GSAS @NIDA : Deep Learning

MLP 1

Assoc.Prof.Thitirat Siriborvornratanakul, Ph.D.

Email: thitirat@as.nida.ac.th
Website: http://as.nida.ac.th/~thitirat/

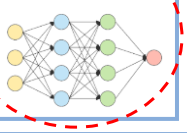
1

DL models in this course

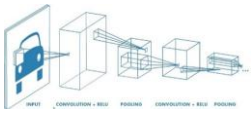
FUNDAMENTAL

~~DISCRIMINATIVE~~

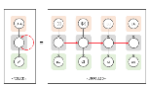
MLP



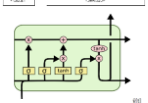
CNN



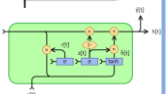
RNN



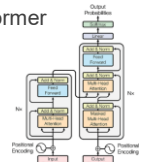
LSTM



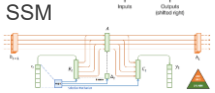
GRU



Transformer

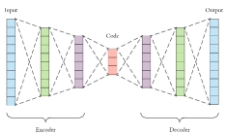


SSM

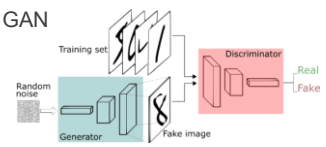


GENERATIVE

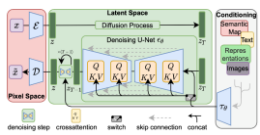
VAE



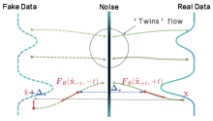
GAN



Diffusion



Flow matching

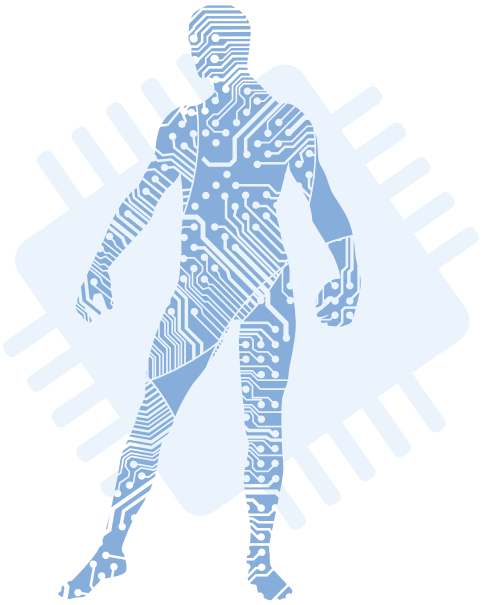


ChatGPT

2

3

OUTLINE



01 Forward Pass
Learn MLP's components relating to the forward propagation

02 Backward Pass
Learn MLP's components relating to the backward propagation

03 Coding
Combine everything and implement a program with PyTorch

4

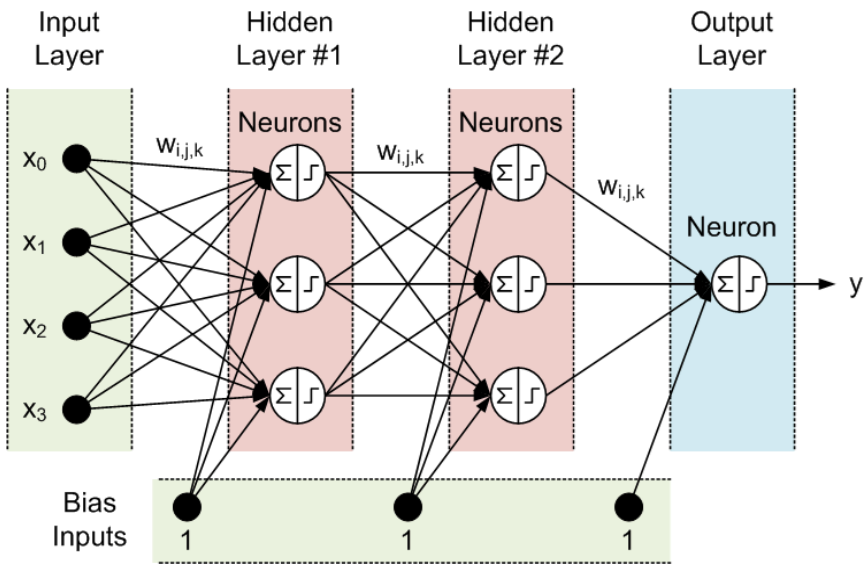
FWD Pass

Learn MLP's components relating to the forward propagation

5

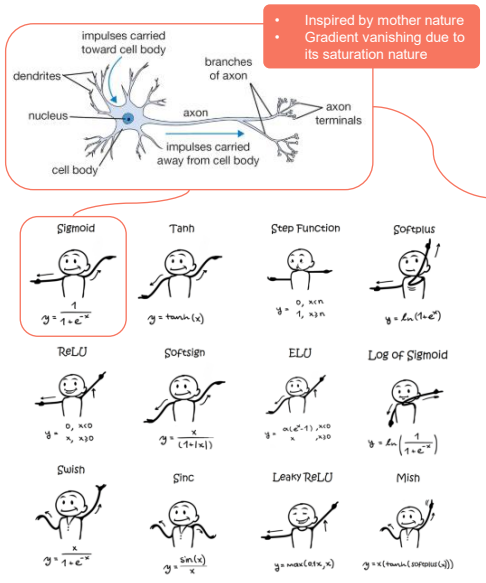
MULTI-LAYER PERCEPTRON

- **Layer**
  - Input layer
  - Output layer
  - Hidden layer
- **Neuron (node)**
  - Linear operation
  - Activation function
- **Trainable parameter**
  - Weight
  - Bias (in TensorFlow and PyTorch, there is one bias value per neuron)



6

ACTIVATION FUNCTION



- Inspired by mother nature
- Gradient vanishing due to its saturation nature

Name	Plot	Equation	Derivative
Identity		$f(x) = x$ Equal to no activation function	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

7

ACTIVATION FUNCTION

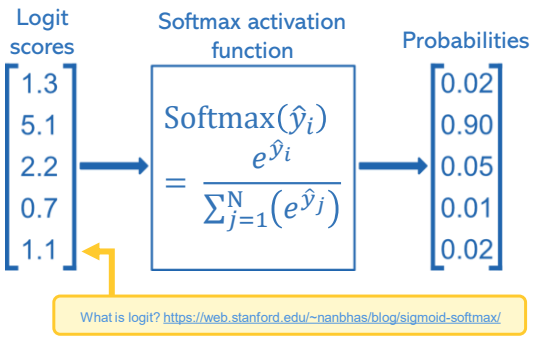
- Two ways to use activation functions in :
  - **The module-based API (`torch.nn`):** used when defining network architectures in classes, as this is better for model printing and summaries. For example, use `nn.Sigmoid()` as a layer within `nn.Sequential()` blocks.
  - **The functional API (`torch.nn.functional`):** used for direct mathematical operations on tensors (e.g., `F.sigmoid(x)`). Best for use inside the forward method.

Activation	Module call (`nn.`)	Functional call (`F.`)	Common use case
Sigmoid	`nn.Sigmoid()`	`F.sigmoid()`	Binary classification (output layer)
ReLU	`nn.ReLU()`	`F.relu()`	Hidden layers (standard choice)
Leaky ReLU	`nn.LeakyReLU()`	`F.leaky_relu()`	Fixing “dying ReLU” issues
Tanh	`nn.Tanh()`	`F.tanh()`	Scaling outputs between -1 and 1
Softmax	`nn.Softmax(dim)`	`F.softmax(dim)`	Multi-class classification (output layer)
ELU	`nn.ELU()`	`F.elu()`	Avoiding vanishing gradients; smooth curves
PReLU	`nn.PReLU()`	`F.prelu()`	ReLU with learnable negative slope
GELU	`nn.GELU()`	`F.gelu()`	Transformers and BERT architectures
Softplus	`nn.Softplus()`	`F.softplus()`	Smooth approximation of ReLU

8

ACTIVATION FUNCTION

- Softmax activation function:
  - It is popular as an activation function for the output layer in a multi-class classification task (N output nodes). Use of softmax in a hidden layer is possible but less common.
  - Softmax function is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one. Each value in the output of the softmax function is interpreted as the probability of membership for each class.
  - Softmax can be thought of as a probabilistic or "softer" version of the argmax function as it is a smoother version of the winner-takes-all activation model in which the unit with the largest input has output +1 while all other units have output 0.
  - Softmax classifier predicts only one class at a time (multi-class not multi-output), so it should be used only with mutually exclusive classes.



19OCT2020: https://machinelearningmastery.com/softmax-activation-function-with-python/

9

ACTIVATION FUNCTION

- Softmax activation function in :
  - In PyTorch, we usually DON'T put Softmax in the model.
  - **Numerical stability:** Log-Softmax is more stable than Softmax + Log.
    - If we calculate e^x (Softmax) and then immediately take the log (Cross Entropy), we risk "exploding" or "vanishing" numbers. PyTorch uses a mathematical shortcut called the **Log-Sum-Exp trick** to compute them together safely.

