Chapter 3: Fundamentals of Deep Learning

3.1 Introduction

In this chapter, we delve deeper into the fundamental concepts of deep learning, a specialized branch of machine learning. Deep learning is characterized by its capacity to learn from data through a hierarchical approach, emphasizing the creation of successive layers of increasingly meaningful and complex representations.

The history of deep learning is a riveting journey through the evolution of artificial intelligence. The concept of neural networks, which are at the core of deep learning, dates back to the 1940s and 1950s with the pioneering work of Warren McCulloch and Walter Pitts. They created a computational model for neural networks based on mathematics and algorithms called threshold logic. This model laid the foundation for what would become the field of deep learning.

Deep learning itself gained prominence in the 1980s when researchers developed algorithms to train multi-layer neural networks, leading to the development of the backpropagation algorithm by Geoffrey Hinton and others. However, it was not until the mid-2000s that deep learning began to emerge as a powerhouse in AI, thanks to the convergence of big data, improvements in computing power, and further algorithmic advances.

One of the significant milestones in deep learning was the development of Generative Adversarial Networks (GANs) by Ian Goodfellow and his colleagues in 2014. GANs consist of two neural networks—the generator and the discriminator—that are trained simultaneously through adversarial processes. The generator creates data that is indistinguishable from real data, while the discriminator tries to distinguish between real and generated data. This innovation has had a profound impact on the fields of computer vision, art generation, and more.

Graph Neural Networks (GNNs) represent another critical development, extending deep learning to the domain of graph data structures. GNNs process data in a manner that maintains the relationships inherent in graph structures, making them incredibly powerful for social network analysis, molecule structure modeling, and recommendation systems.

The introduction of the Transformer architecture in 2017 marked a significant leap in natural language processing. The Transformer, characterized by its self-attention mechanism, enabled the training of models on larger datasets more efficiently than previous sequence-to-sequence models. This architecture became the backbone of models like OpenAI's Generative Pretrained Transformer (GPT)

ML

Data + Label → supervised 监督式学习

Input (特徵) Features x_i = 向量

→ function　$w_1 = w_2 = \cdots w_n = 0$

$w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n = y$　$= 0$

Loss : $E(W) = \frac{1}{n} \sum (y_i - \hat{y})^2$　MSE

Mean square error

Goal : 使它尽可能的小 (Learning Training)

$\underline{\arg}\ \underline{\min}\ E(W)$

gradient decent 梯度下降法

$$W_{n+1} = W_n - \eta \cdot \nabla_{W_n} E$$

<Def> $dW = W_{n+1} - W_n$

$= \eta \cdot \nabla_W E$

<algorithm>

1. random initialize W_0
2. while $dW < \epsilon$ or $Itr > MaxItr$
   - → $E \rightarrow \nabla_W E \rightarrow dW$
   - → update $W_{n+1} = W_n + dW$

stop:

1. $dW < \epsilon$ ← small value
2. Total number of iteration
   > Max iteration

3. output and test (plot.)

