Deep Learning I: Basics and CNN

PS690 Computational Methods in Social Science

Jiawei Fu
Department of Political Science
Duke University

October 21, 2025

Overview

Shallow Neural Networks

- Shallow neural networks are functions look like

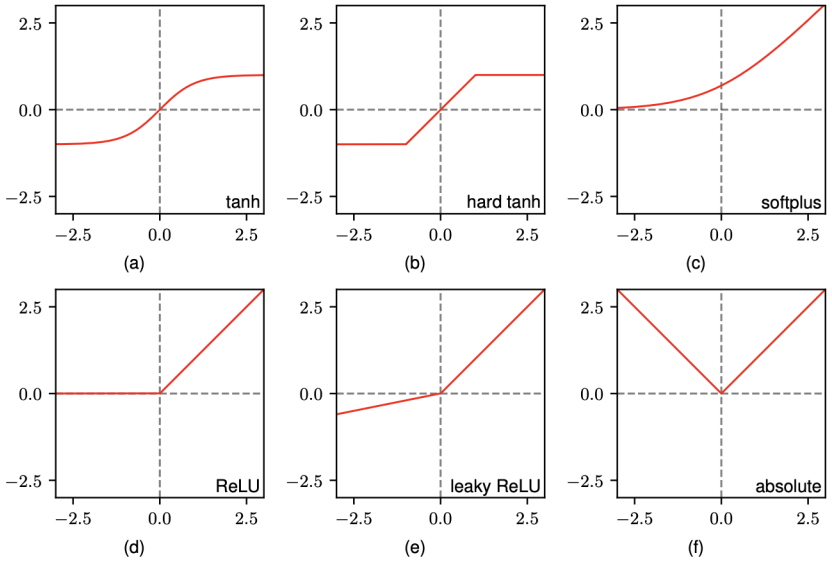
$$\begin{aligned} y &= f(x, \phi) \\ &= \phi_0 + \phi_1 a(\theta_{10} + \theta_{11} x) + \phi_2 a(\theta_{20} + \theta_{21} x) + \phi_3 a(\theta_{30} + \theta_{31} x) \end{aligned}$$

- It has three parts:
  - Linear functions of the input data: $\theta_{i0} + \theta_{i1} x$
  - Pass them into an *activation function*: $a(\cdot)$. The most common choice is the rectified linear unit or ReLU:

$$ReLU(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

  - Sum them together with weight ϕ_i
- The activation function makes function to be **nonlinear**.

Activation Functions



Shallow Neural Networks

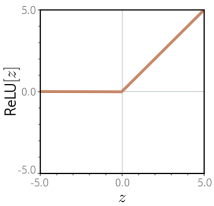


Figure: ReLU function

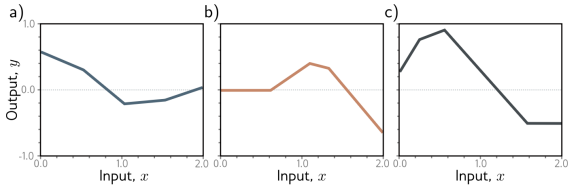


Figure: Shallow neural networks represent a family of continuous piecewise linear functions.

Shallow Neural Networks

- We use $h_i = a(\theta_{i0} + \theta_{i1}x)$ to denote *hidden units*.
- Then, $y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$

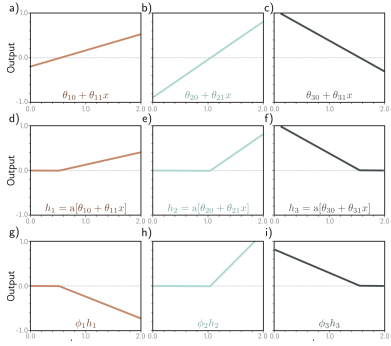


Figure: [Prince, 2023]

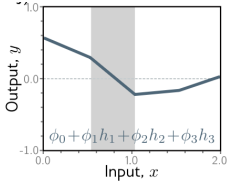


Figure: [Prince, 2023]

