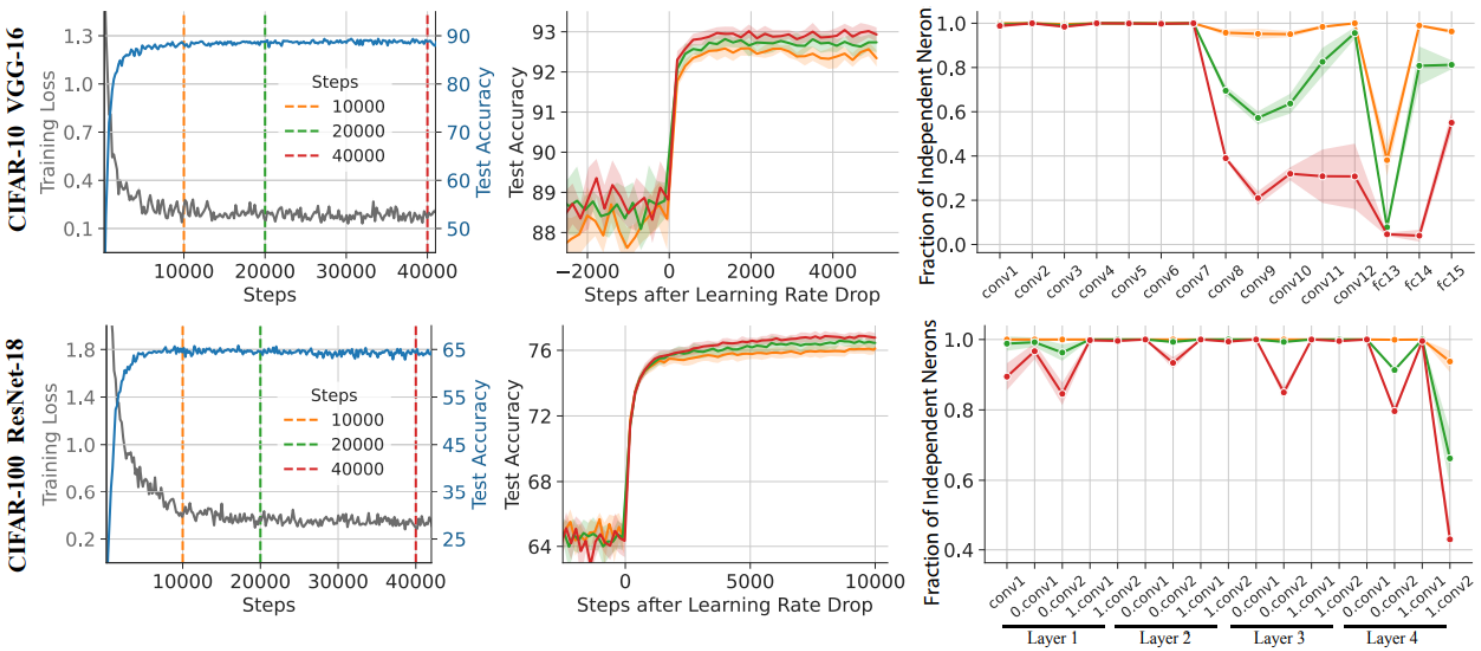
Why?



Implicit Simplification

In larger networks, SGD finds sub-networks that are much simpler, sparser, or low-rank.

- SGD tries to find low-rank solution
- Large learning rates in the beginning improve generalization!
- If effectively finds the best subnetwork for the task.



Stochastic Collapse, Chen, 2023

Generalization of SGD

CIFAR-10, 50k samples, trained with SGD

Architecture	n of parameters	Training loss	Test accuracy
MLP	1.2M	0	51%
Alexnet	1.4M	0	77%
Inception	1.65M	0	86%
Resnet	9M	0	88%

- Over-parameterized networks work better! - counter-intuitive
- No overfitting?
- This is not the case with many algorithms!
- Not understood well yet.

Zhang, 2017, Understanding Deep Learning Requires Rethinking Generalization.

Double Descent

Conventional thinking:

- Larger models are better.
- More data is better.
- Early stopping is good.

