Deep Learning

Class 5: Training Neural Networks

Hédi Hadiji

Université Paris-Saclay - CentraleSupelec

hedi.hadiji@l2s.centralesupelec.fr

November 2025

Table of Contents

Up to Now

We have seen how to train a neural network on a **training dataset** by minimizing a given **loss function**.

True objective is to achieve a low **population loss**, i.e., to **generalize** well to unseen data.

So far, our recipe has been simple:

Choose a model, train it with SGD.

In practice, things are not that easy.

Table of Contents

Training issues

Problem 1: Optimisation of deep nets

There are obstacles to attaining a small training loss

- Vanishing gradients: gradients with respect to first layers are tiny
- Loss landscapes: saddle points and local minima

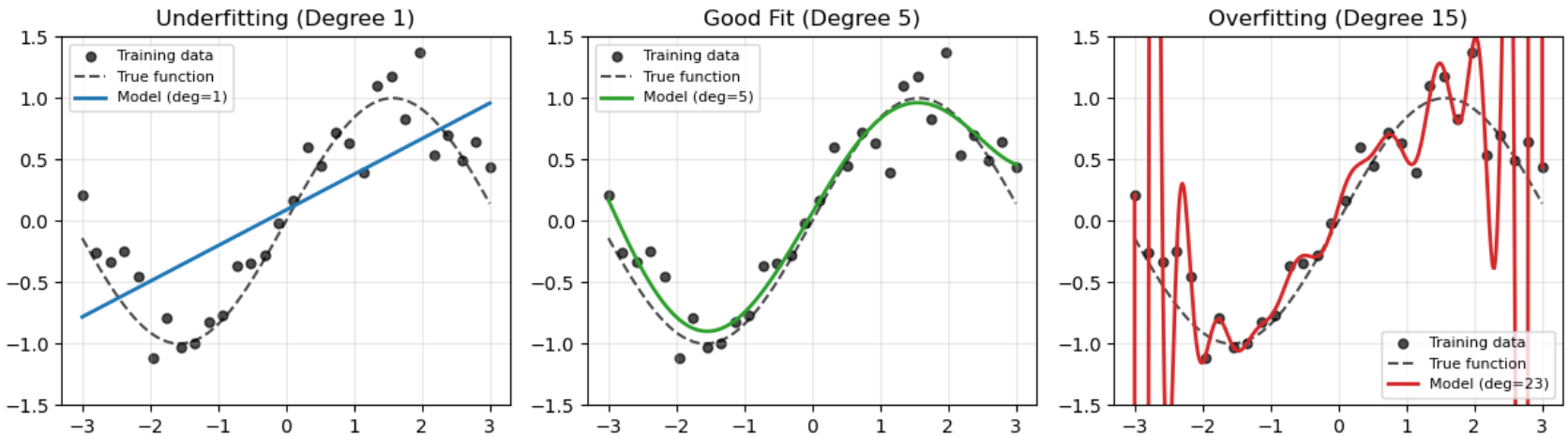
Underfitting, Overfitting in ML

The textbook story:

- You want to do regression, i.e., fit a function to data points.
- Choose a parametric family of models, e.g. polynomials, with d degrees of freedom.
- Take least-squares loss.

Then

- if d is too small: not expressive enough, model will underfit
- if d is too large: not expressive enough, model will overfit



Question: is this toy example relevant for real models and real data?

Overfitting

Overfitting makes sense in the ideal setting described above, but is very ill-defined in practice.

Neural networks work well in the over-parameterized regime: ImageNet is $\sim$ 1 000 000 images, Deep ResNets are up to $\sim$ 60 000 000 parameters.

Problem 1: Learning with many parameters

How can we make highly overparameterized models generalize from low train loss to low test loss?