<Example>

$y = w_1 x_1 + w_0$ with $w_0 \geq 0$

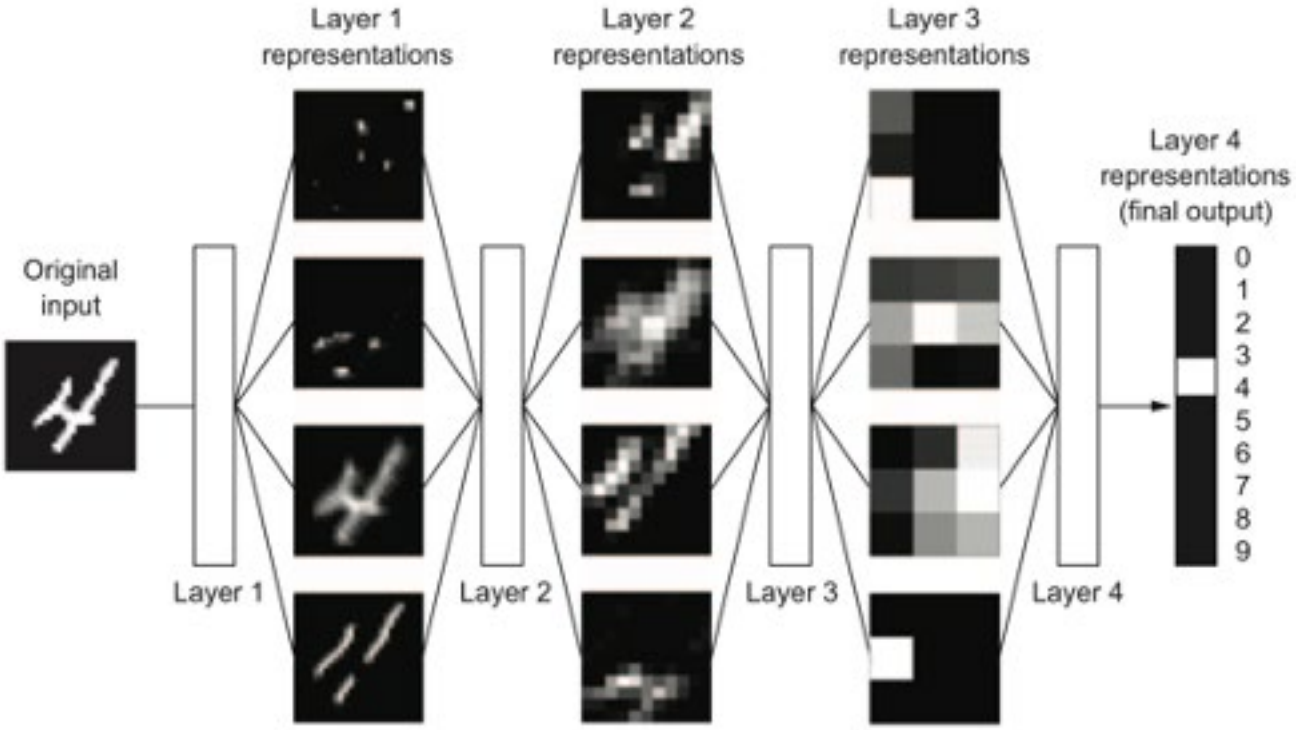
w_1 → slop

w_0 → Bias
-intercept.

series, which started with GPT in 2018 and evolved into more sophisticated versions. GPT models have been groundbreaking in generating human-like text and achieving state-of-the-art results in a variety of language tasks.

In recent years, a novel approach known as diffusion models has gained traction. These models iteratively refine data samples, starting from noise and gradually shaping them into coherent patterns or structures through a reverse diffusion process. This technique has shown remarkable results in generating high-quality images and has the potential to transform the generative modeling landscape.

The evolution of deep learning is characterized by a series of breakthroughs that have expanded the capabilities of neural networks. From simple beginnings to complex systems like GANs, GNNs, Transformers, GPTs, diffusion models, and Sora, deep learning continues to drive significant progress in AI, changing the way we interact with technology and offering a glimpse into the future of intelligent systems.



As illustrated in the figure, a neural network processes an image of a digit through a series of transformations. These transformations occur across multiple layers within the network, each layer abstracting the data further from its original form. In essence, the network distills the image into a hierarchy of features, from the most rudimentary to the most discriminative, that are instrumental for the task at hand—in this case, digit recognition.

Deep neural networks achieve this sophisticated input-to-target mapping by employing a cascade of simple but powerful data transformations, each corresponding to a different layer of the network. These transformations are not manually engineered; instead, they are learned from the data itself. Through exposure to numerous examples during the training process, the network fine-tunes its internal parameters. This tuning is directed by the principle of reducing the discrepancy between the network's predictions and the actual outcomes, iteratively improving the model's performance.

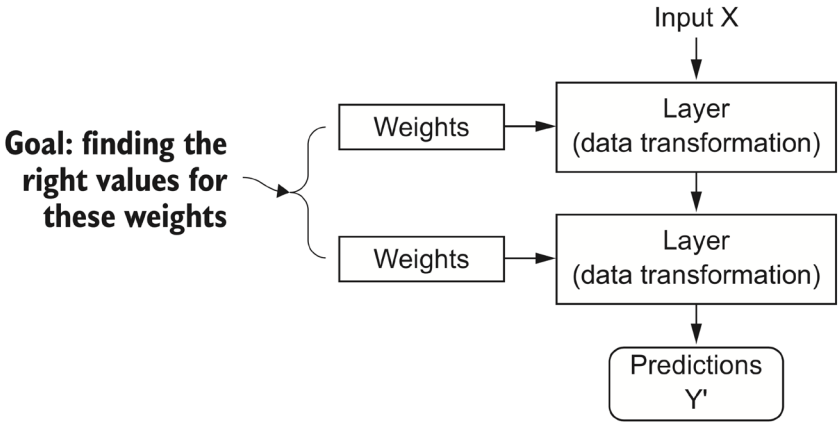
The process by which these networks learn is akin to distilling knowledge, where each layer captures and refines different aspects or features of the data. The initial layers might capture basic forms and edges, intermediate layers may identify more complex structures like textures or patterns, and the final layers synthesize this information to form a high-level understanding necessary for accurate decision-making. This end-to-end learning mechanism, from raw data to actionable insights, is what renders deep learning a powerful tool for a wide array of applications ranging from image and speech recognition to natural language understanding.