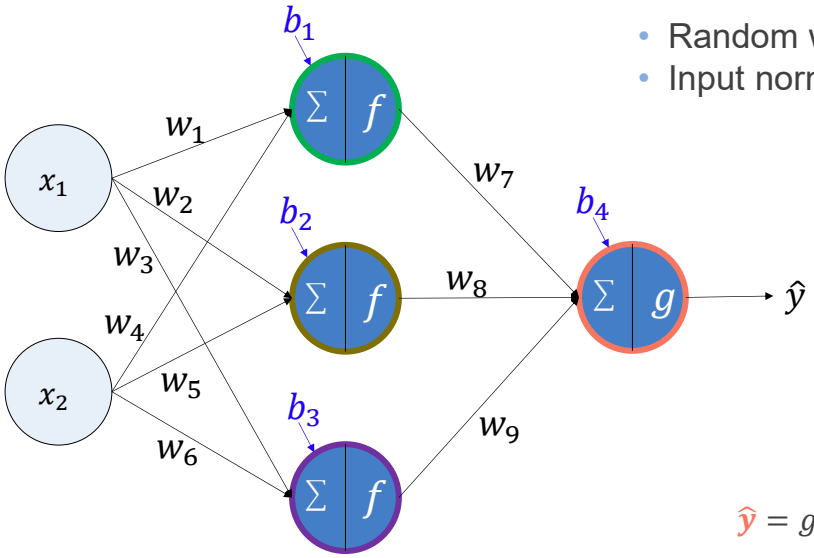
      K [Linear Layer] → [Softmax Layer] → **Probabilities** → [Loss Function]

      [Linear Layer] → **Logits** → [`nn.CrossEntropyLoss` (LogSoftmax + NLLLoss)]

    - Hence, if we use `nn.CrossEntropyLoss` in PyTorch, our model should output raw scores (logits). If we add a Softmax layer at the end of the model AND use CrossEntropyLoss, we are calculating Softmax twice, which will hurt our model's performance.
  - When to use the Softmax layer in PyTorch?
    - When using a different loss function that doesn't include it (like `nn.NLLLoss`).
    - In Inference mode (after training), when we need actual probability scores (e.g., "98% Cat") to show the user.

10

FORWARD PROPAGATION



- Random weight/bias initialization
- Input normalization/standardization

$$\boldsymbol{h_1} = f(w_1 x_1 + w_4\, x_2 + b_1)$$
$$\boldsymbol{h_2} = f(w_2 x_1 + w_5\, x_2 + b_2)$$
$$\boldsymbol{h_3} = f(w_3 x_1 + w_6 x_2 + b_3)$$
$$\hat{\boldsymbol{y}} = g(w_7 \boldsymbol{h_1} + w_8\, \boldsymbol{h_2} + w_9\, \boldsymbol{h_3} + b_4)$$

11

CODE REVIEW

`class torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)` [source]

Applies an affine linear transformation to the incoming data: $y = xA^T + b$.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Parameters:

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{\text{in}})$ where $*$ means any number of dimensions including none and $H_{\text{in}} = \text{in\_features}$.
- Output: $(*, H_{\text{out}})$ where all but the last dimension are the same shape as the input and $H_{\text{out}} = \text{out\_features}$.

Synonyms:

- Dense layer (Keras, TensorFlow)
- Linear layer (PyTorch)
- Fully-connected layer

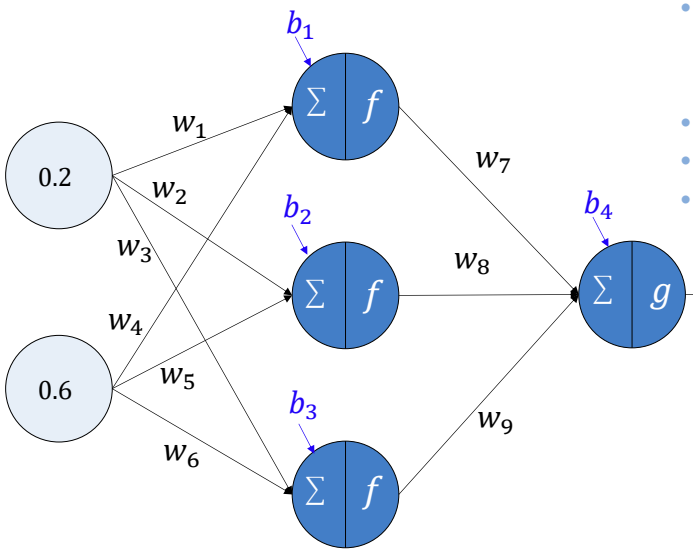
12

BKWD Pass

Learn MLP's components relating to the backward propagation

13

BACKWARD PROPAGATION (1986)



- Gradient Descent
  - Objective/Cost/Loss function
  - Partial derivation and chain rule
- Mini-batch gradient descent
- Epoch/Iteration
- Optimizer

$\hat{y} = 0.3$

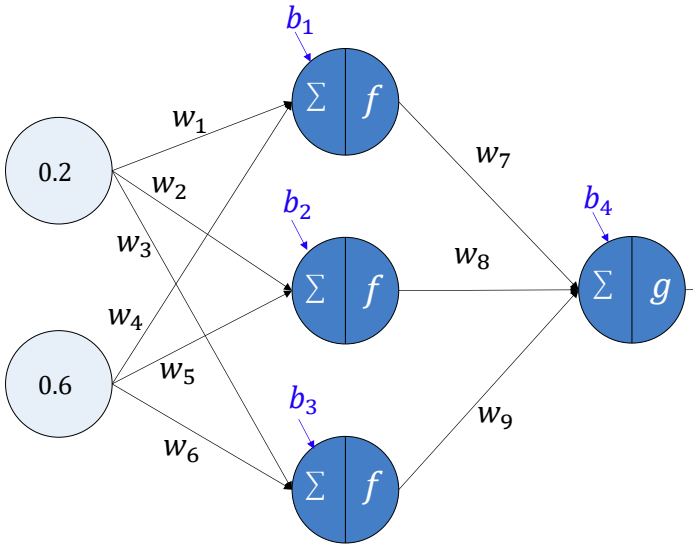
Incorrect prediction

Training dataset

#	x_1	x_2	y
1	0.2	0.6	1.0
2	0.8	0.4	0.0
3	0.5	0.5	0.3

14

OBJECTIVE / COST / LOSS



- Objective function
- Cost function (often an average of the loss across the training set)
- Loss function (one training sample)

Predicted value

$\hat{y} = 0.3$

$y = 1.0$

Target value (ground truth, label)

Use a loss function to measure an error regarding one training sample

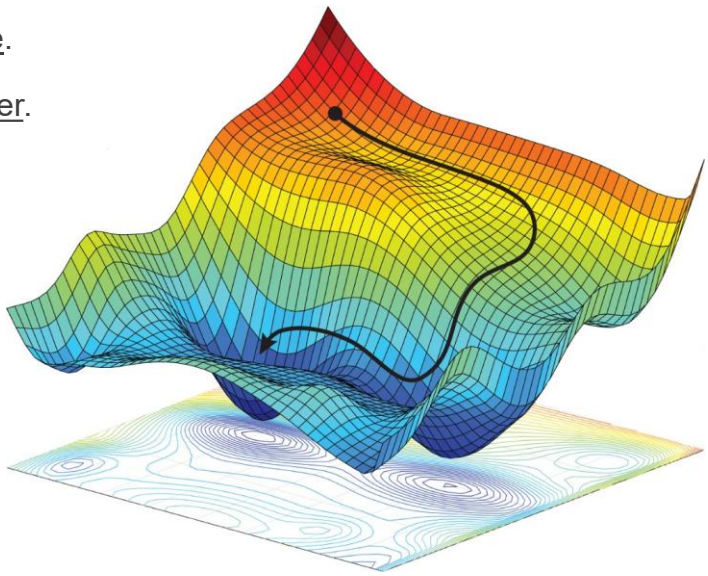
Training dataset

#	x_1	x_2	y
1	0.2	0.6	1.0
2	0.8	0.4	0.0
3	0.5	0.5	0.3

15

OBJECTIVE / COST / LOSS

- Loss function must be differentiable.
- The smaller the loss value, the better.
  - In supervised DL, most standard loss functions are non-negative. Hence, zero loss represents a perfect prediction.
  - Negative loss is mathematically possible in specific fields (like RL), in a standard DL, a negative loss usually suggests a bug in codes.
- Real-world DL loss surfaces are highly non-convex (complex valleys).



16

Can't we optimize the accuracy?

As a mathematical function, we can find the derivative (gradient) of accuracy. However, the derivative is useless for optimization.

1. **The zero-gradient problem:**
   - Accuracy is a step function.
   - Where it is flat, the derivative is exactly 0 (Optimizer: "I don't see a slope. I guess I'll just stay exactly where I am"). Where it jumps, the derivative is undefined (or infinite).

2. **Information density:** The smooth loss function provides a continuous signal that tells the model it is getting warmer or colder, even before it gets a single answer "correct."
   - Accuracy: "You got 0% right." (Provides no hint on how to improve).
   - Cross-Entropy: "You are 0% right, but your 'confidence' in the wrong answer was 0.9. If you nudge your weights this way, that confidence drops to 0.8."

Only attempt to use non-smooth functions for optimization as a last choice. Even then, we usually find a way to smooth it out first.

17

Can't we optimize the accuracy?

As a mathematical function, we can find t… derivative is useless for optimization.

1. **The zero-gradient problem:**
   - Accuracy is a step function.
   - Where it is flat, the derivative is ex… where I am"). Where it jumps, the deri…

2. **Information density:** The smooth loss… it is getting warmer or colder, even bef…
   - Accuracy: "You got 0% right." (Provide…
   - Cross-Entropy: "You are 0% right, but … your weights this way, that confidence…

Only attempt to use non-smooth functio… usually find a way to smooth it out first.