Generic theoretical **upper bounds on generalization error** are larger when the number of parameters is larger and we pick an ERM.
But we do not know if these bounds are tight, and we do not select just any ERM: we use SGD + training techniques.
Some question whether overfitting exists at all. [see this blog series]

Table of Contents

Vanishing / Exploding Gradients

Due to the chain rule, gradients can vanish or explode in deep networks. Recall gradients at layer ℓ:

$$\partial_{W_{\ell-1}} \mathcal{L} = (\partial_{x_\ell} \mathcal{L}) \partial_{W_{\ell-1}} x_\ell = \left((\partial_{x_L} \mathcal{L}) \prod_{k=\ell}^{L-1} \partial_{x_k} x_{k+1} \right) \partial_{W_{\ell-1}} x_\ell$$

If the Jacobians $\partial_{x_k} x_{k+1}$ have small (resp. large) singular values, the gradients will **vanish** (resp. **explode**) with depth.

Residual Connections

Idea: To handle vanishing gradients. Let each layer learn a *residual function* instead of a full transformation.

Residual connection

Given a layer F with weights W

$$x_{\ell+1} = x_\ell + F(x_\ell; W)$$

Then

$$\partial_{x_{\ell+1}} \mathcal{L} = \left(\partial_{x_\ell} \mathcal{L}\right)\left(I_d + \partial_{x_\ell} F\right)$$

Therefore even if activation lie in flat regions of the graph of F, the gradients flow.

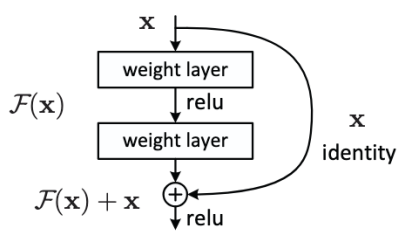
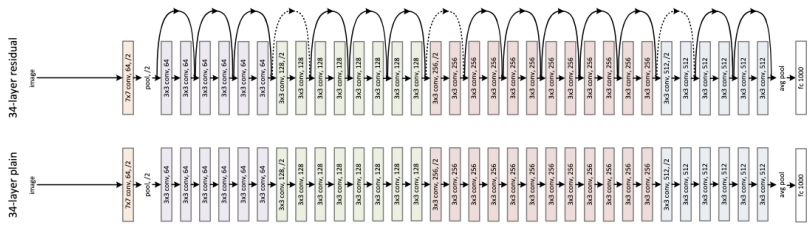
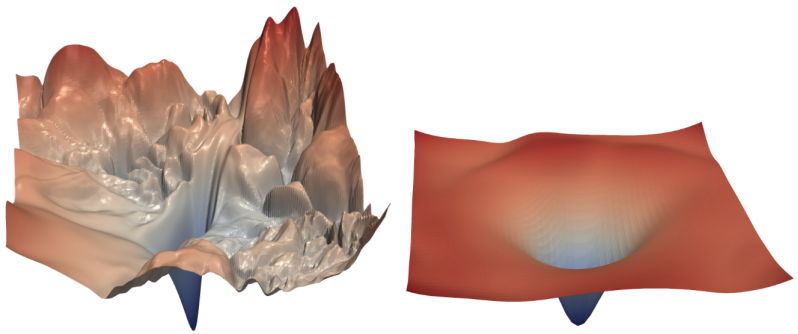


Figure 2. Residual learning: a building block.

Skip connections: Illustration



[Original Resnet Paper] 'solved' the Imagenet dataset.



