Yale University

Multilayer Perceptrons, Backpropagation, and Training

S&DS 265 · Introductory Machine Learning · Lecture 11

Zhuoran Yang

Fall 2026

AI usage: figures were generated, and these slides polished, with the help of AI tools.

Outline

1. Why Deep Learning Now?
2. Depth and Learned Representations
3. Backpropagation and Automatic Differentiation
4. Stabilizing Deep Networks
5. Regularization and the Training Handoff

Learning Objectives

In this lecture you will learn:

1. Why deep learning emerged from the joint growth of architectures, data, computation, and trainability;
2. How depth, width, and nonlinear activations define the MLP model class and its cost;
3. How a scalar chain rule becomes a clear layer-by-layer backpropagation algorithm;
4. How initialization and residual pathways keep signals and gradients usable across depth;
5. Why dropout targets overfitting rather than unstable optimization.

Outline

1. Why Deep Learning Now?
2. Depth and Learned Representations
3. Backpropagation and Automatic Differentiation
4. Stabilizing Deep Networks
5. Regularization and the Training Handoff

Motivating Example: Recognizing Handwritten Digits

An image has $28 \times 28 = 784$ pixel values, while the target is one of ten digits. A linear classifier can weight pixels, but it cannot build a hierarchy of strokes, loops, and shapes.

How can a model learn the intermediate features it needs?

Example data: MNIST handwritten digits, LeCun et al. (1998).

The MNIST Data

Source: MNIST examples; LeCun, Cortes, and Burges.

Observed input

$x_i \in [0,1]^{784}$: a flattened 28×28 grayscale image.

Target

$y_i \in \{0, \dots, 9\}$: the digit identity.

Standard MNIST contains 60,000 training images and 10,000 test images.

Reading One Image

One observation is not a symbol already understood by the model. It is a vector of intensities

$$x_i = (x_{i1}, \ldots, x_{i,784}) \in [0,1]^{784}.$$

The label supplies only the final answer y_i; it does not identify strokes, endpoints, loops, or any other useful intermediate feature.

Problem Setup

Given $\mathcal{D}_n = \{(x_i, y_i)\}_{i=1}^n$, the population target is the conditional distribution $p^*(y \mid x)$. We restrict the estimator to an architecture-dependent MLP class $\mathcal{F}_{\text{MLP}}$ and fit

$$\hat{\theta} \in \arg\min_{\theta} \widehat{R}_n(\theta), \qquad \widehat{R}_n(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \log p_\theta(y_i \mid x_i).$$

Earlier models fixed the features. An MLP learns features and classifier jointly. Its price is many parameters, nonconvex optimization, and representations that are harder to inspect. Held-out cross-entropy and error remain the evidence.

Neural Networks Have a Long History

- The basic neuron and layered computation predate modern software;
- Backpropagation made hidden representations trainable by gradient descent;
- Later breakthroughs changed architectures and the scale of data and computation.

Source: Rosenblatt (1958); Rumelhart, Hinton, and Williams (1986); LeCun et al. (1998); Krizhevsky et al. (2012); He et al. (2016); Vaswani et al. (2017).

1958: The Perceptron Learns a Linear Decision Rule

Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

The first of these questions is in the province of sensory physiology, and is the only one for which appreciable understanding has been achieved. This article will be concerned primarily with the second and third questions, which are still subject to a vast amount of speculation, and where the few relevant facts currently supplied by neurophysiology have not yet been integrated into an acceptable theory.

With regard to the second question, two alternative positions have been maintained. The first suggests that storage of sensory information is in the form of coded representations or images, with some sort of one-to-one mapping between the sensory stimulus and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an organism remembers by reconstructing the original sensory patterns from the "memory traces" which they have left, much as we might develop a photographic negative, or translate the pattern of electrical charges in the "memory" of a digital computer. This hypothesis is appealing in its simplicity and ready intelligibility, and a large family of theoretical brain models has been developed around the idea of a coded, representational memory (**2**, **3**, **9**, **14**). The alternative approach, which stems from the tradition of British empiricism, hazards the guess that the images of stimuli may never really be recorded at all, and that the central nervous system simply acts as an intricate switching network, where retention takes the form of new connections, or pathways, between centers of activity. In many of the more recent developments of this position (Hebb's "cell assembly," and Hull's "cortical anticipatory goal response," for example) the "responses" which are associated to stimuli may be entirely contained within the CNS itself. In this case the response represents an "idea" rather than an action. The important feature of this approach is that there is never any simple mapping of the stimulus into memory, according to some code which would permit its later reconstruction. Whatever in-

[1] The development of this theory has been carried out at the Cornell Aeronautical Laboratory, Inc., under the sponsorship of the Office of Naval Research, Contract Nonr-2381(00). This article is primarily an adaptation of material reported in Ref. 15, which constitutes the first full report on the program.

386

The perceptron adapts a weighted sum of inputs:

$$s(x) = w^\top x + b, \qquad \hat{y} = \mathbf{1}\{s(x) > 0\}.$$

This is an important ancestor of modern networks, but one such unit still produces a linear boundary. Useful intermediate features must either be supplied by hand or learned by additional layers.

Source: First page of Rosenblatt, Psychological Review 65(6), 1958; DOI 10.1037/h0042519.

1986: Backpropagation Learns Hidden Representations

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily interconnected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not have representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom, any number of intermediate layers, and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this input line is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.

Rumelhart, Hinton, and Williams showed how the chain rule can assign a learning signal to hidden units whose desired states are not provided by the labels.

The algorithm learns both the representation and the final prediction rule from one objective.

Source: First page of Rumelhart, Hinton, and Williams, Nature 323, 1986; DOI 10.1038/323533a0.

1998: LeNet Shows End-to-End Recognition

Gradient-Based Learning Applied to Document Recognition

YANN LECUN, MEMBER, IEEE, LÉON BOTTOU, YOSHUA BENGIO, AND PATRICK HAFFNER

Invited Paper

Multilayer neural networks trained with the back-propagation algorithm constitute the best example of a successful gradient-based learning technique. Given an appropriate network architecture, gradient-based learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns, such as handwritten characters, with minimal preprocessing. This paper reviews various methods applied to handwritten character recognition and compares them on a standard handwritten digit recognition task. Convolutional neural networks, which are specifically designed to deal with the variability of two-dimensional (2-D) shapes, are shown to outperform all other techniques.