In the subsequent sections, we will explore the architecture of neural networks, the nature of the transformations between layers, and the algorithms that govern learning, all of which contribute to the effectiveness of deep learning models. Through this exploration, we will gain a comprehensive understanding of how deep learning models develop their intricate understanding of data and the world.

3.1.1 Understanding how deep learning works, in three figures

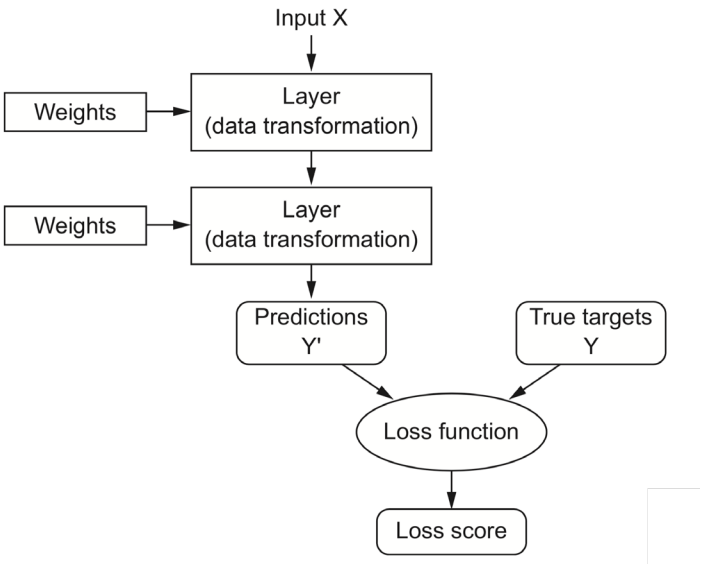
First Figure

The concept of a layer within a neural network and its operational mechanism is encapsulated by what is known as the **layer's weights**. These weights play a pivotal role in how the layer processes and transforms the input data it receives. In the fascinating world of neural networks, the process of "learning" is essentially the journey of meticulously searching for and identifying an optimal set of values for these weights across all layers within the network. This optimization is crucial because it directly influences the network's ability to accurately map, or translate, given example inputs into their correct associated outcomes or targets. Achieving this level of accuracy is the cornerstone of creating effective and reliable neural network models that can perform a wide range of tasks, from image recognition to predicting market trends. Thus, understanding and refining the weights of a neural network is at the heart of its learning process, ensuring that the network not only learns from its inputs but also improves its performance over time in mapping these inputs to their correct targets.



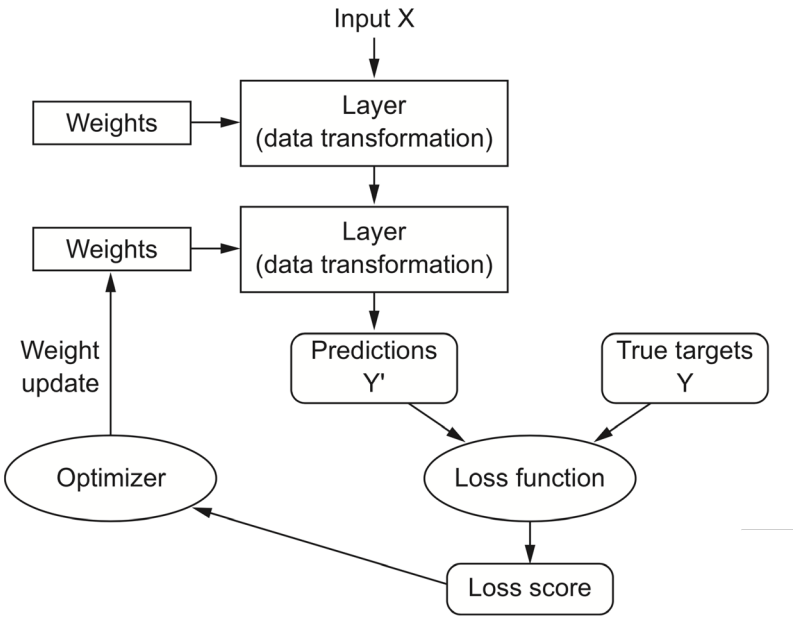
Second Figure

To effectively manage and direct the output of a neural network, it is crucial to have a means of assessing how closely the network's output aligns with the anticipated results. This critical role is fulfilled by what is known as the loss function of the network, which is alternatively referred to as the error function, cost function, or objective function. The essence of the loss function lies in its ability to take the network's predictions and compare them with the actual target values—the outcomes that were desired from the network. By doing so, the loss function calculates a distance score, which serves as a quantitative measure of the network's performance on a given example. This score is instrumental in capturing the degree of success or accuracy with which the network has managed to produce outputs that match the expected targets. Through this process, the loss function provides invaluable feedback that helps in guiding the training of the neural network, ensuring that its predictions become progressively more accurate over time.