Depicting Neural Networks

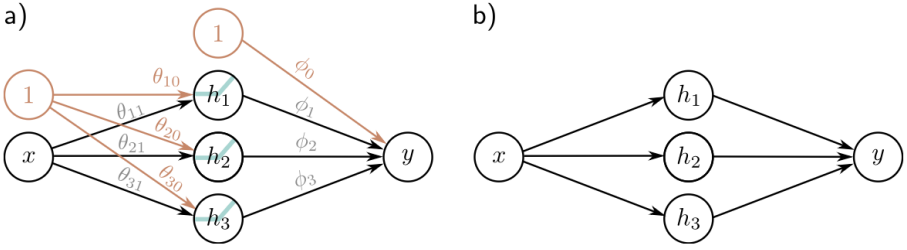


Figure: [Prince, 2023]

Universal approximation theorem

- Now, consider total $D > 0$ hidden units:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

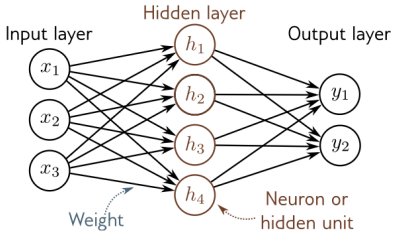
- The number of hidden units in a shallow network is a measure of the network capacity.
- With ReLU activation functions, the network has at most D joints and so is a piecewise linear function with at most $D + 1$ linear regions.
- Indeed, with enough capacity (hidden units), a shallow network can describe any continuous 1D function defined on a compact subset of the real line to arbitrary precision.
- The universal approximation theorem proves that for any continuous function, there exists a shallow network that can approximate this function to any specified precision.

Multivariate inputs and outputs

- The model can be generalized to multiple inputs and outputs.

$$h_d = a(\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i)$$

$$y_j = \phi_{j0} + \sum_{i=1}^{D} \phi_{jd} h_d$$



- Neural networks have a lot of associated jargon.
- They are often referred to in terms of layers.

Universal approximation theorem

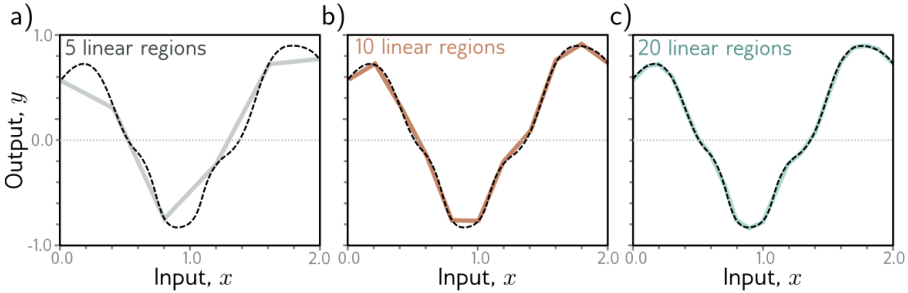


Figure: [Prince, 2023]

Universal approximation theorem

- There are many versions of the theorem.
- We consider one version that activation function is squashing function (non-decreasing, $\lim_{x\to-\infty} a(x) = 0$,$\lim_{x\to\infty} a(x) = 1$), for example ReLu.

Theorem

Let a be a squashing function and K be a compact subset of R^d. Then, for every continuous function $f : R^d \to R$ and every $\epsilon > 0$, there exists a neural network $y(x) = \phi_0 + \sum_{i=1}^{k} \phi_i a(\theta_{d0} + \theta_d^T x)$ such that $\sup_{x\in K} |f(x) - y(x)| < \epsilon$.

Universal approximation theorem

Theorem

Let a be a squashing function and K be a compact subset of R^d. Then, for every continuous function $f : R^d \to R$ and every $\epsilon > 0$, there exists a neural network $y(x) = \phi_0 + \sum_{i=1}^{k} \phi_i a(\theta_{d0} + \theta_d^T x)$ such that $\sup_{x \in K} |f(x) - y(x)| < \epsilon$.

- The proof has three main steps.
- First, we use stone-Weierstrass theorem to show that networks of the form $\sum_{i=1}^{k} \phi_i cos(\theta_{d0} + \theta_d^T x)$ are uniformly dense on compacts in the space of continuous functions.
- Then, we show cosine functions can be approximated in sup norm by networks with the cosine squasher.
- Finally, we show that netwroks with the cosine squasher can be approximated in sup norm by networks with an arbitrary suqashing function a.

Deep neural networks

- Networks with multiple hidden layers are referred to as deep neural networks.
- Deep networks can produce many more linear regions than shallow networks for a given number of parameters; therefore, can represent a broader family of functions.

