Lecture 10. Deep Learning (Chapter 10)

Ping Yu

HKU Business School
The University of Hong Kong

Introduction

- The cornerstone of deep learning is the neural network (NN).
- Neural networks became popular in the late 1980s.
- Lots of successes, hype, and great conferences: NeurIPS[1] (Neural Information Processing Systems).[2]
- Then along came SVMs, RF and Boosting in the 1990s, and Neural Networks took a back seat.
- Re-emerged around 2010 as Deep Learning; by 2020s very dominant and successful, especially in image and video recognization, and speech and text modeling.
- Part of success due to vast improvements in computing power, larger training sets (due to digitization), and software: Tensorflow (by Google) and PyTorch (by FaceBook, now Meta).
- Much of the credit goes to three pioneers and their students: Yann LeCun, Geoffrey Hinton and Yoshua Bengio, who received the 2018 ACM (the Association for Computing Machinery) Turing Award for their work in Neural Networks. [figure here]
- We will concentrate on supervised deep learning (both regression and classification); for unsupervised deep learning, see the Appendix.

[1]NeurIPS rather than NIPS because of the latter's association with the word nipples.

[2]Reflecting its origins at Snowbird, Utah, in 1986, the conference was accompanied by workshops organized at a nearby ski resort up until 2013.

History of Deep Learning

Figure: Yann LeCun (1960-, Meta and NYU), Geoffrey Hinton (1947-, Google and UToronto) and Yoshua Bengio (1964-, UMontréal)

- An Introduction: LeCun, Bengio and Hinton, 2015, Deep Learning, *Nature*, 521: 436-444.

Single Layer Neural Networks

(Section 10.1)

Single Layer Neural Networks

- Different from other nonlinear models in Lectures 7 and 8, a NN uses a particular structure to estimate $f(X) = E[Y|X]$.
- A feed-forward neural network for modeling a quantitative Y using $p = 4$ predictors is shown in Figure 10.1.
- In this simple NN,

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right),$$

  where each of the p features in the input layer feeds into each of the K hidden units (or nodes) in the hidden layer.
  - linear$\longrightarrow$nonlinear$\longrightarrow$linear$\longrightarrow \cdots$.
- The K activations, $A^T = (A_1, \cdots, A_K)$, in the hidden layer,

$$A_k = h_k(X) = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right),$$

  where $g(z)$ is a nonlinear activation function.

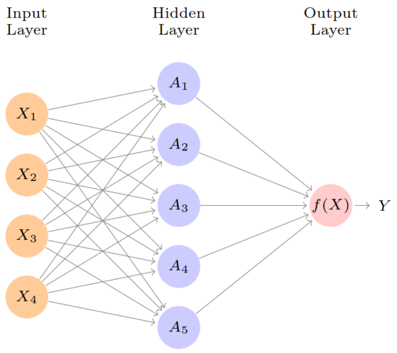


FIGURE 10.1. *Neural network with a single hidden layer. The hidden layer computes activations $A_k = h_k(X)$ that are nonlinear transformations of linear combinations of the inputs $X_1, X_2, \ldots, X_p$. Hence these A_k are not directly observed. The functions $h_k(\cdot)$ are not fixed in advance, but are learned during the training of the network. The output layer is a linear model that uses these activations A_k as inputs, resulting in a function $f(X)$.*

Activation Functions

- A_k's are like the basis functions (i.e., some transformations of X, or derived features) in Lecture 7, but they are not predetermined and learned during the training of the NN, much like the trees in Lecture 8.
- The unknown parameters include $\beta_0, \beta_1, \cdots, \beta_K$ and $w_{10}, \cdots, w_{Kp}$, totally

$$(p+1)K + K + 1$$

  parameters.
- The activation $g(z)$ is known beforehand.
- The sigmoid function is popular in early years of NN [see Figure 10.2]:

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}},$$

  which is the same function used in logistic regression to convert a linear function into probabilities between zero and one.
  - (**) Another smooth activation function is the hyperbolic tangent,

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

  which ranges from -1 to 1 and looks similar as sigmoid.

Why Nonlinear $g(\cdot)$?

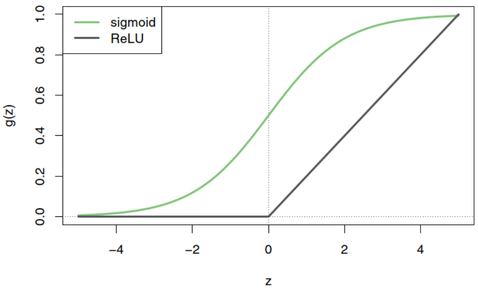


FIGURE 10.2. *Activation functions. The piecewise-linear* ReLU *function is popular for its efficiency and computability. We have scaled it down by a factor of five for ease of comparison.*

- If $g(\cdot)$ is linear, then $f(\cdot)$ is linear in X.
- Nonlinear $g(\cdot)$ can model interaction effects (which linear models cannot): e.g., when $p = 2$ and $K = 2$, set $g(z) = z^2$, and then for an appropriate choice of parameters,

$$\left[\frac{1}{4}(X_1 + X_2)^2 - \frac{1}{4}(X_1 - X_2)^2\right] = X_1 \cdot X_2.$$

The ReLU Activation

- In the 1990s, much efforts are spent on choosing among different g's, but the consensus nowadays is that if you have enough nodes and layers, the specific g does not matter as long as it is nonlinear, so a simple and computationally convenient g like ReLU below can be used.
- The ReLU (rectified linear unit) function is favored in modern era [see Figure 10.2]:

$$g(z) = (z)_+ := \max[0, z],^3$$

  - The ReLU activation can be computed and stored more efficiently than a sigmoid activation because $g(z)$ and $g'(z)$, which are needed in fitting an NN and predicting for a test point, take simple forms.[4]
  - Thresholding at 0 does not matter since the constant term w_{k0} will shift this inflection point.

[3] (*) This should be reminiscent of the hinge loss in SVM.

[4] For ReLU, $g'(z) = 1(z \geq 0)$, where the nondifferentiability at the origin is often ignored.

Where is the name "Neural Network" from?

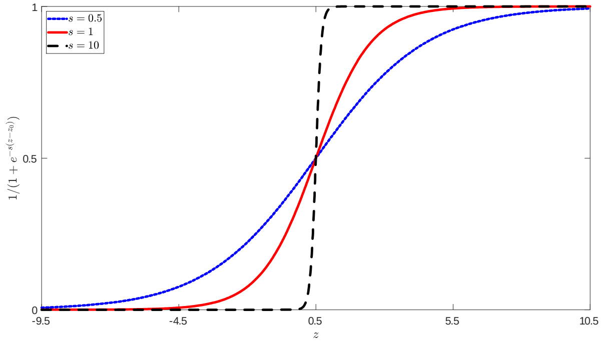


Figure: The Sigmoid Function with $z_0 = 0.5$, and $s = 0.5, 1$, and 10

- Originally, $g(\cdot)$ is a step function (sigmoid $g(s(z - z_0))$ as $s \to \infty$), used to model the neurons (nodes in Figure 10.1) to get fired when the total signal passed to that unit through synapses (links in Figure 10.1) exceeded a certain threshold z_0.

Loss Functions

- If Y is quantitative, then minimize
$$\sum_{i=1}^{n} (y_i - f(x_i))^2,$$
where $f(X) = \beta_0 + \sum_{\ell=1}^{K} \beta_\ell A_\ell$.
- Predict y at the test point x as $\hat{f}(x)$.
- If Y is qualitative, taking M values, then generate M dummy variables, $Y_m = I(Y = m)$, $m = 1, \cdots, M$, and estimate M class probabilities $f_m(X) = \Pr(Y = m|X) = E[Y_m|X]$.
- Note that one and only one Y_m equals 1, so $\{Y_m\}_{m=1}^{M}$ is known as one-hot encoding.
- Note also that $Y_1, \cdots, Y_M$ are quite dependent.
- This is a multi-task learning where a single NN can be used to predict different responses simultaneously.
- Use the softmax function
$$f_m(X) = \frac{e^{Z_m}}{\sum_{\ell=1}^{M} e^{Z_\ell}}$$
where $Z_m = \beta_{m0} + \sum_{\ell=1}^{K} \beta_{m\ell} A_\ell$, $f_m(X) \in [0,1]$, and $\sum_{m=1}^{M} f_m(X) = 1$.

Continued

- When $M = 2$,

$$f_1(X) = \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2}} = \frac{e^{(\beta_{10}-\beta_{20})+\sum_{\ell=1}^{K}(\beta_{1\ell}-\beta_{2\ell})A_\ell}}{1 + e^{(\beta_{10}-\beta_{20})+\sum_{\ell=1}^{K}(\beta_{1\ell}-\beta_{2\ell})A_\ell}} = \frac{1}{1 + e^{-(\beta_{10}-\beta_{20})-\sum_{\ell=1}^{K}(\beta_{1\ell}-\beta_{2\ell})A_\ell}},$$

  and $f_2(X) = 1 - f_1(X)$, where $Y = 2$ is selected as the baseline, so the softmax function is over-parametrized, but regularization and the SGD discussed below will constrain the solutions so that this is not a problem.
  - The softmax function treats all M classes symmetrically without selecting a baseline class.

- Then minimize the cross-entropy [see Appendix A of Lecture 8], i.e., the negative multinomial log-likelihood,

$$-\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im}\log(f_m(x_i)) = -\sum_{m=1}^{M}\sum_{i:y_{im}=1}\log(f_m(x_i)),$$

  a generalization of the log-likelihood for the two-class logistic regression.
  - Predict y at the test point x as $\arg\max_m \hat{f}_m(x)$.

Why a Hidden Layer in Classification?

- The hidden layer can be seen as distorting the input in a non-linear way so that categories become linearly separable by the last layer, just like in SVM.