Model-wise/epoch-wise/sample-wise double descent.

Nakkiran, 2019, Deep Double Descent: Where Bigger Models and More Data Hurt.

Double Descent Overview

- Classic U-shaped bias-variance tradeoff is extended by double descent.
- Occurs when model complexity surpasses a critical point.

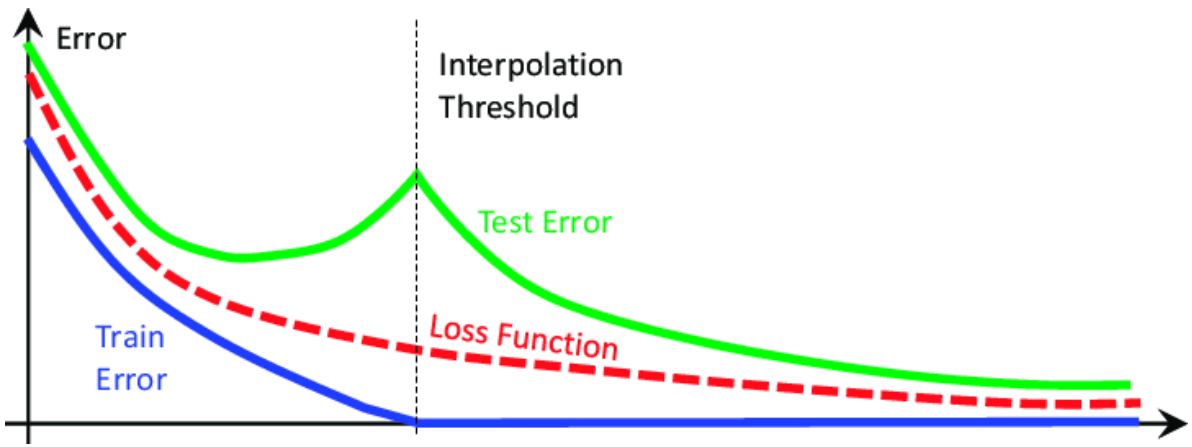


Figure: Double Descent Curve

Key Points of Double Descent

1. **Pre-critical regime:** Performance worsens as complexity increases.
2. **Critical point:** Overfitting peaks, then loss starts to decrease.
3. **Post-critical regime:** Adding more parameters improves performance, despite overparameterization.

Practical Implications

- **Overparameterization:** Large models can still generalize well, challenging traditional bias-variance tradeoff views.
- **Regularization:** Importance of techniques like dropout or weight decay to navigate the peak at interpolation threshold.
- **Data Augmentation:** More data can shift the interpolation threshold, helping mitigate the peak error.
- **Model Selection:** Careful choice between model complexity and training data size to optimize generalization.

Some "improvements" to SGD

- Momentum
- Adaptive Gradient

Momentum

1. Initialize the parameters W_j and set the initial velocity $V_j^0 = 0$ for each parameter $j = 1, \ldots, N$.
2. For each iteration $k = 0, 1, 2, \ldots$:
   - 2.1 Compute the gradient of the loss function with respect to each parameter W_j:

$$\nabla_{W_j} L(W_1^k, \ldots, W_N^k), \quad j = 1, \ldots, N.$$

   - 2.2 Update the velocity for each parameter W_j using the momentum coefficient μ and the current gradient:

$$V_j^{k+1} = \mu V_j^k + \eta_k \nabla_{W_j} L(W_1^k, \ldots, W_N^k), \quad j = 1, \ldots, N.$$

   - 2.3 Update the parameter W_j by subtracting the updated velocity:

$$W_j^{k+1} = W_j^k - V_j^{k+1}, \quad j = 1, \ldots, N.$$

Here, η_k is the learning rate at iteration k, μ is the momentum term that helps to accelerate SGD in the relevant direction and dampens oscillations, and N is the number of parameters.