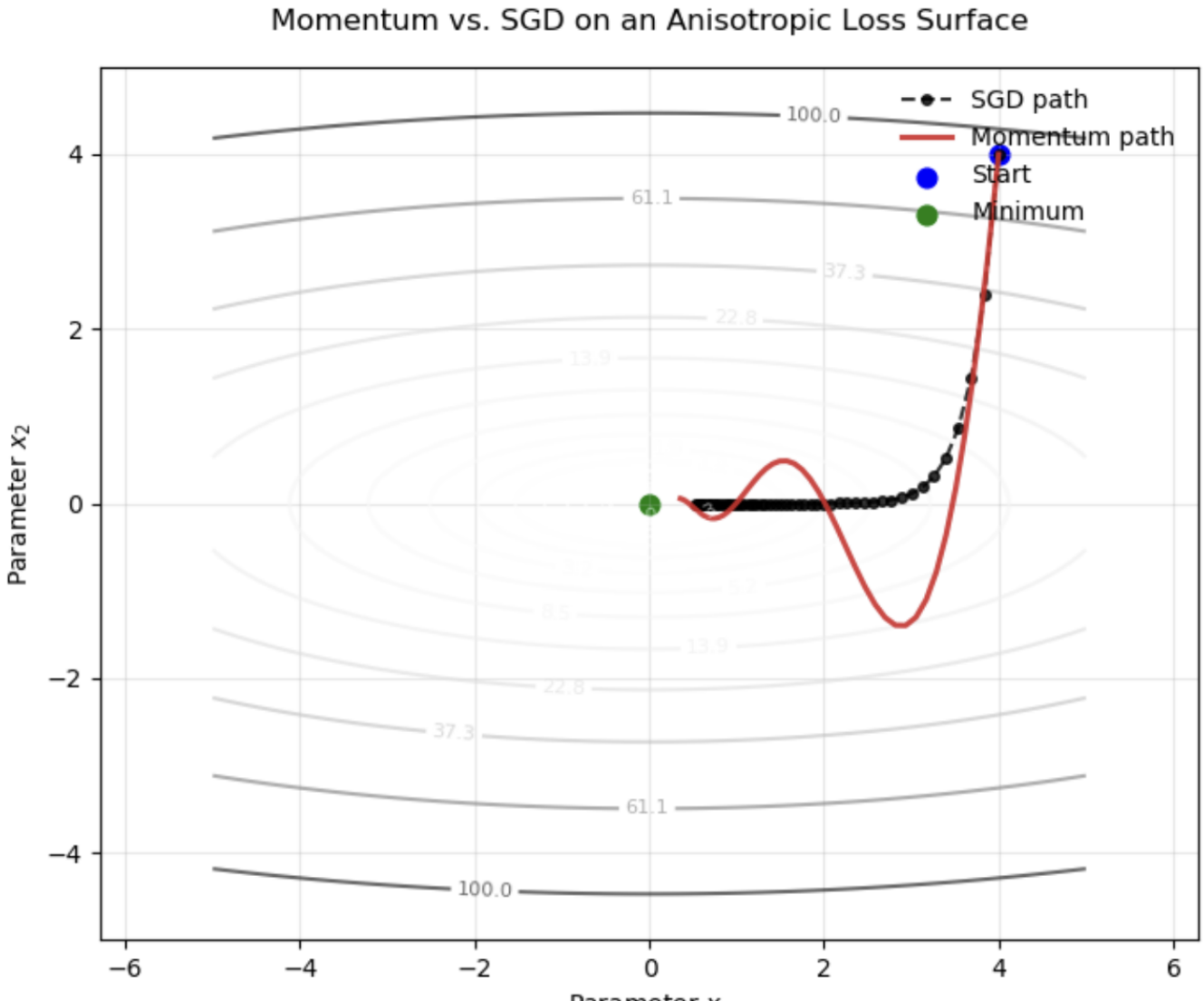
(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

[Visualizing Loss Landscape]

Optimizers + Learning Rate Schedules

Momentum avoids saddle-points.



Adam

Adam uses momentum + adaptive learning rates.