1. **The binary nature of correctness:** In DL, a model outputs a continuous probability (e.g., 0.72), but accuracy requires a hard decision.
   - **The Threshold:** Usually, if the output is ≥0.5, we call it "Class 1." If it's 0.499, we call it "Class 0."
   - **The Jump:** A tiny change in a weight might move the output from 0.499 to 0.501. For the loss function (like CE), this is a small, smooth change. But for accuracy, the model just jumped from "Wrong" to "Right."

2. **The counting problem:** Accuracy is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Predictions}}$$

   - Because the "Number of Correct Predictions" must be an integer (1, 2, 3, …), the accuracy can only change in fixed increments.
   - For example, in a dataset of 100 items, accuracy can only be 70%, 71%, 72%, etc. There is no such thing as 70.5% accuracy if you have 100 samples.

18

Task	PyTorch Class	Equation	Input Activation	Label Type /Shape	Usage Warning
Regression (Standard)	`nn.MSELoss()`	$\frac{1}{\text{output size}} \sum_{i=1}^{\text{output size}} (y_i - \hat{y}_i)^2$	**None** (Linear)	`float32` (Same shape as output)	• Great for clean data. • But outliers will pull the model's attention too much due to the squaring.
Regression (Robust)	`nn.L1Loss()`	$\frac{1}{\text{output size}} \sum_{i=1}^{\text{output size}} \lvert y_i - \hat{y}_i \rvert$	**None** (Linear)	`float32`	• Less sensitive to extreme values than MSE because it doesn't square the error.
Binary Class	`nn.BCELoss()` 	$-\sum_{i=1}^{\text{output size}} [y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$			• Used internally by `BCEWithLogitsLoss`. • RuntimeError, if there is no `Sigmoid` layer explicitly at the end of the model.
Binary Class	`nn.BCEWithLogitsLoss()`	`Sigmoid + nn.BCELoss`	**None** (Logits)	`float32` (Usually `[Batch, 1]`)	• Do not add a `Sigmoid` layer to the model. `BCELoss` applies it internally for better numerical stability.
Multi-Class	`nn.NLLLoss()` NLL: Negative Log Likelihood 	$-\sum_{i=1}^{\text{output size}} (y_i \cdot \log \hat{y}_i)$			• Used internally by `CrossEntropyLoss`. • `NLLLoss` expects Log-Probabilities. Values are typically negative, where 0 is the maximum probability. • Without a `LogSoftmax` layer at the end of the model, `NLLLoss` becomes mathematically invalid, leading to a silent failure where the model stops learnings.
Multi-Class	`nn.CrossEntropyLoss()`	`nn.LogSoftmax + nn.NLLLoss`	**None** (Logits)	`long` (Integer Indices: 0, 1, 2...)	• No Softmax allowed. This loss already combines `LogSoftmax` and `NLLLoss`. • By default, expect class indices, not one-hot vectors.
Multi-Label	`nn.BCEWithLogitsLoss()`	`Sigmoid + nn.BCELoss`	**None** (Logits)	`float32` (Multi-hot: [0, 1, 1, 0])	• Used when a sample can belong to multiple categories at once. • Each output neuron is treated as an independent binary choice.

19

Cross-Entropy Loss

- **Cross-Entropy (CE) Loss:**
  - Measure the difference between two sets of two or more values that sum to 1.0.
  - Measure the difference between a correct probability distribution and a predicted distribution.

$$-\sum_{i=1}^{n\_classes} [y_i * \log(\hat{y}_i)]$$

#No.	y (ground truth)	$\hat{y}$ (prediction)	Correct?
1	(0.2, 0.5, 0.3)	(0.1, 0.3, 0.6)	No
2	(0.5, 0.4, 0.1)	(0.5, 0.3, 0.2)	Yes
3	(1, 0, 0)	(0.3, 0.6, 0.1)	No
4	(0, 0, 1)	(0.3, 0.3, 0.4)	Yes

$$\begin{aligned}\#1 &= -[0.2 * \log(0.1) + 0.5 * \log(0.3) + 0.3 * \log(0.6)] \\ &= -[(0.2 * -2.30) + (0.5 * -1.20) + (0.3 * -0.51)] \\ &= -[(-0.46) + (-0.60) + (-0.15)] \\ &= 1.21\end{aligned}$$

$$\begin{aligned}\#2 &= -[0.5 * \log(0.5) + 0.4 * \log(0.3) + 0.1 * \log(0.2)] \\ &= -[(0.5 * -0.69) + (0.4 * -1.20) + (0.1 * -1.61)] \\ &= -[(-0.35) + (-0.48) + (-0.16)] \\ &= 0.99\end{aligned}$$

$$\begin{aligned}\#3 &= -[1 * \log(0.3) + 0 * \log(0.6) + 0 * \log(0.1)] \\ &= -[(1 * -1.20) + 0 + 0] \\ &= 1.20\end{aligned}$$

$$\begin{aligned}\#4 &= -[0 * \log(0.3) + 0 * \log(0.3) + 1 * \log(0.4)] \\ &= -[0 + 0 + (1 * -0.92)] \\ &= 0.92\end{aligned}$$

20

Cross-Entropy Loss

- **Cross-Entropy (CE) Loss for binary classification:**

$$-\sum_{i=1}^{n\_classes} [y_i * \log(\hat{y}_i)]$$

#No.	y (ground truth)	$\hat{y}$ (prediction)	Correct?
1	(1, 0)	(0.4, 0.6)	No
2	(0, 1)	(0.3, 0.7)	Yes
3	(0, 1)	(0.8, 0.2)	No

$$\#1 = -[1 * \log(0.4) + 0 * \log(0.6)] = -[(1 * -0.92) + 0] = 0.92$$

$$\#2 = -[0 * \log(0.3) + 1 * \log(0.7)] = -[0 + (-0.36)] = 0.36$$

$$\#3 = -[0 * \log(0.8) + 1 * \log(0.2)] = -[0 + (-1.61)] = 1.61$$

- **Binary Cross-Entropy (BCE) Loss:**
  - In the context of binary classification, knowing y and $\hat{y}$ of one class automatically implies the values of y and $\hat{y}$ of the other class. Hence, there is no need to encode as pairs of values.
  - The CE equation can be reduced to the following equation, when y and $\hat{y}$ refer to probabilities of the positive class. See that when $y = 1$ (positive class) the first term is added to our loss whereas when $y = 0$ (negative class) the second term is instead added.

$$(y)(-\log(\hat{y})) + (1 - y)(-\log(1 - \hat{y}))$$

21

Why not MSE loss for classification?

Network 1

#No.	y (ground truth)	$\hat{y}$ (prediction)	Correct?		CE Loss	MSE Loss
1	(0, 0, 1)	(0.3, 0.3, 0.4)	Yes	⇒	0.92	0.54
2	(0, 1, 0)	(0.3, 0.4, 0.3)	Yes	⇒	0.92	0.54
3	(1, 0, 0)	(0.1, 0.2, 0.7)	No	⇒	2.30	1.34
				AVG	**1.38**	**0.81**

Network 2

#No.	y (ground truth)	$\hat{y}$ (prediction)	Correct?		CE Loss	MSE Loss
1	(0, 0, 1)	(0.1, 0.2, 0.7)	Yes	⇒	0.36	0.14
2	(0, 1, 0)	(0.1, 0.7, 0.2)	Yes	⇒	0.36	0.14
3	(1, 0, 0)	(0.3, 0.4, 0.3)	No	⇒	1.20	0.74
				AVG	**0.64**	**0.34**

- Both networks have the same classification error of 1/3=0.33, or equivalently a classification accuracy of 2/3 = 0.67. **BUT** the second network is better because it nails the first two training items and just barely misses the third training item.
- MSE is not a bad approach but compared to CE, MSE gives too much emphasis to the incorrect outputs while CE rewards/penalizes probabilities of correct classes only.

22

OBJECTIVE / COST / LOSS

`nn.L1Loss`	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
`nn.CrossEntropyLoss`	This criterion computes the cross entropy loss between input logits and target.
`nn.CTCLoss`	The Connectionist Temporal Classification loss.
`nn.NLLLoss`	The negative log likelihood loss.
`nn.PoissonNLLLoss`	Negative log likelihood loss with Poisson distribution of target.
`nn.GaussianNLLLoss`	Gaussian negative log likelihood loss.
`nn.KLDivLoss`	The Kullback-Leibler divergence loss.
`nn.BCELoss`	Creates a criterion that measures the Binary Cross Entropy between

https://docs.pytorch.org/docs/stable/nn.html#loss-functions

23

Harmonic Loss to Replace CE Loss

- **Harmonic loss** has two desirable mathematical properties that enable faster convergence and improved interpretability:
  1) Scale invariance
  2) A finite convergence point, which can be interpreted as a class center

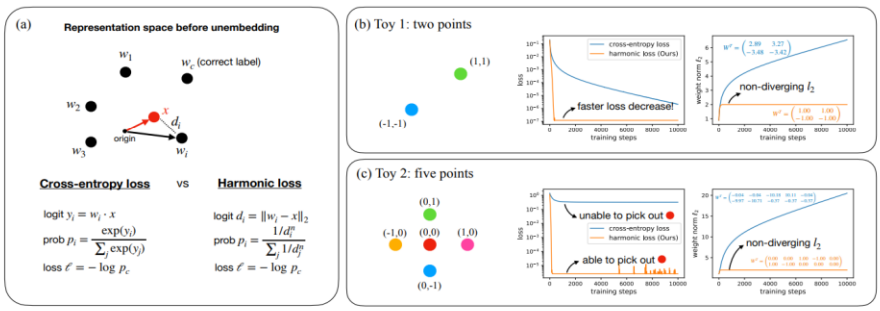


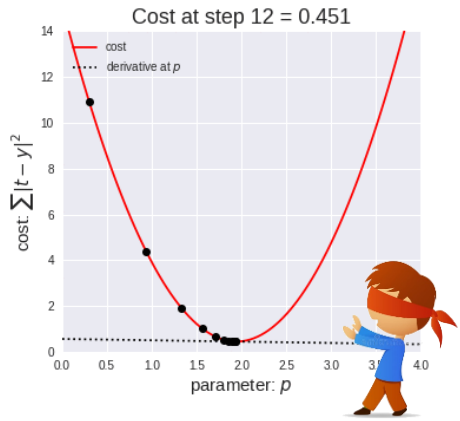
Figure 1. Cross-entropy loss versus harmonic loss (ours). (a) Definitions. Cross-entropy loss leverages the inner product as the similarity metric, whereas the harmonic loss uses Euclidean distance. (b) Toy case 1 with two points (classes). Both the harmonic loss and the l_2 weight norm converge faster for the harmonic loss. (c) Toy case 2 with five points (classes). Harmonic loss can pick out the red point in the middle. By contrast, the cross-entropy loss cannot, since the red point is not linearly separable from other points. The weight matrices are also more interpretable with harmonic loss than with cross-entropy loss.