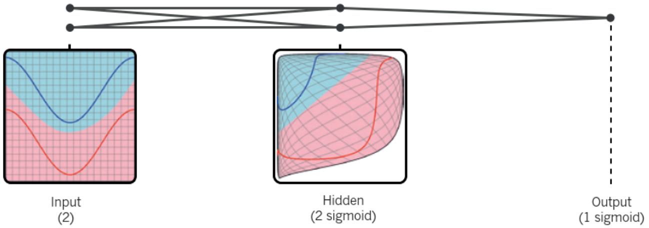


Figure: A single layer neural network (shown by the connected dots) can distort the input space to make the classes of data (examples of which are on the red and blue lines) linearly separable. Note how a regular grid (shown on the left) in input space is also transformed (shown in the middle panel) by hidden units.

Multilayer Neural Networks

(Section 10.2)

Multilayer Neural Networks: An Example

- Although a single hidden layer with many hidden units (i.e., a wide NN) can approximate any continuous function (i.e., is a universal approximator), it is much easier to find a good solution with multiple layers each of modest size, so-called deep neural nets (DNN).
- The famous and publicly available MNIST (Modified National Institute of Standards and Technology) handwritten digit dataset. [see Figure 10.3 for some examples]
- $X \in \mathbb{R}^{28\times 28=784}$ with $X_j \in \{0,1,\cdots,255\}$; note the structural arrangement of X.
- $Y = (Y_0, Y_1, \cdots, Y_9)$ of 10 dummy variables for $\{0,1,\cdots,9\}$ (i.e., $M = 10$).
- 60K train, 10K test images.
- Goal: build a classifier to predict the image class.
- We build a two-layer network with $K_1 = 256$ units at first layer, $K_2 = 128$ units at second layer, and 10 units at output layer. [see Figure 10.4]
  - The number of layers is greater than one, which is the name "deep" learning from.
- Along with intercepts (called biases) there are $235,146$ parameters (referred to as weights[5]).
  - $(p+1)\times K_1 + (K_1+1)\times K_2 + (K_2+1)\times M = 235,146$.

[5]Often, only the slopes are called weights. They can be seen as "knobs" that define the input-output function of the machine.

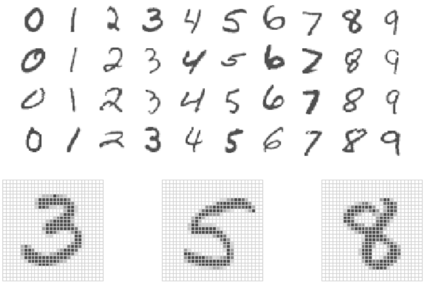


FIGURE 10.3. *Examples of handwritten digits from the* MNIST *corpus. Each grayscale image has* 28×28 *pixels, each of which is an eight-bit number (0–255) which represents how dark that pixel is. The first 3, 5, and 8 are enlarged to show their 784 individual pixel values.*

- Historically, digit recognition problems were the catalyst for the development of NN.

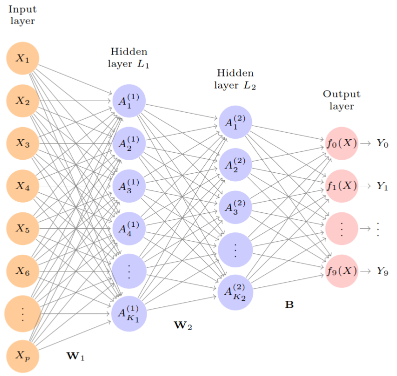
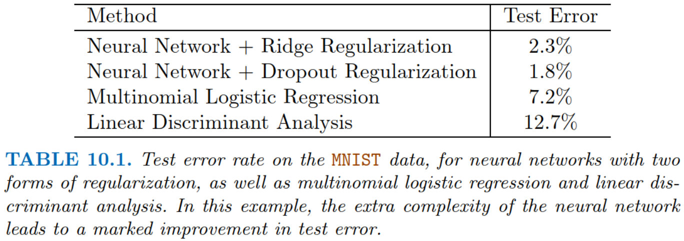


FIGURE 10.4. *Neural network diagram with two hidden layers and multiple outputs, suitable for the* MNIST *handwritten-digit problem.*

- $A_k^{(1)} = h_k^{(1)}(X) = g(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j)$, $k = 1, \cdots, K_1$, and $A_\ell^{(2)} = h_\ell^{(2)}(X) = g(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)})$, $\ell = 1, \cdots, K_2$.
- $Z_m = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} h_\ell^{(2)}(X) = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}$.
- $\mathbf{W}_1$ collects $w_{kj}^{(1)}$, $j = 0, 1, \cdots, p$, $k = 1, \cdots, K_1$, $\mathbf{W}_2$ collects $w_{\ell k}^{(2)}$, $k = 0, 1, \cdots, K_1$, $\ell = 1, \cdots, K_2$, and $\mathbf{B}$ collects $\beta_{m\ell}$, $\ell = 0, 1, \cdots, K_2$, $m = 0, 1, \cdots, 9$.



Method	Test Error
Neural Network + Ridge Regularization	2.3%
Neural Network + Dropout Regularization	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

TABLE 10.1. *Test error rate on the* MNIST *data, for neural networks with two forms of regularization, as well as multinomial logistic regression and linear discriminant analysis. In this example, the extra complexity of the neural network leads to a marked improvement in test error.*

- The number of weights is $235,146 \gg 60,000 = n$, so use regularization to avoid overfitting.
- The number of weights is also$\gg 785 \times 9 = 7,065$, the number of parameters in multinomial logistic regression.
- Early success for neural networks in the 1990s.
- Very overworked problem – best reported rates are $< 0.5\%$!
- Human error rate is reported to be around 0.2%, or 20 of the 10K test images.

Convolutional Neural Networks

(Section 10.3)

Convolutional Neural Networks: An Example

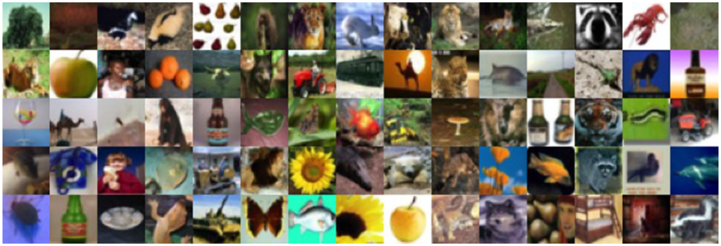


FIGURE 10.5. *A sample of images from the* CIFAR100 *database: a collection of natural images from everyday life, with 100 different classes represented.*

- CIFAR100 (Canadian Institute for Advanced Research) database includes 20 superclasses (e.g., aquatic mammals) with 5 classes per superclass (beaver, dolphin, otter, seal, whale), so $M = 100$.
- $p = 32 \times 32 \times 3$ (red, green, blue) array of 8-bit numbers, so a three-dimensional array called a feature map.
  - The first two axes are spatial, and the third is the channel axis.
- $n_{\text{train}} = 50\text{K}$, and $n_{\text{test}} = 10\text{K}$.
- Major success story for classifying images.

Selectivity-Invariance Dilemma

- A good classifier requires a good feature extractor that produces representations that are selective to the aspects of the image that are important for discrimination (i.e., even tiny differences in these aspects are critical for classification, e.g., the faces of two look-alike people), but that are invariant to irrelevant aspects (e.g., the pose of animal or the accent of speech).
- The conventional option is to hand design good feature extractors, which requires a considerable amount of engineering skill and domain expertise.
- CNN avoids this difficulty by learning good features automatically.

How CNNs Work?

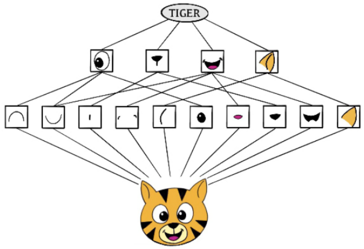


FIGURE 10.6. *Schematic showing how a convolutional neural network classifies an image of a tiger. The network takes in the image and identifies local features. It then combines the local features in order to create compound features, which in this example include eyes and ears. These compound features are used to output the label "tiger".*

- The CNN builds up an image in a hierarchical fashion, mimicing how humans classify images.
- Edges and shapes are recognized and pieced together to form more complex shapes, eventually assembling the target image.
- This hierarchical construction is achieved using convolution and pooling layers.

Convolution Layers

- Input Image$=\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$, and Convolution Filter$=\begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$.
- Convolved Image$=\begin{bmatrix} a\alpha+b\beta+d\gamma+e\delta & b\alpha+c\beta+e\gamma+f\delta \\ d\alpha+e\beta+g\gamma+h\delta & e\alpha+f\beta+h\gamma+i\delta \\ g\alpha+h\beta+j\gamma+k\delta & h\alpha+i\beta+k\gamma+l\delta \end{bmatrix}$.
- The filter is itself an image, and represents a small shape, edge, etc.
- We slide it around the input image, scoring for matches.
- The scoring is done via dot-products (i.e., inner product), illustrated above.
- If the subimage of the input image is similar to the filter, the score is high, otherwise low, (which is a property of the dot-product), i.e., the convolved image highlights regions of the original image that resemble the convolution filter. [see Figure 10.7]
- The filters are learned during training; in the usual image processing, the filters are predefined rather than learned.
- Return to the single-layer NN framework with one hidden unit for each pixel in the convolved image: the parameters are the convolution filter, so operate on localized patches (i.e., many zeros), and the same weights in a given filter are reused for all patches (i.e., highly constrained), so-called weight sharing.

(**) Why the name "Convolution"?