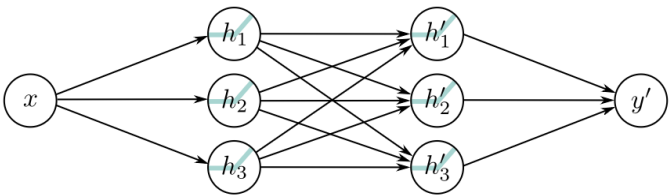


Figure: Neural network with one input, one output, and two hidden layers, each containing three hidden units.

Deep neural networks

The first layer is defined by:

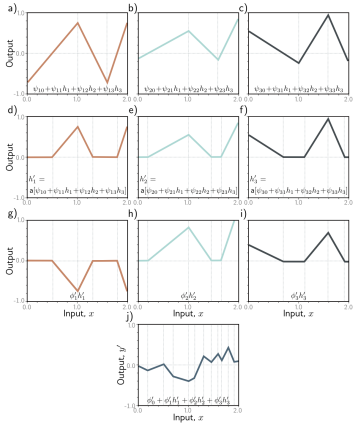
$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned}$$

the second layer by:

$$\begin{aligned} h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3], \end{aligned}$$

and the output by:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.$$



Deep neural networks

- Modern networks might have more than a hundred layers with thousands of hidden units at each layer.
- The number of hidden units in each layer is referred to as the width of the network.
- The number of hidden layers as the depth.
- Both deep and shallow networks can model arbitrary functions, but some functions can be approximated much more efficiently with deep networks.
- With enough hidden units, shallow networks can describe arbitrarily complex functions in high dimensions. However, it turns out that for some functions, the required number of hidden units is impractically large.
- Functions have been identified that require a shallow network with exponentially more hidden units to achieve an equivalent approximation to that of a deep network.

Images Data

- Images have three properties that suggest the need for specialized model architecture.
- First, they are high-dimensional. An image comprises a rectangular array of pixels, in which each pixel has either a grey-scale intensity or more commonly a triplet of red, green, and blue channels each with its own intensity value.
- A typical image for a classification task contains 224×224 RGB values (i.e., 150,528 input dimensions).
- So even for a shallow fully connected network, the number of weights would exceed 150,5282, or 22 billion.
- Second, as we mentioned in the first lecture, real informative figure lies in the low dimension.
- One implication is that nearby image pixels are statistically related. Fully connected net-works have no notion of "nearby" and treat the relationship between every input equally.
- Third, the interpretation of an image is stable under geometric transformations. An image of a tree is still an image of a tree if we shift it leftwards by a few pixels.

A Typical DNN for Image Classification

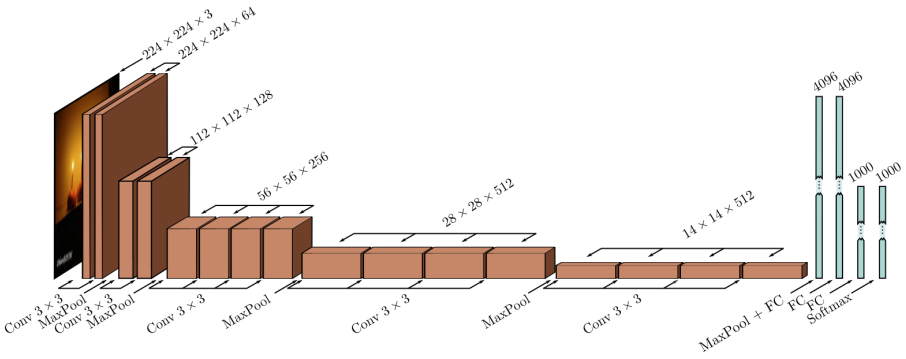
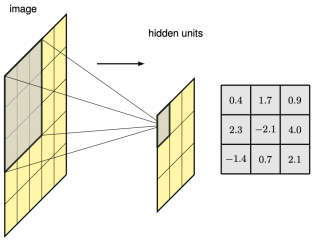


Figure: VGG (Visual Geometry Group) network: 19 hidden layers and 144 million parameters.

Feature detectors

- For simplicity we will initially restrict our attention to grey-scale images (single channel).
- Consider a single unit in the first layer of a neural network that takes as input just the pixel values from a small rectangular region, or patch, from the image.