Real-life document recognition systems are composed of multiple modules including field extraction, segmentation, recognition, and language modeling. A new learning paradigm, called graph transformer networks (GTN's), allows such multimodule systems to be trained globally using gradient-based methods so as to minimize an overall performance measure.

Two systems for online handwriting recognition are described. A large multimodule system using GTN's is described for reading bank checks. It is deployed commercially and reads several millions checks per day.

Keywords—Convolutional neural networks, document recognition, finite-state transducers, gradient-based learning, graph transformer networks, machine learning, neural networks, optical character recognition (OCR).

NOMENCLATURE

GT	Graph transformer.
GTN	Graph transformer network.
HMM	Hidden Markov model.
HOS	Heuristic oversegmentation.
K-NN	K-nearest neighbor.
NN	Neural network.
OCR	Optical character recognition.
PCA	Principal component analysis.
RBF	Radial basis function.
RS-SVM	Reduced-set support vector method.
SDNN	Space displacement neural network.
SVM	Support vector method.
TDNN	Time delay neural network.
V-SVM	Virtual support vector method.

I. INTRODUCTION

Over the last several years, machine learning techniques, particularly when applied to NN's, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the availability of learning techniques and the availability of fast computers are two key factors which have enabled the widespread deployment of sophisticated pattern recognition systems. For example, many of the current commercial systems for reading bank checks or recognizing handwritten postal codes rely on some form of trainable classifier based on neural networks. Similarly, most of the systems that have been demonstrated as successful in competitions such as the U.S. Postal Service recognition contest [1] or the NIST OCR contest [2] are based on learning techniques. Finally, most of the online handwriting recognition systems on pen-based computers use learning techniques of some kind. There are several reasons for this trend. One is the steady improvement of feature extraction methods for speech and image recognition over the last few years. But perhaps more importantly, it has become apparent that no matter how much engineering is applied to the design of feature extractors, a significant amount of tuning of the parameters (or learning) is required to obtain the best performance. Another reason is the recent progress in techniques and algorithms for learning complex decision surfaces. Gradient-based learning techniques applied to neural networks are discussed briefly in Section II. Section III shows why gradient-based learning techniques are more promising than previous methods for automatic learning and less so for hand-designed heuristics. This is made possible by recent progress in machine learning methods and by the availability of fast graphical computers. Section IV shows that hand-crafted feature extraction can be advantageously replaced by carefully designed learning machines that operate directly on pixel images. Using document understanding as a case study, we show that these learning machines can be organized to synergistically integrate individually designed modules so that the overall performance is improved. Section V proposes to go one step further by combining modules into multimodule systems trained globally to minimize a single criterion. Section VI shows that the promise of learning techniques has been largely realized in practical character recognition systems. Because of the modularity of the systems we describe, their performance can be improved dynamically as more data becomes available, and their capacity can be adapted to various tasks by changing some high-level parameters. Section VII discusses in detail the architecture of a system for reading bank checks that is currently deployed commercially and reads millions of checks per day. Section VIII presents a similar system for recognizing handwriting entered directly into computers through pen-based interfaces. Section IX discusses the issue of training very large systems efficiently. Section X concludes with a discussion.

Manuscript received November 1, 1997; revised April 17, 1998. Y. LeCun and L. Bottou are with the Image Processing Services Research Laboratory, AT&T Labs-Research, Red Bank, NJ 07701 USA.

Y. Bengio is with the Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Québec H3C 3J7 Canada.

Publisher Item Identifier S 0018-9219(98)07863-3.

0018–9219/98$10.00 © 1998 IEEE

2278 PROCEEDINGS OF THE IEEE, VOL. 86, NO. 11, NOVEMBER 1998

LeNet combined local connectivity, shared weights, nonlinear layers, and backpropagation for document recognition. It learned features directly from pixels rather than depending only on a hand-designed pipeline.

Lecture 13 will turn these image-specific ideas into convolution and translation-aware feature learning.

Source: First page of LeCun, Bottou, Bengio, and Haffner, Proceedings of the IEEE 86(11), 1998.

2012–2017: Depth Becomes a Scalable Design Principle


ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky, University of Toronto, kriz@cs.utoronto.ca
Ilya Sutskever, University of Toronto, ilya@cs.utoronto.ca
Geoffrey E. Hinton, University of Toronto, hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

1 Introduction

Current approaches to object recognition make essential use of machine learning methods. To improve their performance, we can collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting. Until recently, datasets of labeled images were relatively small — on the order of tens of thousands of images (e.g., NORB [16], Caltech-101/256 [8, 9], and CIFAR-10/100 [12]). Simple recognition tasks can be solved quite well with datasets of this size, especially if they are augmented with label-preserving transformations. For example, the current best error rate on the MNIST digit-recognition task (<0.3%) approaches human performance. But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets. And indeed, the shortcomings of small image sets have been widely recognized (e.g., Pinto et al. [21]), but it has only recently become possible to collect labeled datasets with millions of images. The new larger datasets include LabelMe [23], which consists of hundreds of thousands of fully-segmented images, and ImageNet [6], which consists of over 15 million labeled high-resolution images in over 22,000 categories.

To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don't have. Convolutional neural networks (CNNs) constitute one such class of models [16, 11, 13, 18, 15, 22, 26]. Their capacity can be controlled by varying their depth and breadth, and they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies). Thus, compared to standard feedforward neural networks with similarly-sized layers, CNNs have much fewer connections and parameters and so they are easier to train, while their theoretically-best performance is likely to be only slightly worse.

1

AlexNet, 2012


Deep Residual Learning for Image Recognition

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [41] but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

1. Introduction

Deep convolutional neural networks [22, 21] have led a series of breakthroughs for image classification [21, 49, 39]. Deep networks naturally integrate low/mid/high-level features [49] and classifiers in an end-to-end multi-layer fashion, and the "levels" of features can be enriched by the number of stacked layers (depth). Recent evidence [41, 44] reveals that network depth is of crucial importance, and the leading results [41, 44, 13, 16] on the challenging ImageNet dataset [37] all exploit "very deep" [41] models, with a depth of sixteen [41] to thirty [16]. Many other nontrivial visual recognition tasks [7, 11, 6, 34, 25] have also greatly benefited from very deep models.

Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

greatly benefited from very deep models.

Driven by the significance of depth, a question arises: Is learning better networks as easy as stacking more layers? An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [14, 1, 8], which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization [23, 8, 36, 12] and intermediate normalization layers [16], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with back-propagation [22].

