A Selective Overview of Deep Learning

Jianqing Fan* Cong Ma‡ Yiqiao Zhong*

October 11, 2019

Abstract

Deep learning has arguably achieved tremendous success in recent years. In simple words, deep learning uses the composition of many nonlinear functions to model the complex dependency between input features and labels. While neural networks have a long history, recent advances have greatly improved their performance in computer vision, natural language processing, etc. From the statistical and scientific perspective, it is natural to ask: What is deep learning? What are the new characteristics of deep learning, compared with classical methods? What are the theoretical foundations of deep learning?

To answer these questions, we introduce common neural network models (e.g., convolutional neural nets, recurrent neural nets, generative adversarial nets) and training techniques (e.g., stochastic gradient descent, dropout, batch normalization) from a statistical point of view. Along the way, we highlight new characteristics of deep learning (including depth and over-parametrization) and explain their practical and theoretical benefits. We also sample recent results on theories of deep learning, many of which are only suggestive. While a complete understanding of deep learning remains elusive, we hope that our perspectives and discussions serve as a stimulus for new statistical research.

Keywords: neural networks, over-parametrization, stochastic gradient descent, approximation theory, generalization error.

Contents

Author names are sorted alphabetically.

*Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544, USA; Email: {jqfan, congm, yiqiaoz}@princeton.edu.

1 Introduction

Modern machine learning and statistics deal with the problem of *learning from data*: given a training dataset $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the input and $y_i \in \mathbb{R}$ is the output[1], one seeks a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ from a certain function class $\mathcal{F}$ that has good prediction performance on test data. This problem is of fundamental significance and finds applications in numerous scenarios. For instance, in image recognition, the input $\boldsymbol{x}$ (reps. the output y) corresponds to the raw image (reps. its category) and the goal is to find a mapping $f(\cdot)$ that can classify future images accurately. Decades of research efforts in statistical machine learning have been devoted to developing methods to find $f(\cdot)$ efficiently with provable guarantees. Prominent examples include linear classifiers (e.g., linear / logistic regression, linear discriminant analysis), kernel methods (e.g., support vector machines), tree-based methods (e.g., decision trees, random forests), nonparametric regression (e.g., nearest neighbors, local kernel smoothing), etc. Roughly speaking, each aforementioned method corresponds to a different function class $\mathcal{F}$ from which the final classifier $f(\cdot)$ is chosen.

Deep learning [**?**], in its simplest form, proposes the following *compositional* function class:

$$\left\{ f(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{W}_L \boldsymbol{\sigma}_L(\mathbf{W}_{L-1} \cdots \boldsymbol{\sigma}_2(\mathbf{W}_2 \boldsymbol{\sigma}_1(\mathbf{W}_1 \boldsymbol{x}))) \;\middle|\; \boldsymbol{\theta} = \{\mathbf{W}_1, \ldots, \mathbf{W}_L\} \right\}. \tag{1}$$

Here, for each $1 \le l \le L$, $\boldsymbol{\sigma}_\ell(\cdot)$ is some nonlinear function, and $\boldsymbol{\theta} = \{\mathbf{W}_1, \ldots, \mathbf{W}_L\}$ consists of matrices with appropriate sizes. Though simple, deep learning has made significant progress towards addressing the problem of learning from data over the past decade. Specifically, it has performed close to or better than humans in various important tasks in artificial intelligence, including image recognition [**?**], game playing [**?**], and machine translation [**?**]. Owing to its great promise, the impact of deep learning is also growing rapidly in areas beyond artificial intelligence; examples include statistics [**?**, **?**, **?**, **?**, **?**], applied mathematics [**?**, **?**], clinical research [**?**], etc.

Table 1: Winning models for ILSVRC image classification challenge.

Model	Year	# Layers	# Params	Top-5 error
Shallow	< 2012	—	—	$> 25\%$
AlexNet	2012	8	61M	16.4%
VGG19	2014	19	144M	7.3%
GoogleNet	2014	22	7M	6.7%
ResNet-152	2015	152	60M	3.6%

To get a better idea of the success of deep learning, let us take the ImageNet Challenge [**?**] (also known as ILSVRC) as an example. In the classification task, one is given a training dataset consisting of 1.2

[1] When the label y is given, this problem is often known as *supervised learning*. We mainly focus on this paradigm throughout this paper and remark sparingly on its counterpart, *unsupervised learning*, where y is not given.

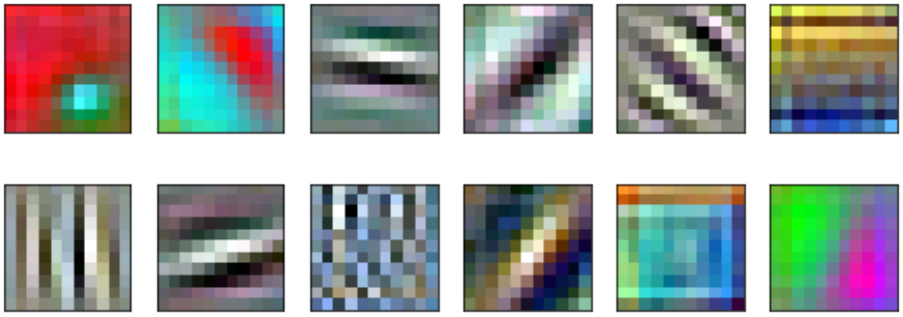


Figure 1: Visualization of trained filters in the first layer of AlexNet. The model is pre-trained on ImageNet and is downloadable via PyTorch package `torchvision.models`. Each filter contains $11 \times 11 \times 3$ parameters and is shown as an RGB color map of size 11×11.

million color images with 1000 categories, and the goal is to classify images based on the input pixels. The performance of a classifier is then evaluated on a test dataset of 100 thousand images, and in the end the top-5 error[2] is reported. Table 1 highlights a few popular models and their corresponding performance. As can be seen, deep learning models (the second to the last rows) have a clear edge over shallow models (the first row) that fit linear models / tree-based models on handcrafted features. This significant improvement raises a foundational question:

Why is deep learning better than classical methods on tasks like image recognition?

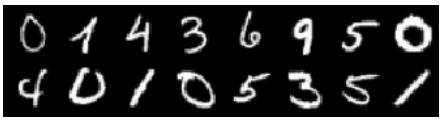
1.1 Intriguing new characteristics of deep learning

It is widely acknowledged that two indispensable factors contribute to the success of deep learning, namely (1) huge datasets that often contain millions of samples and (2) immense computing power resulting from clusters of graphics processing units (GPUs). Admittedly, these resources are only recently available: the latter allows to train larger neural networks which reduces biases and the former enables variance reduction. However, these two alone are not sufficient to explain the mystery of deep learning due to some of its "dreadful" characteristics: (1) *over-parametrization*: the number of parameters in state-of-the-art deep learning models is often much larger than the sample size (see Table 1), which gives them the potential to overfit the training data, and (2) *nonconvexity*: even with the help of GPUs, training deep learning models is still NP-hard [**?**] in the worst case due to the highly nonconvex loss function to minimize. In reality, these characteristics are far from nightmares. This sharp difference motivates us to take a closer look at the salient features of deep learning, which we single out a few below.

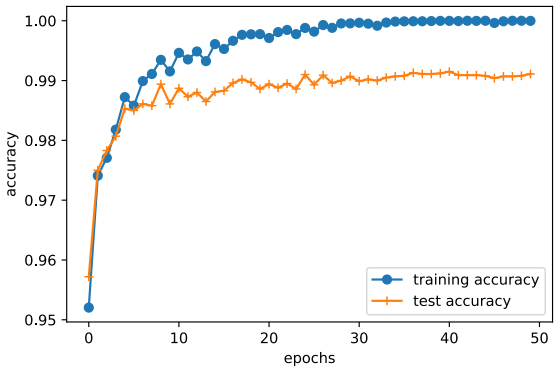
1.1.1 Depth

Deep learning expresses complicated nonlinearity through composing many nonlinear functions; see (1). The rationale for this multilayer structure is that, in many real-world datasets such as images, there are different levels of features and lower-level features are building blocks of higher-level ones. See [**?**] for a visualization of trained features of convolutional neural nets; here in Figure 1, we sample and visualize weights from a pre-trained AlexNet model. This intuition is also supported by empirical results from physiology and neuroscience [**?**, **?**]. The use of function composition marks a sharp difference from traditional statistical methods such as projection pursuit models [**?**] and multi-index models [**?**, **?**]. It is often observed that depth helps efficiently extract features that are representative of a dataset. In comparison, increasing width (e.g., number of basis functions) in a shallow model leads to less improvement. This suggests that deep learning models excel at representing a very different function space that is suitable for complex datasets.

[2]The algorithm makes an error if the true label is not contained in the 5 predictions made by the algorithm.



(a) MNIST images



(b) training and test accuracies

Figure 2: (a) shows the images in the public dataset MNIST; and (b) depicts the training and test accuracies along the training dynamics. Note that the training accuracy is approaching 100% and the test accuracy is still high (no overfitting).

1.1.2 Algorithmic regularization

The statistical performance of neural networks (e.g., test accuracy) depends heavily on the particular optimization algorithms used for training [**?**]. This is very different from many classical statistical problems, where the related optimization problems are less complicated. For instance, when the associated optimization problem has a relatively simple structure (e.g., convex objective functions, linear constraints), the solution to the optimization problem can often be unambiguously computed and analyzed. However, in deep neural networks, due to over-parametrization, there are usually many local minima with different statistical performance [**?**]. Nevertheless, common practice runs stochastic gradient descent with random initialization and finds model parameters with very good prediction accuracy.

1.1.3 Implicit prior learning

It is well observed that deep neural networks trained with only the raw inputs (e.g., pixels of images) can provide a useful representation of the data. This means that after training, the units of deep neural networks can represent features such as edges, corners, wheels, eyes, etc.; see [**?**]. Importantly, the training process is automatic in the sense that no human knowledge is involved (other than hyper-parameter tuning). This is very different from traditional methods, where algorithms are designed after structural assumptions are posited. It is likely that training an over-parametrized model efficiently learns and incorporates the prior distribution $p(\boldsymbol{x})$ of the input, even though deep learning models are themselves discriminative models. With automatic representation of the prior distribution, deep learning typically performs well on similar datasets (but not very different ones) via transfer learning.

1.2 Towards theory of deep learning

Despite the empirical success, theoretical support for deep learning is still in its infancy. Setting the stage, for any classifier f, denote by $\mathbb{E}(f)$ the expected risk on fresh sample (a.k.a. test error, prediction error or generalization error), and by $\mathbb{E}_n(f)$ the empirical risk / training error averaged over a training dataset. Arguably, the key theoretical question in deep learning is

why is $\mathbb{E}(\hat{f}_n)$ small, where $\hat{f}_n$ is the classifier returned by the training algorithm?

We follow the conventional approximation-estimation decomposition (sometimes, also bias-variance trade-off) to decompose the term $\mathbb{E}(\hat{f}_n)$ into two parts. Let $\mathcal{F}$ be the function space expressible by a family of neural nets. Define $f^* = \operatorname{argmin}_f \mathbb{E}(f)$ to be the best possible classifier and $f^*_{\mathcal{F}} = \operatorname{argmin}_{f\in\mathcal{F}} \mathbb{E}(f)$ to be the

best classifier in $\mathcal{F}$. Then, we can decompose the excess error $\mathcal{E} \triangleq \mathbb{E}(\hat{f}_n) - \mathbb{E}(f^*)$ into two parts:

$$\mathcal{E} = \underbrace{\mathbb{E}(f^*_{\mathcal{F}}) - \mathbb{E}(f^*)}_{\text{approximation error}} + \underbrace{\mathbb{E}(\hat{f}_n) - \mathbb{E}(f^*_{\mathcal{F}})}_{\text{estimation error}}. \tag{2}$$

Both errors can be small for deep learning (cf. Figure 2), which we explain below.

- The *approximation error* is determined by the function class $\mathcal{F}$. Intuitively, the larger the class, the smaller the approximation error. Deep learning models use many layers of nonlinear functions (Figure 3)that can drive this error small. Indeed, in Section 5, we provide recent theoretical progress of its representation power. For example, deep models allow efficient representation of interactions among variable while shallow models cannot.
- The *estimation error* reflects the generalization power, which is influenced by both the complexity of the function class $\mathcal{F}$ and the properties of the training algorithms. Interestingly, for *over-parametrized* deep neural nets, stochastic gradient descent typically results in a near-zero training error (i.e., $\mathbb{E}_n(\hat{f}_n) \approx 0$; see e.g. left panel of Figure 2). Moreover, its generalization error $\mathbb{E}(\hat{f}_n)$ remains small or moderate. This "counterintuitive" behavior suggests that for over-parametrized models, gradient-based algorithms enjoy benign statistical properties; we shall see in Section 7 that gradient descent enjoys *implicit regularization* in the over-parametrized regime even without explicit regularization (e.g., ℓ_2 regularization).

The above two points lead to the following heuristic explanation of the success of deep learning models. The large depth of deep neural nets and heavy over-parametrization lead to small or zero training errors, even when running simple algorithms with moderate number of iterations. In addition, these simple algorithms with moderate number of steps do not explore the entire function space and thus have limited complexities, which results in small generalization error with a large sample size. Thus, by combining the two aspects, it explains heuristically that the test error is also small.

1.3 Roadmap of the paper

We first introduce basic deep learning models in Sections 2–4, and then examine their representation power via the lens of approximation theory in Section 5. Section 6 is devoted to training algorithms and their ability of driving the training error small. Then we sample recent theoretical progress towards demystifying the generalization power of deep learning in Section 7. Along the way, we provide our own perspectives, and at the end we identify a few interesting questions for future research in Section 8. The goal of this paper is to present suggestive methods and results, rather than giving conclusive arguments (which is currently unlikely) or a comprehensive survey. We hope that our discussion serves as a stimulus for new statistics research.

2 Feed-forward neural networks

Before introducing the vanilla feed-forward neural nets, let us set up necessary notations for the rest of this section. We focus primarily on classification problems, as regression problems can be addressed similarly. Given the training dataset $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ where $y_i \in [K] \triangleq \{1, 2, \ldots, K\}$ and $\boldsymbol{x}_i \in \mathbb{R}^d$ are independent across $i \in [n]$, supervised learning aims at finding a (possibly random) function $\hat{f}(\boldsymbol{x})$ that predicts the outcome y for a new input $\boldsymbol{x}$, assuming $(y, \boldsymbol{x})$ follows the same distribution as $(y_i, \boldsymbol{x}_i)$. In the terminology of machine learning, the input $\boldsymbol{x}_i$ is often called the *feature*, the output y_i called the *label*, and the pair $(y_i, \boldsymbol{x}_i)$ is an *example*. The function $\hat{f}$ is called the *classifier*, and estimation of $\hat{f}$ is *training* or *learning*. The performance of $\hat{f}$ is evaluated through the prediction error $\mathbb{P}(y \neq \hat{f}(\boldsymbol{x}))$, which can be often estimated from a separate test dataset.

As with classical statistical estimation, for each $k \in [K]$, a classifier approximates the conditional probability $\mathbb{P}(y = k|\boldsymbol{x})$ using a function $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k)$ parametrized by $\boldsymbol{\theta}_k$. Then the category with the highest probability is predicted. Thus, learning is essentially estimating the parameters $\boldsymbol{\theta}_k$. In statistics, one of the most popular methods is (multinomial) logistic regression, which stipulates a specific form for the functions $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k)$: let $z_k = \boldsymbol{x}^\top \boldsymbol{\beta}_k + \alpha_k$ and $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k) = Z^{-1} \exp(z_k)$ where $Z = \sum_{k=1}^K \exp(z_k)$ is a normalization

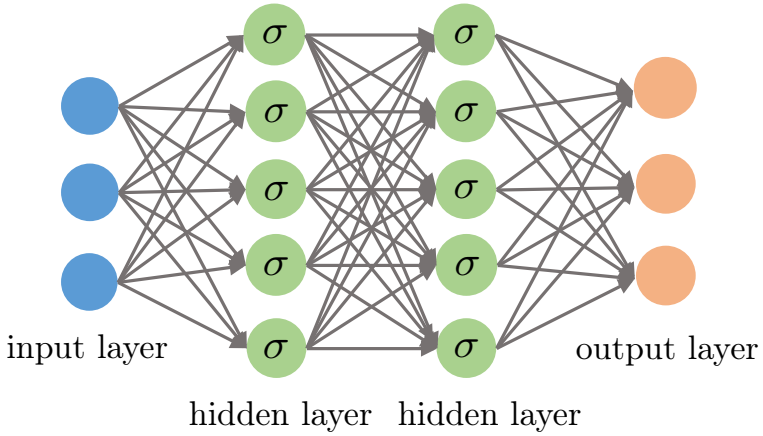


Figure 3: A feed-forward neural network with an input layer, two hidden layers and an output layer. The input layer represents raw features $\{\boldsymbol{x}_i\}_{1\le i\le n}$. Both hidden layers compute an affine transform (a.k.s. indices) of the input and then apply an element-wise activation function $\boldsymbol{\sigma}(\cdot)$. Finally, the output returns a linear transform followed by the softmax activation (resp. simply a linear transform) of the hidden layers for the classification (resp. regression) problem.

factor to make $\{f_k(\boldsymbol{x};\boldsymbol{\theta}_k)\}_{1\le k\le K}$ a valid probability distribution. It is clear that logistic regression induces linear decision boundaries in $\mathbb{R}^d$, and hence it is restrictive in modeling nonlinear dependency between y and $\boldsymbol{x}$. The deep neural networks we introduce below provide a flexible framework for modeling nonlinearity in a fairly general way.