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*, ϵ (epsilon)
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $v_0^{max} \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda \theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big)$
  if *amsgrad*
    $v_t^{max} \leftarrow \mathbf{max}(v_{t-1}^{max}, v_t)$
    $\widehat{v_t} \leftarrow v_t^{max} / \big(1 - \beta_2^t\big)$
  else
    $\widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big)$
  $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

---

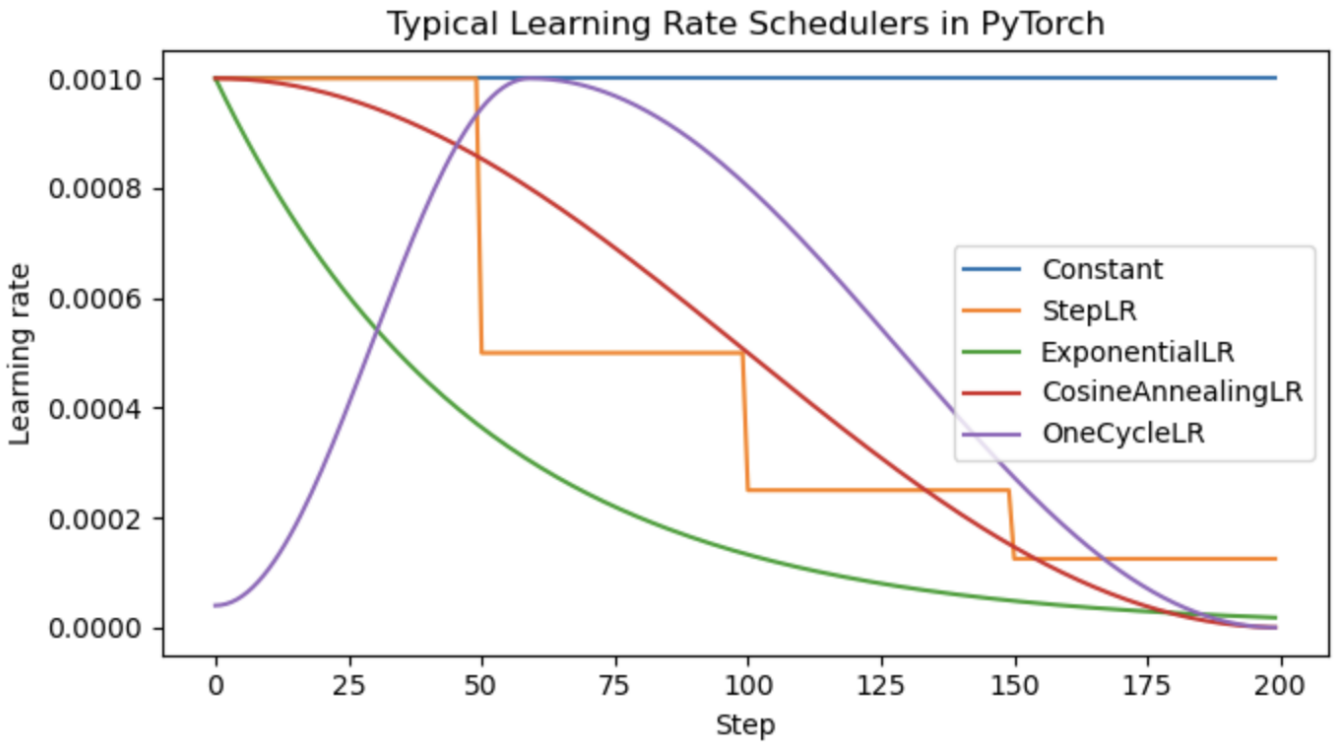
return θ_t

[pytorch doc]

Learning Rate Schedules

Change the learning rate during training to improve convergence:

- Start with a large learning rate to make rapid progress.
- Decrease it over time to refine the solution.



Another Ingredient: Correct Initialization

Initialization sets the starting point of optimization, and the **scale** at which leaning will occur.

Two goals:

- Break symmetry between neurons.
- Maintain the typical scale of activations and gradients across layers.

First observation: With zero-initialization, all neurons in a layer compute the same output and receive the same gradient: no learning occurs within a layer.

Initialization Schemes

Goal: preserve the typical scale of activations and gradients across layers.