When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error, as reported in [10, 42] and thoroughly verified by our experiments. Fig. 1 shows a typical example.

The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution by construction to the deeper model: the added layers are identity mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart. But experiments show that our current solvers on hand are unable to find solutions that are comparably good or better than the constructed solution (or unable to do so in feasible time).

1

ResNet, 2015/16


Attention Is All You Need

Ashish Vaswani*, Google Brain, avaswani@google.com
Noam Shazeer*, Google Brain, noam@google.com
Niki Parmar*, Google Research, nikip@google.com
Jakob Uszkoreit*, Google Research, usz@google.com
Llion Jones*, Google Research, llion@google.com
Aidan N. Gomez*, University of Toronto, aidan@cs.toronto.edu
Łukasz Kaiser*, Google Brain, lukaszkaiser@google.com
Illia Polosukhin*, illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

1 Introduction

Recurrent neural networks, long short-term memory [13] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation [35, 2, 5]. Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures [38, 24, 15].

Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Recent work has achieved significant improvements in computational efficiency through factorization tricks [21] and conditional computation [32], while also improving model performance in case of the latter. The fundamental constraint of sequential computation, however, remains.

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

†Work performed while at Google Brain.
‡Work performed while at Google Research.

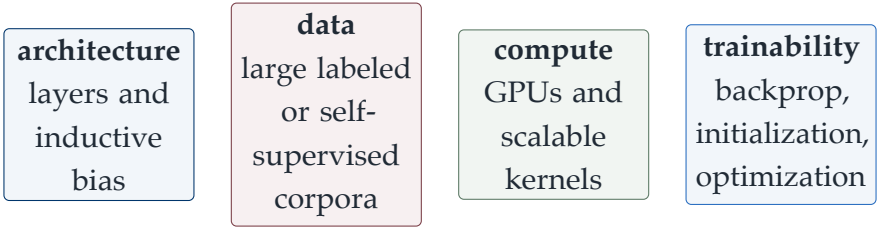
31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

Transformer, 2017

AlexNet paired a large labeled dataset with GPU training; ResNet made much greater depth easier to optimize; Transformers made attention the main sequence-modeling primitive.

Source: First pages of Krizhevsky et al. (NeurIPS 2012), He et al. (CVPR 2016), and Vaswani et al. (NeurIPS 2017).

Modern Success Required Four Ingredients



The mathematics of the chain rule was not new in 2012. What changed was the ability to fit large compositions reliably on rich datasets with useful architectural structure.

Deep learning is architecture, computation, and scale working together.

Outline

From One Hidden Layer to Several

The depth-one regression network from Lecture 6 was

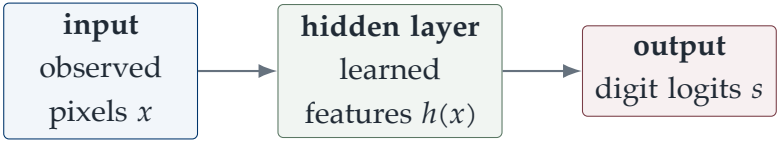
$$h(x) = \sigma(W_1 x + b_1), \qquad f_\theta(x) = a^\top h(x).$$

One hidden layer already turns learned features into a prediction. For digit recognition, we extend the same construction in two ways:

1. Repeat the hidden transformation several times;
2. Use ten output logits and a softmax cross-entropy loss.

Depth lets later features reuse earlier ones. The new question is how to differentiate and train the longer composition.

A Hidden Layer Is an Intermediate Representation



The input and label are observed. Hidden coordinates are not labeled as "stroke" or "loop." Their meaning is determined indirectly by whichever values help reduce the final prediction loss.

Hidden means internal to the model, not mysterious or unobservable to us.

An MLP Is a Composition of Layers

Let $h^{(0)} = x$. For hidden layers $\ell = 1, \dots, L$,

$$z^{(\ell)} = W^{(\ell)} h^{(\ell-1)} + b^{(\ell)}, \qquad h^{(\ell)} = \sigma_\ell\big(z^{(\ell)}\big).$$

The output layer produces logits and probabilities:

$$s = W^{(L+1)} h^{(L)} + b^{(L+1)}, \qquad p_k = \frac{e^{s_k}}{\sum_{j=1}^{10} e^{s_j}}.$$

Every coordinate in one layer connects to every coordinate in the next, hence **fully connected**.

Activations Keep the Composition Nonlinear

Lecture 6 introduced the main choices:

$$\mathrm{ReLU}(z) = \max(0, z), \qquad \mathrm{GELU}(z) = z\Phi(z), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

- ReLU is cheap and piecewise linear.
- GELU is a smooth gate used in many modern transformer MLP blocks.
- Sigmoid is useful for probabilities, but its flat tails can attenuate hidden-layer gradients.

Without an activation, several affine layers collapse to one affine map; the nonlinear gate is what enlarges the function class.

The MNIST MLP Has Explicit Tensor Shapes

For a minibatch of B images and widths $784 \to 256 \to 128 \to 10$,

$$X : [B, 784] \longrightarrow H^{(1)} : [B, 256] \longrightarrow H^{(2)} : [B, 128] \longrightarrow S : [B, 10].$$

The parameters have shapes

$$W^{(1)} : [256, 784], \quad W^{(2)} : [128, 256], \quad W^{(3)} : [10, 128],$$

plus one bias per output coordinate.

Writing shapes before code catches many broadcasting and orientation errors.

A Small MLP Already Has Many Parameters

The parameter count is

$$(784 \cdot 256 + 256) + (256 \cdot 128 + 128) + (128 \cdot 10 + 10)$$
$$= \boxed{235{,}146}.$$

That is modest by modern standards, but already enough that writing and updating each derivative separately is impractical.

Architecture determines the function class and the cost of parameters, activations, gradients, and optimizer state.

Width and Depth Enlarge the Function Class Differently

Width

Adds parallel features within a layer. A sufficiently wide one-hidden- layer network can approximate many continuous functions on compact sets.

Depth

Composes features. Later units can reuse patterns built by earlier units, often representing compositional structure more efficiently.

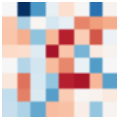
Universal approximation is an existence result; it does not say that finite data and a chosen optimizer will find the useful function.