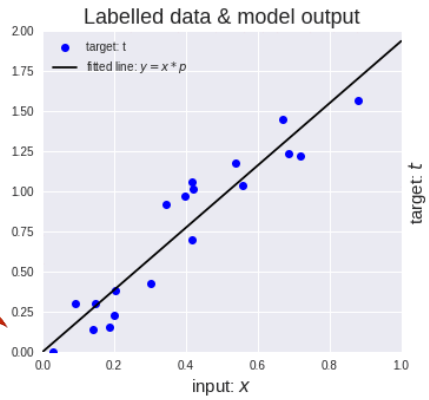
MIT, "Harmonic Loss Trains Interpretable AI Models," arXiv 3FEB2025 https://arxiv.org/abs/2502.01628, Transactions on ML Research (TMLR)

24

GRADIENT DESCENT

- **Gradient descent** is an optimization algorithm based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.





$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

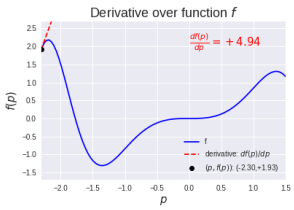
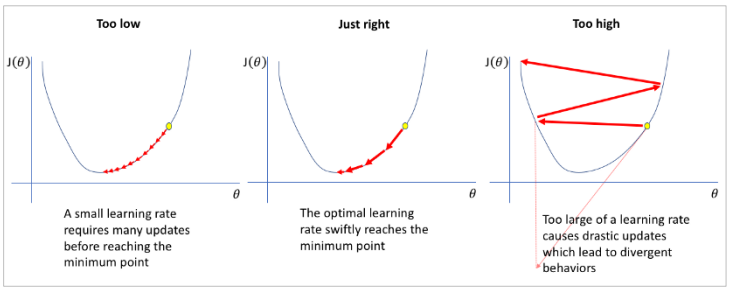


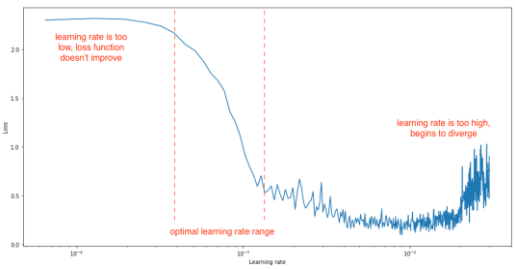
Image credit: https://medium.com/onfido-tech/machine-learning-101-be2e0a86c96a

25

GRADIENT DESCENT

- How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.

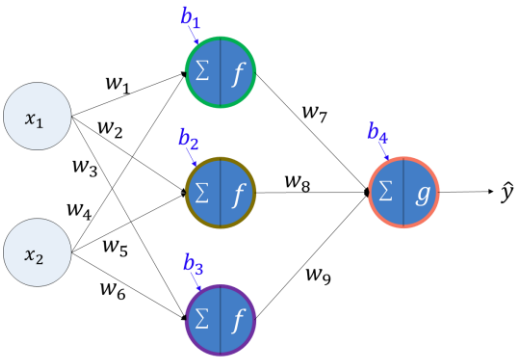




26

BACKWARD PROPAGATION (1986)

$$w_i \leftarrow w_i - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right) \quad , \quad b_i \leftarrow b_i - \left(lr * \frac{\partial(\text{Loss})}{\partial b_i}\right)$$



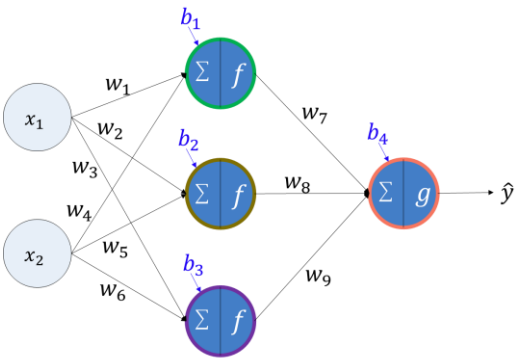
Example of updating w_7:

1. To update w_7, we have to compute
   $$w_7 = w_7 - \left(lr * \frac{\partial(\text{Loss})}{\partial w_7}\right)$$
2. In order to apply the chain rule, we first unfold the loss function until reaching w_7:
   $$\text{Loss} = (y - \hat{y})^2 \quad \texttt{\# Use a squared error for simplification}$$
   $$\hat{y} = \boldsymbol{h_1} w_7 + \boldsymbol{h_2} w_8 + \boldsymbol{h_3} w_9 + b_4$$
3. Compute the gradient value by finding the partial derivation of loss with respect to w_7:
   $$\frac{\partial(\text{Loss})}{\partial w_7} = \frac{\partial(\text{Loss})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_7} = (2 \cdot (y - \hat{y}) \cdot (-1)) \cdot (\boldsymbol{h_1})$$
4. Update w_7 in the opposite direction of the gradient

27

BACKWARD PROPAGATION (1986)

$$w_i \leftarrow w_i - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right) \quad , \quad b_i \leftarrow b_i - \left(lr * \frac{\partial(\text{Loss})}{\partial b_i}\right)$$



Example of updating w_1:

1. To update w_1, we have to compute
   $$w_1 = w_1 - \left(lr * \frac{\partial(\text{Loss})}{\partial w_1}\right)$$
2. In order to apply the chain rule, we first unfold the loss function until reaching w_1:
   $$\text{Loss} = (y - \hat{y})^2 \quad \texttt{\# Use a squared error for simplification}$$
   $$\hat{y} = \boldsymbol{h_1} w_7 + \boldsymbol{h_2} w_8 + \boldsymbol{h_3} w_9 + b_4$$
   $$\boldsymbol{h_1} = x_1 w_1 + x_2 w_4 + b_1$$
3. Compute the gradient value by finding the partial derivation of loss with respect to w_1:
   $$\frac{\partial(\text{Loss})}{\partial w_1} = \frac{\partial(\text{Loss})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1} = (2 \cdot (y - \hat{y}) \cdot (-1)) \cdot (w_7) \cdot (x_1)$$
4. Update w_1 in the opposite direction of the gradient

28

	Backpropagation	Gradient Descent
Definition	An algorithm for calculating the gradients of the cost function	Optimization algorithm used to find the weights that minimize the cost function
Requirements	Differentiation via the chain rule	•Gradient via Backpropagation •Learning rate
Process	Propagating the error backwards and calculating the gradient of the error function with respect to the weights	Descending down the cost function until the minimum point and find the corresponding weights

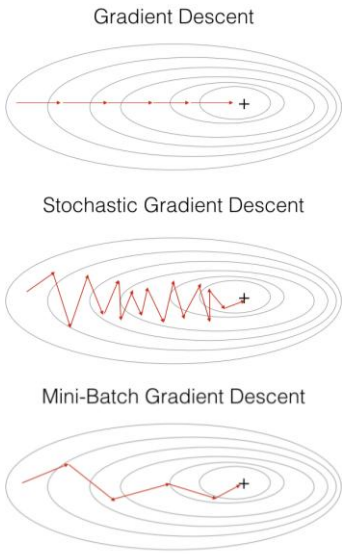
"To put it plainly, gradient descent is the process of using gradients to find the minimum value of the cost function, while backpropagation is calculating those gradients by moving in a backward direction in the neural network. Judging from this, it would be safe to say that gradient descent relies on backpropagation."

