Deep Feedforward Networks

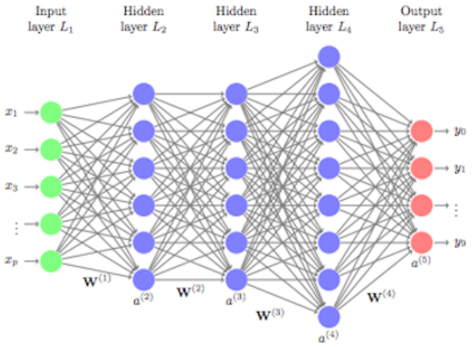
Zichao Yang

Zhongnan University of Economics & Law

Date: October 14, 2024

Deep Feedforward Networks

Deep Feedforward Network is the quintessential deep learning model. And it has many aliases: **feedforward neural network**, **multiplayer perceptrons**(MLPs), or simply **neural network**.



Source: Feedforward Deep Learning Models

Deep Feedforward Networks

The model in the above picture is called **feedforward** because there are no **feedback** connections in which outputs of the model are fed back into itself.

The overall length of the chain gives the **depth** of the model. The name *deep learning* arose from this terminology.

The dimensionality of these hidden layers determines the **width** of the model. The hidden layer is vector valued, and each element of the vector can be interpreted as playing a role analogous to a neuron.

It is best to think of feedforward networks as **function approximation machines** that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

Design A Feedforward Network

One way to understand feedforward networks is to understand how they can overcome the limitations of linear models.

To represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$. Then how to choose the nonlinear transformation ϕ?

- use a very generic ϕ, such as the radial basis function (RBF) kernel. If $\phi(\boldsymbol{x})$ has high enough dimensions, it can work well on the training set, but generalization to the test set often remains poor.
- manually engineer ϕ. It requires decades of human effort for each separate task, and with little transfer between domains.
- use deep learning models to learn ϕ. We parameterize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.

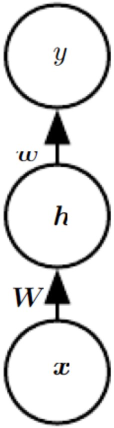
Design A Feedforward Network

(1) We need to make many of the same design decisions as for a linear model: choosing the optimizer, the cost function, and the form of the output units.

(2) Feedforward networks have introduced the concept of a hidden layer, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values.

(3) We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer.

Activation Functions



The vector of hidden units $\boldsymbol{h}$ are computed by:
$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{c}$.
The output layer: $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b) = \boldsymbol{w}^\intercal \boldsymbol{h} + b$

What function should $f^{(1)}$ be?
If $f^{(1)}$ is still linear, then the whole feedforward network will remain as a linear function of its input:
$f^{(2)}(f^{(1)}(\boldsymbol{x})) = (\boldsymbol{W}^\intercal \boldsymbol{x})^\intercal \boldsymbol{w} = \boldsymbol{x}^\intercal \boldsymbol{W} \boldsymbol{w} = \boldsymbol{x}^\intercal \boldsymbol{w}'$, where $\boldsymbol{w}' = \boldsymbol{W} \boldsymbol{w}$. (ignoring the intercept terms, $\boldsymbol{c}$ and b)

How to introduce nonlinearity into the system?
Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an **activation function**.

Activation Functions

Hence, we redefine $\boldsymbol{h}$ as:

$$\boldsymbol{h} = g(f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})) = g(\boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{c})$$

where g is the activation function.

In modern neural networks, the default recommendation for activation function is to use the **rectified linear unit**, or **ReLU**:

$$g(z) = \max\{0, z\}$$

We can now specify our complete network as:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\intercal \max\{0, \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{c}\} + b$$

Gradient-Based Learning

The nonlinearity of a neural network causes most interesting loss function to become non-convex, which means:

(1) Neural networks are usually trained by using **gradient-based optimizers** that merely drive the cost function to a very low value.

(2) Compares to:

(a) the linear equation solvers used to train linear regression models (minimum cost function guaranteed)

(b) the convex optimizatio algorithms used to train logistic regression or SVMs (global convergence guaranteed)

Gradient-Based Learning

Stochastic gradient descent applied to non-convex loss functions has no convergence guarantee, and is sensitive to the values of the initial parameters.

For feedforward neural networks, it is important to initialize all **weights** to **small random values**. The biases may be initialized to zero or to small positive values.

Cost Functions

In deep neural network models, we use the **cross-entropy** between the training data and the model's predictions as the cost function.