Random initialization of weights is standard:

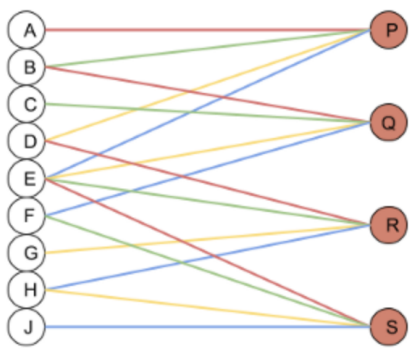
- Independent, zero-mean weights.
- Variance depends on activation function and fan-in/out.

Two main schemes (both can be uniform or normal):

$$\text{Xavier / Glorot:} \quad \text{Var}(W_{ij}) \propto \frac{1}{\text{fan\_in} + \text{fan\_out}}$$

$$\text{He / Kaiming:} \quad \text{Var}(W_{ij}) \propto \frac{1}{\text{fan\_in}}$$

Use Xavier for tanh/sigmoid and He for ReLU-like activations.



Variance Preservation Intuition

At a given layer,

$$\mathrm{Var}((Wx)_i \mid x) = \mathrm{Var}\left(\sum_{i=1}^{d_{\text{fan\_in}}} W_{i,j} x_i\right) = \sum_{i=1}^{d_{\text{fan\_in}}} \mathrm{Var}(W_{i,j})\, x_i^2 \approx \|x\|^2$$

so the variance of activations is preserved layer to layer, and similarly for backpropagated gradients (with the output dimension).

Remark: this variance is with respect to the randomness of initialization only, not the stochasticity of the data. This ensures the *typical scale of activations is preserved.*

[see `torch.nn.init`] for exact constants and functions.

Normalization Layers

Goal: stabilize training and control scale of activations

Motivation:

- Deep networks suffer from drifting activation scales across layers.
- Normalization hard fixes the mean and variance, improving gradient flow.

General form:

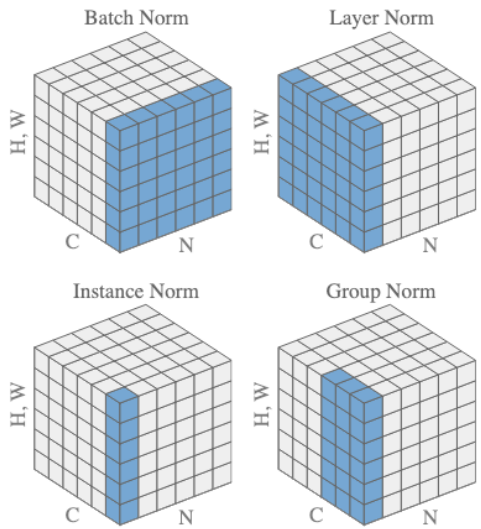
$$y = \frac{x - \mu(x)}{\sigma(x)} \odot \gamma + \beta$$

where μ, σ are computed over specific dimensions and γ, β are learnable parameters restoring scale and shift.

Types of Normalization

Mean and variance can be computed over different axes:

- **BatchNorm:** mean/var over batch and spatial dims.
- **LayerNorm:** over features within each sample.
- **InstanceNorm:** per sample, per channel.
- **GroupNorm:** per group of channels (robust to batch size).

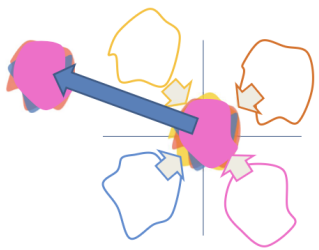


(Batch norm is most common in CNNs, LayerNorm in Transformers, GroupNorm when you have semantically sound ways of grouping channels.)

Batch Normalization intepretation

Historical justification: reduce *internal covariate shift** by normalizing layer inputs, and setting them to a standard location and scale.

* *Internal covariate shift* = distribution of layer inputs changes during training as previous layers update. Think of a layer as trying to learn the mapping from the input distribution to the output.



Batch norm has a weird effect on the computation graph:

- Normalization depends on the entire mini-batch.
- Gradients backpropagate through the mean/variance computation.