- We call it as the **receptive field** of that unit, and it captures the notion of locality.
- The output is $z = ReLU(w^T x + w_0)$, x is the a vector of pixel values.

Feature detectors

- We would like weight values associated with this unit to learn to detect some useful low-level feature.
- Because there is one weight associated with each input pixel, the weights themselves form a small two-dimensional grid known as a **filter**, sometimes also called a **kernel**, as shown in the previous slide.
- Fix w, w_0, and $||x||^2$, we ask for which value of the input image patch x will this hidden unit give the largest output response?
- Then the solution for x that maximizes $w^T x$, and hence maximizes the response of the hidden unit, is of the form $x = \alpha w$ for some coefficient α.
- This says that the maximum output response from this hidden unit occurs when it detects a patch of image that looks like the kernel image.
- Because ReLU generates a non-zero output only when $w^T x$ exceeds a threshold of w_0, and therefore the unit acts as a feature detector that signals when it finds a sufficiently good match to its kernel.

Feature detectors

- For example, suppose we want to detect edges in images.
- Intuitively, we can think of a vertical edge as occurring when there is a significant local change in the intensity between pixels as we move horizontally across the image.
- Therefore, the filters detecting vertical and horizontal edges look like:

−1	0	1
−1	0	1
−1	0	1

Figure: 3 ×3 filter detecting vertical edges

−1	−1	−1
0	0	0
1	1	1

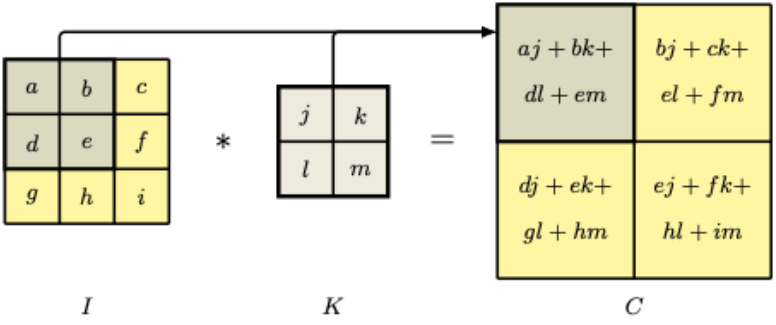
Figure: 3 ×3 filter detecting horizontal edges

Feature detectors

Figure: Illustration of edge detection using convolutional filters form [Bishop and Bishop, 2023]

(a) the original image, (b) the result of convolving with the filter that detects vertical edges, and (c) the result of convolving with the filter that detects horizontal edges.

Why called Convolution?



- Strictly speaking, this operation is called a cross-correlation.

Padding, Stride, Kernel Size, and Dilation

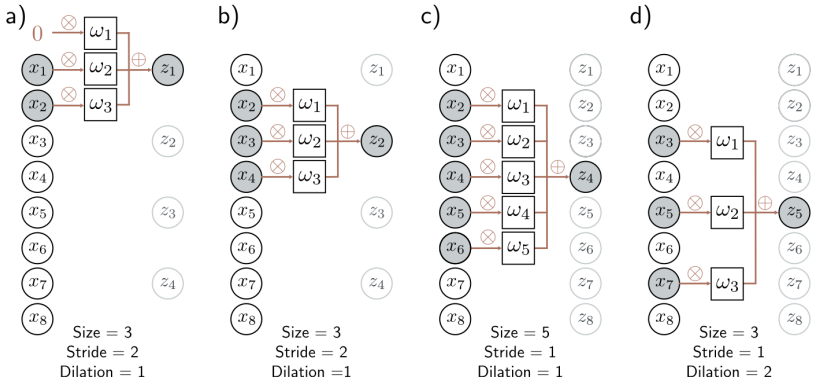


Figure: [Prince, 2023]

Zero padding assumes the input is zero outside its valid range (in a).

Translation equivariance

- If a small patch in a face image represents an eye at that location, then the same set of pixel values in a different part of the image must represent an eye at the new location.
- To achieve this, we can simply replicate the same hidden-unit weight values at multiple locations across the image.
- For example, we can use the same filter to detect edges in images.
- Therefore, the weighting matrix of convolutional layers are pretty sparse, compared to fully connected layers.