2.1 Model setup

From the high level, deep neural networks (DNNs) use composition of a series of simple nonlinear functions to model nonlinearity

$$\boldsymbol{h}^{(L)} = \mathbf{g}^{(L)} \circ \mathbf{g}^{(L-1)} \circ \ldots \circ \mathbf{g}^{(1)}(\boldsymbol{x}),$$

where $\circ$ denotes composition of two functions and L is the number of hidden layers, and is usually called *depth* of a NN model. Letting $\boldsymbol{h}^{(0)} \triangleq \boldsymbol{x}$, one can recursively define $\boldsymbol{h}^{(l)} = \mathbf{g}^{(l)}\big(\boldsymbol{h}^{(l-1)}\big)$ for all $\ell = 1, 2, \ldots, L$. The *feed-forward neural networks*, also called the *multilayer perceptrons* (MLPs), are neural nets with a specific choice of $\mathbf{g}^{(l)}$: for $\ell = 1, \ldots, L$, define

$$\boldsymbol{h}^{(\ell)} = \mathbf{g}^{(l)}\big(\boldsymbol{h}^{(l-1)}\big) \triangleq \boldsymbol{\sigma}\big(\mathbf{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\big), \tag{3}$$

where $\mathbf{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ are the weight matrix and the bias / intercept, respectively, associated with the l-th layer, and $\boldsymbol{\sigma}(\cdot)$ is usually a simple given (known) nonlinear function called the *activation function*. In words, in each layer ℓ, the input vector $\boldsymbol{h}^{(\ell-1)}$ goes through an affine transformation first and then passes through a fixed nonlinear function $\boldsymbol{\sigma}(\cdot)$. See Figure 3 for an illustration of a simple MLP with two hidden layers. The activation function $\boldsymbol{\sigma}(\cdot)$ is usually applied element-wise, and a popular choice is the ReLU (Rectified Linear Unit) function:

$$[\boldsymbol{\sigma}(\boldsymbol{z})]_j = \max\{z_j, 0\}. \tag{4}$$

Other choices of activation functions include leaky ReLU, tanh function [**?**] and the classical sigmoid function $(1 + e^{-z})^{-1}$, which is less used now.

Given an output $\boldsymbol{h}^{(L)}$ from the final hidden layer and a label y, we can define a loss function to minimize. A common loss function for classification problems is the multinomial logistic loss. Using the terminology of deep learning, we say that $\boldsymbol{h}^{(L)}$ goes through an affine transformation and then the *soft-max* function:

$$f_k(\boldsymbol{x};\boldsymbol{\theta}) \triangleq \frac{\exp(z_k)}{\sum_k \exp(z_k)}, \quad \forall\, k \in [K], \qquad \text{where } \boldsymbol{z} = \mathbf{W}^{(L+1)}\boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L+1)} \in \mathbb{R}^K.$$

Then the loss is defined to be the cross-entropy between the label y (in the form of an indicator vector) and the score vector $(f_1(\boldsymbol{x};\boldsymbol{\theta}),\dots,f_K(\boldsymbol{x};\boldsymbol{\theta}))^\top$, which is exactly the negative log-likelihood of the multinomial logistic regression model:

$$\mathcal{L}(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log p_k, \tag{5}$$

where $\boldsymbol{\theta} \triangleq \{\mathbf{W}^{(\ell)},\boldsymbol{b}^{(\ell)} : 1 \le \ell \le L+1\}$. As a final remark, the number of parameters scales with both the depth L and the width (i.e., the dimensionality of $\mathbf{W}^{(\ell)}$), and hence it can be quite large for deep neural nets.

2.2 Back-propagation in computational graphs

Training neural networks follows the *empirical risk minimization* paradigm that minimizes the loss (e.g., (5)) over all the training data. This minimization is usually done via *stochastic gradient descent* (SGD). In a way similar to gradient descent, SGD starts from a certain initial value $\boldsymbol{\theta}^0$ and then iteratively updates the parameters $\boldsymbol{\theta}^t$ by moving it in the direction of the negative gradient. The difference is that, in each update, a small subsample $\mathcal{B} \subset [n]$ called a *mini-batch*—which is typically of size 32–512—is randomly drawn and the gradient calculation is only on $\mathcal{B}$ instead of the full batch $[n]$. This saves considerably the computational cost in calculation of gradient. By the law of large numbers, this stochastic gradient should be close to the full sample one, albeit with some random fluctuations. A pass of the whole training set is called an *epoch*. Usually, after several or tens of epochs, the error on a validation set levels off and training is complete. See Section 6 for more details and variants on training algorithms.

The key to the above training procedure, namely SGD, is the calculation of the gradient $\nabla \ell_{\mathcal{B}}(\boldsymbol{\theta})$, where

$$\ell_{\mathcal{B}}(\boldsymbol{\theta}) \triangleq |\mathcal{B}|^{-1} \sum_{i\in\mathcal{B}} \mathcal{L}(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}),y_i). \tag{6}$$

Gradient computation, however, is in general nontrivial for complex models, and it is susceptible to numerical instability for a model with large depth. Here, we introduce an efficient approach, namely *back-propagation*, for computing gradients in neural networks.

Back-propagation [**?**] is a direct application of the chain rule in networks. As the name suggests, the calculation is performed in a backward fashion: one first computes $\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}^{(L)}$, then $\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}^{(L-1)}$, …, and finally $\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}^{(1)}$. For example, in the case of the ReLU activation function[3], we have the following recursive / backward relation

$$\frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell-1)}} = \frac{\partial \boldsymbol{h}^{(\ell)}}{\partial \boldsymbol{h}^{(\ell-1)}} \cdot \frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell)}} = (\mathbf{W}^{(\ell)})^\top \mathsf{diag}\left(\mathbb{1}\{\mathbf{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \ge \mathbf{0}\}\right) \frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell)}} \tag{7}$$

where $\mathsf{diag}(\cdot)$ denotes a diagonal matrix with elements given by the argument. Note that the calculation of $\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}^{(\ell-1)}$ depends on $\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}^{(\ell)}$, which is the partial derivatives from the next layer. In this way, the derivatives are "back-propagated" from the last layer to the first layer. These derivatives $\{\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}^{(\ell)}\}$ are then used to update the parameters. For instance, the gradient update for $\mathbf{W}^{(\ell)}$ is given by

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \frac{\partial \ell_{\mathcal{B}}}{\partial \mathbf{W}^{(\ell)}}, \quad \text{where} \quad \frac{\partial \ell_{\mathcal{B}}}{\partial W^{(\ell)}_{jm}} = \frac{\partial \ell_{\mathcal{B}}}{\partial h^{(\ell)}_{j}} \cdot \sigma' \cdot h^{(\ell-1)}_{m}, \tag{8}$$

where $\sigma' = 1$ if the j-th element of $\mathbf{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$ is nonnegative, and $\sigma' = 0$ otherwise. The step size $\eta > 0$, also called the *learning rate*, controls how much parameters are changed in a single update.

A more general way to think about neural network models and training is to consider *computational graphs*. Computational graphs are directed acyclic graphs that represent functional relations between variables. They are very convenient and flexible to represent function composition, and moreover, they also

[3]The issue of non-differentiability at the origin is often ignored in implementation.

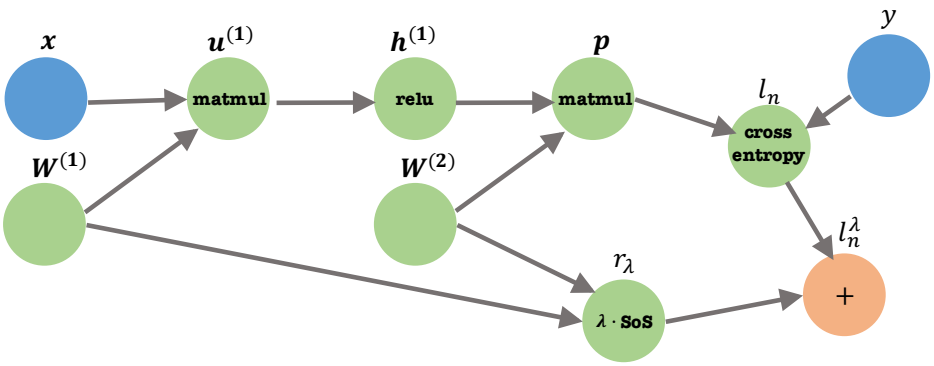


Figure 4: The computational graph illustrates the loss (9). For simplicity, we omit the bias terms. Symbols inside nodes represent functions, and symbols outside nodes represent function outputs (vectors/scalars). `matmul` is matrix multiplication, `relu` is the ReLU activation, `cross entropy` is the cross entropy loss, and `SoS` is the sum of squares.

allow an efficient way of computing gradients. Consider an MLP with a single hidden layer and an ℓ_2 regularization:

$$\ell_{\mathcal{B}}^{\lambda}(\boldsymbol{\theta}) = \ell_{\mathcal{B}}(\boldsymbol{\theta}) + r_{\lambda}(\boldsymbol{\theta}) = \ell_{\mathcal{B}}(\boldsymbol{\theta}) + \lambda\Big(\sum_{j,j'} \big(W_{j,j'}^{(1)}\big)^2 + \sum_{j,j'} \big(W_{j,j'}^{(2)}\big)^2\Big), \tag{9}$$

where $\ell_{\mathcal{B}}(\boldsymbol{\theta})$ is the same as (6), and $\lambda \geq 0$ is a tuning parameter. A similar example is considered in [**?**]. The corresponding computational graph is shown in Figure 4. Each node represents a function (inside a circle), which is associated with an output of that function (outside a circle). For example, we view the term $\ell_{\mathcal{B}}(\boldsymbol{\theta})$ as a result of 4 compositions: first the input data $\boldsymbol{x}$ multiplies the weight matrix $\mathbf{W}^{(1)}$ resulting in $\boldsymbol{u}^{(1)}$, then it goes through the ReLU activation function `relu` resulting in $\boldsymbol{h}^{(1)}$, then it multiplies another weight matrix $\mathbf{W}^{(2)}$ leading to $\boldsymbol{p}$, and finally it produces the cross-entropy with label y as in (5). The regularization term is incorporated in the graph similarly.

A forward pass is complete when all nodes are evaluated starting from the input $\boldsymbol{x}$. A backward pass then calculates the gradients of $\ell_{\mathcal{B}}^{\lambda}$ with respect to all other nodes in the reverse direction. Due to the chain rule, the gradient calculation for a variable (say, $\partial \ell_{\mathcal{B}}/\partial \boldsymbol{u}^{(1)}$) is simple: it only depends on the gradient value of the variables ($\partial \ell_{\mathcal{B}}/\partial \boldsymbol{h}$) the current node points to, and the function derivative evaluated at the current variable value ($\boldsymbol{\sigma}'(\boldsymbol{u}^{(1)})$). Thus, in each iteration, a computation graph only needs to (1) calculate and store the function evaluations at each node in the forward pass, and then (2) calculate all derivatives in the backward pass.

Back-propagation in computational graphs forms the foundations of popular deep learning programming softwares, including TensorFlow [**?**] and PyTorch [**?**], which allows more efficient building and training of complex neural net models.

2.3 Some general remarks

We give a few remarks about general characteristics of deep neural network models and training. Specific deep neural network models and training are detailed in later sections. Before we move on to introducing various popular models in deep learning, we pause here to single out several distinctive characteristics of deep neural nets that are widely believed to contribute towards the success of deep learning. Further theoretical justifications are left to Section **??**. [CM: The following three paragraphs need to be revised.]

[CM: I think overparametrization might be a major reason, and it seems relevant to Statistics.]

First, during training, stochastic gradient descent (SGD) and its variants are widely used. The subsamples of SGD are typically of size 32–512, and are drawn without replacement from a training set of size from thousands to millions. A pass of the whole training set is called an *epoch*. Usually, after several or tens of epochs, the validation error levels off and training is complete. [add a figure showing the typical training/test curve] There are at least two reasons why SGD is widely used: (1) it usually speeds up computation in large machine learning tasks [**?**], which has been well known; (2) it can achieve optimal statistical accuracy, and

even outperforms the full-batch gradient descent (that is, $\mathcal{B} = [n]$) [?]. The latter is particularly intriguing from the perspective of statistics, which stirs widespread interest in recent years. See Section 6.1 for details of SGD. [discuss in Section 6.]

Second, prediction performance generally improves as neural nets become deeper (e.g., 5-30 layers). Deep neural networks are usually over-parametrized, which means that the number of parameters is much larger the sample size. The benefits of depth are at least two-fold: (1) deep neural nets form a larger function space, and thus training error is typically smaller due to better fitting; (2) with explicit or even implicit regularization, the test error is also usually smaller. With regard to the first point, an interesting phenomenon is that the effect of increasing depth is much better than of increasing width. In particular, for many problems, DNNs do not seem to suffer from the curse of dimensionality. Thus, in this respect, DNNs brings new and powerful tools for efficient data fitting. The second point is intriguing from the statistical perspective, because over-parametrization would normally run the risk of severe overfitting in other models. Section 5 and 7 provide recent theories for the the effects of depth and over-parametrization.

It is easy to understand that a larger model has more capacity for data fitting, so (1) should be well expected. For deep nets, the benefit in data fitting, namely (1), is much more significant compared with wide shallow nets [def!]. Many traditional statistical methods such as basis expansion is analogous to shallow nets [need to discuss this before]. Thus, in this respect, deep neural nets brings very new and powerful tools for efficient data fitting. The benefit in generalization, namely (2), is exciting from the statistics standpoint. While explicit regularization through ℓ_2 or ℓ_1 has been well studied, implicit regularization such as SGD is recognized only recently [check; refs].

[CM: My friend Chenxi's comment: I wouldn't agree with the second point of "several distinctive characteristics", that the deeper the better. In fact, the current best ImageNet models (e.g. mine https://arxiv.org/abs/1712.00559; better than ResNet 5% top-1) only has maximum depth of less than 30. "The deeper the better" may only be true in networks that rely heavily on residual connections.]

Third, the ReLU activation function is a crucial element in most deep learning architectures. [add a figure to show different activation functions] Its popularity is largely due to the fact that its derivative is either 0 or 1, which makes training more efficient. Note that in back-propagation, gradient calculation is multiplicative due to the recursive relation (7). Thus, it is important to keep gradients from being "killed", which means that gradients become approximately zero if in the multiplication a factor σ' is very small due to an input with large magnitude. Historically, the sigmoid function (a.k.a. the logistic function) was the common choice for the activation function, but for deep neural nets, it tends to kill gradients and has inferior training performance [?, ?]. In contrast, ReLU or its variants (e.g., leaky ReLU) is much better for training as well as parameter initialization.