[Recent work] questions the internal covariate shift explanation, and argues benefits come from smoother optimization landscape.

Table of Contents

Explicit Regularization

Goal: penalize large or complex parameter values. (aka structural risk minimization, control the capacity of the family of models we are training.)

L2 penalty (weight decay):

$$\mathcal{L}' = \mathcal{L} + \frac{\lambda}{2}\|\theta\|^2$$

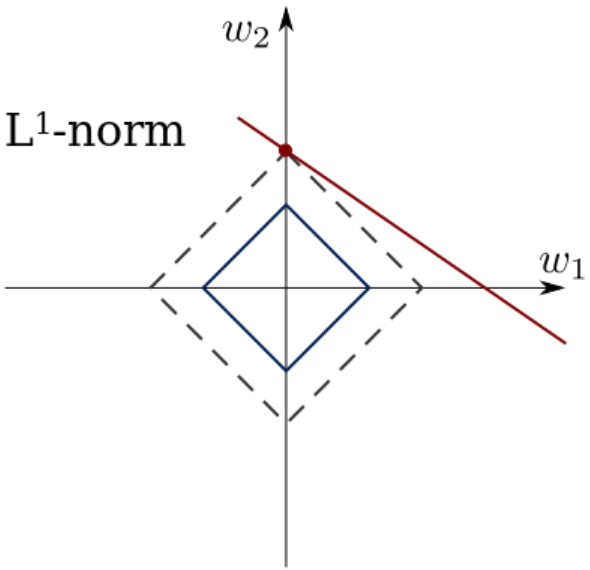
- Shrinks weights smoothly toward zero.
- Encourages small, distributed representations.
- Acts as a Gaussian prior on weights (Bayesian view) (loss is negative log-posterior and training is maximum a posteriori).

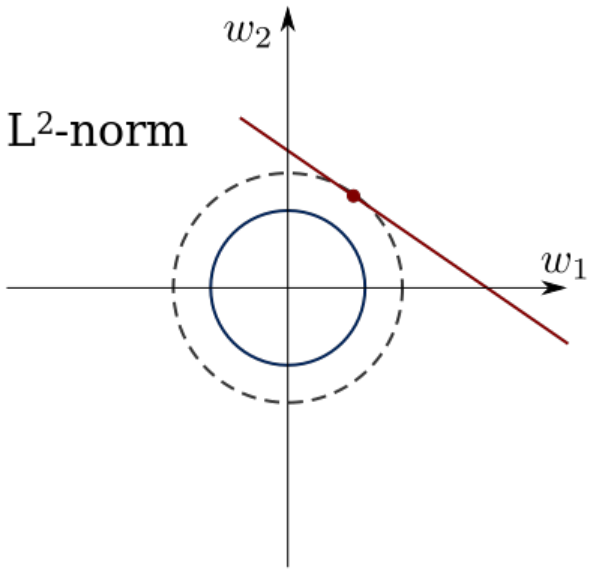
L1 penalty (Lasso):

$$\mathcal{L}' = \mathcal{L} + \lambda\|\theta\|_1$$

- Promotes sparsity: many weights exactly zero.
- Can be for feature selection.
- Acts as a Laplace prior (spiky, heavy-tailed).

Regularization Illustration





Explicit Regularization in Optimizers

Weight decay:

- Implementation of L2 regularization inside optimizers:
  $\theta \leftarrow (1 - \eta\lambda)\theta - \eta\nabla_\theta L$
- Especially in `AdamW`, the decay term is applied directly to parameters, not to gradients : improves stability.

Dropout Principle

Goal: prevent co-adaptation between neurons and improve generalization through noise injection.