- Recall that the convolution of two functions f and g on $\mathbb{R}$ is

$$(f*g)(t)=\int_{-\infty}^{\infty}f(\tau)g(t-\tau)d\tau,$$

  i.e., $(f*g)(t)$ is the area under the function $f(\tau)$ weighted by the function $g(-\tau)$ shifted by the amount t (i.e., shift invariance).
  - As t changes, the weighting function $g(t-\tau)$ emphasizes different parts of the input function $f(\tau)$.
  - When f and g are supported on $[0,\infty]$, then $(f*g)(t)=\int_0^t f(\tau)g(t-\tau)d\tau$.

  convolution $\longrightarrow$ visual explanation

- Convolution filter conducts a discrete convolution on a rectangle area of $\mathbb{Z}^2$.

Convolution Example

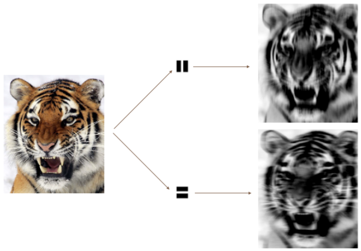


FIGURE 10.7. *Convolution filters find local features in an image, such as edges and small shapes. We begin with the image of the tiger shown on the left, and apply the two small convolution filters in the middle. The convolved images highlight areas in the original image where details similar to the filters are found. Specifically, the top convolved image highlights the tiger's vertical stripes, whereas the bottom convolved image highlights the tiger's horizontal stripes. We can think of the original image as the input layer in a convolutional neural network, and the convolved images as the units in the first hidden layer.*

- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.

More Details

- The two dimensions of the filter, say (ℓ_1, ℓ_2), are usually small positive integers but need not be the same.
  - For CIFAR100, we use $\ell_1 = \ell_2 = 3$.
- Note that for an $L \times L$ image, if convolved by a $\ell \times \ell$ filter, then we get an $(L - \ell + 1) \times (L - \ell + 1)$ image. To get an image with the same dimensions as the original image, we need padding.
- If we pad each side p pixels, then solving $L + 2\mathrm{p} - \ell + 1 = L$, we have $\mathrm{p} = \frac{\ell - 1}{2}$. This is why ℓ is usually odd.
  - We can pad either a zero or the closest pixel.
- For each (color) channel, use a different filter (or the whole filter is $\ell \times \ell \times 3$); the three convolutions are summed to form a two-dimensional image; the color information is used separately only at this point.
- If K different filters are used, we get a single $L \times L \times K$ feature map (from a $L \times L \times 3$ feature map), i.e., there are K channels now.
- Return to the single-layer NN framework: this $L \times L \times K$ feature map is like the activations in a hidden layer except organized and produced in a spatially structured way.
- Typically, a bias term is added to each channel, and apply ReLU to generate a separate layer afterwards, called a detector layer where a positive value indicates presence of a feature.

Pooling Layers

- Max pool: $\begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$.
- Each non-overlapping 2×2 block is replaced by its maximum.
- This sharpens the feature identification by condensing a large image into a smaller summary image.
- Allows for locational invariance: a small neighborhood is likely to contain similar information, so only the largest value in the 2×2 block is maintained in the reduced image.
- Reduces the dimension by a factor of 4 – i.e. factor of 2 in each dimension.

Architecture of a CNN

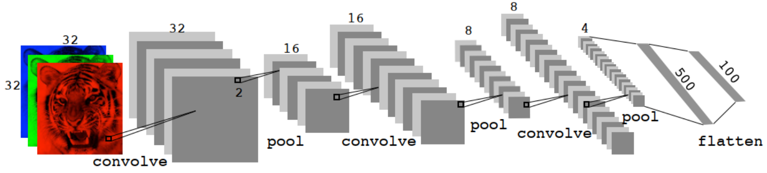


FIGURE 10.8. *Architecture of a deep CNN for the* CIFAR100 *classification task. Convolution layers are interspersed with* 2×2 *max-pool layers, which reduce the size by a factor of 2 in both dimensions.*

- Many convolve + pool layers.

More Details

- Filters are typically small, e.g., each channel 3×3.
- Each filter creates a new channel in convolution layer.
  - The second convolve layer is like the first, but one $\ell \times \ell$ filter applies to all K channels (because no color information now) and then adds up.
- As pooling reduces size, the number of filters/channels in convolution is typically increased ($3 \to 6 \to 12 \to 24$ in Figure 10.8).
- Sometimes repeat several convolve layers before a pool layer to increase the dimension of the filter (i.e., operate on more pixels).
- When the pooling reduces the dimension of each channel to a few (4 in the figure) pixels, then flattening: all pixels in all channels ($4 \times 4 \times 24$ in the figure) are fed into one or more (2 in the figure) fully-connected (i.e., not locally-connected) layers before reaching the output layer.
- Number of layers can be very large, e.g., the resnet50 (see the next slide) classifier trained on imagenet 1000-class image data base has 50 layers (48 convolutional layers, one MaxPool layer, and one average pool layer before softmax)!

(**) Deep Residual Networks

- When the network is very deep, there is a phenomenon called exploding/vanishing gradients because the gradient of the first hidden layer is the product of the gradients in the remaining layers in the backpropagation algorithm.
- To avoid this problem, residual networks add an identity map to an existing fragment, say $\mathbf{F}(\mathbf{x})$, of the network.
- In the figure below, for every two layers, an identity map is added:

$$\mathbf{x} \mapsto \mathbf{g}(\mathbf{x} + \mathbf{F}(\mathbf{x})) = \mathbf{g}\left(\mathbf{x} + \mathbf{W}'\mathbf{g}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{b}'\right),$$

  where $\mathbf{x}$ can be hidden nodes from any layer, $\mathbf{W}$ and $\mathbf{W}'$ are the weights, and $\mathbf{b}$ and $\mathbf{b}'$ are the biases.
- The name comes from the fact that if $-\mathbf{F}(\mathbf{x})$ is a fit of $\mathbf{x}$, then $\mathbf{x} + \mathbf{F}(\mathbf{x})$ is the residual of the fitting.
- The motivation can be explained from the angle of numerical stability: if the magnitude of $\mathbf{F}(\mathbf{x})$ is small, then the spectra (singular values) of Jacobian of $\mathbf{x} + \mathbf{F}(\mathbf{x})$ is close to 1, so we expect better numerical stability.
- By repeating this structure throughout all layers, we are able to train neural nets with hundreds of layers easily, which overcome well-observed training difficulties in deep neural nets.

(**) Skip Connections

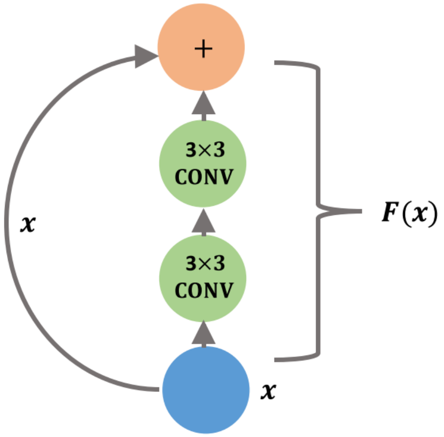
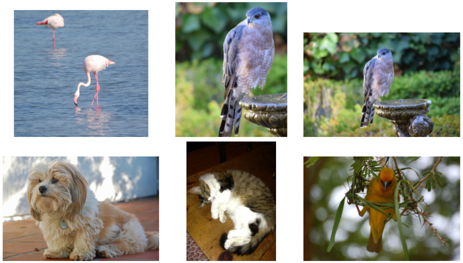


Figure: Skip Connections: after skipping two layers, reconnect to **x**

Using Pretrained Networks to Classify Images



flamingo		Cooper's hawk		Cooper's hawk	
flamingo	0.83	kite	0.60	fountain	0.35
spoonbill	0.17	great grey owl	0.09	nail	0.12
white stork	0.00	robin	0.06	hook	0.07
Lhasa Apso		**cat**		**Cape weaver**	
Tibetan terrier	0.56	Old English sheepdog	0.82	jacamar	0.28
Lhasa	0.32	Shih-Tzu	0.04	macaw	0.12
cocker spaniel	0.03	Persian cat	0.04	robin	0.12

FIGURE 10.10. *Classification of six photographs using the* `resnet50` *CNN trained on the* `imagenet` *corpus. The table below the images displays the true (intended) label at the top of each panel, and the top three choices of the classifier (out of 100). The numbers are the estimated probabilities for each choice. (A kite is a raptor, but not a hawk.)*

- Weight freezing: use the weights from resnet50 (and train just the last few layers).

Document Classification

(Section 10.4)

Document Classification: IMDB Movie Reviews

- The IMDB corpus consists of user-supplied movie ratings for a large collection of movies. Each has been labeled for sentiment as positive or negative. Here is the beginning of a negative review:
  This has to be one of the worst films of the 1990s. When my friends & I were watching this film (being the target audience it was aimed at) we just sat & watched the first half an hour with our jaws touching the floor at how bad it really was. The rest of the time, everyone else in the theater just started talking to each other, leaving or generally crying into their popcorn $\cdots$
- We have labeled training and test sets, each consisting of 25K reviews, and each balanced with regard to sentiment.
- We wish to build a classifier to predict the sentiment of a review.

Featurization: Bag-of-Words

- Documents have different lengths, and consist of sequences of words. How do we create features X to characterize a document? [see Lecture 6 for more details]
  - From a dictionary, identify the 10K most frequently occurring words.
  - Create a binary vector of length $p = 10\text{K}$ for each document, and score a 1 in every position that the corresponding word occurred (counts or proportions may be more informative).
  - With n documents, we now have a $n \times p$ sparse feature matrix $\mathbf{X}$ (only 1.3% nonzeros in this example).
- We compare a lasso logistic regression model to a two-hidden-layer NN (actually a nonlinear logistic regression) on the next slide. (No convolutions here!)
  - glmnet was used to fit the lasso model, and is very effective because it can exploit sparsity in the $\mathbf{X}$ matrix.
  - For NN, $K_1 = K_2 = 16$ ReLU units appear in the two hidden layers.
  - 2K validation data from 25K training data are used to choose tuning parameters.
  - Simpler lasso logistic regression model works as well as NN in this case, about 88% accuracy (i.e., $1-$classification error).
- Bag-of-words are unigrams. We can instead use bigrams (occurrences of adjacent word pairs), and in general *m*-grams to take into account of context.