Translation equivariance

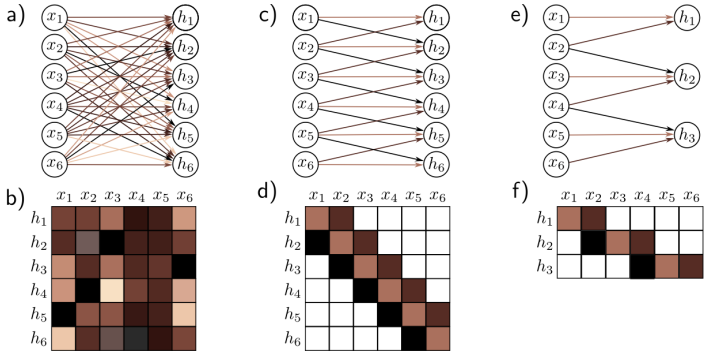
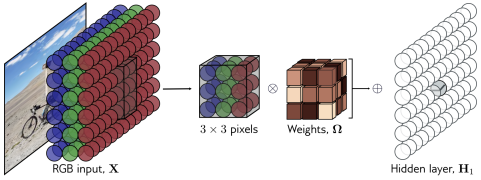


Figure: The figure shows different connected layer and associated weighting matrix. (C) is a convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs. (e) is a convolutional layer with kernel size three and stride two computes a weighted sum at every other position.

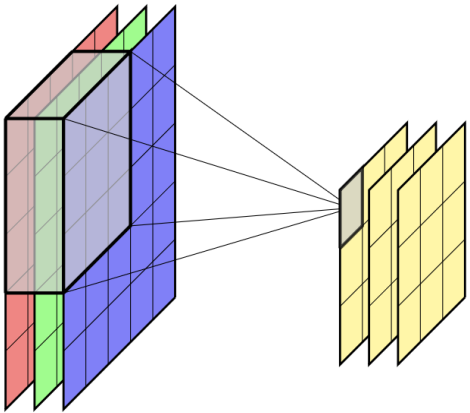
Multi-dimensional Convolutions

- So far we have considered convolutions over a single grey-scale image. For a colour image there will be three channels corresponding to the red, green, and blue colours.
- We can easily extend convolutions to cover multiple channels by extending the dimensionality of the filter.
- An image with $J \times K$ pixels and C channels will be described by a tensor of dimensionality $J \times K \times C$.
- We can introduce a filter described by a tensor of dimensionality $M \times M \times C$ comprising a separate $M \times M$ filter for each of the C channels.



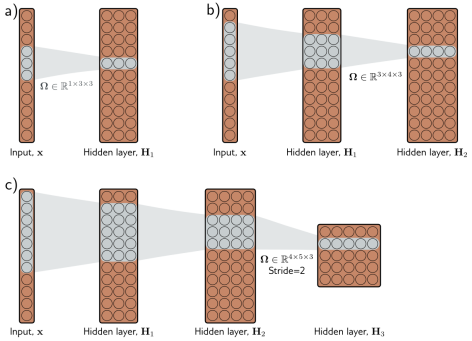
Multi-dimensional Convolutions

- Also, a filter is analogous to a single hidden node in a fully connected network, and it can learn to detect only one kind of feature and is therefore very limited.
- Hence, it is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, termed a feature map or channel.



Multilayer convolutions

- We have learned that deep networks, which consisted of a sequence of fully connected layers.
- Similarly, convolutional networks comprise a sequence of convolutional layers.
- The effective receptive field of a unit in later layers in the network becomes much larger than those in earlier layers.



Multilayer convolutions

- Intuitively, multilayer convolutions can learn hierarchical structure, in which complex features at a particular level are built up from simpler features at the previous level.

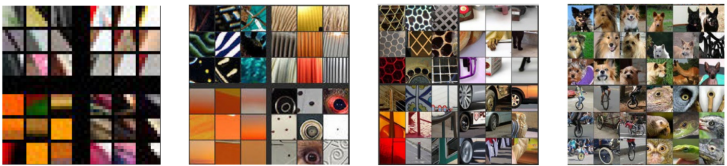
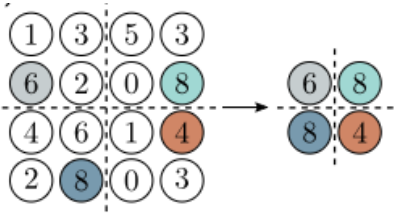


Figure: The first layer responding to edges, the second layer responding to textures and simple shapes, layer 3 showing components of objects (such as wheels), and layer 5 showing entire objects