Mechanism:

- During training, each activation is randomly zeroed:

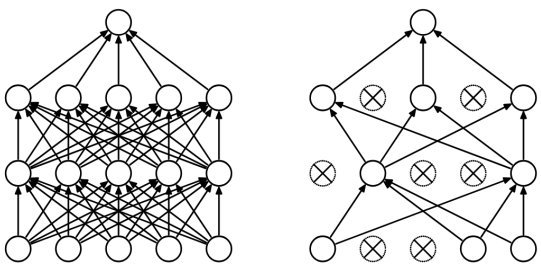
$$h_i \leftarrow m_i h_i, \qquad m_i \sim \text{Bernoulli}(p)$$

  where p is the probability of *keeping* a unit.

- At test time, all units are used but scaled by p:

$$\widehat{h}_i = p\, h_i$$

  ensuring the same expected activation.

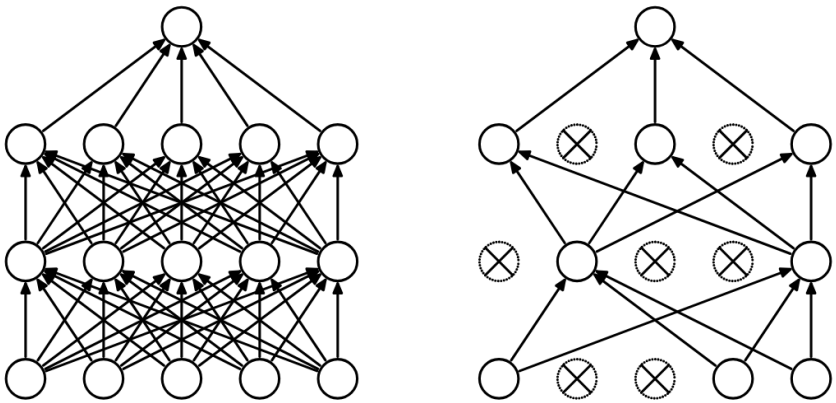


(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right** An example of a thinned net produced by applying dropout to the network on the left Crossed units have been dropped

Srivastava et al., JMLR 2014

Dropout: Effects and Interpretation



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right** An example of a thinned net produced by applying dropout to the network on the left Crossed units have been dropped

Multiple interpretations of dropout:

- Implicitly trains **ensemble of subnetworks** that share weights.
- Acts as an **explicit regularizer**, reducing overfitting.
- Dropout introduces stochasticity that forces robustness: each neuron must perform well under random thinning of its inputs.

Early Stopping

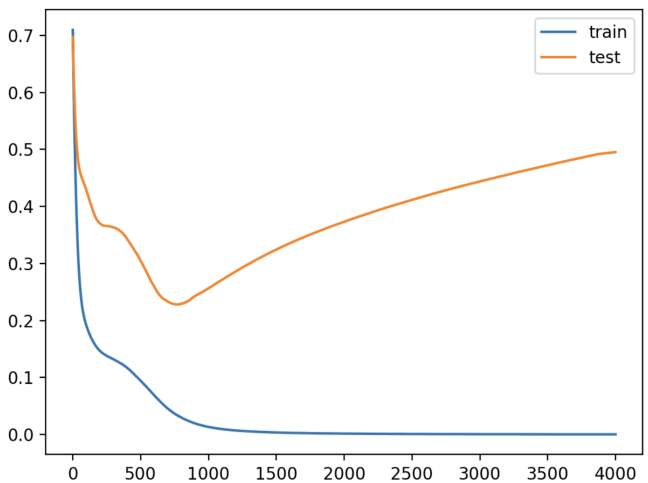
Goal: prevent overfitting by halting training when validation performance stops improving.

Mechanism:

- Split data into training and validation sets.
- Monitor the validation loss L_{val} and stop when L_{val} increases.

Effect:

- Limits effective model capacity (not all weights can be reached within a limited number of steps).
- Reduces risk of memorizing noise in the training set.