In information theory, cross-entropy is used to quantify the difference between two probability distributions. In the context of machine learning, it is used as a measure of error for classification problems. Cross-entropy is defined as:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\sum_{\boldsymbol{y}\in\text{classes}} \hat{p}_{\text{data}}(\boldsymbol{y}) \log p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}) \\ &= -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}) \end{aligned}$$

where $\hat{p}_{\text{data}}(\boldsymbol{y})$ is the true probability distribution we observed in the training set. $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ is your model's predicted probability distribution based on features $(\boldsymbol{x})$ and weights & biases $(\boldsymbol{\theta})$.

Cost Functions

The intuition behind the cross-entropy cost function is that the values generated by our model should be the most likely encountered values in the training set. And we add a negative sign so our goal is to minimize the cost function.

One recurring theme throughout natural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that **saturate** (become very flat) undermine this objective because they make the gradient become very small.

Vanishing Gradient Problem: during the training, the gradient magnitude typically is expected to decrease (or grow uncontrollably), slowing the training process. In the worst case, this may completely stop the neural network from further training.

Linear Units for Gaussian Output Distribution

The choice of cost function, $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$, is tightly coupled with the choice of output unit. Here we introduce three common output units.

One simple kind of output unit is based on an affine transformation with no nonlinearity. These are often called **linear units**.

$$p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{y};\hat{\boldsymbol{y}},\boldsymbol{I})$$

where $\hat{\boldsymbol{y}} = \boldsymbol{W}^\intercal\boldsymbol{x} + \boldsymbol{b}$, and $\boldsymbol{I}$ is a vector of variances chosen by users.

Linear units do not saturate, they pose little difficulty for gradient-based optimization algorithm and may be used with a wide variety of optimization algorithms.

Sigmoid Units for Bernoulli Output Distribution

The maximum likelihood approach for predicting the value of a binary variable y is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$.

To define a Bernoulli distribution, we utilize a **sigmoid** output unit:

$$\hat{y} = \sigma(\boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{b})$$

where σ is the logistic sigmoid function.

Suppose $z = \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{b}$, then the unnormalized probability distribution is: $\log \bar{P}(y) = yz \Rightarrow \bar{P}(y) = \exp(yz)$. Then we normalize it to yield a Bernoulli distribution:

$$\begin{aligned} P(y) &= \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)} \\ &= \sigma((2y-1)z) \end{aligned}$$

Sigmoid Units for Bernoulli Output Distribution

Then the loss function for maximum likelihood learning of a Bernoulli parameterized by a sigmoid is:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\log P(y|\boldsymbol{x}) \\ &= -\log \ sigma((2y-1)z) \\ &= \zeta((1-2y)z) \end{aligned}$$

Softmax Units for Multinoulli Output Distribution

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the **softmax** function.

First, a linear layer predicts unnormalized log probabilities:
$\hat{\boldsymbol{z}} = \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{b}$, where $z_i = \log \tilde{P}(y = i|\boldsymbol{x})$.

Then the softmax function exponentiate and normalize z to obtain the desired $\bar{\boldsymbol{y}}$, we want to achieve: (1) each element of $\hat{y}_i \in [0, 1]$; (2) the entire vector sums to 1.

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$$

Softmax Units for Multinoulli Output Distribution

$$\log \operatorname{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$$

The first term shows that the input z_i always has a direct contribution to the cost function. Because the term cannot saturate, we know that learning can proceed.

The name “softmax” can be confusing. The function is more closely related to the *argmax* function than the *max* function. The term “soft” derives from the fact that the softmax function is continuous and differentiable.

Hidden Units

The design of hidden units does not yet have many definitive guiding theoretical principles. Rectified linear unit (ReLU: $g(z) = \max\{0, z\}$) is an excellent default choice of hidden unit.

ReLU is not differentiable at all input points. But in practice it is not a big concern because:

- We do not expect training to actually reach a point where the gradient is $\mathbf{0}$.
- Even for the point $z = 0$, where the left derivative of z is 0, and the right one is 1, software usually return one of the one-sided derivatives rather than raising an error.

The important point is that in practice we can safely disregard the non-differentiability of the hidden unit activation functions.

Architecture Design

Most neural networks are organized into groups of units called **layers**.

Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by:

$$\boldsymbol{h}^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)\intercal}\boldsymbol{x} + \boldsymbol{b}^{(1)})$$

The second layer is given by:

$$\boldsymbol{h}^{(2)} = g^{(1)}(\boldsymbol{W}^{(2)\intercal}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)})$$

and so on.

Architecture Design

The universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as sigmoid, ReLU) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

TL;DR:
The universal approximation theorem means that **regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function**.