Lasso versus Neural Network – IMDB Reviews

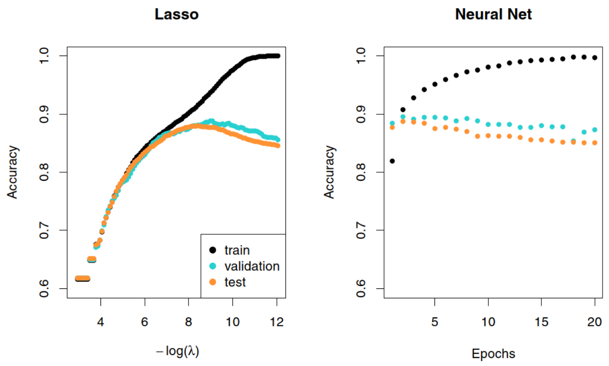


FIGURE 10.11. *Accuracy of the lasso and a two-hidden-layer neural network on the* IMDb *data. For the lasso, the x-axis displays* $-\log(\lambda)$*, while for the neural network it displays epochs (number of times the fitting algorithm passes through the training set). Both show a tendency to overfit, and achieve approximately the same test accuracy.*

Recurrent Neural Networks

(Section 10.5)

Recurrent Neural Networks

- Often data arise as sequences:
  - Documents are sequences of words such as the IMDB example in the last section, and their relative positions have meaning.
  - Time-series such as weather data or financial indices.
  - Recorded speech (e.g., give a text transcription or a language translation) or music (e.g., assess its quality).
  - Handwriting, such as doctor's notes.
- RNNs build models that take into account this sequential nature of the data, and build a memory of the past.
- The feature for each observation is a sequence of vectors, e.g., in document classification, $X = \{X_1, \cdots, X_L\}$ with X_ℓ representing a one-hot encoding for the ℓth word.
- The target Y is often of the usual kind – e.g., a single variable such as sentiment, or a one-hot vector for multiclass.
- However, Y can also be a sequence, such as the same document in a different language.

Architecture of a Simple RNN

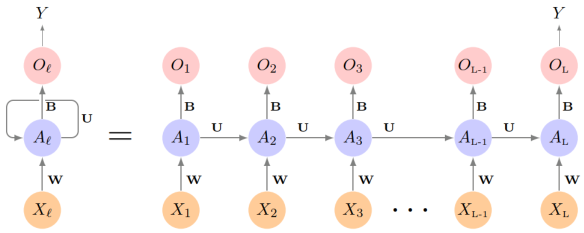


FIGURE 10.12. *Schematic of a simple recurrent neural network. The input is a sequence of vectors* $\{X_\ell\}_1^L$*, and here the target is a single response. The network processes the input sequence* X *sequentially; each* X_ℓ *feeds into the hidden layer, which also has as input the activation vector* $A_{\ell-1}$ *from the previous element in the sequence, and produces the current activation vector* A_ℓ*. The same collections of weights* $\mathbf{W}$, $\mathbf{U}$ *and* $\mathbf{B}$ *are used as each element of the sequence is processed. The output layer produces a sequence of predictions* O_ℓ *from the current activation* A_ℓ*, but typically only the last of these,* O_L*, is of relevance. To the left of the equal sign is a concise representation of the network, which is* unrolled *into a more explicit version on the right.*

- An RNN can be thought of as multiple copies of the same network, each passing a message to a successor.

More Details

- The hidden layer is a sequence of vectors $\{A_\ell\}_{\ell=1}^{L}$, receiving as input X_ℓ as well as $A_{\ell-1}$. A_ℓ produces an output O_ℓ.
- Suppose $X_\ell^T = (X_{\ell 1}, \cdots, X_{\ell p})$, and $A_\ell^T = (A_{\ell 1}, \cdots, A_{\ell K})$.
- $A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right)$: $(p+1) \times K + K \times K$ parameters
- $O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}$: $(K+1)$ parameters
- The same weights **W**, **U** and **B** are used at each step in the sequence – hence the term recurrent.
  - This is a form of weight sharing, like the filtering in CNN.
  - (*) Weight sharing is related to the notion of translational invariance in CNN and stationarity in RNN.
- The A_ℓ sequence represents an evolving model for the response that is updated as each element X_ℓ is processed.
  - As we proceed from beginning to end, the activations A_ℓ accumulate a history of what has been seen before, so that the learned context can be used for prediction.
- Often we are concerned only with the prediction O_L at the last unit.
  - Define $a_{iLk} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{iLj} + \sum_{s=1}^{K} u_{ks} a_{i,L-1,s}\right)$.

Continued

- For the squared error loss, and n sequence/response pairs, we would minimize

$$\sum_{i=1}^{n}(y_i - o_{iL})^2 = \sum_{i=1}^{n}\left(y_i - \left(\beta_0 + \sum_{k=1}^{K}\beta_k a_{iLk}\right)\right)^2.$$

- For the cross-entropy loss, we would minimize

$$-\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im}\log(f_{iLm}) = -\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im}\log\left(\frac{e^{o_{iLm}}}{\sum_{l=1}^{M} e^{o_{iLl}}}\right),$$

  where $y_i^T = (y_{i1}, \cdots, y_{iM})$, and $o_{iL}^T = (o_{iL1}, \cdots, o_{iLM})$ with $o_{iLm} = \beta_{m0} + \sum_{k=1}^{K}\beta_{mk}a_{iLk}$.
  - **B** contains $M \times (K+1)$ parameters now.

- Return to the single-layer NN framework: $X \in \mathbb{R}^{p\times L}$, and $A \in \mathbb{R}^{K\times L}$, but A is generated sequentially.
- O_ℓ is reported because it comes for free, and in some learning tasks (e.g., text transcription), the response is also a sequence, so the output sequence $\{O_1, \cdots, O_L\}$ is explicitly needed.

[Example] Document Classification

- In the bag-of-word model, $X \in \mathbb{R}^p$ with $p = 10\text{K}$ and L ones appearing in X, but in RNN, $X \in \mathbb{R}^{p \times L}$ with only one 1 in each column.
  - If the document has more than L words, use the last L words, and if less than L words, pad with zeros at the beginning to have L words.
- This results in an extremely sparse feature representation, and would not work well.
- Instead we use a lower-dimensional pretrained[6] word embedding matrix $\mathbf{E} \in \mathbb{R}^{m \times 10\text{K}}$. [see Figure 10.13].
  - Embeddings are pretrained on very large corpora of documents, using methods similar to PCA in Lecture 5 on the so-called word co-occurrence matrix.
  - The idea is that the positions of words in the embedding space preserve semantic meaning; e.g., synonyms should appear near each other. [see Lecture 6]
  - **E** can be learned (as part of the optimization) by adding an embedding layer before feeding in X_ℓ.
- This reduces the binary feature vector of length 10K to a real feature vector of dimension $m \ll 10\text{K}$ (e.g., m in the low hundreds), typically with nonzero elements.

[6] This is the weight freezing in CNN. (GPT)

Word Embedding

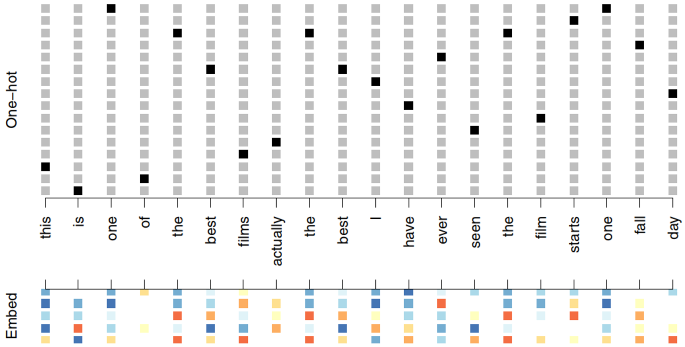


FIGURE 10.13. *Depiction of a sequence of 20 words representing a single document: one-hot encoded using a dictionary of 16 words (top panel) and embedded in an m-dimensional space with $m = 5$ (bottom panel).*

RNN on IMDB Reviews

- For the IMDB data, use RNN with **E** learned ($m = 32$ and $K = 32$) and only achieve 76% accuracy.
- We then fit a more exotic RNN than the one displayed – a LSTM with long and short term memory.
  - Here, A_ℓ receives input from $A_{\ell-1}$ (short term memory) as well as from a version that reaches further back in time (long term memory) to avoid early signals being washed out.
  - Compare "the clouds are in the *sky*" and “I grew up in France... I speak fluent *French*.”.
  - For a tutorial on LSTM, see

  LSTM

  - (**) The sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. A recent model architecture, called the Transformer, eschews recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output, which allows for significantly more parallelization. (GPT)
- Now we get 87% accuracy, slightly less than the 88% achieved by glmnet.
- These data have been used as a benchmark for new RNN architectures:

IMDB leaderboard

[Example] Time Series Forecasting

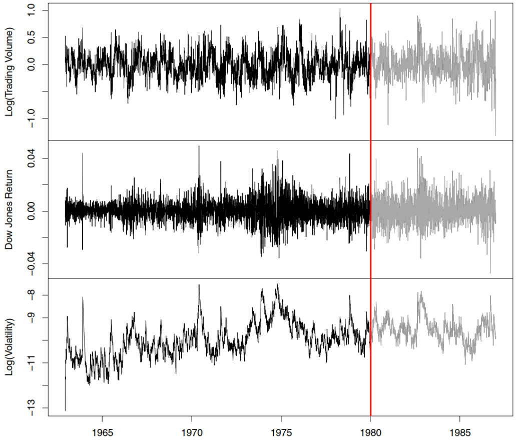


FIGURE 10.14. *Historical trading statistics from the New York Stock Exchange. Daily values of the normalized log trading volume, DJIA return, and log volatility are shown for a 24-year period from 1962–1986. We wish to predict trading volume on any day, given the history on all earlier days. To the left of the red bar (January 2, 1980) is training data, and to the right test data.*

NYSE Data: Dec. 3, 1962 – Dec. 31, 1986

- Log trading volume (v_t): This is the fraction of all outstanding shares that are traded on that day, relative to a 100-day moving average of past turnover, on the log scale.
- Dow Jones returns (r_t): This is the difference between the log of the DJIA on consecutive trading days.
- Log volatility (z_t): This is based on the absolute values of daily price movements.
- Goal: predict Log trading volume tomorrow, given its observed values up to today, as well as those of Dow Jones returns and Log volatility.
  - Predicting stock prices is extremely hard, but predicting trading volume is much easier and useful for planning trading strategies.
- $n_{\text{train}} = 4,281$, and $n_{\text{test}} = 1,770$.