Third Figure

The essential technique in deep learning involves utilizing the score produced by the loss function as a feedback mechanism. This mechanism adjusts the weights of the neural network slightly, aiming to decrease the loss score for the analyzed example. This crucial adjustment process is managed by a component known as the optimizer. The optimizer executes a key algorithm in deep learning, known as the **backpropagation algorithm**. This algorithm plays a pivotal role in the learning process by influencing the direction and magnitude of the weight adjustments. Backpropagation essentially allows the network to learn from its errors by optimizing the weights in such a way that the overall performance of the network improves over time. Later, we will delve deeper into the workings of backpropagation, providing a clearer understanding of its significance and how it facilitates the learning process within deep learning models.

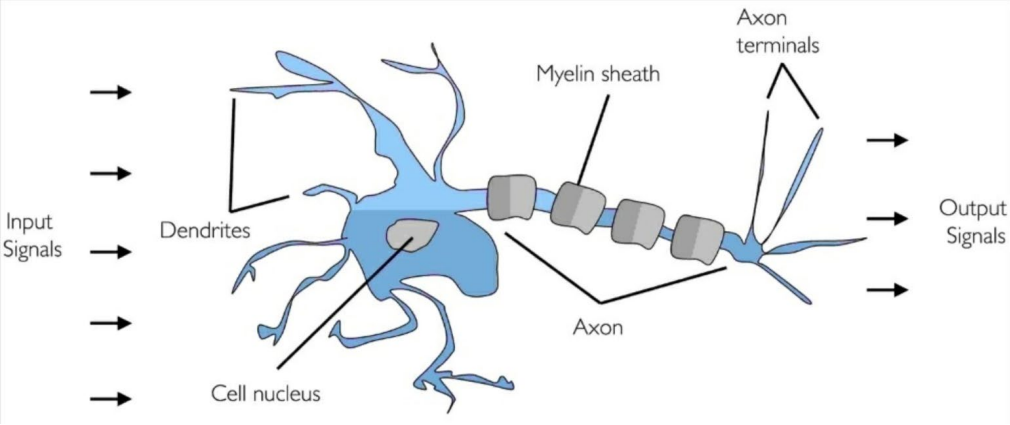


Remark: Initially, the network's weights are set to random values, leading to the network performing a sequence of random transformations. As a result, its output significantly deviates from the desired outcome, reflected in a high loss score. However, with each example processed, the network's weights undergo slight adjustments towards the correct direction, leading to a gradual decrease in the loss score. This process constitutes the training loop. When repeated enough times—often through tens of iterations across thousands of examples—this loop results in weight values that effectively minimize the loss function. Achieving minimal loss means the network's outputs closely match the intended targets, indicating a successfully trained network. This process, though straightforward in principle, when executed on a large scale, appears remarkably transformative, almost like magic.

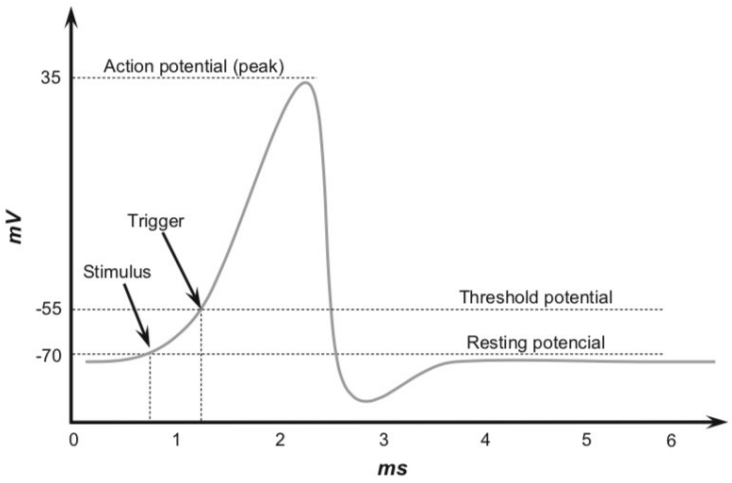
3.2 Artificial Neural Networks: The Perceptron (感知器) Network

Deep learning is essentially based on the framework of deep neural networks, and gaining a comprehensive understanding of deep learning necessitates a grasp of certain fundamental concepts and mathematical frameworks inherent to neural networks, also known as artificial neural networks (ANNs). This section aims to introduce ANNs, beginning with an exploration of one of the earliest ANN architectures, the Perceptron. Following that, we will delve into the more complex Multi-Layer Perceptrons (MLPs).