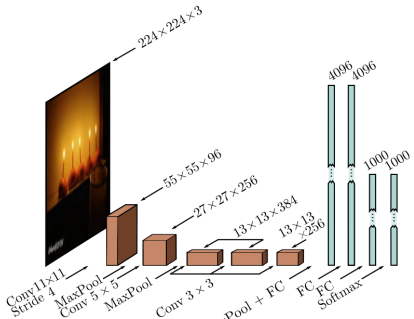
Pooling

- Pooling summarizes the information in a neighborhood.
- Pooling introduces translation invariance, helping models recognize objects even when they appear in different positions within an image.
- Pooling is not strictly necessary, but historically it has been very effective and conceptually simple.



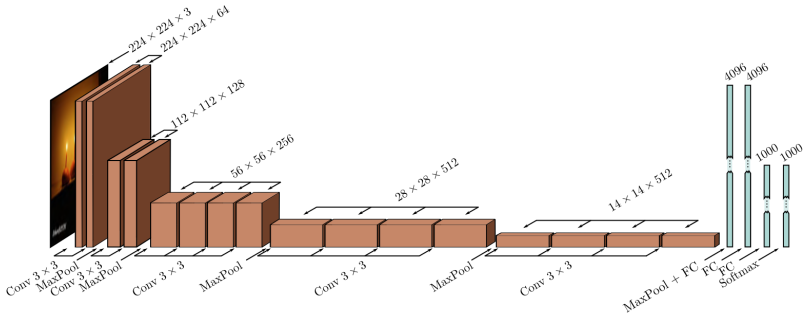
Example network architectures

- Much of the pioneering work on deep learning in computer vision focused on image classification using the ImageNet dataset (1,281,167 training images, 50,000 validation images, and 100,000 test images).
- In 2012, AlexNet was the first convolutional network to perform well on this task.
- Pooling summarizes the information in a neighborhood It is not strictly necessary, but historically it has been very effective and conceptually simple.



Example network architectures

- The VGG (Visual Geometry Group) network was also targeted at classification in the ImageNet task and achieved a considerably better performance.
- The most important change between AlexNet and VGG was the depth of the network.



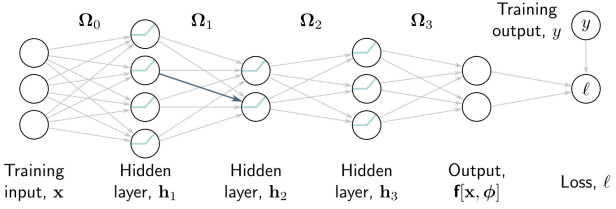
Problem Definitions

- Consider a network $f[x, \phi]$ with multivariate input x, parameters ϕ, and three hidden layers h_1, h_2, h_3:

$$
\begin{aligned}
h_1 &= a[\beta_0 + \Omega_0 x] \\
h_2 &= a[\beta_1 + \Omega_1 h_1] \\
h_3 &= a[\beta_2 + \Omega_2 h_2] \\
f[x, \phi] &= \beta_3 + \Omega_3 h_3
\end{aligned}
$$

Note, those are all vectors/matrices. For example, h_1 is

$$
h_1 = a\left[\begin{bmatrix}\theta_{10}\\ \theta_{20}\\ \theta_{30}\\ \theta_{40}\end{bmatrix} + \begin{bmatrix}\psi_{11} & \psi_{12} & \psi_{13}\\ \psi_{21} & \psi_{22} & \psi_{23}\\ \psi_{31} & \psi_{32} & \psi_{33}\\ \psi_{41} & \psi_{42} & \psi_{43}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}\right]
$$



Gradient Descent

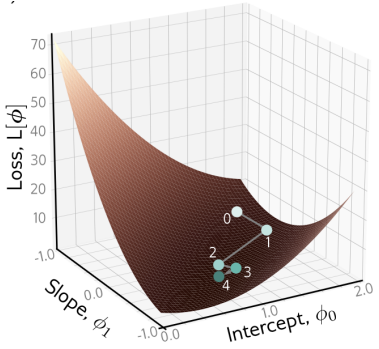
- For supervised learning, we need a loss function $L[\phi] = \sum_{i=1}^{I} l_i$.
- Loss function is consistent of individual loss terms l_i, which return the negative log-likelihood of the ground truth label y_i given the model prediction $f[x_i, \phi]$ for training input x_i.
- For example, it might be the least squares loss $l_i = (f[x_i, \phi] - y_i)^2$.
- The goal is to find parameters $\hat{\phi}$ that minimize the loss.
- The simplest method is **gradient descent**.