Autocorrelation

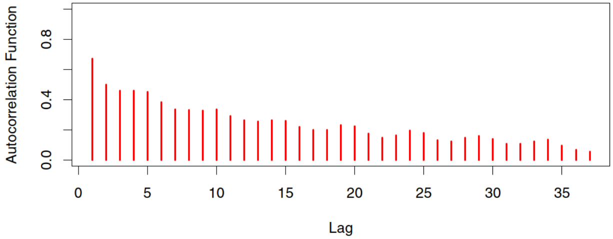


FIGURE 10.15. *The autocorrelation function for* `log_volume`*. We see that nearby values are fairly strongly correlated, with correlations above* 0.2 *as far as 20 days apart.*

- The autocorrelation at lag ℓ is the correlation of all pairs $(v_t, v_{t-\ell})$ that are ℓ trading days apart.
- These sizable correlations give us confidence that past values will be helpful in predicting the future.
  - Think of the IMDB example.
- This is a curious prediction problem: the response v_t is also a feature $v_{t-\ell}$!

RNN Forecaster

- We only have one series of data! How do we set up for an RNN?
- We extract many short mini-series of input sequences $X = (X_1, \cdots, X_L) \in \mathbb{R}^{3\times L}$ with a predefined length L known as the lag:

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \cdots, X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \text{ and } Y = v_t.$$

- Note that this L is the same as the L in Figure 10.12, and there are $n = T - L + 1 = 6046$ observations in total, where $T = n_{\text{train}} + n_{\text{test}} = 6,051$, and $L = 5$.
- We fit an RNN with $K = 12$ hidden units per lag step (i.e. per A_ℓ.)
- L and K are tuning parameters and can be chosen by CV.

RNN Results for NYSE Data

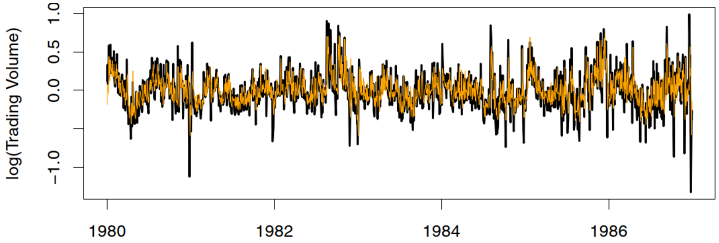


FIGURE 10.16. *RNN forecast of* `log_volume` *on the* `NYSE` *test data. The black lines are the true volumes, and the superimposed orange the forecasts. The forecasted series accounts for 42% of the variance of* `log_volume`.

- $R^2 = 0.42$ for RNN. Recall that $R^2 = 1 - \frac{\sum_{t=1}^{n_{\text{test}}} \left(y_{n_{\text{train}}+t} - \widehat{y}_{n_{\text{train}}+t}\right)^2}{\sum_{t=1}^{n_{\text{test}}} \left(y_{n_{\text{train}}+t} - \bar{y}_{n_{\text{train}}+\cdot}\right)^2}$.
- $R^2 = 0.18$ for straw man, – use yesterday's value of Log trading volume to predict that of today.
- Note that when predicting y_{4281+t}, we are using $\left(x_{4281+t-L}, \cdots, x_{4281+t-1}\right)$ rather than $\left(\hat{x}_{4281+t-L}, \cdots, \hat{x}_{4281+t-1}\right)$.

Autoregression Forecaster

- The RNN forecaster is similar in structure to a traditional autoregression (AR) procedure.
- Define $\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ \vdots \\ v_T \end{bmatrix}$ and $\mathbf{M} = \begin{bmatrix} 1 & v_L & \cdots & v_1 \\ 1 & v_{L+1} & \cdots & v_2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & v_{T-1} & \cdots & v_{T-L} \end{bmatrix}$.
- Fit an OLS regression of $\mathbf{y}$ on $\mathbf{M}$, giving

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \cdots + \hat{\beta}_L v_{t-L},$$

  known as an order-L autoregression model or AR(L).
- For the NYSE data we can include lagged versions of DJ_return and log_volatility in matrix $\mathbf{M}$, resulting in $3L+1$ columns.
  - Using the terminology of RNN, AR(L) is a kind of flattening.
- $R^2 = 0.41$ for AR(5) model ($p \times L + 1 = 16$ parameters).
- $R^2 = 0.42$ for RNN model ($K \times (1 + p + K) + K + 1 = 205$ parameters).
- $R^2 = 0.42$ for nonlinear AR(5) model fit by a single-layer NN with $K = 32$.
- $R^2 \approx 0.46$ for all models if we include day of week (Mondays and Fridays have high trading volumes) of day being predicted.
- LSTM can improve R^2 of RNN up to 1% in this example.

Summary of RNNs

- We have presented the simplest of RNNs. Many more complex variations exist.
- One variation treats the sequence as a one-dimensional image, and uses CNNs for fitting. For example, a sequence of words using an embedding representation can be viewed as an image, and the CNN convolves by sliding a convolutional filter along the sequence to learn particular phrases or short subsequences.
- Can have additional hidden layers, where each hidden layer is a sequence, and treats the previous hidden layer as an input sequence. [Figure here]
- Bidirectional RNNs: scan the document in both directions.
- Can have output also be a sequence, and input and output share the hidden units. So-called Seq2Seq learning are used for language translation (e.g., Google Translate).

Multilayer RNNs

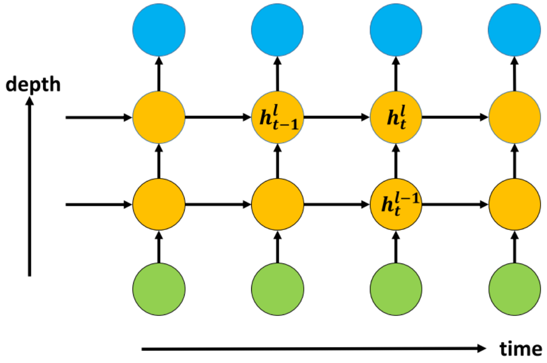


Figure: An RNN with two hidden layers. Higher-level hidden states h_t^l (like our A_t) are determined by the old states h_t^{l-1} and lower-level hidden states h_{t-1}^l. Multilayer RNNs generalize both feed-forward neural nets and one-hidden-layer RNNs.

- Multilayer RNNs usually do not have very large depth (e.g., 2–5), since T is already very large.

When to Use Deep Learning

(Section 10.6)

When to Use Deep Learning?

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis.
  - Examples include digital mammography, ophthalmology eye scans, MRI (magnetic resonance imaging) scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

Should we always use deep learning models?

Continued

- Often the big successes occur when the signal to noise ratio is high – e.g., image recognition and language translation. Datasets are large (so that fitting high-dimensional nonlinear models is possible), and overfitting and/or interpretability is not a big problem.
- For noisier data, simpler models can often work better.
  - On the NYSE data, the AR(5) model is much simpler than an RNN, and performed as well.
  - On the IMDB review data, the linear model fit by glmnet did as well as the NN, and better than the RNN.
  - Read the textbook on the Hitters data example, run Lab 10.9.1, and solve Assignment IV.9.
- We endorse the Occam's razor principle again – we prefer simpler models if they work as well. More interpretable and robust!
  - Whenever possible, try the simpler models as well, and then make a choice based on the performance/complexity tradeoff.

(*) Fitting a Neural Network

(Section 10.7)

Fitting NNs

- Take the single-layer NN for regression as an illustration.
- We try to solve the nonlinear least squares problem:

$$\min_{\{w_k\}_1^K,\beta} \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

  where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\right).$$

- This problem is difficult because the objective is non-convex.
  - Nested arrangement of parameters and the symmetry of the hidden units imply multiple solutions – in Figure 10.17, one is a local minimum, and the other is the global minimum.
- Despite this, effective algorithms have evolved that can optimize complex NN problems efficiently.
- Two general strategies: (i) Slow Learning by using gradient descent; (ii) Regularization such as in ridge or lasso.