Hidden Features Become Class-Specific Templates

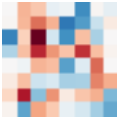
Unit 1

Unit 2

Unit 3



Unit 4

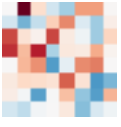


Unit 5

Unit 6

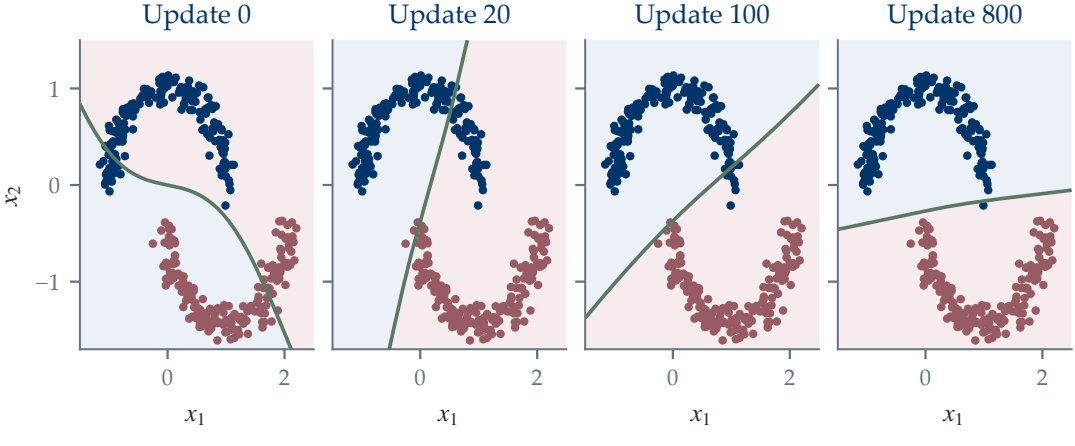
Unit 7

Unit 8



Course calculation from a fitted MLP on handwritten digits.

The Decision Rule Changes Throughout Training



Course from-scratch MLP on a declared two-dimensional classification task.

Classroom Demo: Watch an MLP Learn

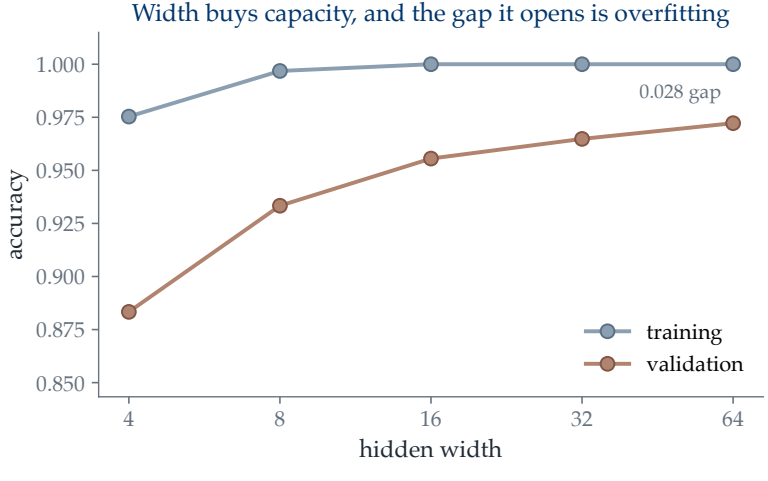
Change one quantity at a time in a two-dimensional classification problem:

1. Compare a narrow and a wider hidden layer;
2. Advance optimization updates and watch the boundary move;
3. Increase the step size until training becomes unstable;
4. Reset the random seed and compare the learned representation.

Question for the class

Which changes alter the function class, and which alter only the training path within that class?

Width and Validation Accuracy



Course fixed-split experiment on handwritten digits.

A Forward Pass Must Retain Intermediate Values

To compute predictions, the model forms

$$X,\ Z^{(1)},\ H^{(1)},\ldots,Z^{(L)},\ H^{(L)},\ S,\ L.$$

Backward rules need selected forward values such as $H^{(\ell-1)}$, $Z^{(\ell)}$, and the logits or probabilities.

Training-memory consequence

Parameters are not the only large objects. Activations scale with minibatch size, sequence or image size, width, and depth; they are normally retained until their gradients have been computed.

Outline

Training Requires One Gradient per Parameter

The loss is one number, but the MLP has many parameters:

$$L(\theta), \qquad \theta = \{W^{(1)}, b^{(1)}, \dots, W^{(L+1)}, b^{(L+1)}\}.$$

Gradient descent needs

$$\nabla_\theta L = \left(\nabla_{W^{(1)}} L, \nabla_{b^{(1)}} L, \dots, \nabla_{W^{(L+1)}} L, \nabla_{b^{(L+1)}} L\right).$$

Computing each derivative independently would repeat most of the same work. **Backpropagation** reuses intermediate calculations while applying the chain rule from the loss back to every parameter.

Begin with the Scalar Chain Rule

Suppose

$$u = f(v), \qquad L = g(u).$$

Then

$$\frac{dL}{dv} = \frac{dL}{du}\frac{du}{dv}.$$

Read the two colored factors as

$$\underbrace{\text{how the loss changes with } u}_{\text{arrives from downstream}} \times \underbrace{\text{how } u \text{ changes with } v}_{\text{local rule}}.$$

Backpropagation repeatedly performs this multiplication in reverse computational order.

A Short Notation for Loss Gradients

For any intermediate value v, define

$$\boxed{\overline{v} := \nabla_v L}.$$

The overline always means **gradient of the final loss with respect to this quantity**. It has the same shape as v.

The scalar chain rule becomes

$$\overline{v} = \overline{u}\frac{du}{dv}.$$

incoming gradient $\times$ local derivative $=$ gradient passed backward

Worked Scalar Neuron: Forward Pass

Consider one ReLU unit with a squared-error loss:

$$z = wx + b, \qquad h = \max(0, z), \qquad L = \frac{1}{2}(h - y)^2.$$

Use

$$x = 2, \qquad w = -1, \qquad b = 3, \qquad y = 4.$$

The forward values are

$$z = (-1)(2) + 3 = 1, \qquad h = 1, \qquad L = \frac{1}{2}(1 - 4)^2 = 4.5.$$

These saved values are exactly what the local backward rules will use.