2.4 Numerical experiments

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten

# the data, split between train and test sets
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

MLP = Sequential([
  Flatten(),
  Dense(512, activation=tf.nn.relu),
  Dense(10, activation=tf.nn.softmax)
])
MLP.compile(optimizer='sgd',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

MLP.fit(x_train, y_train, epochs=5, batch_size=32,
```

```
        verbose=1,
        validation_split=0.1)
score = MLP.evaluate(x_test, y_test)
```

3 Popular models

Moving beyond vanilla feed-forward neural networks, we introduce two other popular deep learning models, namely, the convolutional neural networks (CNNs) and the recurrent neural networks (RNNs). One important characteristic shared by the two models is *weight sharing*, that is some model parameters are identical across locations in CNNs or across time in RNNs. This is related to the notion of translational invariance in CNNs and stationarity in RNNs. At the end of this section, we introduce a modular thinking for constructing more flexible neural nets.

3.1 Convolutional neural networks

The convolutional neural network (CNN) [**?**, **?**] is a special type of feed-forward neural networks that is tailored for image processing. More generally, it is suitable for analyzing data with salient spatial structures. In this subsection, we focus on image classification using CNNs, where the raw input (image pixels) and features of each hidden layer are represented by a 3D tensor $\boldsymbol{X} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$. Here, the first two dimensions d_1, d_2 of $\boldsymbol{X}$ indicate spatial coordinates of an image while the third d_3 indicates the number of channels. For instance, d_3 is 3 for the raw inputs due to the red, green and blue channels, and d_3 can be much larger (say, 256) for hidden layers. Each channel is also called a *feature map*, because each feature map is specialized to detect the same feature at different locations of the input, which we will soon explain. We now introduce two building blocks of CNNs, namely the convolutional layer and the pooling layer.

1. *Convolutional layer (CONV).* A convolutional layer has the same functionality as described in (3), where the input feature $\boldsymbol{X} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ goes through an affine transformation first and then an element-wise nonlinear activation. The difference lies in the specific form of the affine transformation. A convolutional layer uses a number of *filters* to extract local features from the previous input. More precisely, each filter is represented by a 3D tensor $\boldsymbol{F}_k \in \mathbb{R}^{w \times w \times d_3}$ $(1 \le k \le \tilde{d}_3)$, where w is the size of the filter (typically 3 or 5) and $\tilde{d}_3$ denotes the total number of filters. Note that the third dimension d_3 of $\boldsymbol{F}_k$ is equal to that of the input feature $\boldsymbol{X}$. For this reason, one usually says that the filter has size $w \times w$, while suppressing the third dimension d_3. Each filter $\boldsymbol{F}_k$ then convolves with the input feature $\boldsymbol{X}$ to obtain one single feature map $\boldsymbol{O}^k \in \mathbb{R}^{(d_1-w+1)\times(d_1-w+1)}$, where[4]

$$O_{ij}^k = \left\langle [\boldsymbol{X}]_{ij}, \boldsymbol{F}_k \right\rangle = \sum_{i'=1}^{w} \sum_{j'=1}^{w} \sum_{l=1}^{d_3} [\boldsymbol{X}]_{i+i'-1, j+j'-1, l} [\boldsymbol{F}_k]_{i', j', l}. \tag{10}$$

Here $[\boldsymbol{X}]_{ij} \in \mathbb{R}^{w \times w \times d_3}$ is a small "patch" of $\boldsymbol{X}$ starting at location (i, j). See Figure 5 for an illustration of the convolution operation. If we view the 3D tensors $[\boldsymbol{X}]_{ij}$ and $\boldsymbol{F}_k$ as vectors, then each filter essentially computes their inner product with a part of $\boldsymbol{X}$ indexed by i, j (which can be also viewed as convolution, as its name suggests). One then pack the resulted feature maps $\{\boldsymbol{O}^k\}$ into a 3D tensor $\boldsymbol{O}$ with size $(d_1 - w + 1) \times (d_1 - w + 1) \times \tilde{d}_3$, where

$$[\boldsymbol{O}]_{ijk} = [\boldsymbol{O}^k]_{ij}. \tag{11}$$

The outputs of convolutional layers are then followed by nonlinear activation functions. In the ReLU case, we have

$$\tilde{X}_{ijk} = \sigma(O_{ijk}), \qquad \forall\, i \in [d_1 - w + 1], j \in [d_2 - w + 1], k \in [\tilde{d}_3]. \tag{12}$$

The convolution operation (10) and the ReLU activation (12) work together to extract features $\tilde{\boldsymbol{X}}$ from the input $\boldsymbol{X}$. Different from feed-forward neural nets, the filters $\boldsymbol{F}_k$ are shared across all locations (i, j). A patch $[\boldsymbol{X}]_{ij}$ of an input responds strongly (that is, producing a large value) to a filter $\boldsymbol{F}_k$ if they are positively correlated. Therefore intuitively, each filter $\boldsymbol{F}_k$ serves to extract features similar to $\boldsymbol{F}_k$.

[4] To simplify notation, we omit the bias/intercept term associated with each filter.

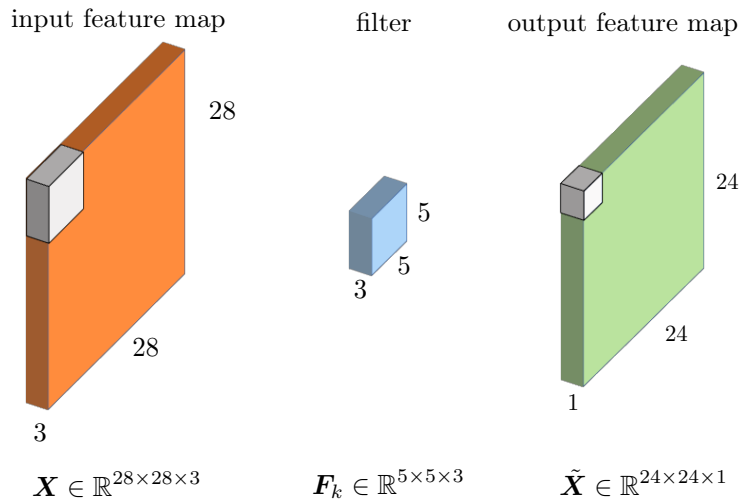


Figure 5: $\boldsymbol{X} \in \mathbb{R}^{28\times28\times3}$ represents the input feature consisting of 28×28 spatial coordinates in a total number of 3 channels / feature maps. $\boldsymbol{F}_k \in \mathbb{R}^{5\times5\times3}$ denotes the k-th filter with size 5×5. The third dimension 3 of the filter automatically matches the number 3 of channels in the previous input. Every 3D patch of $\boldsymbol{X}$ gets convolved with the filter $\boldsymbol{F}_k$ and this as a whole results in a single output feature map $\tilde{X}_{:,:,k}$ with size $24 \times 24 \times 1$. Stacking the outputs of all the filters $\{\boldsymbol{F}_k\}_{1\le k\le K}$ will lead to the output feature with size $24 \times 24 \times K$.

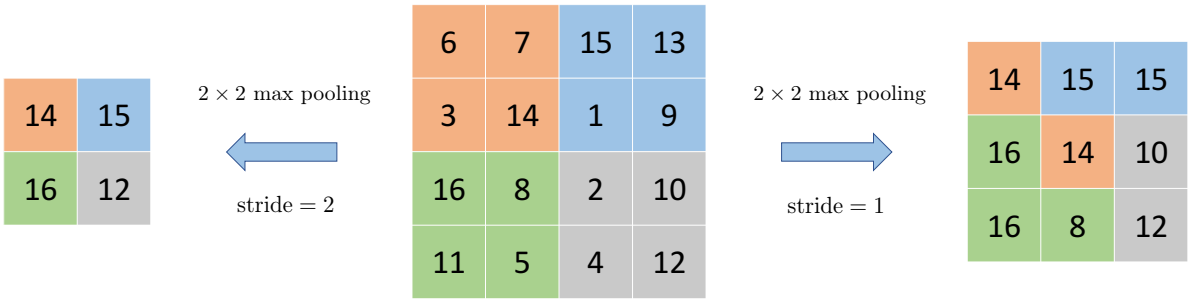


Figure 6: A 2×2 max pooling layer extracts the maximum of 2 by 2 neighboring pixels / features across the spatial dimension.

As a side note, after the convolution (10), the spatial size $d_1 \times d_2$ of the input $\boldsymbol{X}$ shrinks to $(d_1 - w + 1) \times (d_2 - w + 1)$ of $\tilde{\boldsymbol{X}}$. However one may want the spatial size unchanged. This can be achieved via *padding*, where one appends zeros to the margins of the input $\boldsymbol{X}$ to enlarge the spatial size to $(d_1 + w - 1) \times (d_2 + w - 1)$. In addition, a *stride* in the convolutional layer determines the gap $i' - i$ and $j' - j$ between two patches $\boldsymbol{X}_{ij}$ and $\boldsymbol{X}_{i'j'}$: in (10) the stride is 1, and a larger stride would lead to feature maps with smaller sizes.

2. *Pooling layer (POOL).* A pooling layer aggregates the information of nearby features into a single one. This downsampling operation reduces the size of the features for subsequent layers and saves computation. One common form of the pooling layer is composed of the 2×2 max-pooling filter. It computes $\max\{X_{i,j,k}, X_{i+1,j,k}, X_{i,j+1,k}, X_{i+1,j+1,k}\}$, that is, the maximum of the 2×2 neighborhood in the spatial coordinates; see Figure 6 for an illustration. Note that the pooling operation is done separately for each feature map k. As a consequence, a 2×2 max-pooling filter acting on $\boldsymbol{X} \in \mathbb{R}^{d_1\times d_2\times d_3}$ will result in an output of size $d_1/2 \times d_2/2 \times d_3$. In addition, the pooling layer does not involve any parameters to optimize. Pooling layers serve to reduce redundancy since a small neighborhood around a location (i, j) in a feature map is likely to contain the same information.

In addition, we also use fully-connected layers as building blocks, which we have already seen in Section 2. Each fully-connected layer treats input tensor $\boldsymbol{X}$ as a vector $\mathrm{Vec}(\boldsymbol{X})$, and computes $\tilde{\boldsymbol{X}} = \boldsymbol{\sigma}(\mathbf{W}\mathrm{Vec}(\boldsymbol{X}))$. A

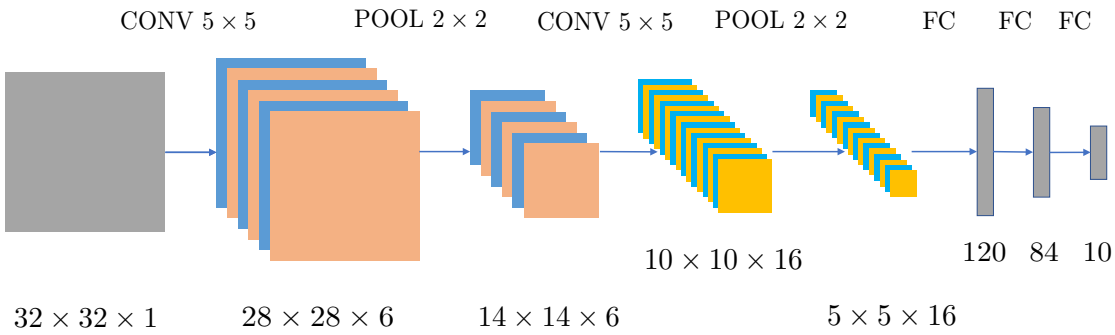
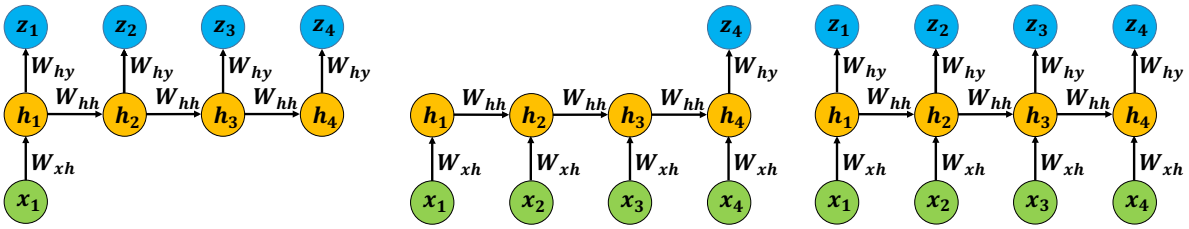


Figure 7: LeNet is composed of an input layer, two convolutional layers, two pooling layers and three fully-connected layers. Both convolutions are valid and use filters with size 5×5. In addition, the two pooling layers use 2×2 average pooling.



(a) One-to-many (b) Many-to-one (c) Many-to-many

Figure 8: Vanilla RNNs with different inputs/outputs settings. (a) has one input but multiple outputs; (b) has multiple inputs but one output; (c) has multiple inputs and outputs. Note that the parameters are shared across time steps.

fully-connected layer does not use weight sharing and is often used in the last few layers of a CNN. As an example, Figure 7 depicts the well-known LeNet 5 [**?**], which is composed of two sets of CONV-POOL layers and three fully-connected layers.

3.2 Recurrent neural networks

Recurrent neural nets (RNNs) are another family of powerful models, which are designed to process time series data and other sequence data. RNNs have successful applications in speech recognition [**?**], machine translation [**?**], genome sequencing [**?**], etc. The structure of an RNN naturally forms a computational graph, and can be easily combined with other structures such as CNNs to build large computational graph models for complex tasks. Here we introduce vanilla RNNs and improved variants such as long short-term memory (LSTM).

3.2.1 Vanilla RNNs

Suppose we have general time series inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$. A vanilla RNN models the "hidden state" at time t by a vector $\boldsymbol{h}_t$, which is subject to the recursive formula

$$\boldsymbol{h}_t = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t). \tag{13}$$

Here, $f_{\boldsymbol{\theta}}$ is generally a nonlinear function parametrized by $\boldsymbol{\theta}$. Concretely, a vanilla RNN with one hidden layer has the following form[5]

$$\boldsymbol{h}_t = \mathbf{tanh}\left(\mathbf{W}_{hh}\boldsymbol{h}_{t-1} + \mathbf{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_{\boldsymbol{h}}\right), \qquad \text{where } \tanh(a) = \tfrac{e^{2a}-1}{e^{2a}+1},$$

[5]Similar to the activation function $\boldsymbol{\sigma}(\cdot)$, the function $\mathbf{tanh}(\cdot)$ means element-wise operations.

$$\boldsymbol{z}_t = \boldsymbol{\sigma}\left(\mathbf{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_{\boldsymbol{z}}\right),$$

where $\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}$ are trainable weight matrices, $\boldsymbol{b}_{\boldsymbol{h}}, \boldsymbol{b}_{\boldsymbol{z}}$ are trainable bias vectors, and $\boldsymbol{z}_t$ is the output at time t. Like many classical time series models, those parameters are shared across time. Note that in different applications, we may have different input/output settings (cf. Figure 8). Examples include

- **One-to-many:** a single input with multiple outputs; see Figure 8(a). A typical application is image captioning, where the input is an image and outputs are a series of words.
- **Many-to-one:** multiple inputs with a single output; see Figure 8(b). One application is text sentiment classification, where the input is a series of words in a sentence and the output is a label (e.g., positive vs. negative).
- **Many-to-many:** multiple inputs and outputs; see Figure 8(c). This is adopted in machine translation, where inputs are words of a source language (say Chinese) and outputs are words of a target language (say English).

As the case with feed-forward neural nets, we minimize a loss function using back-propagation, where the loss is typically

$$\ell_{\mathcal{T}}(\boldsymbol{\theta}) = \sum_{t\in\mathcal{T}} \mathcal{L}(y_t, \boldsymbol{z}_t) = -\sum_{t\in\mathcal{T}}\sum_{k=1}^{K} \mathbb{1}\{y_t = k\} \log\left(\frac{\exp([\boldsymbol{z}_t]_k)}{\sum_k \exp([\boldsymbol{z}_t]_k)}\right),$$

where K is the number of categories for classification (e.g., size of the vocabulary in machine translation), and $\mathcal{T} \subset [T]$ is the length of the output sequence. During the training, the gradients $\partial\ell_{\mathcal{T}}/\partial\boldsymbol{h}_t$ are computed in the reverse time order (from T to t). For this reason, the training process is often called *back-propagation through time*.

One notable drawback of vanilla RNNs is that, they have difficulty in capturing long-range dependencies in sequence data when the length of the sequence is large. This is sometimes due to the phenomenon of *exploding / vanishing gradients*. Take Figure 8(c) as an example. Computing $\partial\ell_{\mathcal{T}}/\partial\boldsymbol{h}_1$ involves the product $\prod_{t=1}^{3}(\partial\boldsymbol{h}_{t+1}/\partial\boldsymbol{h}_t)$ by the chain rule. However, if the sequence is long, the product will be the multiplication of many Jacobian matrices, which usually results in exponentially large or small singular values. To alleviate this issue, in practice, the forward pass and backward pass are implemented in a shorter sliding window $\{t_1, t_1+1, \ldots, t_2\}$, instead of the full sequence $\{1, 2, \ldots, T\}$. Though effective in some cases, this technique alone does not fully address the issue of long-term dependency.

3.2.2 GRUs and LSTM

There are two improved variants that alleviate the above issue: gated recurrent units (GRUs) [**?**] and long short-term memory (LSTM) [**?**].

- A **GRU** refines the recursive formula (13) by introducing *gates*, which are vectors of the same length as $\boldsymbol{h}_t$. The gates, which take values in $[0, 1]$ elementwise, multiply with $\boldsymbol{h}_{t-1}$ elementwise and determine how much they keep the old hidden states.
- An **LSTM** similarly uses gates in the recursive formula. In addition to $\boldsymbol{h}_t$, an LSTM maintains a *cell state*, which takes values in $\mathbb{R}$ elementwise and are analogous to counters.

Here we only discuss LSTM in detail. Denote by $\odot$ the element-wise multiplication. We have a recursive formula in replace of (13):

$$\begin{pmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{o}_t \\ \boldsymbol{g}_t \end{pmatrix} = \begin{pmatrix} \boldsymbol{\sigma} \\ \boldsymbol{\sigma} \\ \boldsymbol{\sigma} \\ \mathbf{tanh} \end{pmatrix} \mathbf{W} \begin{pmatrix} \boldsymbol{h}_{t-1} \\ \boldsymbol{x}_t \\ 1 \end{pmatrix},$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t,$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \mathbf{tanh}(\boldsymbol{c}_t),$$

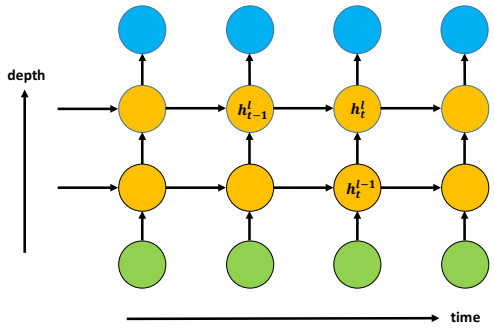


Figure 9: A vanilla RNN with two hidden layers. Higher-level hidden states $\boldsymbol{h}_t^\ell$ are determined by the old states $\boldsymbol{h}_{t-1}^\ell$ and lower-level hidden states $\boldsymbol{h}_t^{\ell-1}$. Multilayer RNNs generalize both feed-forward neural nets and one-hidden-layer RNNs.

where $\mathbf{W}$ is a big weight matrix with appropriate dimensions. The cell state vector $\boldsymbol{c}_t$ carries information of the sequence (e.g., singular/plural form in a sentence). The forget gate $\boldsymbol{f}_t$ determines how much the values of $\boldsymbol{c}_{t-1}$ are kept for time t, the input gate $\boldsymbol{i}_t$ controls the amount of update to the cell state, and the output gate $\boldsymbol{o}_t$ gives how much $\boldsymbol{c}_t$ reveals to $\boldsymbol{h}_t$. Ideally, the elements of these gates have nearly binary values. For example, an element of $\boldsymbol{f}_t$ being close to 1 may suggest the presence of a feature in the sequence data. Similar to the skip connections in residual nets, the cell state $\boldsymbol{c}_t$ has an additive recursive formula, which helps back-propagation and thus captures long-range dependencies.

3.2.3 Multilayer RNNs