Data Augmentation

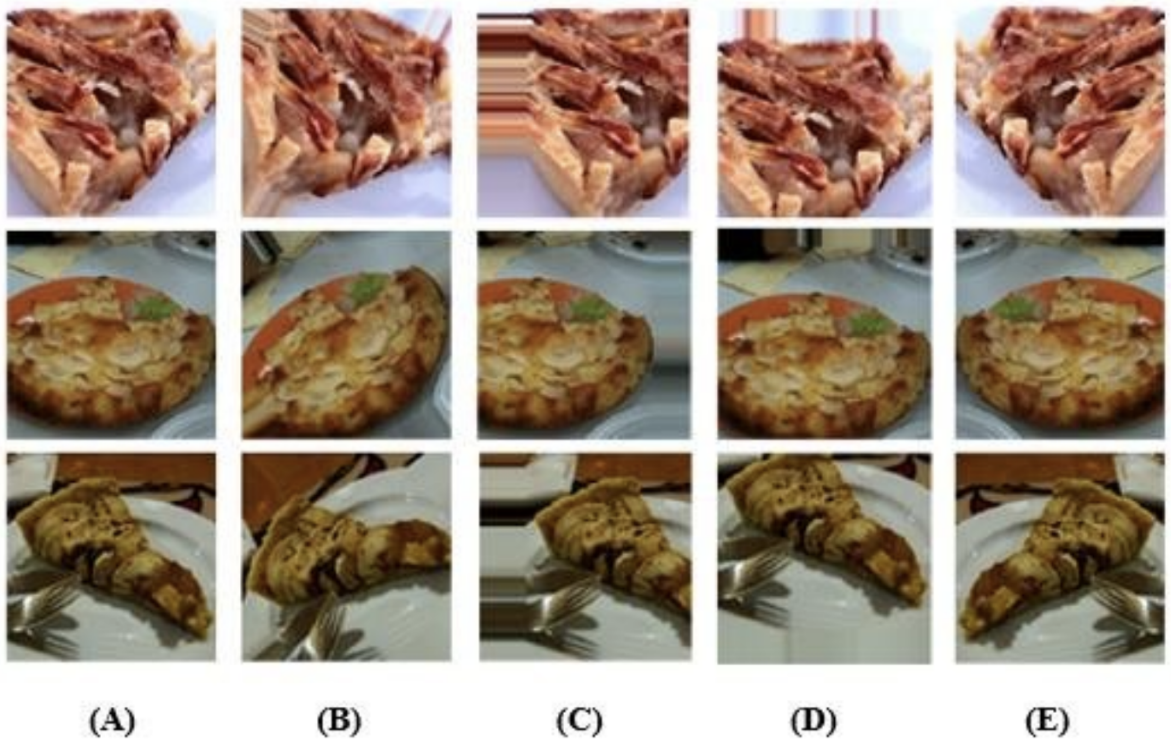
- **Goal:** artificially enlarge the training set by applying label-preserving transformations.
- Encourages **invariance** (e.g. to rotations, translations) and reduces overfitting.
- Common for vision and audio tasks; can also be used for text and time series.

Typical augmentations for images

- Geometric: rotations, flips, crops, translations, elastic deformations
- Photometric: brightness, contrast, hue, noise injection, blurring

No information injected: best to think of it as a regularization method that increases robustness and generalization, or that we teach the model invariances that we want it to have.

Data Augmentation



[A paper with data augmentation]

Implicit Regularization

Observation: the model generalizes well even without explicit penalties because the *training dynamics themselves* favor simple solutions.

Sources of implicit regularization:

- **GD + loss implicit bias** GD + square loss favor small weights solutions when there are multiple global minimizers in overparameterized models.
- **Stochasticity of SGD:** noise in gradients might act as regularization (small batches sometimes outperform large ones).
- **Everything we said before** Eearly stopping, normalization, initialization, data augmentation.

Table of Contents

Conclusion

Many tricks and techniques exist to stabilize and improve training of deep networks.

- Initialization and normalization layers improve gradient flow.
- Regularization techniques (explicit and implicit) help generalization.
- Data augmentation is a powerful way to increase effective training data size.

Many more in practice