Worked Scalar Neuron: Backward Pass

Start at the loss and move backward:

$$\overline{h} = \frac{\partial L}{\partial h} = h - y = -3,$$
$$\overline{z} = \overline{h}\mathbf{1}\{z > 0\} = -3,$$
$$\overline{w} = \overline{z}\frac{\partial z}{\partial w} = (-3)x = -6,$$
$$\overline{b} = \overline{z}\frac{\partial z}{\partial b} = -3.$$

Gradient descent increases w because $w \leftarrow w - \eta\overline{w}$ and $\overline{w} < 0$, moving the prediction toward $y = 4$.

One Vector Layer Has Two Local Operations

For an input vector $h \in \mathbb{R}^d$ and m output units,

$$\underbrace{z = Wh + b}_{\text{affine map}}, \qquad \underbrace{u = \sigma(z)}_{\text{elementwise activation}},$$

with

$$W \in \mathbb{R}^{m \times d}, \qquad b, z, u \in \mathbb{R}^m.$$

The backward pass receives $\overline{u} = \nabla_u L$ and must compute

$$\overline{z}, \qquad \overline{W}, \qquad \overline{b}, \qquad \overline{h}.$$

We handle the activation and affine map separately.

Backward through an Elementwise Activation

Each output coordinate satisfies $u_j = \sigma(z_j)$, so

$$\frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial u_j} \sigma'(z_j).$$

Stacking all coordinates gives

$$\boxed{\overline{z} = \overline{u} \odot \sigma'(z)}.$$

The Hadamard product $\odot$ multiplies coordinate by coordinate.

For ReLU, $\sigma'(z_j)$ is 1 when $z_j > 0$ and 0 when $z_j < 0$; the gate either passes or blocks that coordinate's gradient.

Backward through an Affine Map

For $z = Wh + b$, the incoming gradient is $\bar{z}$. The local rules are

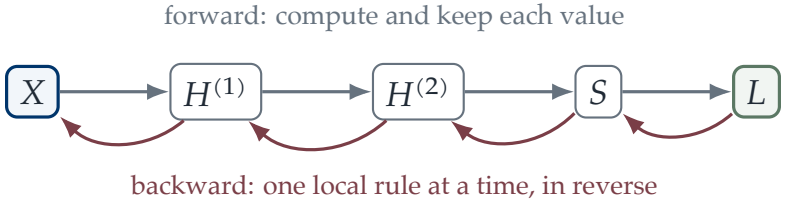
$$\boxed{\overline{W} = \bar{z}h^\top}, \qquad \boxed{\overline{b} = \bar{z}}, \qquad \boxed{\overline{h} = W^\top \bar{z}}.$$

Their shapes verify the formulas:

$$[m \times d] = [m \times 1][1 \times d], \qquad [d \times 1] = [d \times m][m \times 1].$$

The outer product $\bar{z}h^\top$ assigns a gradient to every connection between the input and output coordinates.

Backpropagation Traverses the Graph in Reverse



The forward pass computes and saves intermediate values. The backward pass begins with the scalar loss and applies each local rule in reverse order. Contributions add when several downstream paths depend on the same value.

A One-Hidden-Layer Classifier Backpropagates in Order

For one example,

$$z^{(1)} = W^{(1)}x + b^{(1)}, \quad h^{(1)} = \sigma(z^{(1)}), \quad s = W^{(2)}h^{(1)} + b^{(2)}.$$

Softmax cross-entropy starts the reverse pass with

$$\bar{s} = p - e_y.$$

Then compute, in this order,

$$\overline{W}^{(2)} = \bar{s}\, h^{(1)\top}, \qquad \bar{h}^{(1)} = W^{(2)\top}\bar{s},$$
$$\bar{z}^{(1)} = \bar{h}^{(1)} \odot \sigma'(z^{(1)}), \qquad \overline{W}^{(1)} = \bar{z}^{(1)}x^{\top}.$$

The Same Gradient Pattern Appears in Every Model

model	error signal	times feature
least squares (L04)	$X\theta - y$	x_i
logistic (L09)	$p_i - y_i$	x_i
softmax (L09)	$p_i - e_{y_i}$	x_i
MLP layer ℓ	$\bar{z}^{(\ell)}$	$h^{(\ell-1)}$

At the output, labels provide the error signal $p - e_y$. A hidden layer has no target label; its error signal is produced by the chain rule from the layers that follow it.

A Minibatch Produces Matrix Gradients

Store observations as rows:

$$Z = HW^\top + \mathbf{1}b^\top, \qquad H \in \mathbb{R}^{B\times d}, \quad Z, \overline{Z} \in \mathbb{R}^{B\times m}.$$

Summing the B example contributions gives

$$\boxed{\overline{W} = \overline{Z}^\top H}, \qquad \boxed{\overline{b} = \overline{Z}^\top \mathbf{1}}, \qquad \boxed{\overline{H} = \overline{Z}W}.$$

If the loss averages the minibatch, its factor $1/B$ is already contained in $\overline{Z}$.

Automatic Differentiation Executes These Local Rules

During the forward pass, PyTorch records the operations actually executed and the values required by their backward rules. Calling

```
loss.backward()
```

traverses that graph in reverse and accumulates every leaf parameter's gradient in its `.grad` field.

Autodiff removes derivative bookkeeping; it does not choose the loss, initialize the model, select the optimizer, or guarantee a correct program.

Source: The demo notebook checks the hand-derived gradients against `loss.backward()` and against finite differences on a small deterministic problem.

One PyTorch Training Step

```
optimizer.zero_grad()
logits = model(x_batch)        # forward; save activations
loss = loss_fn(logits, y_batch)
loss.backward()                # reverse mode; fill .grad
optimizer.step()               # update parameters and state
```

The four operations have different roles. In particular, `backward()` computes gradients but does not update the model.

Outline

One Problem, Several Remedies

Backpropagation multiplies the gradient by one factor per layer, so depth compounds whatever that factor is. Too small and the early layers stop learning; too large and the updates blow up.

One Problem, Several Remedies

Backpropagation multiplies the gradient by one factor per layer, so depth compounds whatever that factor is. Too small and the early layers stop learning; too large and the updates blow up.

Guiding principle: keep that factor near one, forward and backward alike.

One Problem, Several Remedies

Backpropagation multiplies the gradient by one factor per layer, so depth compounds whatever that factor is. Too small and the early layers stop learning; too large and the updates blow up.

Guiding principle: keep that factor near one, forward and backward alike.

- **Initialization** sets the factor to about one at the start, through the scale of the random weights.
- **Skip connections** keep a factor of exactly one available at all times, through an identity path.
- **Clipping** and **schedules** bound the step when the factor is wrong anyway.