Architecture Design

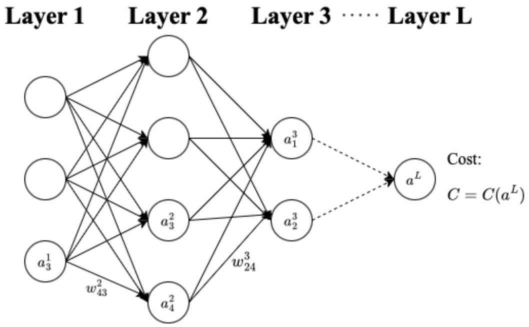
However, we are not guaranteed that the training algorithm will be able to **learn** that function, because:

- The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.
- The training algorithm might choose the wrong function due to overfitting.

In theory, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.

Empirically, greater depth does seem to result in better generalization for a wide variety of tasks.

Back-Propagation



Note: here we use a MLP model to demonstrate the back-propagation method, but back-propagation is widely used in different ML models.

a_j^l denotes the j^{th} neuron in the l^{th} layer.

w_{jk}^l denotes the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.

Each neuron contains two operations: sum & activation:
$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$, where $\sigma(\cdot)$ is the activation function, b_j^l is the bias.

Back-Propagation: Gradient Descent

Recall: in previous lecture, we talked about **gradient descent**, a method allows us to decrease the value of a function $f(\boldsymbol{x})$ by moving in the direction of the negative gradient, $-\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

Gradient descent method proposes a new point $\boldsymbol{x}'$ to minimize $f(\boldsymbol{x})$ based on:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

where ϵ is called as **learning rate**, a positive scalar determining the size of the step.

Back-propagation is built on this thought. And the function $f(\boldsymbol{x})$ we want to minimize here is the **cost/loss function** of the neural network, and $\boldsymbol{x} = [\boldsymbol{w}, \boldsymbol{b}]$.

Two Assumptions About The Cost Function

The goal of back-propagation is to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the cost function C with respect to any weight w or bias b in the network.

For back-propagation to work we need to make two main assumptions about the form of the cost function:

- The cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$ over cost function C_x for individual training examples, x.
- The cost function should be a function of the outputs from the neural network.

Four Fundamental Equations Behind Back-Propagation

We define the intermediate quantity $z_j^l \equiv \sum_k w_{jk}^l a_k^{l-1} + b_j^l$, and we call z^l the weighted input to the neurons in layer l.

Also, we define the error, δ_j^l, of neuron j in layer l by: $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$, hence we have the following two fundamental equations:

(1) rate of change of the cost w.r.t. any bias in the net-work:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \cdot 1 = \delta_j^l$$

(2) rate of change of the cost w.r.t. any weight in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \cdot a_k^{l-1} = a_k^{l-1} \delta_j^l$$

Four Fundamental Equations Behind Back-Propagation

Based on the chain rule, we can rewrite δ_j^l in a more well-defined format. Hence, we can get:

(3) For the output layer L, the equation for the error, δ_j^L, can be written as:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

The first term, $\frac{\partial C}{\partial a_j^L}$, measures how fast the cost is changing as a function of the j^{th} output activation.

The second term, $\sigma'(z_j^L)$, measures how fast the activation function $\sigma(\cdot)$ is changing at z_j^L.

Four Fundamental Equations Behind Back-Propagation

Proof: $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$?

Four Fundamental Equations Behind Back-Propagation

We can further rewrite the above equation in a matrix-based form, as:

$$\delta^L = \nabla_a C \odot \sigma'(z_j^L)$$

where $\nabla_a C$ is a vector whose components are $\frac{\partial C}{\partial a_j^L}, j = 1, 2, ...$

$\odot$ is called **Hadamard product**, and it denotes element-wise multiplication, for example:

$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1 \times 3\\2 \times 4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix}$$

Four Fundamental Equations Behind Back-Propagation

The last equation shows how to calculate the error, δ^l, in the hidden layers.

(4) for the in-between layer, l, we can back out its error, δ^l, using the error in the next layer, δ^{l+1}:

$$\delta^l = ((w^{l+1})^\top \delta^{l+1}) \odot \sigma'(z^l)$$

We can think intuitively of $(w^{l+1})^\top \delta^{l+1}$ as moving the error δ^{l+1} backward through the network, giving us some sort of measure of the error at the output of the l^{th} layer.

We then take the Hadamard product $\odot \sigma'(z^l)$. This moves the error backward through the activation function in layer l, giving us the error δ^l in the weighted input to layer l.

Four Fundamental Equations Behind Back-Propagation

Proof: $\delta^l = ((w^{l+1})^\top \delta^{l+1}) \odot \sigma'(z^l)$?

Summary: Four Fundamental Equations of Back-Propagation

$$\delta^L = \nabla_a C \odot \sigma'(z_j^L) \tag{1}$$

$$\delta^l = ((w^{l+1})^\top \delta^{l+1}) \odot \sigma'(z^l) \tag{2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{4}$$

The Back-Propagation Algorithm

1. **Input $\boldsymbol{x}$**: Set the corresponding activation a^1 for the input layer.