Multilayer RNNs are generalization of the one-hidden-layer RNN discussed above. Figure 9 shows a vanilla RNN with two hidden layers. In place of (13), the recursive formula for an RNN with L hidden layers now reads

$$\boldsymbol{h}_t^\ell = \mathbf{tanh}\left[\mathbf{W}^\ell \begin{pmatrix} \boldsymbol{h}_t^{\ell-1} \\ \boldsymbol{h}_{t-1}^\ell \\ 1 \end{pmatrix}\right], \quad \text{for all } \ell \in [L], \qquad \boldsymbol{h}_t^0 \triangleq \boldsymbol{x}_t.$$

Note that a multilayer RNN has two dimensions: the sequence length T and depth L. Two special cases are the feed-forward neural nets (where $T = 1$) introduced in Section 2, and RNNs with one hidden layer (where $L = 1$). Multilayer RNNs usually do not have very large depth (e.g., 2–5), since T is already very large.

Finally, we remark that CNNs, RNNs, and other neural nets can be easily combined to tackle tasks that involve different sources of input data. For example, in image captioning, the images are first processed through a CNN, and then the high-level features are fed into an RNN as inputs. Theses neural nets combined together form a large computational graph, so they can be trained using back-propagation. This generic training method provides much flexibility in various applications.

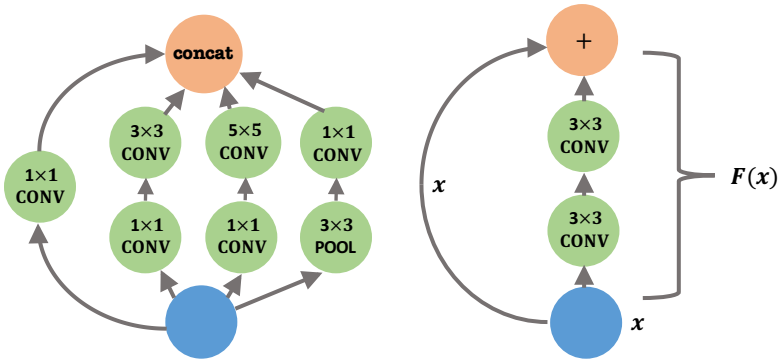
3.3 Modules

Deep neural nets are essentially composition of many nonlinear functions. A component function may be designed to have specific properties in a given task, and it can be itself resulted from composing a few simpler functions. In LSTM, we have seen that the building block consists of several intermediate variables, including cell states and forget gates that can capture long-term dependency and alleviate numerical issues.

This leads to the idea of designing *modules* for building more complex neural net models. Desirable modules usually have low computational costs, alleviate numerical issues in training, and lead to good statistical accuracy. Since modules and the resulting neural net models form computational graphs, training follows the same principle briefly described in Section 2.

Here, we use the examples of *Inception* and *skip connections* to illustrate the ideas behind modules. Figure 10(a) is an example of "Inception" modules used in GoogleNet [**?**]. As before, all the convolutional layers are followed by the ReLU activation function. The concatenation of information from filters with

different sizes give the model great flexibility to capture spatial information. Note that 1×1 filters is an $1 \times 1 \times d_3$ tensor (where d_3 is the number of feature maps), so its convolutional operation does not interact with other spatial coordinates, only serving to aggregate information from different feature maps at the same coordinate. This reduces the number of parameters and speeds up the computation. Similar ideas appear in other work [**?**, **?**].



(a) "Inception" module (b) Skip connections

Figure 10: (a) The "Inception" module from GoogleNet. `Concat` means combining all features maps into a tensor. (b) Skip connections are added every two layers in ResNets.

Another module, usually called *skip connections*, is widely used to alleviate numerical issues in very deep neural nets, with additional benefits in optimization efficiency and statistical accuracy. Training very deep neural nets are generally more difficult, but the introduction of skip connections in *residual networks* [**?**, **?**] has greatly eased the task.

The high level idea of skip connections is to add an identity map to an existing nonlinear function. Let $\mathbf{F}(\boldsymbol{x})$ be an arbitrary nonlinear function represented by a (fragment of) neural net, then the idea of skip connections is simply replacing $\mathbf{F}(\boldsymbol{x})$ with $\boldsymbol{x}+\mathbf{F}(\boldsymbol{x})$. Figure 10(b) shows a well-known structure from residual networks [**?**]—for every two layers, an identity map is added:

$$\boldsymbol{x} \longmapsto \boldsymbol{\sigma}(\boldsymbol{x} + \mathbf{F}(\boldsymbol{x})) = \boldsymbol{\sigma}(\boldsymbol{x} + \mathbf{W}'\boldsymbol{\sigma}(\mathbf{W}\boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{b}'), \tag{14}$$

where $\boldsymbol{x}$ can be hidden nodes from any layer and $\mathbf{W}, \mathbf{W}', \boldsymbol{b}, \boldsymbol{b}'$ are corresponding parameters. By repeating (namely composing) this structure throughout all layers, [**?**, **?**] are able to train neural nets with hundreds of layers easily, which overcomes well-observed training difficulties in deep neural nets. Moreover, deep residual networks also improve statistical accuracy, as the classification error on ImageNet challenge was reduced by 46% from 2014 to 2015. As a side note, skip connections can be used flexibly. They are not restricted to the form in (14), and can be used between any pair of layers ℓ, ℓ' [**?**].

4 Deep unsupervised learning

In supervised learning, given labelled training set $\{(y_i, \mathbf{x}_i)\}$, we focus on discriminative models, which essentially represents $\mathbb{P}(y \,|\, \boldsymbol{x})$ by a deep neural net $f(\boldsymbol{x}; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$. Unsupervised learning, in contrast, aims at extracting *information* from *unlabeled* data $\{\boldsymbol{x}_i\}$, where the labels $\{y_i\}$ are absent. In regard to this information, it can be a low-dimensional embedding of the data $\{\boldsymbol{x}_i\}$ or a generative model with latent variables to approximate the distribution $\mathbb{P}_{\mathbf{X}}(\boldsymbol{x})$. To achieve these goals, we introduce two popular unsupervised deep leaning models, namely, autoencoders and generative adversarial networks (GANs). The first one can be viewed as a dimension reduction technique, and the second as a density estimation method. DNNs are the key elements for both of these two models.

4.1 Autoencoders

Recall that in dimension reduction, the goal is to reduce the dimensionality of the data and at the same time preserve its salient features. In particular, in principal component analysis (PCA), the goal is to embed the data $\{\boldsymbol{x}_i\}_{1\le i\le n}$ into a low-dimensional space via a linear function $\boldsymbol{f}$ such that maximum variance can be explained. Equivalently, we want to find linear functions $\boldsymbol{f}:\mathbb{R}^d\to\mathbb{R}^k$ and $\boldsymbol{g}:\mathbb{R}^k\to\mathbb{R}^d$ ($k\le d$) such that the difference between $\boldsymbol{x}_i$ and $\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}_i))$ is minimized. Formally, we let

$$\boldsymbol{f}(\boldsymbol{x})=\boldsymbol{W}_f\boldsymbol{x}\triangleq\boldsymbol{h}\quad\text{and}\quad\boldsymbol{g}(\boldsymbol{h})=\boldsymbol{W}_g\boldsymbol{h},\quad\text{where}\quad\boldsymbol{W}_f\in\mathbb{R}^{k\times d}\text{ and }\boldsymbol{W}_g\in\mathbb{R}^{d\times k}.$$

Here, for simplicity, we assume that the intercept/bias terms for $\boldsymbol{f}$ and $\boldsymbol{g}$ are zero. Then, PCA amounts to minimizing the quadratic loss function

$$\text{minimize}_{\boldsymbol{W}_f,\boldsymbol{W}_g}\qquad\frac{1}{n}\sum_{i=1}^n\|\boldsymbol{x}_i-\boldsymbol{W}_f\boldsymbol{W}_g\boldsymbol{x}_i\|_2^2\,. \tag{15}$$

It is the same as minimizing $\|\mathbf{X}-\mathbf{W}\mathbf{X}\|_{\mathrm{F}}^2$ subject to $\mathrm{rank}(\mathbf{W})\le k$, where $\mathbf{X}\in\mathbb{R}^{p\times n}$ is the design matrix. The solution is given by the singular value decomposition of $\mathbf{X}$ [**?**, Thm. 2.4.8], which is exactly what PCA does. It turns out that PCA is a special case of autoencoders, which is often known as the *undercomplete linear autoencoder*.

More broadly, autoencoders are neural network models for (nonlinear) dimension reduction, which generalize PCA. An autoencoder has two key components, namely, the encoder function $\boldsymbol{f}(\cdot)$, which maps the input $\boldsymbol{x}\in\mathbb{R}^d$ to a hidden code/representation $\boldsymbol{h}\triangleq\boldsymbol{f}(\boldsymbol{x})\in\mathbb{R}^k$, and the decoder function $\boldsymbol{g}(\cdot)$, which maps the hidden representation $\boldsymbol{h}$ to a point $\boldsymbol{g}(\boldsymbol{h})\in\mathbb{R}^d$. Both functions can be multilayer neural networks as (3). See Figure 11 for an illustration of autoencoders. Let $\mathcal{L}(\boldsymbol{x}_1,\boldsymbol{x}_2)$ be a loss function that measures the difference between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ in $\mathbb{R}^d$. Similar to PCA, an autoencoder is used to find the encoder $\boldsymbol{f}$ and decoder $\boldsymbol{g}$ such that $\mathcal{L}(\boldsymbol{x},\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x})))$ is as small as possible. Mathematically, this amounts to solving the following minimization problem

$$\text{minimize}_{\boldsymbol{f},\boldsymbol{g}}\quad\frac{1}{n}\sum_{i=1}^n\mathcal{L}\left(\boldsymbol{x}_i,\boldsymbol{g}\left(\boldsymbol{h}_i\right)\right)\quad\text{with}\quad\boldsymbol{h}_i=\boldsymbol{f}\left(\boldsymbol{x}_i\right),\quad\text{for all }\ i\in[n]. \tag{16}$$

One needs to make structural assumptions on the functions $\boldsymbol{f}$ and $\mathbf{g}$ in order to find useful representations of the data, which leads to different types of autoencoders. Indeed, if no assumption is made, choosing $\boldsymbol{f}$ and $\boldsymbol{g}$ to be identity functions clearly minimizes the above optimization problem. To avoid this trivial solution, one natural way is to require that the encoder f maps the data onto a space with a smaller dimension, i.e., $k<d$. This is the *undercomplete autoencoder* that includes PCA as a special case. There are other structured autoencoders which add desired properties to the model such as sparsity or robustness, mainly through regularization terms. Below we present two other common types of autoencoders.

- *Sparse autoencoders.* One may believe that the dimension k of the hidden code $\boldsymbol{h}_i$ is larger than the input dimension d, and that $\boldsymbol{h}_i$ admits a sparse representation. As with LASSO [**?**] or SCAD [**?**], one may add a regularization term to the reconstruction loss $\mathcal{L}$ in (16) to encourage sparsity [**?**]. A sparse autoencoder solves

$$\min_{\boldsymbol{f},\boldsymbol{g}}\ \underbrace{\frac{1}{n}\sum_{i=1}^n\mathcal{L}\left(\boldsymbol{x}_i,\boldsymbol{g}\left(\boldsymbol{h}_i\right)\right)}_{\text{loss}}+\underbrace{\lambda\left\|\boldsymbol{h}_i\right\|_1}_{\text{regularizer}}\quad\text{with}\quad\boldsymbol{h}_i=\boldsymbol{f}\left(\boldsymbol{x}_i\right),\ \text{for all }i\in[n].$$

  This is similar to *dictionary learning*, where one aims at finding a sparse representation of input data on an overcomplete basis. Due to the imposed sparsity, the model can potentially learn useful features of the data.

- *Denoising autoencoders.* One may hope that the model is robust to noise in the data: even if the input data $\boldsymbol{x}_i$ are corrupted by small noise $\boldsymbol{\xi}_i$ or miss some components (the noise level or the missing probability is typically small), an ideal autoencoder should faithfully recover the original data. A denoising

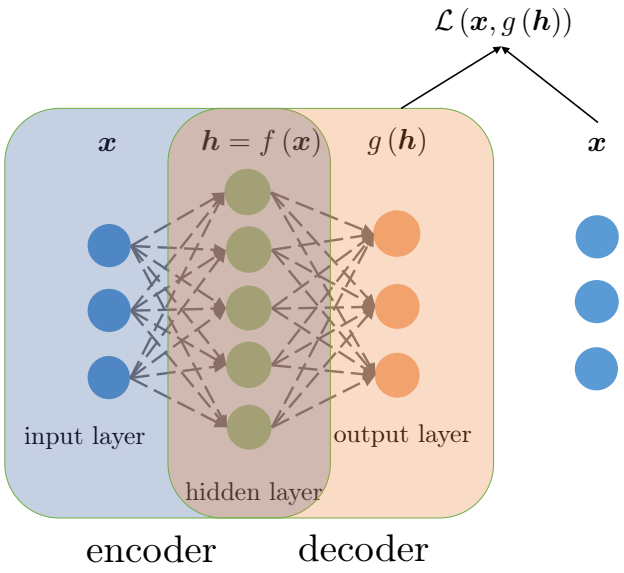


Figure 11: First an input $\boldsymbol{x}$ goes through the decoder $\boldsymbol{f}(\cdot)$, and we obtain its hidden representation $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x})$. Then, we use the decoder $\boldsymbol{g}(\cdot)$ to get $\boldsymbol{g}(\boldsymbol{h})$ as a reconstruction of $\boldsymbol{x}$. Finally, the loss is determined from the difference between the original input $\boldsymbol{x}$ and its reconstruction $\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$.

autoencoder [**?**] achieves this robustness by explicitly building a noisy data $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{\xi}_i$ as the new input, and then solves an optimization problem similar to (16) where $\mathcal{L}(\boldsymbol{x}_i, \boldsymbol{g}(\boldsymbol{h}_i))$ is replaced by $\mathcal{L}(\boldsymbol{x}_i, \boldsymbol{g}(\boldsymbol{f}(\tilde{\boldsymbol{x}}_i)))$. A denoising autoencoder encourages the encoder/decoder to be stable in the neighborhood of an input, which is generally a good statistical property. An alternative way could be constraining f and g in the optimization problem, but that would be very difficult to optimize. Instead, sampling by adding small perturbations in the input provides a simple implementation. We shall see similar ideas in Section 6.3.3.

4.2 Generative adversarial networks

Given unlabeled data $\{\boldsymbol{x}_i\}_{1 \le i \le n}$, density estimation aims to estimate the underlying probability density function $\mathbb{P}_{\boldsymbol{X}}$ from which the data is generated. Both parametric and nonparametric estimators [**?**] have been proposed and studied under various assumptions on the underlying distribution. Different from these classical density estimators, where the density function is explicitly defined in relatively low dimension, generative adversarial networks (GANs) [**?**] can be categorized as an *implicit* density estimator in much higher dimension. The reasons are twofold: (1) GANs put more emphasis on sampling from the distribution $\mathbb{P}_{\boldsymbol{X}}$ than estimation; (2) GANs define the density estimation implicitly through a source distribution $\mathbb{P}_{\boldsymbol{Z}}$ and a generator function $g(\cdot)$, which is usually a deep neural network. We introduce GANs from the perspective of sampling from $\mathbb{P}_{\boldsymbol{X}}$ and later we will generalize the vanilla GANs using its relation to density estimators.

4.2.1 Sampling view of GANs

Suppose the data $\{\boldsymbol{x}_i\}_{1 \le i \le n}$ at hand are all real images, and we want to generate *new* natural images. With this goal in mind, GAN models a *zero-sum* game between two players, namely, the generator $\mathcal{G}$ and the discriminator $\mathcal{D}$. The generator $\mathcal{G}$ tries to generate fake images akin to the true images $\{\boldsymbol{x}_i\}_{1 \le i \le n}$ while the discriminator $\mathcal{D}$ aims at differentiating the fake ones from the true ones. Intuitively, one hopes to learn a generator $\mathcal{G}$ to generate images where the *best* discriminator $\mathcal{D}$ cannot distinguish. Therefore the payoff is higher for the generator $\mathcal{G}$ if the probability of the discriminator $\mathcal{D}$ getting wrong is higher, and correspondingly the payoff for the discriminator correlates positively with its ability to tell wrong from truth.

Mathematically, the generator $\mathcal{G}$ consists of two components, an source distribution $\mathbb{P}_{\boldsymbol{Z}}$ (usually a standard multivariate Gaussian distribution with hundreds of dimensions) and a function $\mathbf{g}(\cdot)$ which maps a sample $\boldsymbol{z}$ from $\mathbb{P}_{\boldsymbol{Z}}$ to a point $\mathbf{g}(\boldsymbol{z})$ living in the same space as $\boldsymbol{x}$. For generating images, $\mathbf{g}(\boldsymbol{z})$ would be a

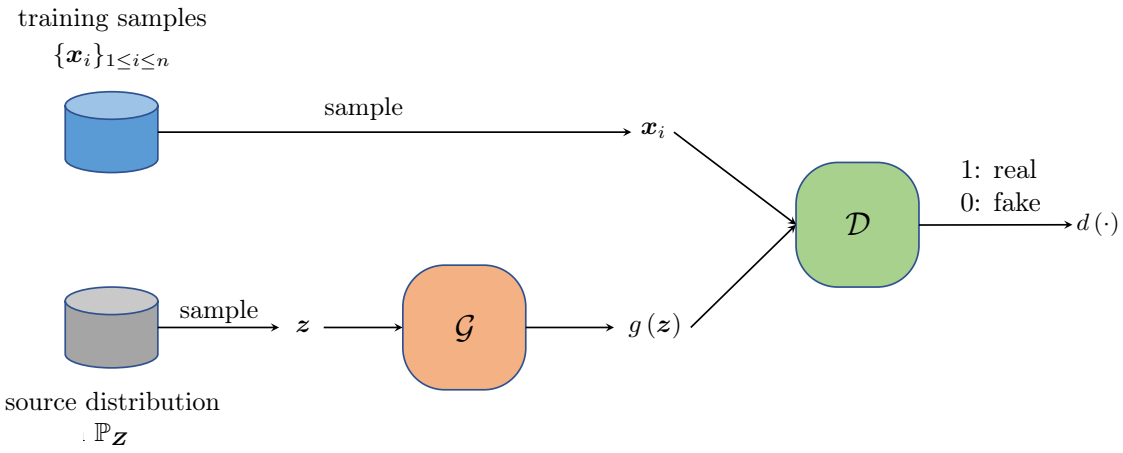


Figure 12: GANs consist of two components, a generator $\mathcal{G}$ which generates fake samples and a discriminator $\mathcal{D}$ which differentiate the true ones from the fake ones.

3D tensor. Here $\mathbf{g}(\boldsymbol{z})$ is the fake sample generated from $\mathcal{G}$. Similarly the discriminator $\mathcal{D}$ is composed of one function which takes an image $\boldsymbol{x}$ (real or fake) and return a number $d(\boldsymbol{x}) \in [0,1]$, the probability of $\boldsymbol{x}$ being a real sample from $\mathbb{P}_{\boldsymbol{X}}$ or not. Oftentimes, both the generating function $\mathbf{g}(\cdot)$ and the discriminating function $d(\cdot)$ are realized by deep neural networks, e.g., CNNs introduced in Section 3.1. See Figure 12 for an illustration for GANs. Denote $\boldsymbol{\theta}_{\mathcal{G}}$ and $\boldsymbol{\theta}_{\mathcal{D}}$ the parameters in $\mathbf{g}(\cdot)$ and $d(\cdot)$, respectively. Then GAN tries to solve the following *min-max* problem:

$$\min_{\boldsymbol{\theta}_{\mathcal{G}}} \max_{\boldsymbol{\theta}_{\mathcal{D}}} \quad \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\boldsymbol{X}}}\left[\log\left(d\left(\boldsymbol{x}\right)\right)\right] + \mathbb{E}_{\boldsymbol{z}\sim\mathbb{P}_{\boldsymbol{Z}}}\left[\log\left(1-d\left(\mathbf{g}\left(\boldsymbol{z}\right)\right)\right)\right]. \tag{17}$$

Recall that $d(\boldsymbol{x})$ models the belief / probability that the discriminator thinks that $\boldsymbol{x}$ is a true sample. Fix the parameters $\boldsymbol{\theta}_{\mathcal{G}}$ and hence the generator $\mathcal{G}$ and consider the inner maximization problem. We can see that the goal of the discriminator is to maximize its ability of differentiation. Similarly, if we fix $\boldsymbol{\theta}_{\mathcal{D}}$ (and hence the discriminator), the generator tries to generate more realistic images $\mathbf{g}(\boldsymbol{z})$ to fool the discriminator.

4.2.2 Density estimation view of GANs

Let us now take a density-estimation view of GANs. Fixing the source distribution $\mathbb{P}_{\boldsymbol{Z}}$, any generator $\mathcal{G}$ induces a distribution $\mathbb{P}_{\mathcal{G}}$ over the space of images. Removing the restrictions on $d(\cdot)$, one can then rewrite (17) as

$$\min_{\mathbb{P}_{\mathcal{G}}} \max_{d(\cdot)} \quad \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\boldsymbol{X}}}\left[\log\left(d\left(\boldsymbol{x}\right)\right)\right] + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\mathcal{G}}}\left[\log\left(1-d\left(\boldsymbol{x}\right)\right)\right]. \tag{18}$$

Observe that the inner maximization problem is solved by the likelihood ratio, i.e.

$$d^*\left(\boldsymbol{x}\right) = \frac{\mathbb{P}_{\boldsymbol{X}}\left(\boldsymbol{x}\right)}{\mathbb{P}_{\boldsymbol{X}}\left(\boldsymbol{x}\right) + \mathbb{P}_{\mathcal{G}}\left(\boldsymbol{x}\right)}.$$

As a result, (18) can be simplified as

$$\min_{\mathbb{P}_{\mathcal{G}}} \quad \mathrm{JS}\left(\mathbb{P}_{\boldsymbol{X}} \parallel \mathbb{P}_{\mathcal{G}}\right), \tag{19}$$

where $\mathrm{JS}(\cdot\|\cdot)$ denotes the Jensen–Shannon divergence between two distributions

$$\mathrm{JS}\left(\mathbb{P}_{\boldsymbol{X}}\|\mathbb{P}_{\mathcal{G}}\right) = \frac{1}{2}\mathrm{KL}\big(\mathbb{P}_{\boldsymbol{X}} \parallel \tfrac{\mathbb{P}_{\boldsymbol{X}}+\mathbb{P}_{\mathcal{G}}}{2}\big) + \frac{1}{2}\mathrm{KL}\big(\mathbb{P}_{\mathcal{G}} \parallel \tfrac{\mathbb{P}_{\boldsymbol{X}}+\mathbb{P}_{\mathcal{G}}}{2}\big).$$

In words, the vanilla GAN (17) seeks a density $\mathbb{P}_{\mathcal{G}}$ that is closest to $\mathbb{P}_{\boldsymbol{X}}$ in terms of the Jensen–Shannon divergence. This view allows to generalize GANs to other variants, by changing the distance metric. Examples include f-GAN [**?**], Wasserstein GAN (W-GAN) [**?**], MMD GAN [**?**], etc. We single out the Wasserstein

GAN (W-GAN) [?] to introduce due to its popularity. As the name suggests, it minimizes the Wasserstein distance between $\mathbb{P}_{\boldsymbol{X}}$ and $\mathbb{P}_{\mathcal{G}}$:

$$\min_{\boldsymbol{\theta}_{\mathcal{G}}} \quad \mathrm{WS}\left(\mathbb{P}_{\boldsymbol{X}} \| \mathbb{P}_{\mathcal{G}}\right) \quad = \quad \min_{\boldsymbol{\theta}_{\mathcal{G}}} \sup_{f: f \text{ 1-Lipschitz}} \mathbb{E}_{\boldsymbol{x} \sim \mathbb{P}_{\boldsymbol{X}}}\left[f\left(\boldsymbol{x}\right)\right] - \mathbb{E}_{\boldsymbol{x} \sim \mathbb{P}_{\mathcal{G}}}\left[f\left(\boldsymbol{x}\right)\right], \tag{20}$$

where $f(\cdot)$ is taken over all Lipschitz functions with coefficient 1. Comparing W-GAN (20) with the original formulation of GAN (17), one finds that the Lipschitz function f in (20) corresponds to the discriminator $\mathcal{D}$ in (17) in the sense that they share similar objectives to differentiate the true distribution $\mathbb{P}_{\boldsymbol{X}}$ from the fake one $\mathbb{P}_{\mathcal{G}}$. In the end, we would like to mention that GANs are more difficult to train than supervised deep learning models such as CNNs [?]. Apart from the training difficulty, how to evaluate GANs objectively and effectively is an ongoing research.

5 Representation power: approximation theory

Having seen the building blocks of deep learning models in the previous sections, it is natural to ask: what is the benefits of composing multiple layers of nonlinear functions. In this section, we address this question from a approximation theoretical point of view. Mathematically, letting $\mathcal{H}$ be the space of functions representable by neural nets (NNs), how well can a function f (with certain properties) be approximated by functions in $\mathcal{H}$. We first revisit universal approximation theories, which are mostly developed for shallow neural nets (neural nets with a single hidden layer), and then provide recent results that demonstrate the benefits of depth in neural nets. Other notable works include Kolmogorov-Arnold superposition theorem [?, ?], and circuit complexity for neural nets [?].

5.1 Universal approximation theory for shallow NNs

The universal approximation theories study the approximation of f in a space $\mathcal{F}$ by a function represented by a one-hidden-layer neural net

$$g(\boldsymbol{x}) = \sum_{j=1}^{N} c_j \sigma_*(\boldsymbol{w}_j^\top \boldsymbol{x} - b_j), \tag{21}$$

where $\sigma_* : \mathbb{R} \to \mathbb{R}$ is certain activation function and N is the number of hidden units in the neural net. For different space $\mathcal{F}$ and activation function σ_*, there are upper bounds and lower bounds on the approximation error $\|f - g\|$. See [?] for a comprehensive overview. Here we present representative results.

First, as $N \to \infty$, any continuous function f can be approximated by some g under mild conditions. Loosely speaking, this is because each component $\sigma_*(\boldsymbol{w}_j^\top \boldsymbol{x} - b_j)$ behaves like a basis function and functions in a suitable space $\mathcal{F}$ admits a basis expansion. Given the above heuristics, the next natural question is: what is the rate of approximation for a finite N?

Let us restrict the domain of $\boldsymbol{x}$ to a unit ball B^d in $\mathbb{R}^d$. For $p \in [1, \infty)$ and integer $m \geq 1$, consider the L^p space and the Sobolev space with standard norms

$$\|f\|_p = \Big[\int_{B^n} |g(\boldsymbol{x})|^p \, d\boldsymbol{x}\Big]^{1/p}, \qquad \|f\|_{m,p} = \Big[\sum_{0 \leq |\boldsymbol{k}| \leq m} \|D^{\boldsymbol{k}} f\|_p^p\Big]^{1/p},$$

where $D^{\boldsymbol{k}} f$ denotes partial derivatives indexed by $\boldsymbol{k} \in \mathbb{Z}_+^d$. Let $\mathcal{F} \triangleq \mathcal{F}_p^m$ be the space of functions f in the Sobolev space with $\|f\|_{m,p} \leq 1$. Note that functions in $\mathcal{F}$ have bounded derivatives up to m-th order, and that smoothness of functions is controlled by m (larger m means smoother). Denote by $\mathcal{H}_N$ the space of functions with the form (21). The following general upper bound is due to [?].

Theorem 1 (Theorem 2.1 in [?]). *Assume $\sigma_* : \mathbb{R} \to \mathbb{R}$ is such that σ_* has arbitrary order derivatives in an open interval I, and that σ_* is not a polynomial on I. Then, for any $p \in [1, \infty)$, $d \geq 2$, and integer $m \geq 1$,*

$$\sup_{f \in \mathcal{F}_p^m} \inf_{g \in \mathcal{H}_N} \|f - g\|_p \leq C_{d,m,p} \, N^{-m/d},$$

where $C_{d,m,p}$ is independent of N, the number of hidden units.

In the above theorem, the condition on $\sigma_*(\cdot)$ is mainly technical. This upper bound is useful when the dimension d is not large. It clearly implies that the one-hidden-layer neural net is able to approximate any smooth function with enough hidden units. However, it is unclear how to find a good approximator g; nor do we have control over the magnitude of the parameters (huge weights are impractical). While increasing the number of hidden units N leads to better approximation, the exponent $-m/d$ suggests the presence of the *curse of dimensionality*. The following (nearly) matching lower bound is stated in [?].

Theorem 2 (Theorem 5 in [?]). *Let $p \geq 1$, $m \geq 1$ and $N \geq 2$. If the activation function is the standard sigmoid function $\sigma(t) = (1 + e^{-t})^{-1}$, then*

$$\sup_{f \in \mathcal{F}_p^m} \inf_{g \in \mathcal{H}_N} \|f - g\|_p \geq C'_{d,m,p} \left(N \log N\right)^{-m/d}, \tag{22}$$

where $C'_{d,m,p}$ is independent of N.

Results for other activation functions are also obtained by [?]. Moreover, the term $\log N$ can be removed if we assume an additional continuity condition [?].

For the natural space $\mathcal{F}_p^m$ of smooth functions, the exponential dependence on d in the upper and lower bounds may look unappealing. However, [?] showed that for a different function space, there is a good dimension-free approximation by the neural nets. Suppose that a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ has a Fourier representation

$$f(\boldsymbol{x}) = \int_{\mathbb{R}^d} e^{i\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} \tilde{f}(\boldsymbol{\omega}) \, d\boldsymbol{\omega}, \tag{23}$$

where $\tilde{f}(\boldsymbol{\omega}) \in \mathbb{C}$. Assume that $f(\mathbf{0}) = 0$ and that the following quantity is finite

$$C_f = \int_{\mathbb{R}^d} \|\boldsymbol{\omega}\|_2 |\tilde{f}(\boldsymbol{\omega})| \, d\boldsymbol{\omega}. \tag{24}$$

[?] uncovers the following dimension-free approximation guarantee.

Theorem 3 (Proposition 1 in [?]). *Fix a $C > 0$ and an arbitrary probability measure μ on the unit ball B^d in $\mathbb{R}^d$. For every function f with $C_f \leq C$ and every $N \geq 1$, there exists some $g \in \mathcal{H}_N$ such that*

$$\left[\int_{B^d} (f(\boldsymbol{x}) - g(\boldsymbol{x}))^2 \, \mu(d\boldsymbol{x})\right]^{1/2} \leq \frac{2C}{\sqrt{N}}.$$

Moreover, the coefficients of g may be restricted to satisfy $\sum_{j=1}^N |c_j| \leq 2C$.

The upper bound is now independent of the dimension d. However, C_f may implicitly depend on d, as the formula in (24) involves an integration over $\mathbb{R}^d$ (so for some functions C_f may depend exponentially on d). Nevertheless, this theorem does characterize an interesting function space with an improved upper bound. Details of the function space are discussed by [?]. This theorem can be generalized; see [?] for an example.

To help understand why a dimensionality-free approximation holds, let us appeal to a heuristic argument given by Monte Carlo simulations. It is well-known that Monte Carlo approximation errors are independent of dimensionality in evaluation of high-dimensional integrals. Let us generate $\{\boldsymbol{\omega}_j\}_{1 \leq j \leq N}$ randomly from a given density $p(\cdot)$ in $\mathbb{R}^d$. Consider the approximation to (23) by

$$g_N(\boldsymbol{x}) = \frac{1}{N} \sum_{j=1}^N c_j e^{i\langle \boldsymbol{\omega}_j, \boldsymbol{x} \rangle}, \qquad c_j = \frac{\tilde{f}(\boldsymbol{\omega}_j)}{p(\boldsymbol{\omega}_j)}. \tag{25}$$

Then, $g_N(\boldsymbol{x})$ is a one-hidden-layer neural network with N units and the sinusoid activation function. Note that $\mathbb{E} g_N(\boldsymbol{x}) = f(\boldsymbol{x})$, where the expectation is taken with respect to randomness $\{\boldsymbol{\omega}_j\}$. Now, by independence, we have

$$\mathbb{E}(g_N(\boldsymbol{x}) - f(\boldsymbol{x}))^2 = \frac{1}{N} \mathrm{Var}(c_j e^{i\langle \boldsymbol{\omega}_j, \boldsymbol{x} \rangle}) \leq \frac{1}{N} \mathbb{E} c_j^2,$$

if $\mathbb{E} c_j^2 < \infty$. Therefore, the rate is independent of the dimensionality d, though the constant can be.

5.2 Approximation theory for multi-layer NNs

The approximation theory for multilayer neural nets is less understood compared with neural nets with one hidden layer. Driven by the success of deep learning, there are many recent papers focusing on expressivity of deep neural nets. As studied by [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**], deep neural nets excel at representing *composition* of functions. This is perhaps not surprising, since deep neural nets are themselves defined by composing layers of functions. Nevertheless, it points to a new territory rarely studied in statistics before. Below we present a result based on [**?**, **?**].

Suppose that the inputs $\boldsymbol{x}$ have a bounded domain $[-1,1]^d$ for simplicity. As before, let $\sigma_* : \mathbb{R} \to \mathbb{R}$ be a generic function, and $\boldsymbol{\sigma}_* = (\sigma_*, \cdots, \sigma_*)^\top$ be element-wise application of σ_*. Consider a neural net which is similar to (3) but with scaler output: $g(\boldsymbol{x}) = \mathbf{W}_\ell \boldsymbol{\sigma}_*(\cdots \boldsymbol{\sigma}_*(\mathbf{W}_2 \boldsymbol{\sigma}_*(\mathbf{W}_1 \boldsymbol{x}))\cdots)$. A unit or neuron refers to an element of vectors $\boldsymbol{\sigma}_*(\mathbf{W}_k \cdots \boldsymbol{\sigma}_*(\mathbf{W}_2 \boldsymbol{\sigma}_*(\mathbf{W}_1 \boldsymbol{x}))\cdots)$ for any $k = 1, \ldots, \ell - 1$. For a multivariate polynomial p, define $m_k(p)$ to be the smallest integer such that, for any $\epsilon > 0$, there exists a neural net $g(\boldsymbol{x})$ satisfying $\sup_{\boldsymbol{x}} |p(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon$, with k hidden layers (i.e., $\ell = k + 1$) and no more than $m_k(p)$ neurons in total. Essentially, $m_k(p)$ is the minimum number of neurons required to approximate p arbitrarily well.

Theorem 4 (Theorem 4.1 in [**?**]). *Let $p(\boldsymbol{x})$ be a monomial $x_1^{r_1} x_2^{r_2} \cdots x_d^{r_d}$ with $q = \sum_{j=1}^d r_j$. Suppose that σ_* has derivatives of order $2q$ at the origin, and that they are nonzero. Then,*
(i) $m_1(p) = \prod_{j=1}^d (r_j + 1)$;
(ii) $\min_k m_k(p) \le \sum_{j=1}^d (7\lceil \log_2(r_j) \rceil + 4)$.

This theorem reveals a sharp distinction between shallow networks (one hidden layer) and deep networks. To represent a monomial function, a shallow network requires exponentially many neurons in terms of the dimension d, whereas linearly many neurons suffice for a deep network (with bounded r_j). The exponential dependence on d, as shown in Theorem 4(i), is resonant with the curse of dimensionality widely seen in many fields; see [**?**]. One may ask: how does depth help? Depth circumvents this issue, at least for certain functions, by allowing us to represent function composition efficiently. Indeed, Theorem 4(ii) offers a nice result with clear intuitions: it is known that the product of two scalar inputs can be represented using 4 neurons [**?**], so by composing multiple products, we can express monomials with $O(d)$ neurons.

Recent advances in nonparametric regressions also support the idea that deep neural nets excel at representing composition of functions [**?**, **?**]. In particular, [**?**] considered the nonparametric regression setting where we want to estimate a function $\hat{f}_n(\boldsymbol{x})$ from i.i.d. data $\mathcal{D}_n = \{(y_i, \boldsymbol{x}_i)\}_{1 \le i \le n}$. If the true regression function $f(\boldsymbol{x})$ has certain hierarchical structure with intrinsic dimensionality[6] d^*, then the error

$$\mathbb{E}_{\mathcal{D}_n} \mathbb{E}_{\boldsymbol{x}} \left| \hat{f}_n(\boldsymbol{x}) - f(\boldsymbol{x}) \right|^2$$