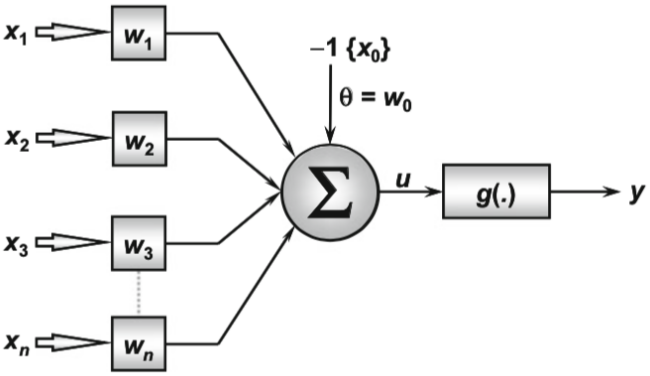
The Perceptron network represents one of the simplest forms of ANN architecture. It was developed in 1957 by Frank Rosenblatt, drawing inspiration from the biological structure and function of neurons. In the brain, neurons are interconnected nerve cells responsible for the processing and transmission of both chemical and electrical signals. This biological process is mirrored in the Perceptron network's design, where the goal is to simulate the way neurons activate and communicate with each other to process information. The foundational idea behind the Perceptron was to create a machine that could learn from and adapt to a set of inputs in a manner analogous to the learning and adaptive processes observed in biological neural networks.



When a nerve cell, or neuron, receives stimulation through an impulse that exceeds its activation threshold of -55 millivolts (mV), it is due to changes in the internal concentrations of sodium (Na^+) and potassium (K^+) ions. This change initiates an electrical impulse that propagates along the neuron's axon, reaching a peak amplitude of 35 mV. The process involves several stages of action potential changes, reflecting the neuron's excitation phases. These stages are crucial for the transmission of signals across the neural network, allowing neurons to communicate and perform complex processing tasks. The dynamics of these voltage variations during the neuron's excitation are depicted in the figure below, illustrating the intricate processes that underlie neural activity and signaling within the brain. This mechanism is fundamental to understanding how signals are transmitted in biological systems and provides a basis for mimicking these processes in artificial neural networks.



The Perceptron network is structured around a single artificial neuron, embodying one of the most basic forms of an artificial neural network. The architecture of a Perceptron is designed to handle n input signals, each input representing a different aspect of the problem at hand. These inputs feed directly into the neuron, which processes them to produce a single output. The illustration below provides a visual representation of a Perceptron network, showcasing how multiple inputs are integrated and processed by the neuron to yield a singular output. This design highlights the Perceptron's capability to make decisions or classifications based on the provided inputs, serving as a fundamental building block for more complex neural network architectures that comprise multiple layers and neurons.



In mathematical notation, the inner processing performed by the Perceptron can be described by the following expressions:

$$\begin{cases} u = \sum_{i=1}^{n} w_i x_i - \theta \\ y = g(u) \end{cases}$$

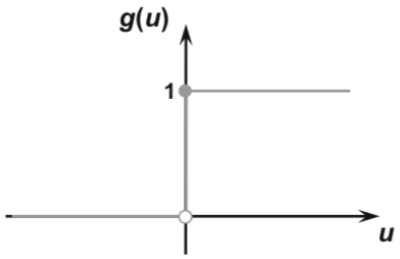
where x_i is a network input, w_i is the weight associated with the i^{th} input, θ is the activation threshold (or bias), $g(.)$ is the activation function and u is the activation potential or sometimes called input signal.

The activation function, *g(.)*, is crucial as it determines how input signals are converted into an output. The most employed activation functions in Perceptrons include the step function and the bipolar step function. These functions are binary in nature; the step function outputs either a 0 or 1, while the bipolar step function outputs -1 or 1, depending on whether the activation potential u crosses the threshold θ. This binary output mechanism enables the Perceptron to make clear distinctions or classifications based on the processed input signals.

The most common activation functions used in Perceptrons are the step and bipolar step functions:

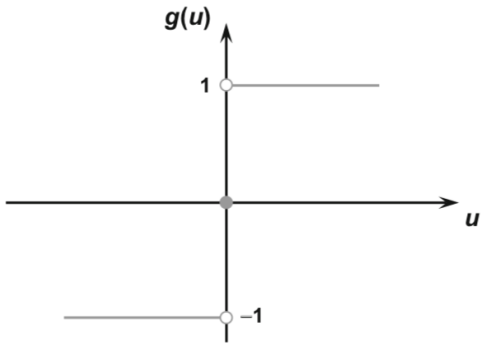
Step Function

$$g(u) = \begin{cases} 0 & \text{if } u < 0 \\ 1 & \text{if } u \geq 0 \end{cases}$$



Bipolar Step Function

$$g(u) = \begin{cases} -1 & \text{if } u < 0 \\ 1 & \text{if } u \geq 0 \end{cases}$$



Q: What are the possible output values possibly produced by the Perceptron?