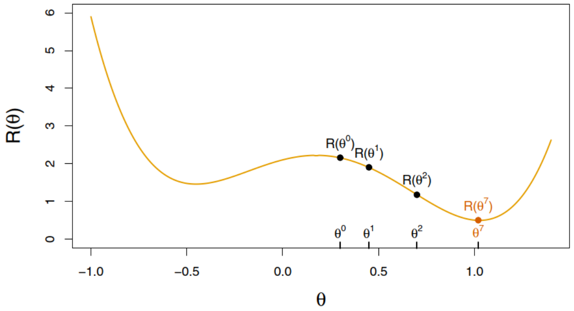


FIGURE 10.17. *Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill — against the gradient — until it cannot go down any further. Here gradient descent reached the global minimum in 7 steps.*

Non Convex Functions and Gradient Descent

- Let
$$R(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - f_\theta(x_i)\right)^2,$$
where $\theta = \left(\{w_k\}_1^K, \beta\right)$.
- Step 1: Start with a guess θ_0 for all the parameters in θ, and set $t = 0$.
- Step 2: Iterate until the objective $R(\theta)$ fails to decrease:
  - (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e., $R\left(\theta^{t+1}\right) < R\left(\theta^t\right)$.
  - (b) Set $t \leftarrow t+1$.
- In this simple example we reached the global minimum.
- If we had started a little to the left of θ_0 we would have gone in the other direction, and ended up in a local minimum.
- Although θ is multi-dimensional, we have depicted the process as one-dimensional. It is much harder to identify whether one is in a local minimum in high dimensions.

Gradient Descent Continued

- How to find a direction δ that points downhill? We compute the gradient vector:

$$\nabla R\left(\theta^t\right) = \left.\frac{\partial R\left(\theta^t\right)}{\partial \theta}\right|_{\theta=\theta^t},$$

  i.e., the vector of partial derivatives at the current guess θ^t.

- $\nabla R\left(\theta^t\right)$ is the direction in θ-space in which $R(\theta)$ increases (i.e., goes uphill) most rapidly at θ^t, so the idea of gradient descent is to move θ a little in the opposite direction (i.e., go downhill) [figure here; see also Figure 10.17]:

$$\theta^{t+1} = \theta^t + \delta = \theta^t - \rho \nabla R\left(\theta^t\right), \tag{1}$$

  where ρ is the learning rate (typically small, e.g. $\rho = 0.01$, i.e., slow learning as in boosting).

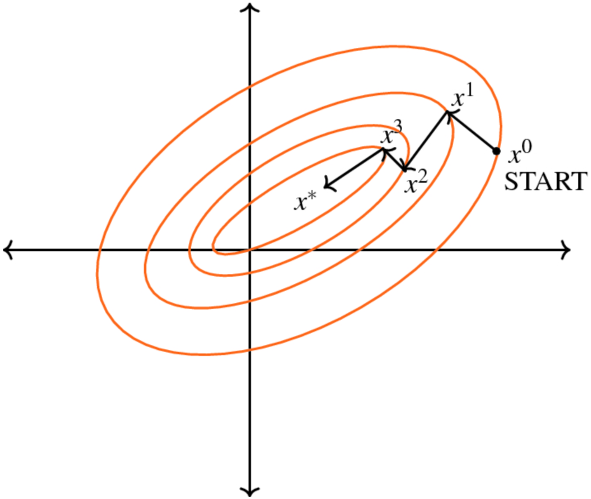


Figure: Steepest Descent: $\dim(\theta) = 2$

Backpropagation

- $R(\theta) = \sum_{i=1}^{n} R_i(\theta)$ is a sum, so gradient is sum of gradients, where

$$R_i(\theta) = \frac{1}{2}\left(y_i - f_\theta(x_i)\right)^2 = \frac{1}{2}\left(y_i - \beta_0 - \sum_{k=1}^{K}\beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\right)\right)^2.$$

- For ease of notation, let $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}$.
- Backpropagation uses the chain rule for differentiation:

$$\frac{\partial R_i(\theta)}{\partial \beta_k} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)}\frac{\partial f_\theta(x_i)}{\partial \beta_k} = -\left(y_i - f_\theta(x_i)\right)g(z_{ik}) =: \delta_i g(z_{ik}), \qquad (2)$$

and

$$\frac{\partial R_i(\theta)}{\partial w_{kj}} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)}\frac{\partial f_\theta(x_i)}{\partial g(z_{ik})}\frac{\partial g(z_{ik})}{\partial z_{ik}}\frac{\partial z_{ik}}{\partial w_{kj}} = -\left(y_i - f_\theta(x_i)\right)\cdot\beta_k\cdot g'(z_{ik})\cdot x_{ij} =: s_{ik}\cdot x_{ij}, \qquad (3)$$

where $-\delta_i$ is the residual $y_i - f_\theta(x_i)$, so both $\frac{\partial R_i(\theta)}{\partial \beta_k}$ and $\frac{\partial R_i(\theta)}{\partial w_{kj}}$ are fractions of the residual.

Two-Pass Algorithm

- δ_i and s_{ki} are "errors" from the current model at the output and hidden layer units, and

$$s_{ik} = g'(z_{ik})\beta_k\delta_i, \tag{4}$$

known as the backpropagation equations.

- Forward Pass: the current $\hat{\theta}$ is fixed and the predicted values are $f_{\hat{\theta}}(x_i)$.
- Backward Pass: the errors δ_i are computed (using $f_{\hat{\theta}}(x_i)$), and then back-progagated via (4) to give the errors s_{ik}.
  - Both sets of errors are then used to compute the gradients for the updates in (1), via (2) and (3).
- This two-pass procedure is what is known as backpropagation.

(**) More General Case

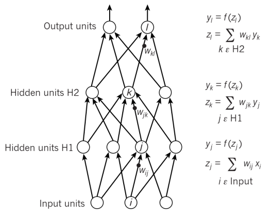
- In general,

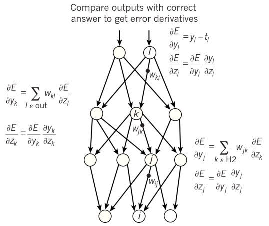
$$\frac{\partial R(\theta)}{\partial \mathbf{h}^{(\ell-1)}} = \frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{h}^{(\ell-1)}} \frac{\partial R(\theta)}{\partial \mathbf{h}^{(\ell)}} = \left(\mathbf{W}^{(\ell)}\right)^T \operatorname{diag}\left\{\mathbf{1}(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \geq 0)\right\} \frac{\partial R(\theta)}{\partial \mathbf{h}^{(\ell)}},$$

and

$$\frac{\partial R(\theta)}{\partial W_{jm}^{(\ell)}} = \frac{\partial R(\theta)}{\partial h_j^{(\ell)}} \cdot g' \cdot h_m^{(\ell-1)},$$

where $\mathbf{h}^{(\ell)}$, $\mathbf{W}^{(\ell)}$ and $\mathbf{b}^{(\ell)}$ collects the hidden nodes, weights and biases in the ℓth hidden layer, $g' = 1$ if the jth element of $\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$ is nonnegative, and $g' = 0$ otherwise.




Compare outputs with correct answer to get error derivatives
$\frac{\partial E}{\partial y_l} = y_l - t_l$
$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$
$\frac{\partial E}{\partial y_k} = \sum_{l \in out} w_{kl} \frac{\partial E}{\partial z_l}$
$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$
$\frac{\partial E}{\partial y_j} = \sum_{k \in H2} w_{jk} \frac{\partial E}{\partial z_k}$
$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$
l
k
j
i
w_{kl}
w_{jk}
w_{ij}

Stochastic Gradient Descent and Regularization

- When n is large, rather than compute the gradient using all the data, use a small minibatch drawn at random at each step.[7] E.g., for MNIST data, with $n = 60$K, $n_{\text{train}} = 48$K and $n_{\text{valid}} = 12$K, we use minibatches of 128 observations.
- This process is known as stochastic gradient descent (SGD).
- An epoch is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e.,$48\text{K}/128 \approx 375$ gradient steps for MNIST. [see Figure 10.18]
- In the first row of Table 10.1, we use ridge regularization (known as weight decay here):

$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i)) + \lambda\sum_j \theta_j^2 .$$

  - As in RR, it is better to standardize the predictors.
  - We can use different λ's for different layers; here, we penalize $\mathbf{W}_1$ and $\mathbf{W}_2$ only and do not penalize $\mathbf{B}$ since $\dim(\mathbf{B})$ is small [recall that the bias terms are not penalized in RR].
  - SGD enforces its own form of approximately quadratic regularization.
  - The validation objective starts to increase by 30 epochs, so early stopping can be used as an additional regularization.

[7] (**) Random draws without replacement have some advantages in theory and practice. The minibatch size m is typically 32–512. Because n/m need not be an integer, the learning rate ρ in the last minibatch should be appropriately adjusted.

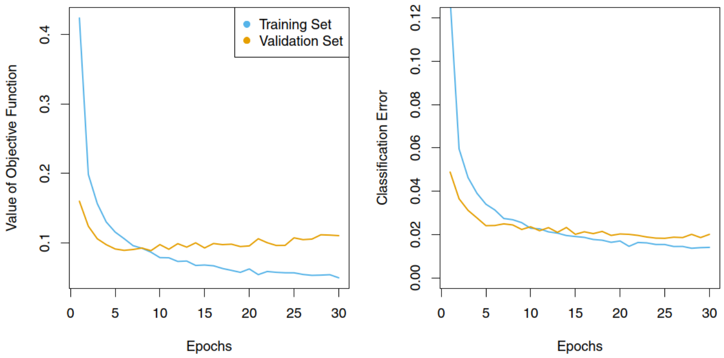


FIGURE 10.18. *Evolution of training and validation errors for the* MNIST *neural network depicted in Figure 10.4, as a function of training epochs.*

- The objective function is $-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log\left(f_m\left(x_i\right)\right)$.
- Two other popular forms of regularization are dropout and augmentation.