has an optimal minimax convergence rate $O(n^{-\frac{2q}{2q+d^*}})$, rather than the usual rate $O(n^{-\frac{2q}{2q+d}})$ that depends on the ambient dimension d. Here q is the smoothness parameter. This provides another justification for deep neural nets: if data are truly hierarchical, then the quality of approximators by deep neural nets depends on the intrinsic dimensionality, which avoids the curse of dimensionality.

We point out that the approximation theory for deep learning is far from complete. For example, in Theorem 4, the condition on σ_* excludes the widely used ReLU activation function, there are no constraints on the magnitude of the weights (so they can be unreasonably large).

6 Training deep neural nets

The *existence* of a good function approximator in the NN function class does not explain why in practice we can easily *find* them. In this section, we introduce standard methods, namely *stochastic gradient descent* (SGD) and its variants, to train deep neural networks (or to find such a good approximator). As with many statistical machine learning tasks, training DNNs follows the *empirical risk minimization* (ERM) paradigm

[6] Roughly speaking, the true regression function can be represented by a tree where each node has at most d^* children. See [**?**] for the precise definition.

which solves the following optimization problem

$$\text{minimize}_{\boldsymbol{\theta}\in\mathbb{R}^p} \qquad \ell_n(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(f\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right),y_i\right). \tag{26}$$

Here $\mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}),y_i)$ measures the discrepancy between the prediction $f(\boldsymbol{x}_i;\boldsymbol{\theta})$ of the neural network and the true label y_i. Correspondingly, denote by $\ell(\boldsymbol{\theta}) \triangleq \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}[\mathcal{L}(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y})]$ the out-of-sample error, where $\mathcal{D}$ is the joint distribution over $(y,\boldsymbol{x})$. Solving ERM (26) for deep neural nets faces various challenges that roughly fall into the following three categories.

- *Scalability and nonconvexity.* Both the sample size n and the number of parameters p can be huge for modern deep learning applications, as we have seen in Table 1. Many optimization algorithms are not practical due to the computational costs and memory constraints. What is worse, the empirical loss function $\ell_n(\boldsymbol{\theta})$ in deep learning is often nonconvex. It is *a priori* not clear whether an optimization algorithm can drive the empirical loss (26) small.

- *Numerical stability.* With a large number of layers in DNNs, the magnitudes of the hidden nodes can be drastically different, which may result in the "exploding gradients" or "vanishing gradients" issue during the training process. This is because the recursive relations across layers often lead to exponentially increasing / decreasing values in both forward passes and backward passes.

- *Generalization performance.* Our ultimate goal is to find a parameter $\hat{\boldsymbol{\theta}}$ such that the out-of-sample error $\ell(\hat{\boldsymbol{\theta}})$ is small. However, in the over-parametrized regime where p is much larger than n, the underlying neural network has the potential to fit the training data perfectly while performing poorly on the test data. To avoid this overfitting issue, proper regularization, whether explicit or implicit, is needed in the training process for the neural nets to generalize.

In the following three subsections, we discuss practical solutions / proposals to address these challenges.

6.1 Stochastic gradient descent

Stochastic gradient descent (SGD) [**?**] is by far the most popular optimization algorithm to solve ERM (26) for large-scale problems. It has the following simple update rule:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t G(\boldsymbol{\theta}^t) \qquad \text{with} \qquad G\left(\boldsymbol{\theta}^t\right) = \nabla\mathcal{L}\left(f\left(\boldsymbol{x}_{i_t};\boldsymbol{\theta}^t\right),y_{i_t}\right) \tag{27}$$

for $t = 0,1,2,\ldots$, where $\eta_t > 0$ is the step size (or learning rate), $\boldsymbol{\theta}^0 \in \mathbb{R}^p$ is an initial point and i_t is chosen randomly from $\{1,2,\cdots,n\}$. It is easy to verify that $G(\boldsymbol{\theta}^t)$ is an unbiased estimate of $\nabla\ell_n(\boldsymbol{\theta}^t)$. The advantage of SGD is clear: compared with gradient descent, which goes over the entire dataset in every update, SGD uses a single example in each update and hence is considerably more efficient in terms of both computation and memory (especially in the first few iterations).

Apart from practical benefits of SGD, how well does SGD perform theoretically in terms of minimizing $\ell_n(\boldsymbol{\theta})$? We begin with the convex case, i.e., the case where the loss function is convex w.r.t. $\boldsymbol{\theta}$. It is well understood in literature that with proper choices of the step sizes $\{\eta_t\}$, SGD is guaranteed to achieve both *consistency* and *asymptotic normality*.

- *Consistency.* If $\ell(\boldsymbol{\theta})$ is a strongly convex function[7], then under some mild conditions[8], learning rates that satisfy

$$\sum_{t=0}^{\infty}\eta_t = +\infty \qquad \text{and} \qquad \sum_{t=0}^{\infty}\eta_t^2 < +\infty \tag{28}$$

[7]For results on consistency and asymptotic normality, we consider the case where in each step of SGD, the stochastic gradient is computed using a fresh sample $(y,\boldsymbol{x})$ from $\mathcal{D}$. This allows to view SGD as an optimization algorithm to minimize the population loss $\ell(\boldsymbol{\theta})$.

[8]One example of such condition can be constraining the second moment of the gradients: $\mathbb{E}\left[\|\nabla\mathcal{L}\left(\boldsymbol{x}_i,y_i;\boldsymbol{\theta}^t\right)\|_2^2\right] \le C_1 + C_2\|\boldsymbol{\theta}^t - \boldsymbol{\theta}^*\|_2^2$ for some $C_1, C_2 > 0$. See [**?**] for details.

guarantee almost sure convergence to the unique minimizer $\boldsymbol{\theta}^* \triangleq \operatorname{argmin}_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta})$, i.e., $\boldsymbol{\theta}^t \xrightarrow{\text{a.s.}} \boldsymbol{\theta}^*$ as $t \to \infty$ [?, ?, ?, ?]. The requirements in (28) can be viewed from the perspective of bias-variance tradeoff: the first condition ensures that the iterates can reach the minimizer (controlled bias), and the second ensures that stochasticity does not prevent convergence (controlled variance).

- *Asymptotic normality.* It is proved by [?] that for robust linear regression with fixed dimension p, under the choice $\eta_t = t^{-1}$, $\sqrt{t}\,(\boldsymbol{\theta}^t - \boldsymbol{\theta}^*)$ is asymptotically normal under some regularity conditions (but $\boldsymbol{\theta}^t$ is not asymptotically efficient in general). Moreover, by averaging the iterates of SGD, [?] proved that even with a *larger* step size $\eta_t \propto t^{-\alpha}, \alpha \in (1/2, 1)$, the averaged iterate $\bar{\boldsymbol{\theta}}^t = t^{-1}\sum_{s=1}^t \boldsymbol{\theta}^s$ is asymptotic efficient for robust linear regression. These strong results show that SGD with averaging performs as well as the MLE asymptotically, in addition to its computational efficiency.

These classical results, however, fail to explain the effectiveness of SGD when dealing with nonconvex loss functions in deep learning. Admittedly, finding global minima of nonconvex functions is computationally infeasible in the worst case. Nevertheless, recent work [?, ?] bypasses the worst case scenario by focusing on losses incurred by over-parametrized deep learning models. In particular, they show that (stochastic) gradient descent converges linearly towards the *global* minimizer of $\ell_n(\boldsymbol{\theta})$ as long as the neural network is sufficiently *over-parametrized*. This phenomenon is formalized below.

Theorem 5 (Theorem 2 in [?]). *Let $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ be a training set satisfying $\min_{i,j:i\neq j} \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 \ge \delta > 0$. Consider fitting the data using a feed-forward neural network (1) with ReLU activations. Denote by L (resp. W) the depth (resp. width) of the network. Suppose that the neural network is sufficiently over-parametrized, i.e.,*

$$W \gg \mathsf{poly}\left(n, L, \frac{1}{\delta}\right), \tag{29}$$

where poly *means a polynomial function. Then with high probability, running SGD (27) with* certain random initialization *and properly chosen step sizes yields $\ell_n(\boldsymbol{\theta}^t) \le \varepsilon$ in $t \asymp \log\frac{1}{\varepsilon}$ iterations.*

Two notable features are worth mentioning: (1) first, the network under consideration is sufficiently over-parametrized (cf. (29)) in which the number of parameters is *much* larger than the number of samples, and (2) one needs to initialize the weight matrices to be in near-isometry such that the magnitudes of the hidden nodes do not blow up or vanish. In a nutshell, *over-parametrization* and *random initialization* together ensure that the loss function (26) has a benign landscape[9] around the initial point, which in turn implies fast convergence of SGD iterates.

There are certainly other challenges for vanilla SGD to train deep neural nets: (1) training algorithms are often implemented in GPUs, and therefore it is important to tailor the algorithm to the infrastructure, (2) the vanilla SGD might converge very slowly for deep neural networks, albeit good theoretical guarantees for well-behaved problems, and (3) the learning rates $\{\eta_t\}$ can be difficult to tune in practice. To address the aforementioned challenges, three important variants of SGD, namely *mini-batch SGD, momentum-based SGD*, and *SGD with adaptive learning rates* are introduced.

6.1.1 Mini-batch SGD

Modern computational infrastructures (e.g., GPUs) can evaluate the gradient on a number (say 64) of examples as efficiently as evaluating that on a single example. To utilize this advantage, mini-batch SGD with batch size $K \ge 1$ forms the stochastic gradient through K random samples:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t G(\boldsymbol{\theta}^t) \qquad \text{with} \qquad G(\boldsymbol{\theta}^t) = \frac{1}{K}\sum_{k=1}^{K} \nabla\mathcal{L}\big(f\big(\boldsymbol{x}_{i_t^k}; \boldsymbol{\theta}^t\big), y_{i_t^k}\big), \tag{30}$$

where for each $1 \le k \le K$, i_t^k is sampled uniformly from $\{1, 2, \cdots, n\}$. Mini-batch SGD, which is an "interpolation" between gradient descent and stochastic gradient descent, achieves the best of both worlds: (1) using $1 \ll K \ll n$ samples to estimate the gradient, one effectively reduces the variance and hence accelerates the convergence, and (2) by taking the batch size K appropriately (say 64 or 128), the stochastic gradient $G(\boldsymbol{\theta}^t)$ can be efficiently computed using the matrix computation toolboxes on GPUs.

[9] In [?], the loss function $\ell_n(\boldsymbol{\theta})$ satisfies the PL condition.

6.1.2 Momentum-based SGD

While mini-batch SGD forms the foundation of training neural networks, it can sometimes be slow to converge due to its oscillation behavior [**?**]. Optimization community has long investigated how to accelerate the convergence of gradient descent, which results in a beautiful technique called *momentum methods* [**?**, **?**]. Similar to gradient descent with moment, *momentum-based SGD*, instead of moving the iterate $\boldsymbol{\theta}^t$ in the direction of the current stochastic gradient $G(\boldsymbol{\theta}^t)$, smooth the past (stochastic) gradients $\{G(\boldsymbol{\theta}^t)\}$ to stabilize the update directions. Mathematically, let $\boldsymbol{v}^t \in \mathbb{R}^p$ be the direction of update in the tth iteration, i.e.,

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t \boldsymbol{v}^t.$$

Here $\boldsymbol{v}^0 = G(\boldsymbol{\theta}^0)$ and for $t = 1, 2, \cdots$

$$\boldsymbol{v}^t = \rho \boldsymbol{v}^{t-1} + G(\boldsymbol{\theta}^t) \tag{31}$$

with $0 < \rho < 1$. A typical choice of ρ is 0.9. Notice that $\rho = 0$ recovers the mini-batch SGD (30), where no past information of gradients is used. A simple unrolling of $\boldsymbol{v}^t$ reveals that $\boldsymbol{v}^t$ is actually an exponential averaging of the past gradients, i.e., $\boldsymbol{v}^t = \sum_{j=0}^{t} \rho^{t-j} G(\boldsymbol{\theta}^j)$. Compared with vanilla mini-batch SGD, the inclusion of the momentum "smoothes" the oscillation direction and accumulates the persistent descent direction. We want to emphasize that theoretical justifications of momentum in the *stochastic* setting is not fully understood [**?**, **?**].

6.1.3 SGD with adaptive learning rates

In optimization, *preconditioning* is often used to accelerate first-order optimization algorithms. In principle, one can apply this to SGD, which yields the following update rule:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t \boldsymbol{P}_t^{-1} G(\boldsymbol{\theta}^t) \tag{32}$$

with $\boldsymbol{P}_t \in \mathbb{R}^{p \times p}$ being a preconditioner at the t-th step. Newton's method can be viewed as one type of preconditioning where $\boldsymbol{P}_t = \nabla^2 \ell(\boldsymbol{\theta}^t)$. The advantages of preconditioning are two-fold: first, a good preconditioner reduces the condition number by changing the local geometry to be more homogeneous, which is amenable to fast convergence; second, a good preconditioner frees practitioners from laboring tuning of the step sizes, as is the case with Newton's method. AdaGrad, an adaptive gradient method proposed by [**?**], builds a preconditioner $\boldsymbol{P}_t$ based on information of the past gradients:

$$\boldsymbol{P}_t = \Big\{ \mathsf{diag}\Big(\sum_{j=0}^{t} G\left(\boldsymbol{\theta}^t\right) G\left(\boldsymbol{\theta}^t\right)^\top \Big) \Big\}^{1/2}. \tag{33}$$

Since we only require the diagonal part, this preconditioner (and its inverse) can be efficiently computed in practice. In addition, investigating (32) and (33), one can see that AdaGrad adapts to the importance of each coordinate of the parameters by setting smaller learning rates for frequent features, whereas larger learning rates for those infrequent ones. In practice, one adds a small quantity $\delta > 0$ (say 10^{-8}) to the diagonal entries to avoid singularity (numerical underflow). A notable drawback of AdaGrad is that the effective learning rate vanishes quickly along the learning process. This is because the historical sum of the gradients can only increase with time. RMSProp [**?**] is a popular remedy for this problem which incorporates the idea of exponential averaging:

$$\boldsymbol{P}_t = \Big\{ \mathsf{diag}\Big(\rho \boldsymbol{P}_{t-1} + (1-\rho) G\left(\boldsymbol{\theta}^t\right) G\left(\boldsymbol{\theta}^t\right)^\top \Big) \Big\}^{1/2}. \tag{34}$$

Again, the decaying parameter ρ is usually set to be 0.9. Later, Adam [**?**, **?**] combines the momentum method and adaptive learning rate and becomes the default training algorithms in many deep learning applications.

6.2 Easing numerical instability

For very deep neural networks or RNNs with long dependencies, training difficulties often arise when the values of nodes have different magnitudes or when the gradients "vanish" or "explode" during back-propagation. Here we discuss three partial solutions to alleviate this problem.

6.2.1 ReLU activation function

One useful characteristic of the ReLU function is that its derivative is either 0 or 1, and the derivative remains 1 even for a large input. This is in sharp contrast with the standard sigmoid function $(1+e^{-t})^{-1}$ which results in a very small derivative when inputs have large magnitude. The consequence of small derivatives across many layers is that gradients tend to be "killed", which means that gradients become approximately zero in deep nets.