A:

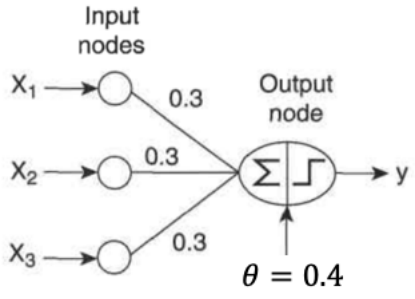
The Perceptron network serves as a foundational model for simple linear binary classification tasks. It operates by calculating a linear combination of its inputs, and based on this computation, it determines whether the result surpasses a certain threshold. If the outcome exceeds this threshold, the Perceptron outputs the positive class; otherwise, it outputs the negative class. This mechanism is akin to how a linear Support Vector Machine (SVM) functions, making it effective for binary decision-making processes.

Training a Perceptron involves adjusting and fine-tuning its parameters to achieve optimal performance. Specifically, this training process focuses on identifying the appropriate values for the weights w_i associated with each input and the threshold θ that dictates the decision boundary. The goal of this optimization is to ensure that the Perceptron can accurately distinguish between the two classes based on the input it receives, thereby enhancing its ability to perform binary classification tasks effectively. This is achieved through iterative updates to the weights and threshold, guided by feedback from the Perceptron's performance on training data.

Example: Consider a Perceptron network shown below. A perceptron computes its output value, y, by performing a weighted sum on its inputs, subtracting a threshold from the sum, and then examining the sign of the result. The model has three input nodes, each of which has an identical weight of 0.3 to the output node and a threshold of 0.4. These values can be found by training covered in the sequel.

X_1	X_2	X_3	y
1	0	0	−1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	−1
0	1	0	−1
0	1	1	1
0	0	0	−1

(a) Data set.



(b) Perceptron.

Q: Construct the model.
A:

3.2.1 Training Process of the Perceptron

The adjustment of Perceptron's weights and thresholds, in order to classify patterns that belong to one of the two possible classes, is performed by the use of Hebb's learning rule (Hebb 1949).

he Perceptron's ability to classify patterns into one of two possible classes hinges on adjusting its weights and thresholds, a process guided by Hebb's learning rule, formulated in 1949. This rule posits a simple yet effective mechanism for learning: if the Perceptron's output matches the desired output, its weights and threshold remain the same, acting under an inhibitory condition. Conversely, if the output differs from the expected value, the weights and threshold are modified proportionally to the input signals. This adjustment is carried out sequentially across all training samples until the Perceptron's output aligns closely with the desired output for every sample.

In mathematical notation, the rules for adjusting the weights w_i and threshold θ can be expressed, respectively, by the following equations:

$$w_i^{\text{current}} = w_i^{\text{previous}} - \eta \cdot \left(y - d^{(k)}\right) \cdot x_i^{(k)}$$

$$\theta^{\text{current}} = \theta^{\text{previous}} - \eta \cdot \left(y - d^{(k)}\right) \cdot (-1)$$

These two equations can be represented by a single vector expression given by:

$$\boldsymbol{w}^{\text{current}} = \boldsymbol{w}^{\text{previous}} - \eta \cdot \left(y - d^{(k)}\right) \cdot \boldsymbol{x}^{(k)}$$

In algorithmic notation, the inner processing performed by the Perceptron can be described by the following expression:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \cdot \left(y - d^{(k)}\right) \cdot \boldsymbol{x}^{(k)}$$

where

$\boldsymbol{w} = [\theta \quad w_1 \quad w_2 \quad \cdots \quad w_n]$ is the vector containing the threshold and weights;
$\boldsymbol{x}^{(k)} = \left[-1 \quad x_1^{(k)} \quad x_2^{(k)} \quad \cdots \quad x_n^{(k)}\right]$ is the k^{th} training sample;
$d^{(k)}$ is the desired value for the k^{th} training sample;
η is a constant that defines the learning rate of the Perceptron.

The learning rate η determines how fast the training process will take to its convergence (stabilization). The choice of η should be done carefully to avoid instabilities in the training process, and it is usually defined within the range $0 < \eta \leq 1$.

3.2.2 Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let's now implement it in Python and apply it to the Iris dataset.

Giving Computers the Ability to Learn from Data.