Zero Initialization Makes Hidden Units Identical

If every row of W_1 and every entry of b_1 begins equal, then every hidden unit produces the same value and receives the same gradient.

Gradient descent preserves this symmetry, so width does not create distinct learned features.

First requirement

Random initialization breaks symmetry among hidden units.

Arbitrary Random Scale Can Destroy the Signal

Suppose $z_j = \sum_{k=1}^{n_{\text{in}}} W_{jk} h_k$, with independent centered terms. Approximately,

$$\operatorname{Var}(z_j) \approx n_{\text{in}} \operatorname{Var}(W_{jk}) \operatorname{Var}(h_k).$$

Repeating the layer can make activations and gradients vanish or explode if this scale changes systematically with depth.

Initialization is an architectural calculation, not just a random seed.

Xavier Balances Forward and Backward Scale

Glorot–Xavier initialization uses

$$\boxed{\mathrm{Var}(W_{jk}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

It compromises between preserving the forward activation variance ($1/n_{\text{in}}$) and the backward gradient variance ($1/n_{\text{out}}$).

This argument assumes centered, weakly dependent signals and is best matched to tanh-like activations.

Source: Glorot and Bengio, 2010.

He Initialization Accounts for ReLU Gating

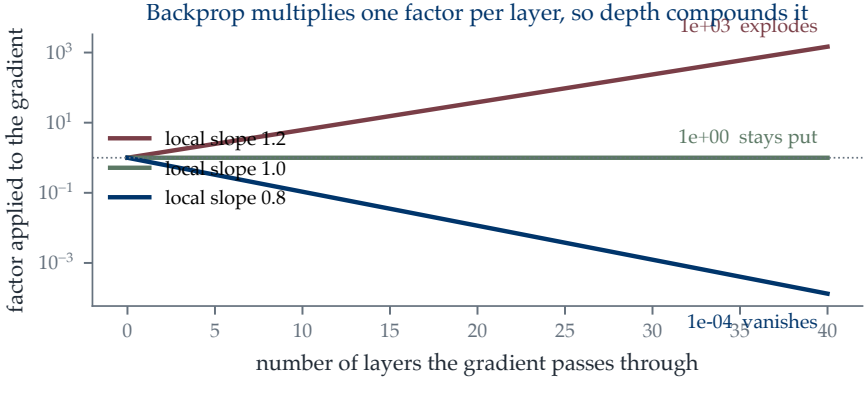
A centered symmetric input passes through a ReLU gate only about half the time, motivating

$$\boxed{\mathrm{Var}(W_{jk}) \approx \frac{2}{n_{\text{in}}}}.$$

PyTorch layers expose initialization explicitly; the appropriate rule depends on the activation and parameterization.

A good initialization places training in a usable scale regime. It does not eliminate the need to monitor activations and gradient norms.

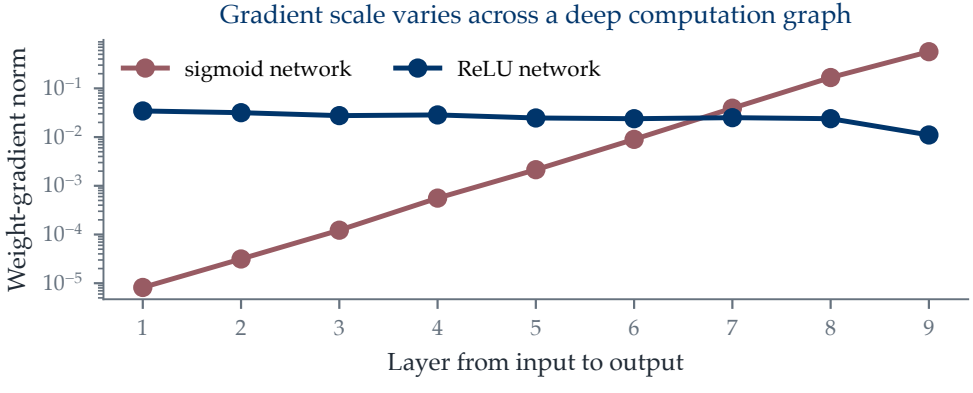
Depth Multiplies Local Gradient Scales



Backpropagation multiplies the gradient by one local slope per layer, so after L layers the factor is $(\text{slope})^{L}$. A slope of 0.8 leaves 10^{-4} of the gradient after 40 layers, 1.2 leaves 10^{3} times too much, and only 1.0 transmits it unchanged.

Constant local slopes, so the product is exact.

Initialization and Activation Change Layerwise Gradients



The same nine-layer network on one digits minibatch, once with sigmoid and once with ReLU. The sigmoid gradient shrinks by four orders of magnitude between the output layer and the input layer, so the early layers barely move; with ReLU the norm is about the same in every layer.

Course calculation on a fixed handwritten-digits minibatch.

A Residual Block Learns a Correction

A plain block must learn the whole mapping, $h^{(\ell+1)} = F_\ell(h^{(\ell)})$. A residual block learns only what to add:

$$\boxed{h^{(\ell+1)} = h^{(\ell)} + F_\ell(h^{(\ell)})}$$

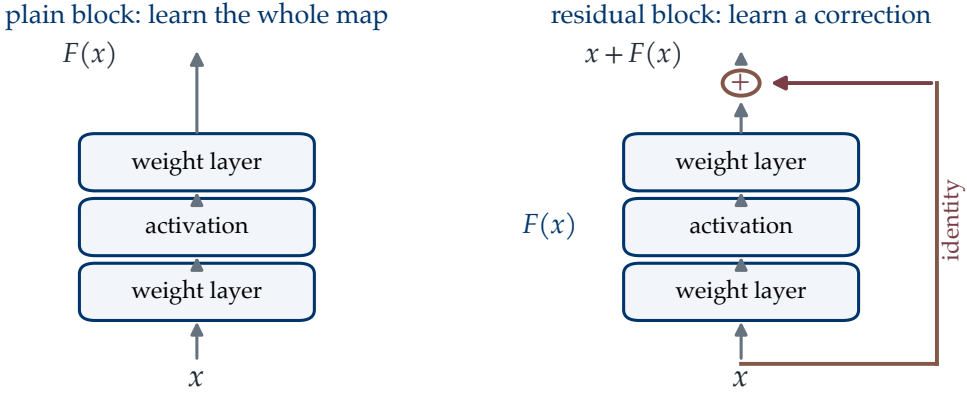
A Residual Block Learns a Correction

A plain block must learn the whole mapping, $h^{(\ell+1)} = F_\ell(h^{(\ell)})$. A residual block learns only what to add:

$$\boxed{h^{(\ell+1)} = h^{(\ell)} + F_\ell(h^{(\ell)})}$$

- If the useful transformation is near the identity, the branch has little left to learn;
- Setting $F_\ell = 0$ recovers the identity exactly, so extra depth cannot hurt the function class.