Adaptive gradient - AdaGrad

1. Initialize the parameters W_j and a small constant ϵ. Initialize the gradient accumulation term $G_j^0 = 0$ for each parameter $j = 1, \ldots, N$.
2. For each iteration $k = 0, 1, 2, \ldots$:
   2.1 Compute the gradients:

$$\nabla_{W_j} L(W_1^k, \ldots, W_N^k), \quad j = 1, \ldots, N.$$

   2.2 Accumulate the squared gradients for each parameter W_j:

$$G_j^{k+1} = G_j^k + (\nabla_{W_j} L(W_1^k, \ldots, W_N^k))^2, \quad j = 1, \ldots, N.$$

   2.3 Update each parameter W_j using the accumulated gradient, adjusting the learning rate for each parameter inversely based on the square root of G_j^{k+1}:

$$W_j^{k+1} = W_j^k - \frac{\eta}{\sqrt{G_j^{k+1}} + \epsilon} \nabla_{W_j} L(W_1^k, \ldots, W_N^k), \quad j = 1, \ldots, N.$$

Momentum and Adaptive gradient

AdaGrad's has the ability to perform smaller updates (i.e., lower learning rates) for parameters associated with frequently occurring features, and larger updates (i.e., higher learning rates).
Momentum speeds up convergence.

- Often added to SGD.
- The goal is "not" to smooth convergence!
- SGD + Momentum oscillates too.
- Momentum allows large batches.
- Adaptive gradient scales the gradients.
- Adam = momentum + adaptive gradient with the second moment (scales the gradient by gradient variance).

Adam vs AdamW

- Adam often generalizes less than SGD. Why?
- Using Adam can hinder L2 regularization.
- The reason: Even the regularization term is "normalized".
- Solution: AdamW.

Standard goto algorithms

- AdamW/Adam/SGD + Nesterov Momentum for machine vision and transformers
- AdamW for NLP
- AdaGrad for recommenders

2nd Order methods

- Minimization of 2nd order local approximation.
- Interesting also to inspect your model.
- BackPACK tool for computing Hessians and interesting other quantities.

Dangel, 2020, BackPACK: Packing more into Backprop

2nd order methods

What do we want from the optimization algorithm:

- converge most of the time
- converge with most initial weights
- converge fast
- generalize well - we don't know!
- be robust to small perturbations to the system - we don't know!
- small memory requirements
- no need to tune hyperparameters

Introduction to AdaHessian

- AdaHessian is a second-order optimization algorithm that adapts the learning rate based on the curvature of the loss landscape, utilizing the Hessian matrix's diagonal elements.
- Unlike first-order methods (e.g., Adam, SGD) that rely solely on gradients, AdaHessian incorporates second-order information, leading to potentially better convergence properties and efficiency in training deep neural networks.
- Key Innovation: Efficient computation of the Hessian's diagonal to adjust the learning rates, making it feasible for large-scale deep learning tasks.

Key Features of AdaHessian

- **Adaptive Learning Rates:** Uses the second-order information from the Hessian matrix's diagonal to adapt learning rates for each parameter dynamically.
- **Efficiency:** Proposes an efficient way to approximate the Hessian's diagonal, making the computation scalable to large models and datasets.
- **Improved Convergence:** By leveraging the curvature information, AdaHessian can achieve faster and potentially more stable convergence, especially in complex optimization landscapes.
- **Robustness:** Demonstrates improved generalization and robustness across a variety of tasks and architectures compared to first-order optimizers.

Comparing AdaHessian with Other Optimizers

- **Second vs. First Order:** AdaHessian incorporates second-order information, providing a deeper insight into the loss landscape compared to first-order methods like Adam and SGD.
- **Adaptive Learning Rate Adjustment:** Unlike Adam, which adjusts learning rates based on first-order moments, AdaHessian uses the curvature of the loss function for more informed adjustments.
- **Performance and Efficiency:** While more computationally intensive than first-order methods, AdaHessian's efficient Hessian approximation techniques make it competitive in terms of computational overhead and performance.
- **Use Cases:** Particularly beneficial in settings where the curvature of the loss landscape plays a critical role in optimization dynamics, such as in ill-conditioned problems.