Dropout Learning

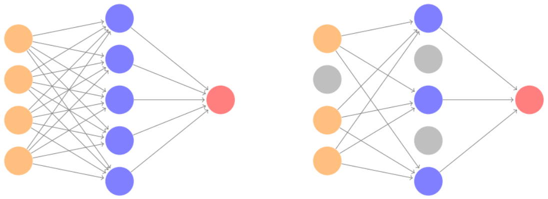
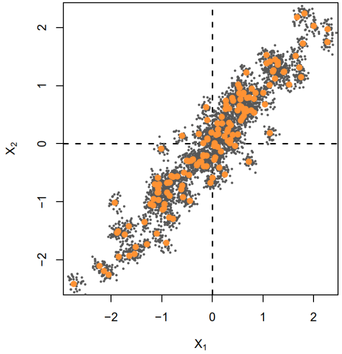


FIGURE 10.19. *Dropout Learning. Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random, and ignored in an instance of training.*

- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.
- In simple scenarios like linear regression, a version of this process can be shown to be equivalent to ridge regularization.
- Dropout has been used in practice to avoid correlation between weights which is in turn due to correlation/multicollinearity between nodes, just as in RR.
- This is also similar to randomly omitting variables when growing trees in RF, preventing nodes from becoming too-specialized.

Ridge and Data Augmentation



- Make many copies of each (x_i, y_i) and add a small amount of Gaussian noise to the x_i – a little cloud around each observation – but leave the copies of y_i alone!
- This makes the fit robust to small perturbations in x_i, and is equivalent to ridge regularization in an OLS setting.

Data Augmentation on the Fly

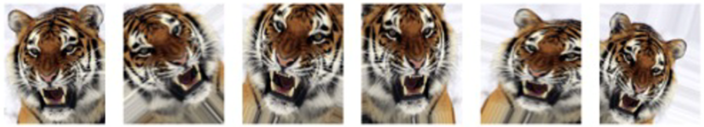


FIGURE 10.9. *Data augmentation. The original image (leftmost) is distorted in natural ways to produce different images with the same class label. These distortions do not fool humans, and act as a form of regularization when fitting the CNN.*

- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural distortions (e.g., zoom, horizontal and vertical shift, shear, small rotations, and in this case horizontal flips) are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.
- The label is left unchanged – in each case still tiger.
- We do not need to store the distorted images in fitting the CNN.
- Improves performance of CNN and is similar to ridge.

Network Tuning

1. The architecture of NN (the number of hidden layers, and the number of units per layer): the number of units per hidden layer can be large, and control overfitting by regularization.
2. Regularization tuning parameters (dropout rate ϕ and the strength λ of lasso and ridge): typically set separately at each layer.
   - How to achieve dropout for each layer? Set the weights associated with the "dropped out" units to zero, while keeping the architecture intact.
3. Details of SGD: batch size, the number of epochs, and details of data augmentation.

(**) Interpolation and Double Descent

(Section 10.8)

Interpolation and Double Descent

- With NNs, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running SGD till zero training error (i.e., interpolate the training data) often gives good out-of-sample error.
- Increasing the number of units or layers and again training till zero error sometimes gives even better out-of-sample error. [brief Figure 10.20]
- Typically, we expect a U-shape test error curve (and a monotonically decreasing training error curve), and advocate an intermediate level of model complexity.

What happened to overfitting and the usual bias-variance trade-off?

- The discussion below is based on "Belkin, Hsu, Ma, and Mandal, 2019, Reconciling Modern Machine Learning and the Bias-Variance Trade-off, *PNAS*, 116, 15849–15854"

Simulation

- $Y = \sin(X) + \varepsilon$ with $X \sim U[-5,5]$ and $\varepsilon \sim N\left(0, 0.3^2\right)$ Gaussian with S.D.$= 0.3$.
  - The signal-to-noise ratio – $Var(f(X))/Var(\varepsilon)$ – is 5.9, quite high.
- Training set $n = 20$, test set very large (10K).
- We fit a natural spline to the data with d degrees of freedom – i.e., a linear regression onto d basis functions:

$$\hat{y}_i = \hat{\beta}_1 N_1(x_i) + \hat{\beta}_2 N_2(x_i) + \cdots + \hat{\beta}_d N_d(x_i).$$ [see Figure 10.21]

- When $d = 20$ we fit the training data exactly, and get all residuals equal to zero.
- When $d > 20$, we still fit the data exactly, but the solution is not unique. Among the zero-residual solutions, we pick the one with minimum norm, i.e., the zero-residual solution with smallest $\sum_{j=1}^{d} \hat{\beta}_j^2$.
  - To achieve a zero-residual solution with $d = 20$ is a real stretch! Easier for larger d (e.g., $d = 42$ and 80 in Figure 10.21).

In Figure 10.20,

- When $d \leq 20$, model is OLS, and we see usual bias-variance trade-off.
- When $d > 20$, we revert to minimum-norm. As d increases above 20, the minimum $\sum_{j=1}^{d} \hat{\beta}_j^2$ decreases since it is easier to achieve zero error, and hence less wiggly solutions.

The Double-Descent Error Curve

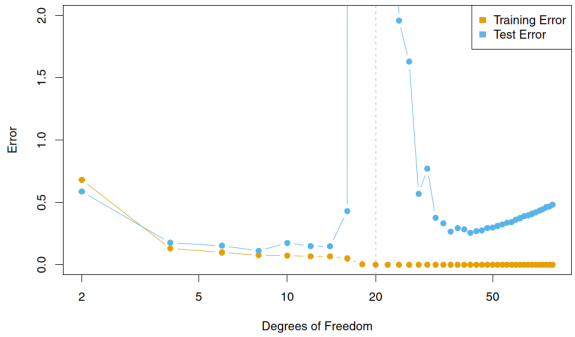


FIGURE 10.20. *Double descent phenomenon, illustrated using error plots for a one-dimensional natural spline example. The horizontal axis refers to the number of spline basis functions on the log scale. The training error hits zero when the degrees of freedom coincides with the sample size $n = 20$, the "interpolation threshold", and remains zero thereafter. The test error increases dramatically at this threshold, but then descends again to a reasonable value before finally increasing again.*

Less Wiggly Solutions

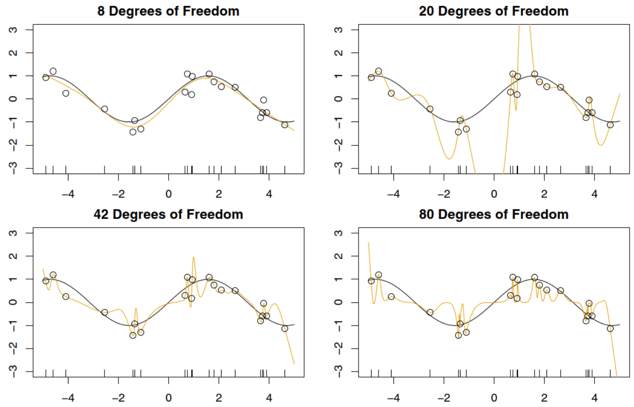


FIGURE 10.21. *Fitted functions $\hat{f}_d(X)$ (orange), true function $f(X)$ (black) and the observed 20 training data points. A different value of d (degrees of freedom) is used in each panel. For $d \geq 20$ the orange curves all interpolate the training points, and hence the training error is zero.*

Some Comments

- The double-descent phenomenon does not contradict the bias-variance trade-off! - The x-axis is d rather than the "flexibility" of models, e.g., $d = 42$ is less "flexible" than $d = 20$, so has lower variance.
- Most of the statistical learning methods in this course do not exhibit double descent, e.g., the regularization approaches typically do not interpolate since they can give great results without interpolating. Think about the example here.
- The maximal margin classifiers and SVMs that have zero training error often achieve good test error. This is because they seek smooth minimum norm solutions, similar to the minimum-norm natural spline here.

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a minimum norm zero-residual solution.
- Stochastic gradient flow – i.e., the entire path of SGD solutions – is somewhat similar to ridge path.
- By analogy, deep and wide NNs fit by SGD down to zero training error often give good solutions that generalize well.
- In particular cases with high signal-to-noise ratio – e.g., image recognition – are less prone to overfitting; the zero-error solution is mostly signal!
- Nonetheless, we typically do not want to rely on double descent in NNs, e.g., through regularization or early stopping.

Lab: Deep Learning

(Section 10.9)

- A Single Layer Network on the Hitters Data
- A Multilayer Network on the MNIST Digit Data
- Convolutional Neural Network
- Using Pretrained CNN Models

- IMDB Document Classification
- Recurrent Neural Networks

Appendix: Deep Unsupervised Learning

Alternative Interpretation of PCA

- We will discuss two unsupervised learning methods in deep learning: Autoencoders and Generative Adversarial Networks (GANs).
- Autoencoders can be interpreted as a nonlinear PCA for dimension reduction; like in PCR, they can be used for pre-training in supervised learning, especially in small datasets, to avoid overfitting.
- First, we express PCA in the following way:

$$\mathbf{X} = (\mathbf{U}_M \mathbf{D}_M) \mathbf{V}_M^T = (\mathbf{X}\mathbf{V}_M) \mathbf{V}_M^T,$$

  where the subscript M means keeping the first M columns of $\mathbf{U}, \mathbf{D}$ and $\mathbf{V}$.
- In other words, $\mathbf{V}_M^T$ and $\mathbf{V}_M$ are the solutions of the following minimization problem:

$$\min_{\mathbf{W}_f, \mathbf{W}_g} \frac{1}{n} \left\| \mathbf{X}^T - \mathbf{W}_f \mathbf{W}_g \mathbf{X}^T \right\|^2 = \min_{\mathbf{W}_f, \mathbf{W}_g} \frac{1}{n} \sum_{i=1}^{n} \left\| \mathbf{x}_i - \mathbf{W}_f \mathbf{W}_g \mathbf{x}_i \right\|^2,$$

  where $\mathbf{W}_g : \mathbb{R}^p \to \mathbb{R}^M$ and $\mathbf{W}_f : \mathbb{R}^M \to \mathbb{R}^p$ are treated as linear functions $f(\mathbf{x}) = \mathbf{W}_f \mathbf{x}$ and $g(\mathbf{h}) = \mathbf{W}_g \mathbf{h}$ for $\mathbf{x} \in \mathbb{R}^p$ and $\mathbf{h} \in \mathbb{R}^M$.
- The encoder function $f(\cdot)$ maps the input $\mathbf{x} \in \mathbb{R}^p$ to a hidden code/representation $\mathbf{h} = f(\mathbf{x}) \in \mathbb{R}^M$, and the decoder function $g(\cdot)$ maps the hidden representation $\mathbf{h}$ to a point $g(\mathbf{h}) \in \mathbb{R}^p$.
- So PCA is often known as the undercomplete linear autoencoder.