5APR2023: https://www.analyticsvidhya.com/blog/2023/01/gradient-descent-vs-backpropagation-whats-the-difference/

29

MINI-BATCH GRADIENT DESCENT

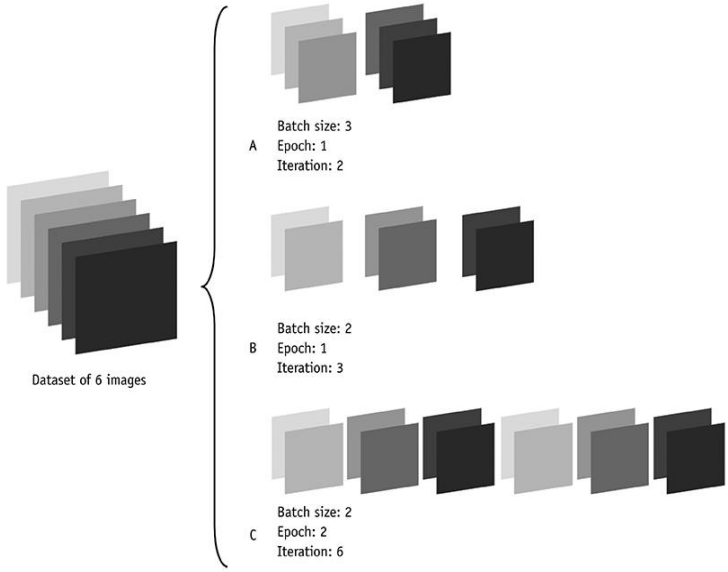
- Batch Gradient Descent:
  - Use the aggregated gradients from all training samples to update all trainable parameters
  - Slow calculation, large memory, and computationally expensive (due to large data)
- Stochastic Gradient Descent:
  - Use the gradient from one training sample to update all trainable parameters
  - Frequent updates and less memory, but noisy gradients, high variance, and computationally expensive (due to frequent updates)
- Mini-batch Gradient Descent:
  - Use the gradient from m training samples to update all trainable parameters, when m is the mini-batch size (`batch_size`)
  - More stable convergence, more efficient gradient calculation, and less memory but not guarantee good convergence



30

EPOCH / ITERATION

- One epoch is when an entire training dataset is passed forward and backward through the neural network only once.
- One iteration (step) is when one (mini) batch of training samples is passed forward and backward through the neural network once.



31

Why research favors iterations (steps) more?

- **The infinite data paradigm:**
  - In large-scale training, datasets are so massive that we might never finish a single epoch.
  - Hence, concluding the training at every epoch is impractical. Iterations provide a more constant measure of computational progress regardless of data volume.
- **High-resolution monitoring:**
  - Relying on epoch-level reporting is inefficient for long-running experiments where a single pass might take days.
  - Measuring by iterations allows for a much higher resolution of the loss curve, enabling researchers to detect training instabilities or loss spikes in real-time and intervene immediately.

- **Hardware and scaling consistency:**
  - In distributed multi-GPU environments, the time to complete an iteration is stable for a fixed batch size, while the duration of an epoch fluctuates if the dataset is sharded or updated.
  - Using iterations ensures consistent benchmarking and learning rate scheduling across different hardware clusters and changing data scales.

32

OPTIMIZER

- **Optimizers** are algorithms or methods used to minimize an error function (loss function) or to maximize the efficiency of production.
- Optimizers help get results faster.
- Gradient descent is an example of the optimizer.

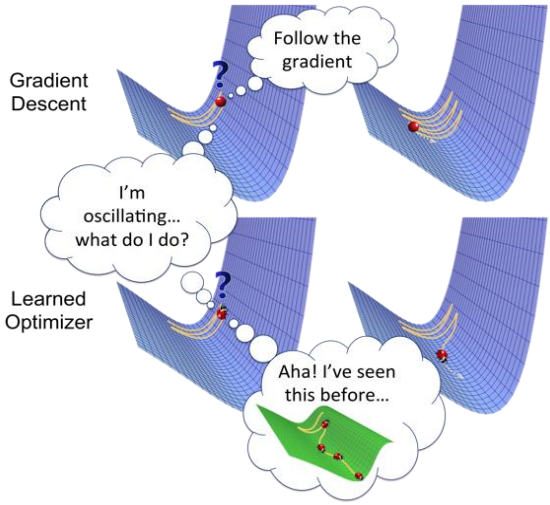


Image credit: (13JAN2019) https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

33

OPTIMIZER

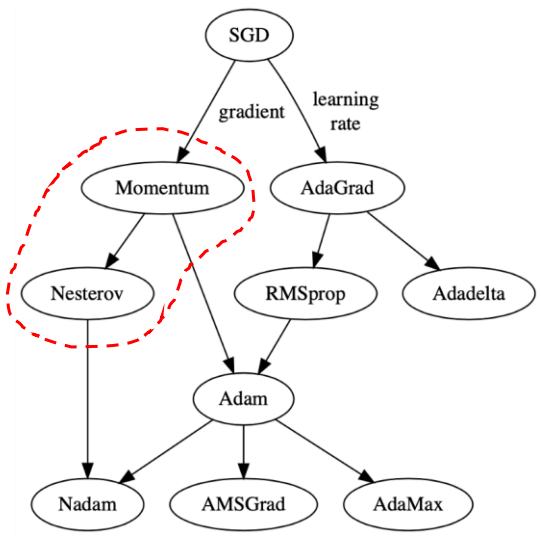
Adadelta	Implements Adadelta algorithm.
Adafactor	Implements Adafactor algorithm.
Adagrad	Implements Adagrad algorithm.
Adam	Implements Adam algorithm.
AdamW	Implements AdamW algorithm, where weight decay does not accumulate in the momentum nor variance.
SparseAdam	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
Adamax	Implements Adamax algorithm (a variant of Adam based on infinity norm).
ASGD	Implements Averaged Stochastic Gradient Descent.
LBFGS	Implements L-BFGS algorithm.

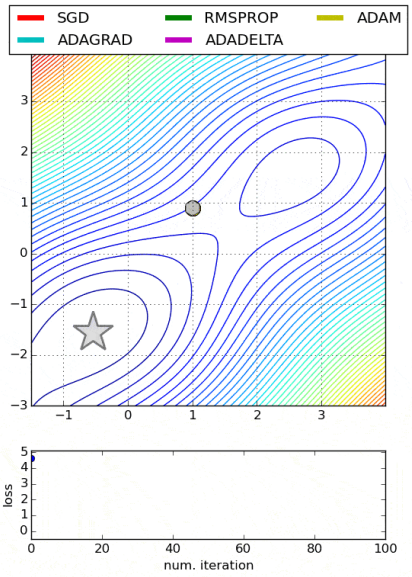
Muon	Implements Muon algorithm.
NAdam	Implements NAdam algorithm.
RAdam	Implements RAdam algorithm.
RMSprop	Implements RMSprop algorithm.
Rprop	Implements the resilient backpropagation algorithm.
SGD	Implements stochastic gradient descent (optionally with momentum).

https://docs.pytorch.org/docs/stable/optim.html#algorithms

34

FASTER OPTIMIZER





Conclusion diagram from https://www.kdnuggets.com/2019/06/gradient-descent-algorithms-cheat-sheet.html

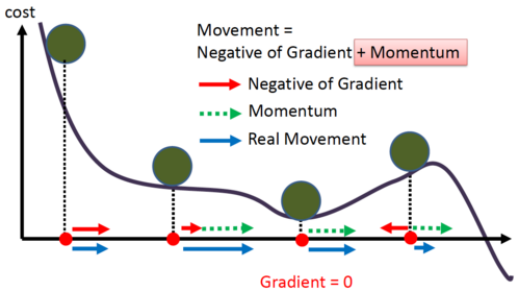
35

MOMENTUM (1964)

- **Vanilla Gradient Descent:**
  - It simply takes small, regular steps down the slope, so the algorithm will take much more time to reach the bottom.
  - It doesn't care much about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

- **Momentum in physics:**
  - An object with motion has some inertia which causes it to tend to move in the direction of motion.
  - Thus, if the optimization algorithm is moving in a general direction, the momentum causes it to 'resist' changes in direction.

- **Momentum optimization (1964):**
  - It cares much about what previous gradients were.
  - The update depends on **(the accumulated past) + (the gradient at that point)**.
  - It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction.



36

MOMENTUM (1964)

- The first few updates may show no real advantage over vanilla Gradient Descent — since there are no previous gradients to utilize for the update.
- As the number of updates increases the momentum kickstarts and allows faster convergence.

- The momentum vector or the velocity
- The initial velocity is set to zero.

The local gradient is used for acceleration.

$\boldsymbol{\beta}$ is a hyperparameter called "momentum."
- Control how quickly the effect of past gradients decay
- Value between 0.0 (high friction) and 1.0 (no friction)
- A typical value is 0.9.

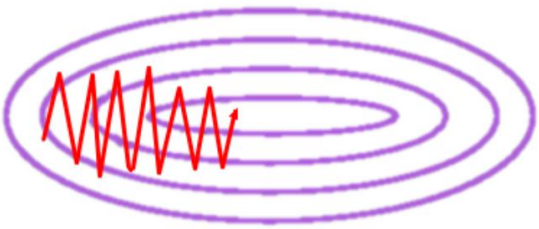
1. $\boldsymbol{m_i} \leftarrow \boldsymbol{\beta m_i} - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right)$
2. $w_i \leftarrow w_i + \boldsymbol{m_i}$

37

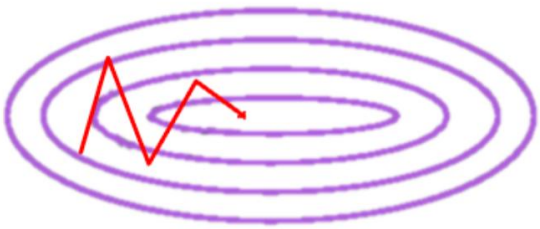
MOMENTUM (1964)

- To use momentum optimization, just use `torch.optim.SGD` and set its `momentum` hyperparameter

$$w_i \leftarrow w_i - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right)$$



$$\boldsymbol{m_i} \leftarrow \boldsymbol{\beta m_i} - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right)$$
$$w_i \leftarrow w_i + \boldsymbol{m_i}$$



(Left) Vanilla SGD, (right) SGD with momentum. Goodfellow et al. (2016)

38

NESTEROV MOMENTUM (1983)

- Problems of the vanilla momentum optimization:
  - The momentum problem near minima regions
  - Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum.

- Nesterov Accelerated Gradient (NAG), also known as Nesterov momentum optimization, is a small variant to momentum optimization and is almost always faster than vanilla momentum optimization.
  - **Look before leap:** Measure the gradient of the cost function not at the local position w_i but slightly ahead in the direction of the momentum, at $w_i + \beta m$
  - This small tweak works because, in general, the momentum vector will be pointing in the right direction (toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original positions.