We will take an object-oriented approach to defining the perceptron interface as a Python class, which will allow us to initialize new Perceptron objects that can learn from data via a fit method and make predictions via a separate predict method. As a convention, we append an underscore (_) to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

The following is the interface of a perceptron in Python, you will need to finish the implementation in our lab:

```
import numpy as np
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    errors_ : list
      Number of misclassifications (updates) in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the number of
          examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
```

```
        self : object

        """
        …… to be finished in lab
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate, eta, and the number of epochs, `n_iter` (passes over the training dataset).

Via the fit method, we initialize the weights in `self.w_` to a vector, , where m stands for the number of dimensions (features) in the dataset, and we add 1 for the first element in this vector that represents the bias unit. Remember that the first element in this vector, `self.w_[0]`, represents the so-called bias unit that we discussed earlier.

Also notice that this vector contains small random numbers drawn from a normal distribution with standard deviation 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

It is important to keep in mind that we don't initialize the weights to zero because the learning rate, η (eta), only influences the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to zero, the learning rate parameter, eta, affects only the scale of the weight vector, not the direction. After the weights have been initialized, the fit method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule that we discussed in the previous section.

The class labels are predicted by the `predict` method, which is called in the `fit` method during training to get the class label for the weight update; but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $w^T x$.

Python Example: Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will restrict the following analyses and examples in the remainder of this chapter to two feature variables (dimensions). Although the perceptron rule is not restricted to two dimensions, considering only two features, sepal length and petal length, will allow us to visualize the decision regions of the trained model in a scatter plot for learning purposes.

Note that we will also only consider two flower classes, Setosa and Versicolor, from the Iris dataset for practical reasons—remember, the perceptron is a binary classifier. However, the perceptron algorithm can be extended to multi-class classification—for example, the one-vs.-all (OvA) technique.

OvA, which is sometimes also called one-vs.-rest (OvR), is a technique that allows us to extend any binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the examples from all other classes are considered negative classes. If we were to classify a new, unlabeled data instance, we would use our n classifiers, where n is the number of class labels, and assign the class label with the highest confidence to the particular instance we want to classify. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

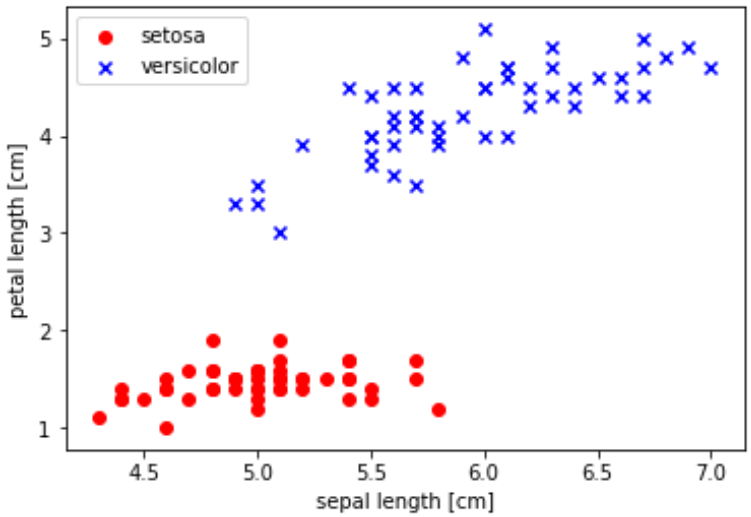
The `scikit-learn` has `Perceptron` class implemented in the `linear_model` module. Let us use parts of `Iris` dataset to demonstrate its simple usage for binary classification. Our goal is to use `sepal` and `petal` lengths to classify the flowers for `Setosa` or `Versicolor`:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris.data[0:100, (0, 2)] # sepal length, petal length
y = iris.target[0:100] # Setosa or Versicolor
```

```
# plot data
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')
```




```
per_clf = Perceptron(random_state=42) #default learning rate = 1.0
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1)
per_clf.fit(X_train,y_train)

y_pred = per_clf.predict(X_test)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
```

```
Misclassified samples: 0
```

The threshold and weights can be found via:

```
print ("threshold: ", per_clf.intercept_)
print ("w1: ", per_clf.coef_[0,0])
print ("w2: ", per_clf.coef_[0,1])
```

```
threshold:  [-0.01]
w1:  -0.027
w2:  0.052
```

You can download the above Python codes `perceptron.ipynb` from the course website.

3.3 Artificial Neural Networks: Multi-Layer Perceptrons (MLPs)

Multilayer Perceptron (MLP) network features, at least, **one intermediate (hidden) neural layer**, which is placed between the input layer and the output layer. Consequently, MLP networks have at least two neural layers, and their neurons are distributed among the intermediate and output layers.