Autoencoders

- The autoencoder amounts to solving the following minimization problem:

$$\min_{f,g} \frac{1}{n} \sum_{i=1}^{n} \mathscr{L}\left(\mathbf{x}_i, g\left(\mathbf{h}_i\right)\right) \text{ with } \mathbf{h}_i = f\left(\mathbf{x}_i\right) \text{ for all } i, \tag{5}$$

  where $\mathscr{L}(\cdot,\cdot)$ is a loss function that measures the difference between the two arguments, $f(\mathbf{x}_i) \in \mathbb{R}^M$ and $g(\mathbf{h}_i) \in \mathbb{R}^p$ can be generated by multilayer neural networks. [figure here]
- Parallel to PCA, if $M < p$, we get the undercomplete autoencoder.
- Generally, we can put some structures on the autoencoder to achieve sparsity (if $M > p$) or robustness (to contamination on $\mathbf{x}_i$).
- Sparse Autoencoders: in (5), add a regularization term to the objective function:

$$\min_{f,g} \frac{1}{n} \sum_{i=1}^{n} \mathscr{L}\left(\mathbf{x}_i, g\left(\mathbf{h}_i\right)\right) + \lambda \left\|\mathbf{h}_i\right\|_1 \text{ with } \mathbf{h}_i = f\left(\mathbf{x}_i\right) \text{ for all } i$$

- Denoising Autoencoders: like data augmentation, solve

$$\min_{f,g} \frac{1}{n} \sum_{i=1}^{n} \mathscr{L}\left(\mathbf{x}_i, g\left(f\left(\widetilde{\mathbf{x}}_i\right)\right)\right),$$

  where $\widetilde{\mathbf{x}}_i = \mathbf{x}_i + \xi_i$ with ξ_i being the noise created on purpose.

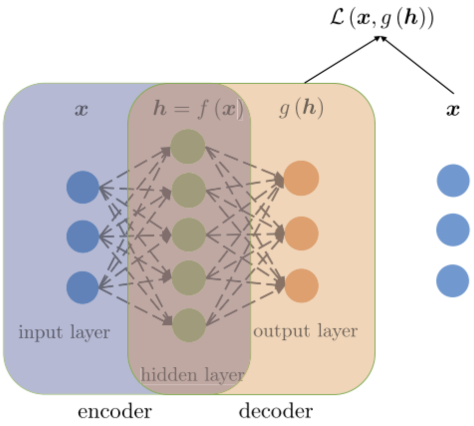


FIG. *A diagram of a (sparse) autoencoder. First, an input $\boldsymbol{x}$ goes through the encoder $\boldsymbol{f}(\cdot)$, and we obtain its hidden representation $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x})$. Then we use the decoder $\boldsymbol{g}(\cdot)$ to get $\boldsymbol{g}(\boldsymbol{h})$ as a reconstruction of $\boldsymbol{x}$. Finally, the loss is determined from the difference between the original input $\boldsymbol{x}$ and its reconstruction $\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$. A regularizer is used if the dimension of $\boldsymbol{h}$ is larger than that of $\boldsymbol{x}$.*

Generative Adversarial Networks (GANs)

- GANs are designed as an implicit density estimator of $\mathbb{P}_X$ when p is large. Why?
  (i) GANs put more emphasis on sampling from the distribution $\mathbb{P}_X$ than estimation.
  (ii) GANs define the density estimation implicitly through a source distribution $\mathbb{P}_Z$ and a generator function $g(\cdot)$.
- The structure of GANs can be summarized in the following figure:

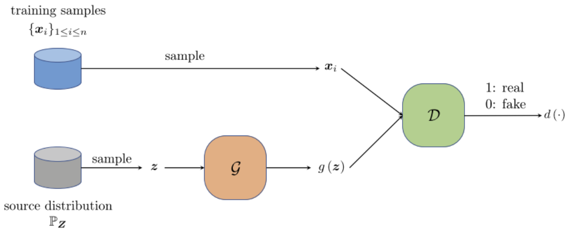


FIG. *GANs consist of two components, a generator $\mathcal{G}$ which generates fake samples and a discriminator $\mathcal{D}$ which differentiate the true ones from the fake ones.*

Sampling View of GANs

- Suppose the data at hand are all real images, and the target of GANs is to generate new natural images.
- With this goal in mind, the generator $\mathscr{G}$ and discriminator $\mathscr{D}$ play a zero-sum game.
- Suppose $g(\cdot)$ and $d(\cdot)$ are constructed by deep neural networks (e.g., CNN for images) with parameters $\theta_{\mathscr{G}}$ and $\theta_{\mathscr{D}}$; then GAN tries to solve the following min-max problem:

$$\min_{\theta_{\mathscr{G}}} \max_{\theta_{\mathscr{D}}} E_{\mathbf{x}\sim\mathbb{P}_X}\left[\log d(\mathbf{x})\right] + E_{\mathbf{z}\sim\mathbb{P}_Z}\left[\log\left(1-d\left(g\left(\mathbf{z}\right)\right)\right)\right], \tag{6}$$

  where $\mathbb{P}_Z$ is usually a standard multivariate normal distribution with $\dim(Z) < p$ in the order of hundreds, $g(\mathbf{z}) \in \mathbb{R}^p$ is a fake sample generated from $\mathscr{G}$, and $d(\cdot) \in [0,1]$ is the probability of a (real or fake) sample being a real sample from $\mathbb{P}_X$.
  - Fix $\theta_{\mathscr{G}}$, $\theta_{\mathscr{D}}$ maximizes the ability of differentiation; fix $\theta_{\mathscr{D}}$, $\theta_{\mathscr{G}}$ tries to generate more realistic sample $g(\mathbf{z})$ to fool $\mathscr{D}$.

Density Estimation View of GANs

- (6) can be re-written as

$$\min_{\mathbb{P}_{\mathscr{G}}} \max_{d(\cdot)} E_{\mathbf{x}\sim\mathbb{P}_X}\left[\log d\left(\mathbf{x}\right)\right] + E_{\mathbf{x}\sim\mathbb{P}_{\mathscr{G}}}\left[\log\left(1-d\left(\mathbf{x}\right)\right)\right], \tag{7}$$

  where $\mathbb{P}_{\mathscr{G}}$ is the distribution induced by $\mathscr{G}$ when $\mathbf{z} \sim \mathbb{P}_Z$.
- The inner maximization problem is solved by the likelihood ratio:

$$d^*\left(\mathbf{x}\right) = \frac{\mathbb{P}_X\left(\mathbf{x}\right)}{\mathbb{P}_X\left(\mathbf{x}\right) + \mathbb{P}_{\mathscr{G}}\left(\mathbf{x}\right)}.$$

- As a result, (7) can be simplified as

$$\min_{\mathbb{P}_{\mathscr{G}}} \text{JS}\left(\mathbb{P}_X \parallel \mathbb{P}_{\mathscr{G}}\right),$$

  where $\text{JS}(\cdot \parallel \cdot)$ denotes the Jensen-Shannon divergence between two distributions

$$\text{JS}\left(\mathbb{P}_X \parallel \mathbb{P}_{\mathscr{G}}\right) = \frac{1}{2}\text{KL}\left(\mathbb{P}_X \parallel \frac{\mathbb{P}_X + \mathbb{P}_{\mathscr{G}}}{2}\right) + \frac{1}{2}\text{KL}\left(\mathbb{P}_{\mathscr{G}} \parallel \frac{\mathbb{P}_X + \mathbb{P}_{\mathscr{G}}}{2}\right)$$

  with $\text{KL}(\mathbb{P}_X \parallel \mathbb{P}_{\mathscr{G}}) = \int \mathbb{P}_X\left(\mathbf{x}\right)\log\left(\frac{\mathbb{P}_X(\mathbf{x})}{\mathbb{P}_{\mathscr{G}}(\mathbf{x})}\right)d\mathbf{x}$ being Kullback–Leibler divergence between $\mathbb{P}_X$ and $\mathbb{P}_{\mathscr{G}}$ which was introduced in Appendix A of Lecture 6.

Continued

- The JS divergence can be replaced by other metrics.
- One popular choice is the Wasserstein distance, which generates the Wasserstein GAN (W-GAN):

$$\min_{\theta_{\mathscr{G}}} \mathrm{WS}\left(\mathbb{P}_X \parallel \mathbb{P}_{\mathscr{G}}\right) = \min_{\theta_{\mathscr{G}}} \sup_{f:f \text{ 1-Lipschitz}} E_{\mathbf{x}\sim\mathbb{P}_X}\left[f\left(\mathbf{x}\right)\right] - E_{\mathbf{x}\sim\mathbb{P}_{\mathscr{G}}}\left[f\left(\mathbf{x}\right)\right],$$

  where $f(\cdot)$ is taken over all Lipschitz functions with coefficient 1, and $\mathbb{P}_{\mathscr{G}}$ is generated by a neural network model $g_{\theta_{\mathscr{G}}}$.
  - Comparing with (6), $f(\cdot)$ corresponds to the discriminator $\mathscr{D}$ in the sense that they share similar objectives to differentiate the true distribution $\mathbb{P}_X$ from the fake one $\mathbb{P}_{\mathscr{G}}$.
- Obviously, GANs are harder to train than supervised deep learning models; how to evaluate GANs objectively and effectively is also hard.
- We can see the close connection of GANs with the supervised density estimation in the appendix of Lecture 6.