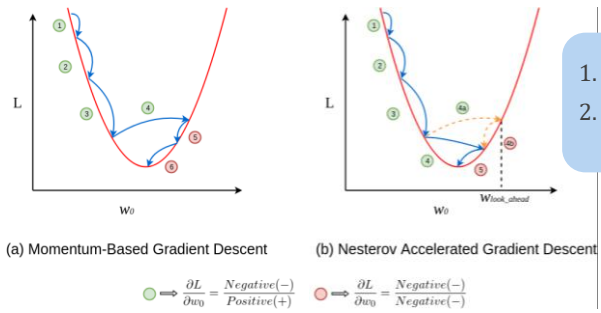
39

NESTEROV MOMENTUM (1983)

- When the momentum pushes the weights across a valley:
  - The gradient at the starting point w_i continues to push farther across the valley, while the gradient at the point $w_i + \beta m$ pushes back toward the bottom of the valley.
  - This helps reduce oscillations and thus NAG converges faster.

Momentum

1. $m_i \leftarrow \beta m_i - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right)$
2. $w_i \leftarrow w_i + m_i$



(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent

$\frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

Nesterov Momentum

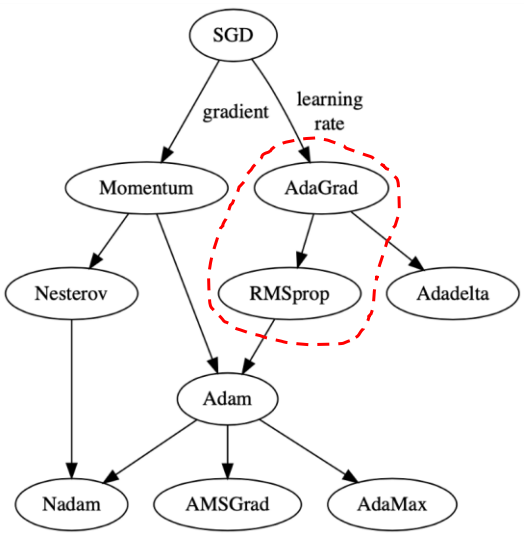
1. $m_i \leftarrow \beta m_i - \left(lr * \frac{\partial(\text{Loss})}{\partial(w_i + \beta m_i)}\right)$
2. $w_i \leftarrow w_i + m_i$

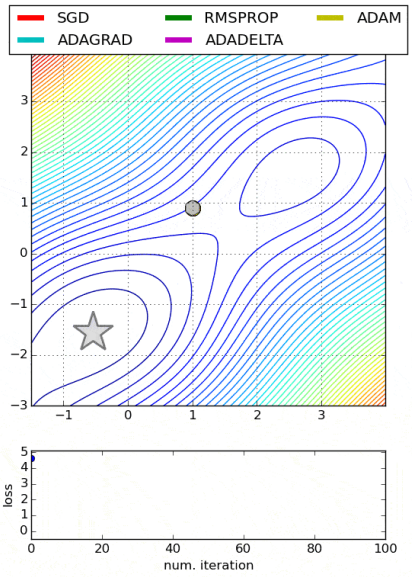
Look-ahead gradient

- To use NAG, simply set `nesterov=True` when creating the `torch.optim.SGD` optimizer

40

FASTER OPTIMIZER





Conclusion diagram from https://www.kdnuggets.com/2019/06/gradient-descent-algorithms-cheat-sheet.html

41

Adaptive Gradient (AdaGrad) (2011)

- Problem of the vanilla Gradient Descent:
  - Gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley.
  - It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum.
- AdaGrad's Solution: `torch.optim.Adagrad`
  - AdaGrad achieves the correction by scaling down the gradient vector along the steepest dimensions.
  - In short, AdaGrad decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an adaptive learning rate.

- s_i accumulates the squares of the loss function w.r.t. w_i.
- If the loss function is steep along the i^{th} dimension, then s_i will get larger and larger at each iteration.
- s is initialized by zero.

$$1.\ \boldsymbol{s_i} \leftarrow \boldsymbol{s_i} + \left(\frac{\partial(\text{Loss})}{\partial w_i}\right)^2$$

$$2.\ w_i \leftarrow w_i - \left(lr \cdot \frac{\partial(\text{Loss})}{\partial w_i} \cdot \frac{1}{\sqrt{\boldsymbol{s_i}+\varepsilon}} \right)$$

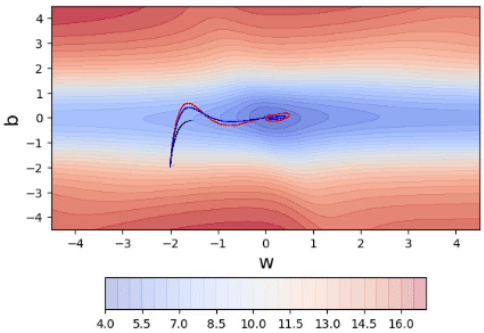
Scale down the gradient vector (a.k.a., learning rate decay)

A smooth term to avoid division by zero, typically set to 10^{-10}

42

Adaptive Gradient (AdaGrad) (2011)

- AdaGrad's limitations:
  - It frequently performs well for simple quadratic problems, but it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.
  - It is recommended that we should not use AdaGrad to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though).



Vanilla GD(black)
Momentum (red)
Nesterov (blue)
AdaGrad (green)

43

RMSProp (2012) by Hinton and Tieleman

- Problem of AdaGrad:
  - It runs the risk of slowing down a bit too fast and never converging to the global optimum.
- Solution of RMSProp: `torch.optim.RMSprop`
  - Accumulate only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training) using exponential decay in the first step

α is a hyperparameter called "smoothing constant (decay factor)." The higher the alpha, the longer the optimizer remembers gradients.

AdaGrad

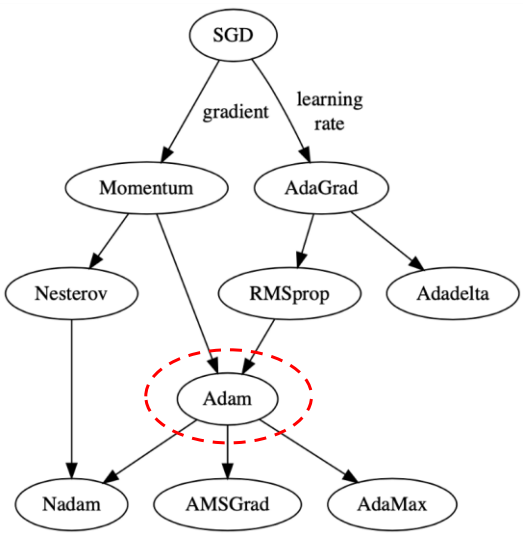
1. $s_i \leftarrow s_i + \left(\frac{\partial(\text{Loss})}{\partial w_i}\right)^2$
2. $w_i \leftarrow w_i - \left(lr \cdot \frac{\partial(\text{Loss})}{\partial w_i} \cdot \frac{1}{\sqrt{s_i}+\varepsilon}\right)$

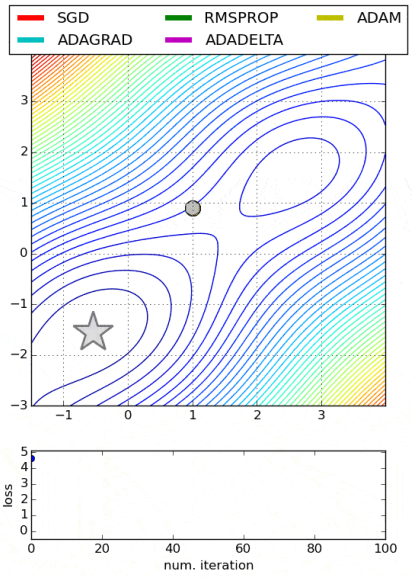
RMSProp

1. $s_i \leftarrow \alpha s_i + (1-\alpha)\left(\frac{\partial(\text{Loss})}{\partial w_i}\right)^2$
2. $w_i \leftarrow w_i - \left(lr \cdot \frac{\partial(\text{Loss})}{\partial w_i} \cdot \frac{1}{\sqrt{s_i}+\varepsilon}\right)$

44

FASTER OPTIMIZER





Conclusion diagram from https://www.kdnuggets.com/2019/06/gradient-descent-algorithms-cheat-sheet.html

45

Adam (arXiv 2014, ICLR 2015)

- Adam (Adaptive moment estimation) combines the ideas of momentum optimization and RMSProp.
  - Just like momentum optimization, Adam keeps track of an exponentially decaying average of past gradients.
  - Just like RMSProp, Adam keeps track of an exponentially decaying average of past squared gradients.
  - Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter lr.

- To use Adam, please refer to `torch.optim.Adam` :
  - The momentum decay hyperparameter `betas[0]` is initialized to 0.9.
  - The scaling decay hyperparameter `betas[1]` is initialized to 0.999.
  - The smoothing term `eps` to prevent division by zero is initialized to a tiny number of 10^{-8}.