Gradient Descent

- This starts with initial parameters: $\phi = [\phi_0, \phi_1, ..., \phi_N]^T$
  1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} & \frac{\partial L}{\partial \phi_1} & \cdots & \frac{\partial L}{\partial \phi_N} \end{bmatrix}^T$$

  2. Update the parameters: $\phi_{t+1} \leftarrow \phi_t - \alpha \frac{\partial L}{\partial \phi}$



Gradient Descent

- Consider a simple east squares loss example with two parameters:

$$\begin{aligned} L[\phi] &= \sum_{i=1}^{I} (f[x_i, \phi] - y_i)^2 \\ &= \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2 \end{aligned}$$

- The derivative of the loss function can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \phi} = \sum_{i=1}^{I} \frac{\partial l_i}{\partial \phi}$$

where

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}$$

Stochastic Gradient Descent

- In DNN, loss function is not that strict convex, which has unique **global minimum**.
- In other words, loss functions have numerous **local minima**.
- One issue of basic gradient descent algorithm is that,if we start in a position and use gradient descent to go downhill, there is no guarantee that we will wind up at the global minimum.
- One of the main problems is that the final destination of a gradient descent algorithm is entirely determined by the starting point.
- **Stochastic gradient descent** (SGD) attempts to remedy this problem by adding some noise to the gradient at each step. The solution still moves downhill on average, but at any given iteration, the direction chosen is not necessarily in the steepest downhill direction.

Stochastic Gradient Descent

- The mechanism for introducing randomness is simple.
- At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone.
- This subset is known as a minibatch or **batch** for short. The batches are usually drawn from the dataset without replacement.
- The update rule is

$$\phi_{t+1} \leftarrow \phi_t - \alpha \sum_{i \in B_t} \frac{\partial l_i[\phi_t]}{\partial \phi}$$

- The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again.
- A single pass through the entire training dataset is referred to as an **epoch**.

Stochastic Gradient Descent

- SGD adds noise to the process and helps prevent the algorithm from getting trapped in a sub-optimal region of parameter space.
- Each iteration is also computationally cheaper since it only uses a subset of the data.
- There are many advanced techniques to further improve the SGD: momentum, Adam, etc.

Backpropagation Algorithm

- Now, let us go back to DNN, which has far more complicated structure and parameters.
- The best way to understand is to go through an example.
- Consider a neural network with one input, one output, one hidden unit at each layer, and different activation functions sin, exp, cos:

$$f[x, \phi] = \beta_3 + w_0 cos[\beta_2 + w_2 exp[\beta_1 + w_1 sin[\beta_0 + w_0 x]]]$$

- We aim to compute a lot of derivatives with respect to each parameter $\phi = \{\beta_0, w_0, \beta_1, w_1, \beta_2, w_2, \beta_3, w_3\}$: $\frac{\partial l_i}{\partial \phi_j}$, where $\phi_j \in \phi$.

Backpropagation Algorithm

- We use chain rule to calculate those derivatives.

$$\begin{aligned}\frac{\partial \ell_i}{\partial \omega_0} &= -2\Big(\beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\big]\Big] - y_i\Big) \\ &\quad \cdot \omega_1\omega_2\omega_3 \cdot x_i \cdot \cos[\beta_0 + \omega_0 \cdot x_i] \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big] \\ &\quad \cdot \sin\Big[\beta_2 + \omega_2 \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big]\Big].\end{aligned}$$

- Such expressions are awkward to derive and code without mistakes and do not exploit the inherent redundancy; notice that the three exponential terms are the same.
- Backpropagation Algorithm is an efficient method for computing all of these derivatives. It consists of
  1. a forward pass, in which we compute and store a series of intermediate values and the network output
  2. a backward pass, in which we calculate the derivatives of each parameter, starting at the end of the network, and reusing previous calculations as we move toward the start.

Backpropagation Algorithm

Figure: Backpropagation forward pass.