Plain Block against Residual Block



Same two weight layers and activation in both. The residual block adds one wire: the input is carried around the block and added to its output, so the block only has to produce the correction.

The construction is from He et al. (2016).

The Skip Path Gives the Gradient a Direct Route

With $y = x + F(x)$ the block Jacobian is $\partial y / \partial x = I + J_F(x)$, so an incoming gradient $\overline{y}$ leaves as

$$\overline{x} = \underbrace{\overline{y}}_{\text{direct}} + \underbrace{J_F(x)^\top \overline{y}}_{\text{through the block}}$$

The Skip Path Gives the Gradient a Direct Route

With $y = x + F(x)$ the block Jacobian is $\partial y / \partial x = I + J_F(x)$, so an incoming gradient $\overline{y}$ leaves as

$$\overline{x} = \underbrace{\overline{y}}_{\text{direct}} + \underbrace{J_F(x)^\top \overline{y}}_{\text{through the block}}$$

- The first term crosses untouched: nothing to shrink it;
- The second adds the learned correction;
- Stacked L deep, the direct terms give a path of factor exactly one.

The Skip Path Gives the Gradient a Direct Route

With $y = x + F(x)$ the block Jacobian is $\partial y / \partial x = I + J_F(x)$, so an incoming gradient $\overline{y}$ leaves as

$$\overline{x} = \underbrace{\overline{y}}_{\text{direct}} + \underbrace{J_F(x)^\top \overline{y}}_{\text{through the block}}$$

- The first term crosses untouched: nothing to shrink it;
- The second adds the learned correction;
- Stacked L deep, the direct terms give a path of factor exactly one.

Trainability only: skips need matching shapes, and set neither the learning rate nor the amount of overfitting.

Two More Tricks, on the Optimizer Side

Initialization and skips keep the per-layer factor near one. Two further devices act on the step itself, when it is not.

Two More Tricks, on the Optimizer Side

Initialization and skips keep the per-layer factor near one. Two further devices act on the step itself, when it is not.

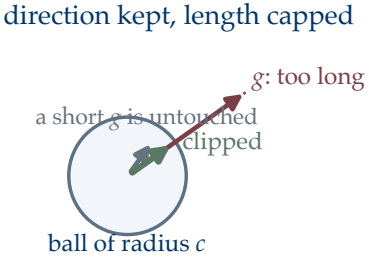
Gradient clipping. If the gradient is too long, shorten it but keep its direction:

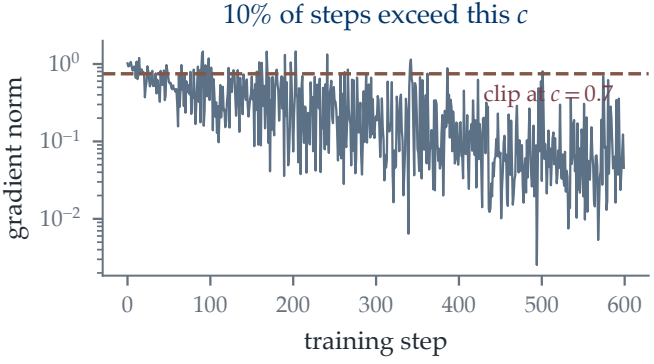
$$g \leftarrow g \cdot \min\Big(1, \frac{c}{\|g\|_2}\Big).$$

A single bad minibatch can then no longer throw the parameters across the landscape.

Standard in language-model training, where a rare long sequence can produce a gradient many times the usual size.

Clipping Caps the Length, Not the Direction





Left: a gradient longer than c is pulled back onto the ball of radius c; a short one is left alone. Right: gradient norms during a real run of a three-layer network — mostly near 0.2, with spikes eight times larger that clipping removes.

Course computation on the handwritten-digits data; c set at the 90th percentile of the observed norms.

Learning-Rate Schedules

The step size need not be constant. A common schedule has two phases:

- **Warmup**: start small and increase over the first few thousand steps, while the parameters are still far from anything sensible and the curvature estimate is poor.
- **Decay**: reduce the rate as training proceeds — cosine or step decay — so late updates refine rather than jump.

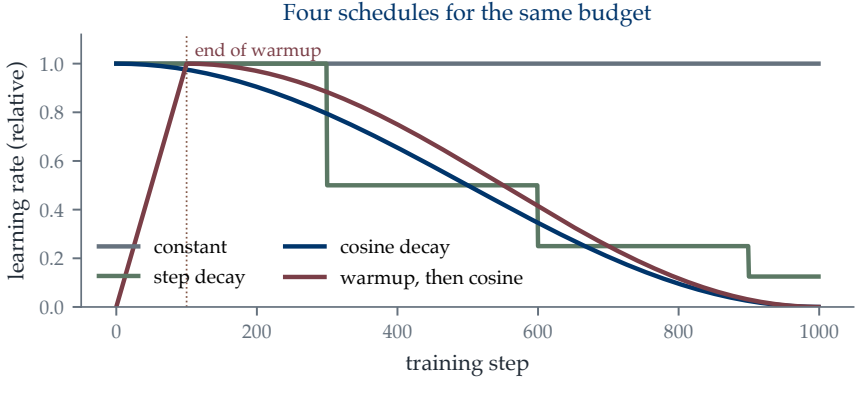
Learning-Rate Schedules

The step size need not be constant. A common schedule has two phases:

- **Warmup**: start small and increase over the first few thousand steps, while the parameters are still far from anything sensible and the curvature estimate is poor.
- **Decay**: reduce the rate as training proceeds — cosine or step decay — so late updates refine rather than jump.

None of the four devices changes the model class. They change whether the optimizer can reach a good member of it.