The popularity of the ReLU activation function and its variants (e.g., leaky ReLU) is largely attributable to the above reason. It has been well observed that the ReLU activation function has superior training performance over the sigmoid function [**?**, **?**].

6.2.2 Skip connections

We have introduced skip connections in Section 3.3. Why are skip connections helpful for reducing numerical instability? This structure does not introduce a larger function space, since the identity map can be also represented with ReLU activations: $\boldsymbol{x} = \boldsymbol{\sigma}(\boldsymbol{x}) - \boldsymbol{\sigma}(-\boldsymbol{x})$.

One explanation is that skip connections bring ease to the training / optimization process. Suppose that we have a general nonlinear function $\mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)$. With a skip connection, we represent the map as $\boldsymbol{x}_{\ell+1} = \boldsymbol{x}_\ell + \mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)$ instead. Now the gradient $\partial \boldsymbol{x}_{\ell+1}/\partial \boldsymbol{x}_\ell$ becomes

$$\frac{\partial \boldsymbol{x}_{\ell+1}}{\partial \boldsymbol{x}_\ell} = \mathbf{I} + \frac{\partial \mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)}{\partial \boldsymbol{x}_\ell} \qquad \text{instead of} \qquad \frac{\partial \mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)}{\partial \boldsymbol{x}_\ell}, \tag{35}$$

where $\mathbf{I}$ is an identity matrix. By the chain rule, gradient update requires computing products of many components, e.g., $\frac{\partial \boldsymbol{x}_L}{\partial \boldsymbol{x}_1} = \prod_{\ell=1}^{L-1} \frac{\partial \boldsymbol{x}_{\ell+1}}{\partial \boldsymbol{x}_\ell}$, so it is desirable to keep the spectra (singular values) of each component $\frac{\partial \boldsymbol{x}_{\ell+1}}{\partial \boldsymbol{x}_\ell}$ close to 1. In neural nets, with skip connections, this is easily achieved if the parameters have small values; otherwise, this may not be achievable even with careful initialization and tuning. Notably, training neural nets with hundreds of layers is possible with the help of skip connections.

6.2.3 Batch normalization

Recall that in regression analysis, one often standardizes the design matrix so that the features have zero mean and unit variance. Batch normalization extends this standardization procedure from the input layer to all the hidden layers. Mathematically, fix a mini-batch of input data $\{(\boldsymbol{x}_i, y_i)\}_{i\in\mathcal{B}}$, where $\mathcal{B} \subset [n]$. Let $\boldsymbol{h}_i^{(\ell)}$ be the feature of the i-th example in the ℓ-th layer ($\ell = 0$ corresponds to the input $\boldsymbol{x}_i$). The batch normalization layer computes the normalized version of $\boldsymbol{h}_i^{(\ell)}$ via the following steps:

$$\boldsymbol{\mu} \triangleq \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \boldsymbol{h}_i^{(\ell)}, \qquad \boldsymbol{\sigma}^2 \triangleq \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \big(\boldsymbol{h}_i^{(\ell)} - \boldsymbol{\mu}\big)^2 \qquad \text{and} \qquad \boldsymbol{h}_{i,\text{norm}}^{(l)} \triangleq \frac{\boldsymbol{h}_i^{(\ell)} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}.$$

Here all the operations are element-wise. In words, batch normalization computes the z-score for each feature over the mini-batch $\mathcal{B}$ and use that as inputs to subsequent layers. To make it more versatile, a typical batch normalization layer has two additional learnable parameters $\boldsymbol{\gamma}^{(\ell)}$ and $\boldsymbol{\beta}^{(\ell)}$ such that

$$\boldsymbol{h}_{i,\text{new}}^{(l)} = \boldsymbol{\gamma}^{(l)} \odot \boldsymbol{h}_{i,\text{norm}}^{(l)} + \boldsymbol{\beta}^{(l)}.$$

Again $\odot$ denotes the element-wise multiplication. As can be seen, $\boldsymbol{\gamma}^{(\ell)}$ and $\boldsymbol{\beta}^{(\ell)}$ set the new feature $\boldsymbol{h}_{i\text{new}}^{(l)}$ to have mean $\boldsymbol{\beta}^{(\ell)}$ and standard deviation $\boldsymbol{\gamma}^{(\ell)}$. The introduction of batch normalization makes the training of neural networks much easier and smoother. More importantly, it allows the neural nets to perform well over a large family of hyper-parameters including the number of layers, the number of hidden units, etc. At test time, the batch normalization layer needs more care. For brevity we omit the details and refer to [**?**].

6.3 Regularization techniques

So far we have focused on training techniques to drive the empirical loss (26) small efficiently. Here we proceed to discuss common practice to improve the generalization power of trained neural nets.

6.3.1 Weight decay

One natural regularization idea is to add an ℓ_2 penalty to the loss function. This regularization technique is known as the weight decay in deep learning. We have seen one example in (9). For general deep neural nets, the loss to optimize is $\ell_n^\lambda(\boldsymbol{\theta}) = \ell_n(\boldsymbol{\theta}) + r_\lambda(\boldsymbol{\theta})$ where

$$r_\lambda(\boldsymbol{\theta}) = \lambda \sum_{\ell=1}^{L} \sum_{j,j'} \left[W_{j,j'}^{(\ell)}\right]^2.$$

Note that the bias (intercept) terms are not penalized. If $\ell_n(\boldsymbol{\theta})$ is a least square loss, then regularization with weight decay gives precisely ridge regression. The penalty $r_\lambda(\boldsymbol{\theta})$ is a smooth function and thus it can be also implemented efficiently with back-propagation.

6.3.2 Dropout

Dropout, introduced by [**?**], prevents overfitting by randomly dropping out subsets of features during training. Take the l-th layer of the feed-forward neural network as an example. Instead of propagating all the features in $\boldsymbol{h}^{(\ell)}$ for later computations, dropout randomly omits some of its entries by

$$\boldsymbol{h}_{\text{drop}}^{(\ell)} = \boldsymbol{h}^{(\ell)} \odot \mathsf{mask}^\ell,$$

where $\odot$ denotes element-wise multiplication as before, and mask^ℓ is a vector of Bernoulli variables with success probability p. It is sometimes useful to rescale the features $\boldsymbol{h}_{\text{inv drop}}^{(\ell)} = \boldsymbol{h}_{\text{drop}}^{(\ell)}/p$, which is called *inverted dropout*. During training, mask^ℓ are i.i.d. vectors across mini-batches and layers. However, when testing on fresh samples, dropout is disabled and the original features $\boldsymbol{h}^{(\ell)}$ are used to compute the output label y. It has been nicely shown by [**?**] that for generalized linear models, dropout serves as adaptive regularization. In the simplest case of linear regression, it is equivalent to ℓ_2 regularization. Another possible way to understand the regularization effect of dropout is through the lens of bagging [**?**]. Since different mini-batches has different masks, dropout can be viewed as training a large ensemble of classifiers at the same time, with a further constraint that the parameters are shared. Theoretical justification remains elusive.

6.3.3 Data augmentation

Data augmentation is a technique of enlarging the dataset when we have knowledge about invariance structure of data. It implicitly increases the sample size and usually regularizes the model effectively. For example, in image classification, we have strong prior knowledge about what invariance properties a good classifier should possess. The label of an image should not be affected by translation, rotation, flipping, and even crops of the image. Hence one can augment the dataset by randomly translating, rotating and cropping the images in the original dataset.

Formally, during training we want to minimize the loss $\ell_n(\boldsymbol{\theta}) = \sum_i \mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ w.r.t. parameters $\boldsymbol{\theta}$, and we know a priori that certain transformation $T \in \mathcal{T}$ where $T : \mathbb{R}^d \to \mathbb{R}^d$ (e.g., affine transformation) should not change the category / label of a training sample. In principle, if computation costs were not a consideration, we could convert this knowledge to a constraint $f_{\boldsymbol{\theta}}(T\boldsymbol{x}_i) = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \forall\, T \in \mathcal{T}$ in the minimization formulation. Instead of solving a constrained optimization problem, data augmentation enlarges the training dataset by sampling $T \in \mathcal{T}$ and generating new data $\{(T\boldsymbol{x}_i, y_i)\}$. In this sense, data augmentation induces invariance properties through sampling, which results in a much bigger dataset than the original one.

7 Generalization power

Section 6 has focused on the in-sample / training error obtained via SGD, but this alone does not guarantee good performance with respect to the out-of-sample / test error. The gap between the in-sample error and the out-of-sample error, namely the *generalization gap*, has been the focus of statistical learning theory since its birth; see [**?**] for an excellent introduction to this topic.

While understanding the generalization power of deep neural nets is difficult [**?**, **?**], we sample recent endeavors in this section. From a high level point of view, these approaches can be divided into two categories, namely *algorithm-independent controls* and *algorithm-dependent control*s. More specifically, algorithm-independent controls focus solely on bounding the *complexity* of the function class represented by certain deep neural networks. In contrast, algorithm-dependent controls take into account the algorithm (e.g., SGD) used to train the neural network.

7.1 Algorithm-independent controls: uniform convergence

The key to algorithm-independent controls is the notion of *complexity* of the function class parametrized by certain neural networks. Informally, as long as the complexity is not too large, the generalization gap of *any* function in the function class is well-controlled. However, the standard complexity measure (e.g., VC dimension [**?**]) is at least proportional to the number of weights in a neural network [**?**, **?**], which fails to explain the practical success of deep learning. The caveat here is that the function class under consideration is *all* the functions realized by certain neural networks, with *no* restrictions on the size of the weights at all. On the other hand, for the class of linear functions with bounded norm, i.e., $\{\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \,|\, \|\boldsymbol{w}\|_2 \le M\}$, it is well understood that the complexity of this function class (measured in terms of the empirical Rademacher complexity) with respect to a random sample $\{\boldsymbol{x}_i\}_{1\le i\le n}$ is upper bounded by $\max_i \|\boldsymbol{x}_i\|_2 M/\sqrt{n}$, which is independent of the number of parameters in $\boldsymbol{w}$. This motivates researchers to investigate the complexity of *norm-controlled* deep neural networks[10] [**?**, **?**, **?**, **?**]. Setting the stage, we introduce a few necessary notations and facts. The key object under study is the function class parametrized by the following fully-connected neural network with depth L:

$$\mathcal{F}_L \triangleq \left\{\boldsymbol{x} \mapsto \boldsymbol{W}_L \boldsymbol{\sigma}\left(\boldsymbol{W}_{L-1}\boldsymbol{\sigma}\left(\cdots \boldsymbol{W}_2 \boldsymbol{\sigma}\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right) \,\middle|\, (\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L) \in \mathcal{W}\right\}. \tag{36}$$

Here $(\boldsymbol{W}_1, \boldsymbol{W}_2, \cdots, \boldsymbol{W}_L) \in \mathcal{W}$ represents a certain constraint on the parameters. For instance, one can restrict the Frobenius norm of each parameter $\boldsymbol{W}_l$ through the constraint $\|\boldsymbol{W}_l\|_{\mathrm{F}} \le M_{\mathrm{F}}(l)$, where $M_{\mathrm{F}}(l)$ is some positive quantity. With regard to the complexity measure, it is standard to use *Rademacher complexity* to control the capacity of the function class of interest.

Definition 1 (Empirical Rademacher complexity)**.** *The empirical Rademacher complexity of a function class $\mathcal{F}$ w.r.t. a dataset $S \triangleq \{\boldsymbol{x}_i\}_{1\le i\le n}$ is defined as*

$$\mathcal{R}_S(\mathcal{F}) = \mathbb{E}_{\boldsymbol{\varepsilon}}\Big[\sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^n \varepsilon_i f(\boldsymbol{x}_i)\Big], \tag{37}$$

where $\boldsymbol{\varepsilon} \triangleq (\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n)$ is composed of i.i.d. Rademacher random variables, i.e., $\mathbb{P}(\varepsilon_i = 1) = \mathbb{P}(\varepsilon_i = -1) = 1/2$.

In words, Rademacher complexity measures the ability of the function class to fit the random noise represented by $\boldsymbol{\varepsilon}$. Intuitively, a function class with a larger Rademacher complexity is more prone to overfitting. We now formalize the connection between the empirical Rademacher complexity and the out-of-sample error; see Chapter 24 in [**?**].

Theorem 6. *Assume that for all $f \in \mathcal{F}$ and all $(y, \boldsymbol{x})$ we have $|\mathcal{L}(f(\boldsymbol{x}), y)| \le 1$. In addition, assume that for any fixed y, the univariate function $\mathcal{L}(\cdot, y)$ is Lipschitz with constant 1. Then with probability at least $1-\delta$ over the sample $S \triangleq \{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n} \overset{\text{i.i.d.}}{\sim} \mathcal{D}$, one has for all $f \in \mathcal{F}$*

$$\underbrace{\mathbb{E}_{(y,\boldsymbol{x})\sim\mathcal{D}}\left[\mathcal{L}\left(f(\boldsymbol{x}), y\right)\right]}_{\text{out-of-sample error}} \le \underbrace{\frac{1}{n}\sum_{i=1}^n \mathcal{L}\left(f(\boldsymbol{x}_i), y_i\right)}_{\text{in-sample error}} + 2\mathcal{R}_S(\mathcal{F}) + 4\sqrt{\frac{\log(4/\delta)}{n}}.$$

In English, the generalization gap of any function f that lies in $\mathcal{F}$ is well-controlled as long as the Rademacher complexity of $\mathcal{F}$ is not too large. With this connection in place, we single out the following complexity bound.

[10] Such attempts have been made in the seminal work [**?**].

Theorem 7 (Theorem 1 in [?]). *Consider the function class $\mathcal{F}_L$ in (36), where each parameter $\boldsymbol{W}_l$ has Frobenius norm at most $M_{\mathrm{F}}(l)$. Further suppose that the element-wise activation function $\sigma(\cdot)$ is 1-Lipschitz and positive-homogeneous (i.e., $\sigma(c \cdot x) = c\sigma(x)$ for all $c \geq 0$). Then the empirical Rademacher complexity (37) w.r.t. $S \triangleq \{\boldsymbol{x}_i\}_{1 \leq i \leq n}$ satisfies*

$$\mathcal{R}_S\left(\mathcal{F}_L\right) \leq \max_i \|\boldsymbol{x}_i\|_2 \cdot \frac{4\sqrt{L}\prod_{l=1}^{L} M_{\mathrm{F}}(l)}{\sqrt{n}}. \tag{38}$$

The upper bound of the empirical Rademacher complexity (38) is in a similar vein to that of linear functions with bounded norm, i.e., $\max_i \|\boldsymbol{x}_i\|_2 M/\sqrt{n}$, where $\sqrt{L}\prod_{l=1}^{L} M_{\mathrm{F}}(l)$ plays the role of M in the latter case. Moreover, ignoring the term $\sqrt{L}$, the upper bound (38) does not depend on the size of the network in an explicit way if $M_F(l)$ sharply concentrates around 1. This reveals that the capacity of the neural network is well-controlled, regardless of the number of parameters, as long as the Frobenius norm of the parameters is bounded. Extensions to other norm constraints, e.g., spectral norm constraints, path norm constraints have been considered by [?, ?, ?, ?, ?]. This line of work improves upon traditional capacity analysis of neural networks in the over-parametrized setting, because the upper bounds derived are often size-independent. Having said this, two important remarks are in order: (1) the upper bounds (e.g., $\prod_{l=1}^{L} M_{\mathrm{F}}(l)$) involve implicit dependence on the size of the weight matrix and the depth of the neural network, which is hard to characterize; (2) the upper bound on the Rademacher complexity offers a uniform bound over all functions in the function class, which is a pure statistical result. However, it stays silent about how and why standard training algorithms like SGD can obtain a function whose parameters have small norms.