2. **Feedforward**: for each $l = 2, 3, ..., L$, compute $z^l = w^l a^{l-1} + b^l$ and $a^{l-1} = \sigma(z^{l-1})$.

3. **Output error δ^L**: compute the vector $\delta^L = \nabla_a C \odot \sigma'(z_j^L)$.

4. **Backpropagate the error**: For each $l = L-1, L-2, ...2$, compute $\delta^l = ((w^{l+1})^\top \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Claculate Gradients**: The gradient of the cost function is given by: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

6. **Update All Weights and Biases**: $w_{jk}^k = -\epsilon\frac{\partial C}{\partial w_{jk}^l}$ and $b_j^l = -\epsilon\frac{\partial C}{\partial b_j^l}$, where ϵ is the learning rate.

Back-Propagation

Let's see how the back-propagation works in python code!

Also you can check out these supplement materials:

1. 3Blue1Brown - What is backpropagation really doing?

2. Andrej Karpathy - The spelled-out intro to neural networks and backpropagation: building micrograd

3. Michael Nielsen - Chapter 2: How the backpropagation algorithm works

MLP Neural Network in Scikit-learn

In Scikit-Learn, you can build a MLP neural network using the MLPClassifier and MLPRegressor classes.

Take a look at the user guide and see how many hyperparameters you can fine tune in a neural network. And let's try it out in python.

Q: how to do model selection here?

Hint: we can still use the GridSearchCV function.

Class in Python

A simple example of defining a class in python.

```
class Dog:
species = "Canis familiaris"   # define class attribute

def __init__(self, name, age): # define instance attributes
self.name = name
self.age = age

# Instance method
def description(self):
return f"{self.name} is {self.age} years old"

# Another instance method
def speak(self, sound):
return f"{self.name} barks {sound}"

# Create an instance of Dog
my_dog = Dog("Ben Ben", 5)
```

Class in Python

Class Inheritance Illustration:

```
# Class Inheritance
class Puppy(Dog):
def __init__(self, name, age):
super().__init__(name, age)

def bark(self, sound="Woof"):
return self.speak(sound)

my_dog = Puppy('Lab', 6)

my_dog.species
my_dog.name
my_dog.age
my_dog.description()
my_dog.speak('Woof')
my_dog.bark('Yin')
```

Neural Network in PyTorch

We usually deploy deep learning models utilizing dedicated python libraries like PyTorch or Tensorflow.

In this class, we use PyTorch since nowadays it is the default deep learning library used in the research community and among developers working on new deep learning models.

PyTorch Library

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR), widely used for applications such as computer vision and natural language processing.

- **Ease of Use**: PyTorch offers a clear and intuitive syntax which makes it accessible for beginners in deep learning.
- **Pythonic Nature**: PyTorch allows developers and researchers to use standard Python features, which makes the process of building and testing neural networks more straightforward.
- **Dynamic Computation Graphs**: PyTorch uses dynamic computation graphs, which helps in automatically calculating gradients.

PyTorch Library

- **Strong GPU Acceleration**: PyTorch seamlessly integrates with CUDA, a parallel computing platform and application programming interface model created by Nvidia.
- **Robust Ecosystem**: PyTorch is supported by a large ecosystem of tools and libraries.
- **Community and Support**: PyTorch benefits from a strong community of developers and researchers who continuously contribute to the library's development.

Now, it is time to install PyTorch in your computer!

CUDA and GPU Computing

Compute Unified Device Architecture (CUDA) is a proprietary parallel computing platform and application programming interface developed by NVIDA.

CUDA is essential to GPU computing:

- **Parallel Processing Capabilities**: GPUs are highly efficient at handling multiple operations concurrently. CUDA provides the necessary tools and frameworks to harness this capability.
- **Optimized Hardware Utilization**: CUDA allows direct access to the GPU's virtual instruction set and parallel computational elements. CUDA-enabled computing is more efficient than CPU-only computing.

Drawback: CUDA is only available on certain NVIDA GPUs.

PyTorch Installation

Check CUDA compatibility.

Install PyTorch (CUDA or CPU). Go to the PyTorch website and follow the instruction.

PyTorch takes care of CUDA and cuDNN installations automatically.

Neural Network in PyTorch

(1) Enable GPU computing.

(2) Create a Custom Dataset for your own project.

(3) Build a Neural Network Model: model structure, activation function, loss function, optimizer.

(4) What is batch processing? What is an epoch? Source: Batch vs Epoch

(5) How to do hyperparameter tuning in PyTorch? optuna