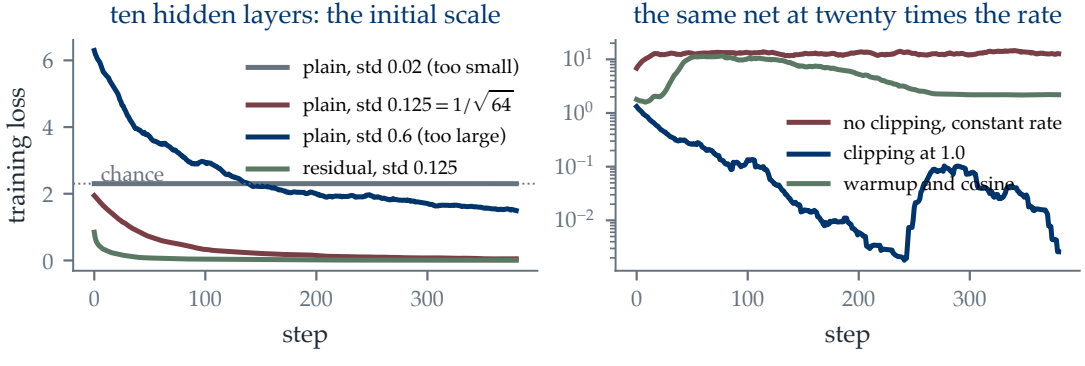
Four Schedules for the Same Budget



A constant rate, step decay, cosine decay, and a short warmup followed by cosine. Warmup keeps the first steps small while the parameters are still arbitrary; decay makes the last steps refine rather than jump.

Relative rates over a fixed budget of 1,000 steps.

The Tricks, Measured



Left: ten tanh layers on digits. At std 0.02 the signal dies and the loss sits at chance; at 0.6 the units saturate and it crawls to 1.5; at $1/\sqrt{64}$ it reaches 0.05, and the residual version 0.006. Right: at twenty times the rate the plain run diverges to 12, while clipping reaches 0.003.

SGD, batch 128, 400 steps, loss smoothed over 20; right panel on a log scale.

Outline

Dropout Randomly Removes Hidden Coordinates

For a hidden vector h and dropout probability p, draw $m_j \sim \text{Bernoulli}(1-p)$ independently and use

$$\boxed{\tilde{h}_j = \frac{m_j}{1-p} h_j}$$

Dropout Randomly Removes Hidden Coordinates

For a hidden vector h and dropout probability p, draw $m_j \sim \text{Bernoulli}(1-p)$ independently and use

$$\boxed{\tilde{h}_j = \frac{m_j}{1-p} h_j}$$

Dividing by $1-p$ keeps the scale: $\mathbb{E}[\tilde{h}_j \mid h_j] = h_j$. Each minibatch therefore trains a randomly thinned network, and no single unit can be relied upon.

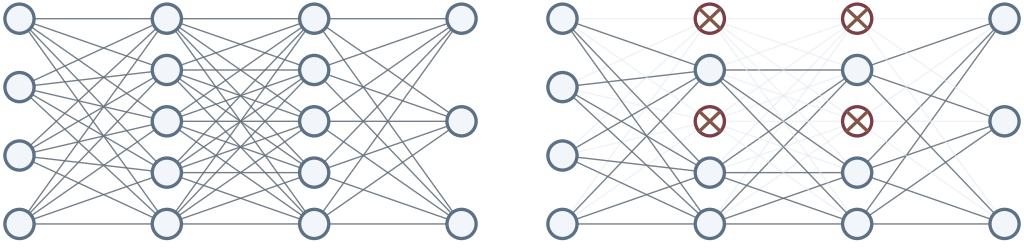
Source: Srivastava et al. (2014).

A Different Thinned Network Every Minibatch

Dropout trains a different thinned network on every minibatch

all units present

one dropout mask



Left: the full network. Right: one draw of the mask, with two hidden units in each layer switched off and every edge touching them idle. The next minibatch draws a different mask.

Dropout is from Srivastava et al. (2014).

Dropout Targets Overfitting, Not Gradient Instability

mode	transformation	purpose
training	$\tilde{h} = m \odot h/(1-p)$	noisy regularized fitting
evaluation	$\tilde{h} = h$	deterministic prediction

Dropout discourages fragile co-adaptation of hidden coordinates and can reduce a train–validation gap. It may also make optimization noisier.

In PyTorch, `model.train()` activates dropout and `model.eval()` disables it. Forgetting the mode switch makes evaluation stochastic and changes the model being assessed.

Different Failure Modes Need Different Interventions

symptom	primary intervention	role
vanishing/exploding scale	Xavier or He	usable initial scale
degradation with depth	residual path	direct signal route
saturated activations	activation/normalization	responsive units
unstable updates	rate/clipping/optimizer	controlled motion
train–validation gap	dropout/decay/augmentation	regularization

These mechanisms solve different problems. Lecture 12 studies the update rule; Lecture 13 develops normalization in the concrete setting of image channels.

Summary: Backpropagation and Initialization

1. For any intermediate v, $\overline{v} = \nabla_v L$ has the same shape as v.
2. Backpropagation combines an incoming gradient with each local derivative.
3. Initialization controls the starting scale; residual paths provide a direct route through depth.

$$\overline{W}^{(\ell)} = \overline{z}^{(\ell)} h^{(\ell-1)\top}, \qquad h^{(\ell+1)} = h^{(\ell)} + F_\ell(h^{(\ell)}).$$

$$\mathrm{Var}(W_{jk}) \approx \frac{2}{d_{\text{in}} + d_{\text{out}}} \text{ (Xavier)}, \qquad \mathrm{Var}(W_{jk}) \approx \frac{2}{d_{\text{in}}} \text{ (He/ReLU)}.$$

Summary: Regularization Is a Separate Question

1. Dropout masks hidden coordinates during training and rescales survivors so the conditional expectation is unchanged.
2. Dropout is disabled for deterministic evaluation.
3. It targets overfitting; residual connections and initialization target trainability.

$$\tilde{h}_j = \frac{m_j}{1-p} h_j, \qquad m_j \sim \text{Bernoulli}(1-p).$$

Next Lecture

Backpropagation supplies a gradient, while initialization and residual paths keep the network trainable. A practical run still depends on which examples define each update, what state the optimizer stores, and how its rate changes.

Lecture 12 — Optimization for Deep Learning connects shuffled data loaders, minibatches, epochs, schedules, and adaptive optimizers in one complete training procedure.

Reading for this lecture: ISLP Chapter 10; D2L Chapter 5, Multilayer Perceptrons; DL Chapter 6.