Credit Assignment without Backpropagation

- Can we train without a backdrop?
- Backprop is very expensive and hard to parallelize
- Backprop is not biologically plausible - there is no such feedback in the brain
- Biologically-motivated:
  - asynchronous updating of weights at different layers of a network
  - reduced memory costs from having to store intermediate layer activation values
  - reduced synaptic wiring in the feedback path

The resulting computational efficiencies can be particularly great on neuromorphic hardware, where forward and backward network weights are represented by physically separate wiring on a circuit.

Forward Gradient

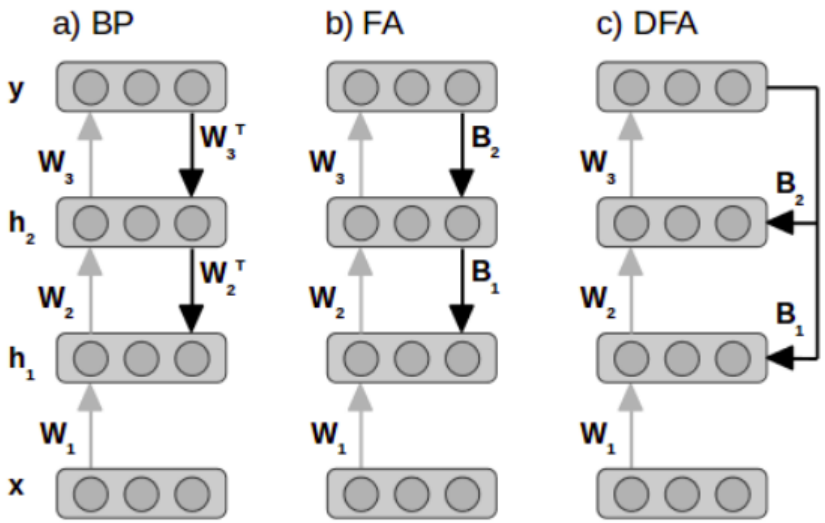
- We can estimate the gradient in forward mode.
- The forward gradient is an unbiased estimation of the standard gradient.
- Can be much faster than GD.

Baydin, 2022, Gradients without Backpropagation

Direct Feedback Alignment

Alignment Provides Learning in Deep Neural Networks the gradient of the last layer is computed and is distributed to all previous layers.

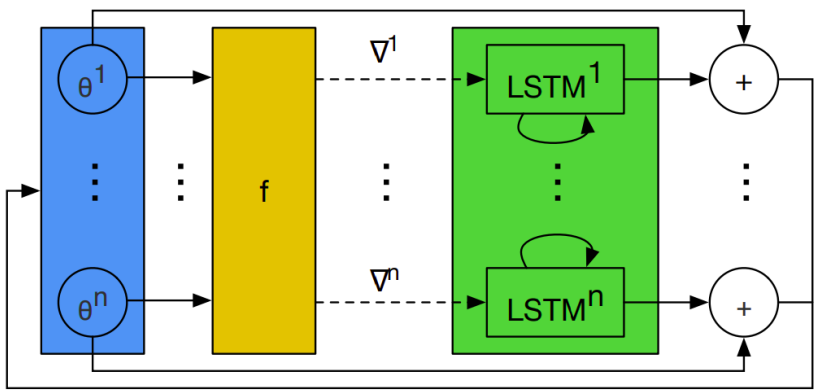
Can be used to solve real-life problems efficiently!



Nokland, 2016, Direct Feedback Alignment Provides Learning in Deep Neural Networks

Learning to Optimize

- Use meta-learning to find some "optimization algorithm".
- Two loops - inner loop optimizes a function, outer loop optimizes an optimizer (LSTM)