Forward pass: We treat the computation of the loss as a series of calculations:

$$
\begin{aligned}
f_0 &= \beta_0 + \omega_0 \cdot x_i \\
h_1 &= \sin[f_0] \\
f_1 &= \beta_1 + \omega_1 \cdot h_1 \\
h_2 &= \exp[f_1] \\
f_2 &= \beta_2 + \omega_2 \cdot h_2 \\
h_3 &= \cos[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3 \\
\ell_i &= (f_3 - y_i)^2.
\end{aligned}
$$

Backpropagation Algorithm

Figure: Backpropagation forward pass.

Forward pass: We treat the computation of the loss as a series of calculations:

$$
\begin{aligned}
f_0 &= \beta_0 + \omega_0 \cdot x_i \\
h_1 &= \sin[f_0] \\
f_1 &= \beta_1 + \omega_1 \cdot h_1 \\
h_2 &= \exp[f_1] \\
f_2 &= \beta_2 + \omega_2 \cdot h_2 \\
h_3 &= \cos[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3 \\
\ell_i &= (f_3 - y_i)^2.
\end{aligned}
$$

Backpropagation Algorithm

Figure: Backpropagation backward pass 1.

Backward pass #1: We now compute the derivatives of ℓ_i with respect to these intermediate variables, but in reverse order:

$$\frac{\partial \ell_i}{\partial f_3}, \quad \frac{\partial \ell_i}{\partial h_3}, \quad \frac{\partial \ell_i}{\partial f_2}, \quad \frac{\partial \ell_i}{\partial h_2}, \quad \frac{\partial \ell_i}{\partial f_1}, \quad \frac{\partial \ell_i}{\partial h_1}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial f_0}. \tag{7.9}$$

The first of these derivatives is straightforward:

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y_i). \tag{7.10}$$

The next derivative can be calculated using the chain rule:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3} \frac{\partial \ell_i}{\partial f_3}. \tag{7.11}$$

Backpropagation Algorithm

$$
\begin{aligned}
\frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right).
\end{aligned}
$$

Backpropagation Algorithm

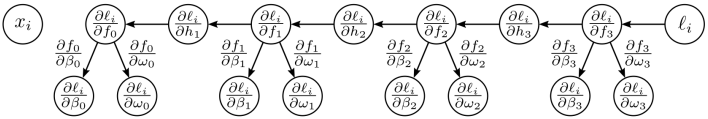


Figure: Backpropagation backward pass 2.

Backward pass #2: Finally, we consider how the loss ℓ_i changes when we change the parameters $\beta_\bullet$ and $\omega_\bullet$. Once more, we apply the chain rule (figure 7.5):

$$\begin{aligned}\frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\ \frac{\partial \ell_i}{\partial \omega_k} &= \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}.\end{aligned} \tag{7.13}$$

In each case, the second term on the right-hand side was computed in equation 7.12. When $k > 0$, we have $f_k = \beta_k + \omega_k \cdot h_k$, so:

$$\frac{\partial f_k}{\partial \beta_k} = 1 \qquad \text{and} \qquad \frac{\partial f_k}{\partial \omega_k} = h_k. \tag{7.14}$$

Backpropagation Algorithm

- Modern deep learning frameworks such as PyTorch and TensorFlow calculate the derivatives automatically, given the model specification.
- The backpropagation algorithm computes the derivatives that are used by stochastic gradient descent.
- There are many other things need to be considered. For example, how to initialize the parameters.
- Recall, $f_k = \beta_k + \Omega_k h_k$. Imagine we initialize all β_k to zero and elements of Ω_k according to a normal distribution with mean zero and variance σ^2.

Backpropagation Algorithm

- If variance σ^2 is very small, $f_k = \beta_k + \Omega_k h_k$ will be a weighted sum if h_k where the weights are very small.
- In addition, the ReLU function clips values less than zero, so the range of h_k will be half that of f_{k-1}.
- Consequently, the magnitudes of the pre-activations at the hidden layers will get smaller and smaller as we progress through the network.
- Similarly, if variance σ^2 is too large, values will get larger.
- This may leads to values that cannot be represented with finite precision floating point arithmetic.
- In the backward pass, the gradient magnitude may also decrease or increase uncontrollably.
- These cases are known as the **vanishing gradient problem** and the **exploding gradient problem** respectively.

References


Bishop, C. M. and Bishop, H. (2023).
Deep learning: Foundations and concepts.
Springer Nature.


Prince, S. J. (2023).
Understanding deep learning.
MIT press.