7.2 Algorithm-dependent controls

In this subsection, we bring computational thinking into statistics and investigate the role of algorithms in the generalization power of deep learning. The consideration of algorithms is quite natural and well motivated: (1) local/global minima reached by different algorithms can exhibit totally different generalization behaviors due to extreme nonconvexity, which marks a huge difference from traditional models, (2) the *effective* capacity of neural nets is possibly not large, since a particular algorithm does not explore the entire parameter space.

These demonstrate the fact that on top of the complexity of the function class, the inherent property of the algorithm we use plays an important role in the generalization ability of deep learning. In what follows, we survey three different ways to obtain upper bounds on the generalization errors by exploiting properties of the algorithms.

7.2.1 Mean field view of neural nets

As we have emphasized, modern deep learning models are highly over-parametrized. A line of work [?, ?, ?, ?, ?, ?] approximates the ensemble of weights by an asymptotic limit as the number of hidden units tends to infinity, so that the dynamics of SGD can be studied via certain partial different equations.

More specifically, let $\hat{f}(\boldsymbol{x}; \boldsymbol{\theta}) = N^{-1}\sum_{i=1}^{N} \sigma(\boldsymbol{\theta}_i^\top \boldsymbol{x})$ be a function given by a one-hidden-layer neural net with N hidden units, where $\sigma(\cdot)$ is the ReLU activation function and parameters $\boldsymbol{\theta} \triangleq [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N]^\top \in \mathbb{R}^{N \times d}$ are suitably randomly initialized. Consider the regression setting where we want to minimize the population risk $R_N(\boldsymbol{\theta}) = \mathbb{E}[(y - \hat{f}(\boldsymbol{x}; \boldsymbol{\theta}))^2]$ over parameters $\boldsymbol{\theta}$. A key observation is that this population risk depends on the parameters $\boldsymbol{\theta}$ only through its empirical distribution, i.e., $\hat{\rho}^{(N)} = N^{-1}\sum_{i=1}^{N} \delta_{\boldsymbol{\theta}_i}$ where $\delta_{\boldsymbol{\theta}_i}$ is a point mass at $\boldsymbol{\theta}_i$. This motivates us to view express $R_N(\boldsymbol{\theta})$ equivalently as $R(\hat{\rho}^{(N)})$, where $R(\cdot)$ is a functional that maps distributions to real numbers. Running SGD for $R_N(\cdot)$—in a suitable scaling limit—results in a gradient flow on the space of distributions endowed with the Wasserstein metric that minimizes $R(\cdot)$. It turns out that the empirical distribution $\hat{\rho}_k^{(N)}$ of the parameters after k steps of SGD is well approximated by the gradient follow, as long as the the neural net is over-parametrized (i.e., $N \gg d$) and the number of steps is not too large. In particular, [?] have shown that under certain regularity conditions,

$$\sup_{k \in [0, T/\varepsilon] \cap \mathbb{N}} \left| R(\hat{\rho}^{(N)}) - R\left(\rho_{k\varepsilon}\right) \right| \lesssim e^T \sqrt{\frac{1}{N} \vee \varepsilon} \cdot \sqrt{d + \log \frac{N}{\varepsilon}},$$

where $\varepsilon > 0$ is an proxy for the step size of SGD and $\rho_{k\varepsilon}$ is the distribution of the gradient flow at time $k\varepsilon$. In words, the out-of-sample error under $\boldsymbol{\theta}^k$ generated by SGD is well-approximated by that of $\rho_{k\varepsilon}$. Viewing the optimization problem from the distributional aspect greatly simplifies the problem conceptually, as the complicated optimization problem is now passed into its limit version—for this reason, this analytical approach is called the mean field perspective. In particular, [**?**] further demonstrated that in some simple settings, the out-of-sample error $R(\rho_{k\varepsilon})$ of the distributional limit can be fully characterized. Nevertheless, how well does $R(\rho_{k\varepsilon})$ perform and how fast it converges remain largely open for general problems.

7.2.2 Stability

A second way to understand the generalization ability of deep learning is through the *stability* of SGD. An algorithm is considered stable if a slight change of the input does not alter the output much. It has long been observed that a stable algorithm has a small generalization gap; examples include k nearest neighbors [**?**, **?**], bagging [**?**, **?**], etc. The precise connection between stability and generalization gap is stated by [**?**, **?**]. In what follows, we formalize the idea of *stability* and its connection with the generalization gap. Let $\mathcal{A}$ denote an algorithm (possibly randomized) which takes a sample $S \triangleq \{(y_i, \boldsymbol{x}_i)\}_{1 \le i \le n}$ of size n and returns an estimated parameter $\hat{\boldsymbol{\theta}} \triangleq \mathcal{A}(S)$. Following [**?**], we have the following definition for *stability*.

Definition 2. *An algorithm (possibly randomized) $\mathcal{A}$ is ε-uniformly stable with respect to the loss function $\mathcal{L}(\cdot, \cdot)$ if for all datasets S, S' of size n which differ in at most one example, one has*

$$\sup_{\boldsymbol{x}, y} \mathbb{E}_{\mathcal{A}} \left[\mathcal{L}\left(f(\boldsymbol{x}; \mathcal{A}\left(S\right)), y\right) - \mathcal{L}\left(f(\boldsymbol{x}; \mathcal{A}\left(S'\right)), y\right) \right] \le \varepsilon.$$

Here the expectation is taken w.r.t. the randomness in the algorithm $\mathcal{A}$ and ε might depend on n. The loss function $\mathcal{L}(\cdot, \cdot)$ takes an example (say $(\boldsymbol{x}, y)$) and the estimated parameter (say $\mathcal{A}(S)$) as inputs and outputs a real value.

Surprisingly, an ε-uniformly stable algorithm incurs small generalization gap *in expectation*, which is stated in the following lemma.

Lemma 1 (Theorem 2.2 in [**?**])**.** *Let $\mathcal{A}$ be ε-uniformly stable. Then the expected generalization gap is no larger than ε, i.e.,*

$$\left| \mathbb{E}_{\mathcal{A}, S} \left[\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(\boldsymbol{x}_i; \mathcal{A}\left(S\right)), y_i) - \mathbb{E}_{(\boldsymbol{x}, y) \sim \mathcal{D}} \left[\mathcal{L}\left(f(\boldsymbol{x}; \mathcal{A}\left(S\right)), y\right) \right] \right] \right| \le \varepsilon. \tag{39}$$

With Lemma 1 in hand, it suffices to prove stability bound on specific algorithms. It turns out that SGD introduced in Section 6 is uniformly stable when solving smooth nonconvex functions.

Theorem 8 (Theorem 3.12 in [**?**])**.** *Assume that for any fixed $(y, \boldsymbol{x})$, the loss function $\mathcal{L}(f(x; \boldsymbol{\theta}), y)$, viewed as a function of $\boldsymbol{\theta}$, is L-Lipschitz and β-smooth. Consider running SGD on the empirical loss function with decaying step size $\alpha_t \le c/t$, where c is some small absolute constant. Then SGD is uniformly stable with*

$$\varepsilon \lesssim \frac{T^{1 - \frac{1}{\beta c + 1}}}{n},$$

where we have ignored the dependency on β, c and L.

Theorem 8 reveals that SGD operating on nonconvex loss functions is indeed uniformly stable as long as the number of steps T is not large compared with n. This together with Lemma 1 demonstrates the generalization ability of SGD in expectation. Nevertheless, two important limitations are worth mentioning. First, Lemma 1 provides an upper bound on the out-of-sample error *in expectation*, but ideally, instead of an on-average guarantee under $\mathbb{E}_{\mathcal{A}, S}$, we would like to have a high probability guarantee as in the convex case [**?**]. Second, controlling the generalization gap alone is not enough to achieve a small out-of-sample error, since it is unclear whether SGD can achieve a small training error within T steps.

7.2.3 Implicit regularization

In the presence of over-parametrization (number of parameters larger than the sample size), conventional wisdom informs us that we should apply some regularization techniques (e.g., ℓ_1 / ℓ_2 regularization) so that the model will not overfit the data. However, in practice, neural networks without explicit regularization generalize well. This phenomenon motivates researchers to look at the regularization effects introduced by training algorithms (e.g., SGD) in this over-parametrized regime. While there might exits multiple, if not infinite global minima of the empirical loss (26), it is possible that practical algorithms tend to converge to solutions with better generalization powers.

Take the underdetermined linear system $\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{y}$ as a starting point. Here $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ and $\boldsymbol{\theta} \in \mathbb{R}^p$ with p much larger than n. Running gradient descent on the loss $\frac{1}{2}\|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2$ from the origin (i.e., $\boldsymbol{\theta}^0 = \boldsymbol{0}$) results in the solution with the minimum Euclidean norm, that is GD converges to

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \quad \|\boldsymbol{\theta}\|_2 \qquad \text{subject to} \quad \boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{y}.$$

In words, without any ℓ_2 regularization in the loss function, gradient descent automatically finds the solution with the least ℓ_2 norm. This phenomenon, often called as *implicit regularization*, not only has been empirically observed in training neural networks, but also has been theoretically understood in some simplified cases, e.g., logistic regression with separable data. In logistic regression, given a training set $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ with $\boldsymbol{x}_i \in \mathbb{R}^p$ and $y_i \in \{1, -1\}$, one aims to fit a logistic regression model by solving the following program:

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \qquad \frac{1}{n}\sum_{i=1}^{n} \ell\big(y_i\boldsymbol{x}_i^\top\boldsymbol{\theta}^t\big). \tag{40}$$

Here, $\ell(u) \triangleq \log(1 + e^{-u})$ denotes the logistic loss. Further assume that the data is separable, i.e., there exists $\boldsymbol{\theta}^* \in \mathbb{R}^p$ such that $y_i\boldsymbol{\theta}^{*\top}\boldsymbol{x}_i > 0$ for all i. Under this condition, the loss function (40) can be arbitrarily close to zero for certain $\boldsymbol{\theta}$ with $\|\boldsymbol{\theta}\|_2 \to \infty$. What happens when we minimize (40) using gradient descent? [**?**] uncovers a striking phenomenon.

Theorem 9 (Theorem 3 in [**?**]). *Consider the logistic regression (40) with separable data. If we run GD*

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta\frac{1}{n}\sum_{i=1}^{n} y_i\boldsymbol{x}_i\ell'\big(y_i\boldsymbol{x}_i^\top\boldsymbol{\theta}^t\big)$$

from any initialization $\boldsymbol{\theta}^0$ with appropriate step size $\eta > 0$, then normalized $\boldsymbol{\theta}^t$ converges to a solution with the maximum ℓ_2 margin. That is,

$$\lim_{t\to\infty} \frac{\boldsymbol{\theta}^t}{\|\boldsymbol{\theta}^t\|_2} = \hat{\boldsymbol{\theta}}, \tag{41}$$

where $\hat{\boldsymbol{\theta}}$ is the solution to the hard margin support vector machine:

$$\hat{\boldsymbol{\theta}} \triangleq \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \|\boldsymbol{\theta}\|_2, \qquad \textit{subject to} \quad y_i\boldsymbol{x}_i^\top\boldsymbol{\theta} \ge 1 \quad \textit{for all } 1 \le i \le n. \tag{42}$$

The above theorem reveals that gradient descent, when solving logistic regression with separable data, implicitly regularizes the iterates towards the ℓ_2 max margin vector (cf. (41)), without any explicit regularization as in (42). Similar results have been obtained by [**?**]. In addition, [**?**] studied algorithms other than gradient descent and showed that coordinate descent produces a solution with the maximum ℓ_1 margin.

Moving beyond logistic regression, which can be viewed as a one-layer neural net, the theoretical understanding of implicit regularization in deeper neural networks is still limited; see [**?**] for an illustration in deep linear convolutional neural networks.

8 Discussion

Due to space limitations, we have omitted several important deep learning models; notable examples include deep reinforcement learning [**?**], deep probabilistic graphical models [**?**], variational autoencoders [**?**], transfer

learning [?], etc. Apart from the modeling aspect, interesting theories on generative adversarial networks [?, ?], recurrent neural networks [?], connections with kernel methods [?, ?] are also emerging. We have also omitted the inverse-problem view of deep learning where the data are assumed to be generated from a certain neural net and the goal is to recover the weights in the NN with as few examples as possible. Various algorithms (e.g., GD with spectral initialization) have been shown to recover the weights successfully in some simplified settings [?, ?, ?, ?, ?, ?].

In the end, we identify a few important directions for future research.

- *New characterization of data distributions.* The success of deep learning relies on its power of efficiently representing complex functions relevant to real data. Comparatively, classical methods often have optimal guarantee if a problem has a certain known structure, such as smoothness, sparsity, and low-rankness [?, ?, ?, ?], but they are insufficient for complex data such as images. How to characterize the high-dimensional real data that can free us from known barriers, such as the curse of dimensionality is an interesting open question?

- *Understanding various computational algorithms for deep learning.* As we have emphasized throughout this survey, computational algorithms (e.g., variants of SGD) play a vital role in the success of deep learning. They allow fast training of deep neural nets and probably contribute towards the good generalization behavior of deep learning in practice. Understanding these computational algorithms and devising better ones are crucial components in understanding deep learning.

- *Robustness.* It has been well documented that DNNs are sensitive to small adversarial perturbations that are indistinguishable to humans [?]. This raises serious safety issues once if deploy deep learning models in applications such as self-driving cars, healthcare, etc. It is therefore crucial to refine current training practice to enhance robustness in a principled way [?].

- *Low SNRs.* Arguably, for image data and audio data where the signal-to-noise ratio (SNR) is high, deep learning has achieved great success. In many other statistical problems, the SNR may be very low. For example, in financial applications, the firm characteristic and covariates may only explain a small part of the financial returns; in healthcare systems, the uncertainty of an illness may not be predicted well from a patient's medical history. How to adapt deep learning models to excel at such tasks is an interesting direction to pursue?

Acknowledgements

J. Fan is supported in part by the NSF grants DMS-1712591 and DMS-1662139, the NIH grant R01-GM072611 and the ONR grant N00014-19-1-2120. We thank Ruying Bao, Yuxin Chen, Chenxi Liu, Weijie Su, Qingcan Wang and Pengkun Yang for helpful comments and discussions.