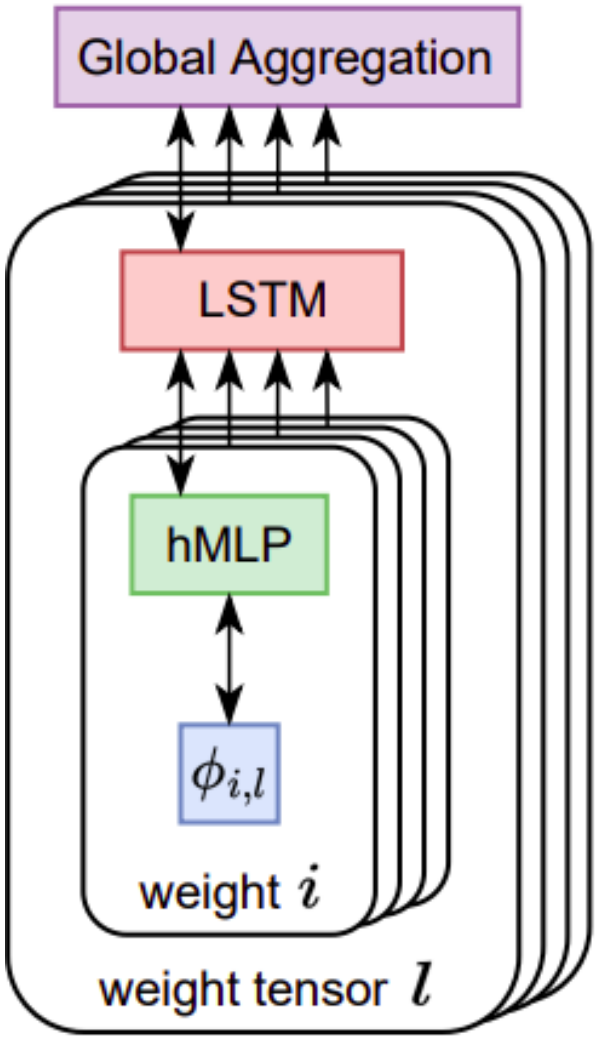
VeLO

First actually useful learned optimizer!

- Trained to solve many different optimization problems
- Uses hypernetworks
- Each hypernetwork ingests multiple features:
  - Exponential moving averages of the gradient and squared gradient
  - Mean and variance of weights and gradients
  - Training stage (info about training process).

Metz, 2022, VeLO: Training Versatile Learned Optimizers by Scaling Up.

VeLO


Global Aggregation
LSTM
hMLP
$\phi_{i,l}$
weight i
weight tensor l

VeLO

- Works very well for "smaller" networks (less than 500M)
- Allows much larger batches (10x)
- VeLO learns implicit learning rate scheduling
- Adapts to training horizon
- 2x memory overhead
- Fails after 200k iterations
- Sometimes fails out of distribution

LION

- Symbolic program assembly takes 45 common operations from numpy
- The program can access usual information - weight, gradient, learning step + some open variables
- Uses an evolutionary algorithm to create new optimization algos
- Uses many tricks - removal by wrong syntax, warm-start (AdamW)
- Funneling process to allow only the most promising algos to go from proxy tasks to large real problems
- 512 TPUs for days!

LION

Lion algorithm:

- Uniform updates to all weights! Adds a lot of noise $\rightarrow$ generalization
- Faster/less memory than AdamW, Adam, and adafactor. And often better.

Chen, 2023, Symbolic Discovery of Optimization Algorithms

Basic Methods

- Grid Search
- Random Search (preferred over grid)
- Evolution Search (CMA-ES)

Bayesian Optimization BO

Principle:

- Global optimization of black-box functions that are expensive to evaluate
- Relies on building a probabilistic model (surrogate) of the objective function, which is then used to make predictions about where in the parameter space good hyper-parameter values are likely to be found.

Surrogate Model: Gaussian Process (GP)
Acquisition Function: Balances exploration and exploitation - Expected Improvement (EI), Probability of Improvement (PI), and Upper Confidence Bound (UCB).

HyperBand

- **Principle**: Hyperband is an optimization algorithm that accelerates random search through adaptive resource allocation and early-stopping.
- **Resource Allocation**: Hyperband evaluates a large number of configurations with a small amount of resources and incrementally allocates more resources to promising configurations in successive rounds.
- **Early Stopping**: This feature allows Hyperband to terminate poor-performing configurations early.

In practice:

For small problems use Random (use Optuna). For large, use BOHB (hpbandster).