46

Adam (arXiv 2014, ICLR 2015)

Momentum

1. $m_i \leftarrow \beta m_i - \left(lr * \frac{\partial(\text{Loss})}{\partial w_i}\right)$
2. $w_i \leftarrow w_i + m_i$

RMSProp

1. $s_i \leftarrow \alpha s_i + (1-\alpha)\left(\frac{\partial(Loss)}{\partial w_i}\right)^2$
2. $w_i \leftarrow w_i - \left(lr \cdot \frac{\partial(Loss)}{\partial w_i} \cdot \frac{1}{\sqrt{s_i}+\varepsilon}\right)$

An exponentially decaying average instead of an exponentially decaying sum

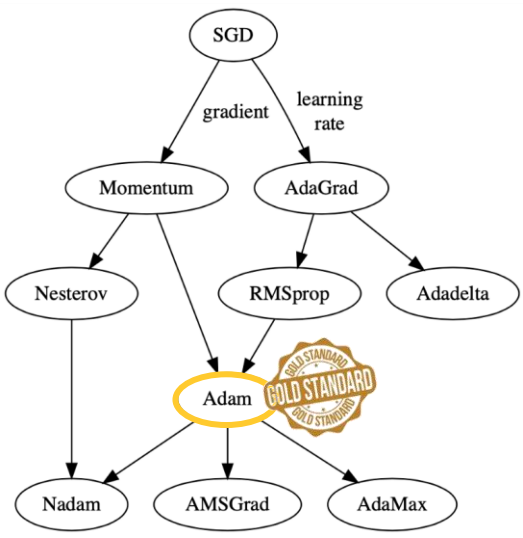
1. $m_i \leftarrow \boldsymbol{\beta_0} m_i + (1-\boldsymbol{\beta_0})\left(\frac{\partial(Loss)}{\partial w_i}\right)$
2. $s_i \leftarrow \boldsymbol{\beta_1} s_i + (1-\boldsymbol{\beta_1})\left(\frac{\partial(\text{Loss})}{\partial w_i}\right)^2$
3. $\hat{m}_i \leftarrow \frac{m_i}{1-\boldsymbol{\beta_0}^t}$
4. $\hat{s}_i \leftarrow \frac{s_i}{1-\boldsymbol{\beta_1}^t}$

- t represents the iteration number starting at 1.
- Since m and s are initialized at 0, they will be biased toward 0 at the beginning of training.
- These two steps will help boost m and s at the beginning of training.

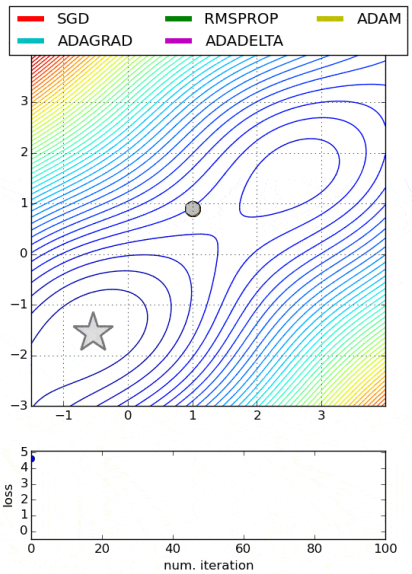
5. $w_i \leftarrow w_i - \left(lr \cdot \frac{\hat{m}_i}{\sqrt{\hat{s}_i}+\boldsymbol{\varepsilon}}\right)$

47

FASTER OPTIMIZER



Conclusion diagram from https://www.kdnuggets.com/2019/06/gradient-descent-algorithms-cheat-sheet.html



48

Adam with Weight Decay (arXiv 2017, ICLR 2019)

- **Adam (2014)** vs. **AdamW (2017):**
  - Both are adaptive optimizers widely used in deep learning.
  - Both are designed to manage momentum and adjust learning rates dynamically.
  - The difference is how they handle weight decay, which impacts their effectiveness in different scenarios.
- **Usage scenarios:** [21OCT2024: https://www.datacamp.com/tutorial/adamw-optimizer-in-pytorch]
  - Adam performs better in tasks where regularization is less critical, or when computational efficiency is prioritized over generalization.
    - For example Small neural networks (on datasets like MNIST or CIFAR-10), simple regression problems, early stage prototyping, less noisy data, short training cycles
  - AdamW excels in scenarios where overfitting is a concern and model size is substantial.
    - For example Large-scale transformers (like GPT), complex computer vision models (on datasets like ImageNet), multi-task learning, generative models (like GAN), reinforcement learning

I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," ICLR 2019, https://arxiv.org/abs/1711.05101

49

Adam with Weight Decay (arXiv 2017, ICLR 2019)

- L2 Regularization vs. Weight Decay:
  - In SGD with momentum, L2 regularization is the same as weight decay.
    - L2 regularization adds λW_i to the gradient-computation equation.
    - Weight decay adds λW_i to the parameter-update equation.
  - **BUT**, in Adam, L2 regularization is not the same as weight decay.
    - L2 regularization leads to weights with large historic parameter and/or gradient amplitudes being regularized less than they would be when using weight decay.
    - It is suggested that, for Adam, we should use weight decay, not L2 regularization.

50

Adam with Weight Decay (arXiv 2017, ICLR 2019)

Adam w/o regularization

0. $g_t \leftarrow \frac{\partial(Loss)}{\partial w_i}$
1. $m_i \leftarrow \beta_0 m_i + (1-\beta_0)\, g_t$
2. $s_i \leftarrow \beta_1 s_i + (1-\beta_1)(g_t)^2$
3. $\hat{m}_i \leftarrow \frac{m_i}{1-\beta_0{}^t}$
4. $\hat{s}_i \leftarrow \frac{s_i}{1-\beta_1{}^t}$
5. $w_i \leftarrow w_i - lr\left(\frac{\hat{m}_i}{\sqrt{\hat{s}_i}+\varepsilon}\right)$

λ is the weight decay factor.

Adam with regularization

0. $g_t \leftarrow \frac{\partial(Loss + \lambda W_i)}{\partial w_i}$
1. $m_i \leftarrow \beta_0 m_i + (1-\beta_0)\, g_t$
2. $s_i \leftarrow \beta_1 s_i + (1-\beta_1)(g_t)^2$
3. $\hat{m}_i \leftarrow \frac{m_i}{1-\beta_0{}^t}$
4. $\hat{s}_i \leftarrow \frac{s_i}{1-\beta_1{}^t}$
5. $w_i \leftarrow w_i - lr\left(\frac{\hat{m}_i}{\sqrt{\hat{s}_i}+\varepsilon}\right)$

- The standard and regularization gradients are coupled, mixing their effects during momentum calculation.
- This coupling can corrupt the direction and magnitude of updates.

AdamW with regularization

0. $g_t \leftarrow \frac{\partial(Loss)}{\partial w_i}$
1. $m_i \leftarrow \beta_0 m_i + (1-\beta_0)\, g_t$
2. $s_i \leftarrow \beta_1 s_i + (1-\beta_1)(g_t)^2$
3. $\hat{m}_i \leftarrow \frac{m_i}{1-\beta_0{}^t}$
4. $\hat{s}_i \leftarrow \frac{s_i}{1-\beta_1{}^t}$
5. $w_i \leftarrow w_i - lr\left(\frac{\hat{m}_i}{\sqrt{\hat{s}_i}+\varepsilon} + \lambda W_i\right)$

The regularized gradient is decoupled and applied after the smoothed standard gradient update to adjust the weights.

51

Muon optimizer

The second-order optimizer that surpasses AdamW in large-scale LLM training.

- Since Adam (2014), new algorithms have come and gone without unseating Adam from its throne, with the exception of AdamW (2017).
- **Muon** is perhaps the leading example of how algorithms targeting specific NN training workloads are starting to make real headway against the formidable AdamW baseline.

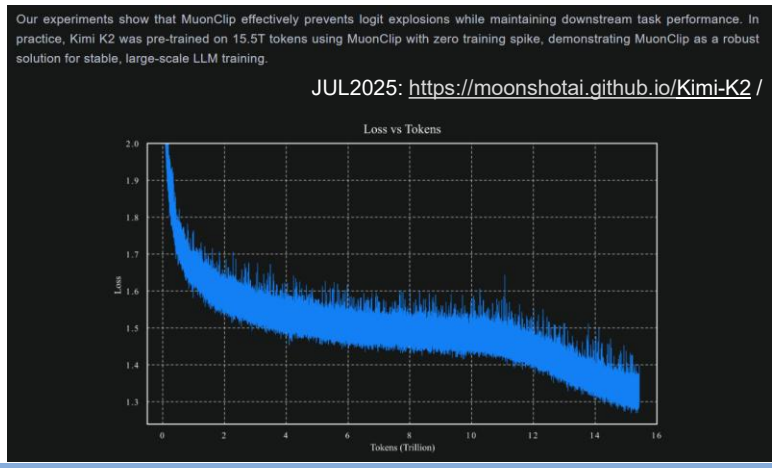
12JUL2025: https://lukemerrick.com/posts/first_order_optimization.html

52

Muon optimizer

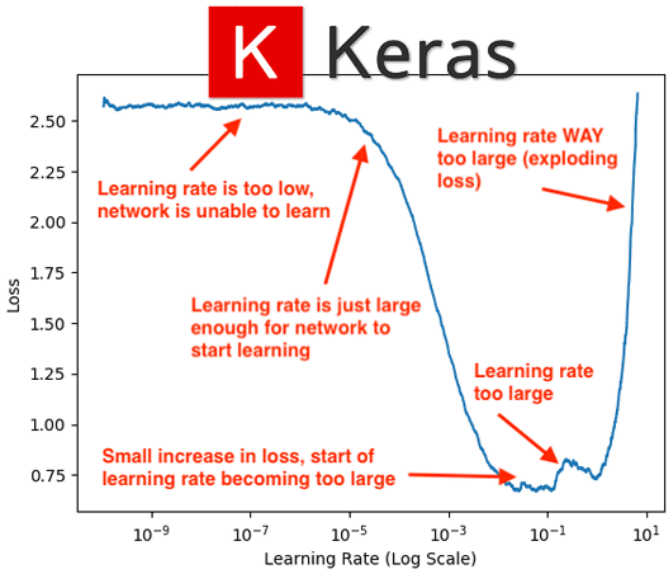
The second-order optimizer that surpasses AdamW in large-scale LLM training.

- **Muon** made a splash by powering impressive gains on NanoGPT speed runs (OCT2024).
- **Muon** is having a big moment as the Kimi K2 model release (JUL2025) backs up the claims made in Moonshot's paper https://arxiv.org/abs/2502.16982 .



12JUL2025: https://lukemerrick.com/posts/first_order_optimization.html

Learning Rate Tuning

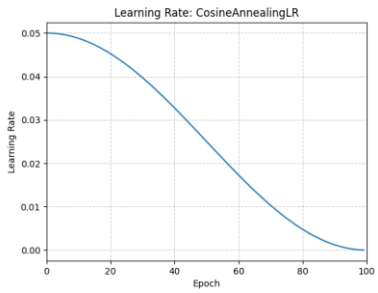


5AUG2019: https://pyimagesearch.com/2019/08/05/keras-learning-rate-finder/

Learning Rate (LR) Scheduler

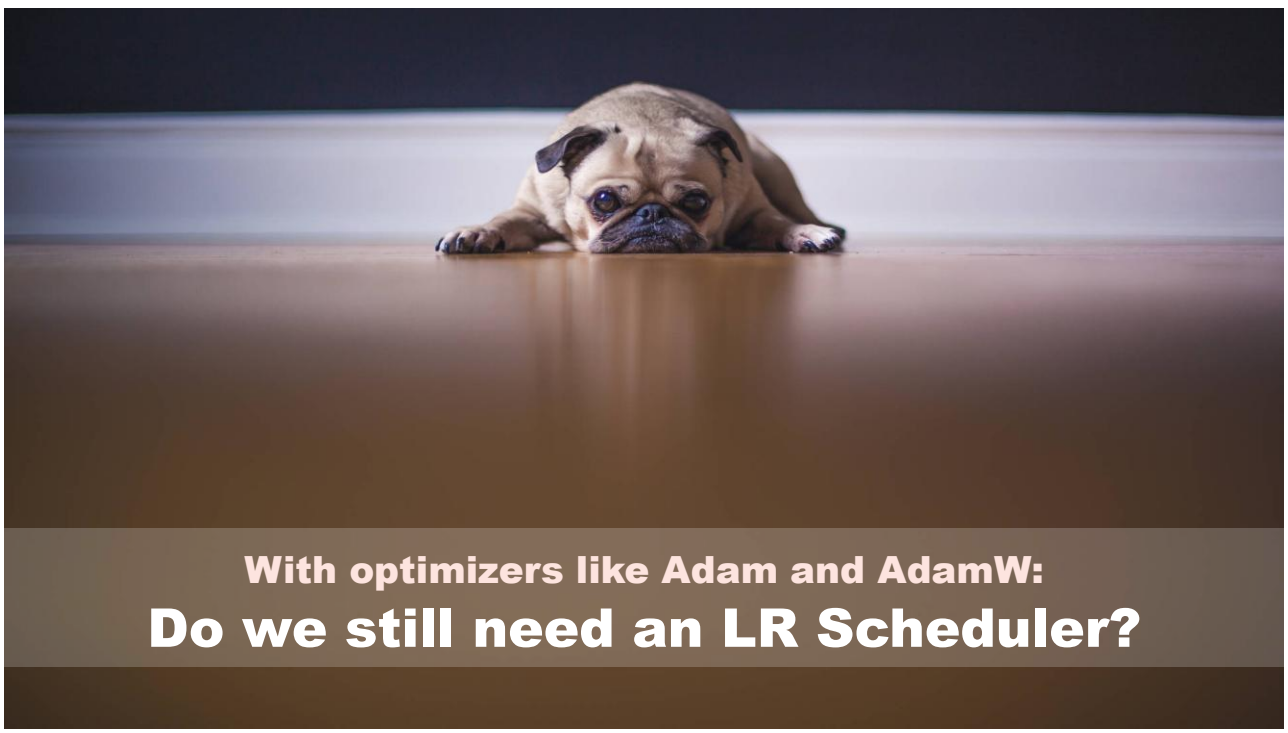
- The big three schedulers in :

1. **`optim.lr_scheduler.StepLR()`: The standard decay**
   - Decays the LR by a factor (`gamma`) every few `step_size`*.
2. **`optim.lr_scheduler.ReduceLROnPlateau()`: The smart decay**
   - Monitors a metric (like validation loss) and reduces LR when the metric stops improving for `patience` steps*.
3. **`optim.lr_scheduler.CosineAnnealingLR()`: The current gold standard for training modern architectures**
   - Most modern papers use this as it helps the model converge to a much better local minimum.
   - In the beginning, a high learning rate helps the model escape local minima and explore the loss landscape. As training progresses, the rate smoothly "anneals (cools down)" to a very small value, allowing the model to settle precisely into a high-quality global minimum.



* = The number of times the `scheduler.step()` is called.

55



56

Adam/AdamW vs. LR Scheduler

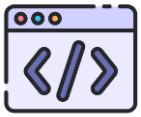
- **Local control:** An adaptive-LR optimizer (like Adam) helps with per-parameter learning rates.
  - It updates each trainable parameter with an individual learning rate. This means that every parameter in the network has a specific learning rate associated with it, ensuring that "busy" parameters move with more stability and "rare" parameters get the boost they need to learn.
  - **BUT** the single learning rate for each parameter is computed using lr (the initial learning rate) as an upper limit. This means that every single learning rate can vary from 0 (no update) to lr (maximum update).
- **Global control:** LR Scheduler refers to a global learning rate multiplier that decays the base learning rate (lr) over time, giving us additional control over the overall training dynamics, improving both performance and stability.

57

Adam/AdamW + LR Scheduler

- Further finetuning:
  - Adam adjusts the learning rate individually for each parameter based on the gradient, but the overall learning rate still affects how much the model updates in each step.
  - An LR scheduler allows us to adjust the global learning rate over time, providing better control, especially during fine-tuning phases when smaller learning rates are crucial.
- Prevent overshooting:
  - While Adam adjusts the learning rates per parameter, there can still be cases where the learning rate is too high for some stages of training.
  - A scheduler can gradually decrease the learning rate as the model converges, helping avoid overshooting the minimum and settling into a better solution.
- Improve generalization:
  - A common technique, such as learning rate warmup or decay, can help models generalize better.
  - By slowly reducing the learning rate over time, we reduce the chance of overfitting and help the model find a more optimal, generalized solution.
- Handling plateaus:
  - Sometimes, training plateaus (stops improving) even with Adam.
  - An LR scheduler can lower the learning rate at key moments to help the model "escape" these plateaus and keep improving.

58



```
# 1. SETUP
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

# 2. THE LOOP
for epoch in range(epochs):
    model.train() # Enable dropout/batchnorm for training

    for batch_idx, (data, target) in enumerate(train_loader):

        # --- THE CORE ITERATION ---
        optimizer.zero_grad()            # 1. Reset gradient buffers
        output = model(data)             # 2. Forward pass
        loss = criterion(output, target) # 3. Calculate error
        loss.backward()                  # 4. Backpropagate error
        optimizer.step()                 # 5. Update weights (Gradient Descent)

    # --- THE SCHEDULING ---
    # Usually called once per epoch to decay the global LR
    scheduler.step()
```

59

Coding

Combine everything and implement a program with PyTorch

60

EX 1: Your First MLP (1/2)

- Use PyTorch to create an MLP model based on a dummy dataset data
- **Strict Data Typing:**
  - PyTorch does not automatically convert data types.
  - Make sure that the dataset is converted into `torch.Tensor` (typically using `torch.float32` for MLP inputs) before passing them to the model.
- **Explicit Model Modes:**
  - Don't forget to manually toggle the model's state using `model.train()` during the training loop and `model.eval()` during inference.
  - This explicitly controls behaviors like Dropout and Batch Normalization that Keras handles automatically.

61

EX 1: Your First MLP (2/2)

- **Comprehensive Reproducibility:**
  - Unlike Keras, PyTorch requires manual seeding across multiple libraries—random, numpy, and torch (for both CPU and GPU)—to ensure fixed weight initialization and consistent results.
- **The Multi-Run Strategy:**
  - To conclude performance reliably, the training logic should be wrapped in a loop that resets the seed and re-initializes the model 3–5 times.

```
import torch
import numpy as np
import random

def set_seeds(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    # Force deterministic algorithms for 100% reproducibility
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# Example of averaging performance
results = []
for i in range(5):
    set_seeds(seed=i)
    # model = MyMLP()...
    # score = train_and_evaluate()...
    results.append(score)

print(f"Average Performance: {np.mean(results)}")
```

62

EX 2: Hand-designed MLPs

- Four separated problems:
  - Regression with 4 numeric inputs in order to produce 3 numeric outputs
  - Binary classification with 4 numeric inputs in order to produce the yes-or-no prediction
  - Single-label multi-class classification with 4 numeric inputs in order to produce likelihoods regarding 6 output classes
  - Multi-label multi-class classification with 4 numeric inputs in order to produce likelihoods regarding 6 output classes

- Hyperparameter checklist:
  - The number of layers: input, hidden, output
  - The number of neurons in each layer
  - ~~Weight initialization~~
  - Input normalization
  - Activation function in each layer
  - ~~Regularization~~
  - Loss function
  - Optimizer
  - Initial learning rate
  - ~~Learning rate scheduler~~
  - (Mini-) Batch size
  - The number of epochs

63

64