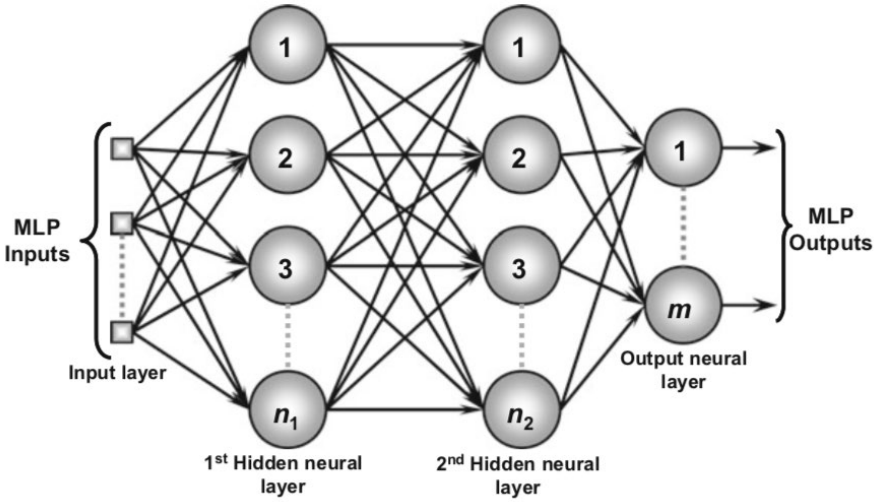
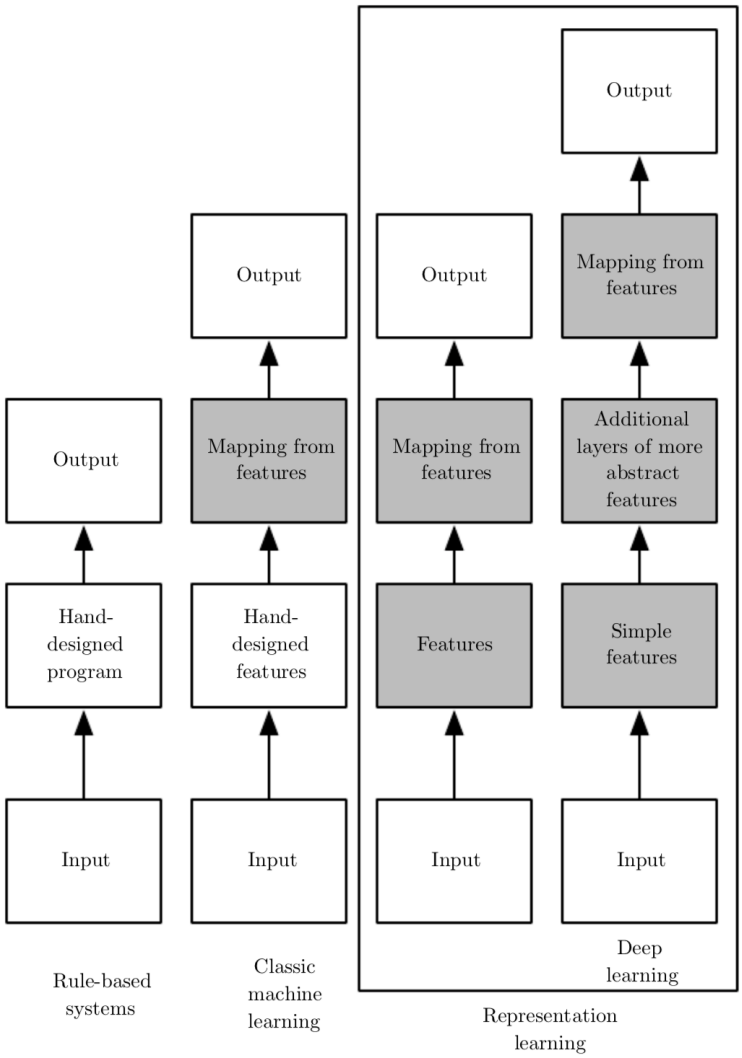


Illustration of a multilayer perceptron network

The hidden layers can be viewed as internal representations or **features** of the raw data. We have long been interested in finding a way to train multilayer artificial neural networks (ANNs) that could **automatically** discover good "internal representations," i.e. features that make learning easier and more accurate. This is called **representation learning** and features can be thought of as the stereotypical input to a specific ANN node that activates that node (i.e. causes it to output a positive value near 1). Since a node's activation is dependent on its incoming weights and bias, we say a node has learned a feature if its weights and bias cause that node to activate when the feature is present in its input.

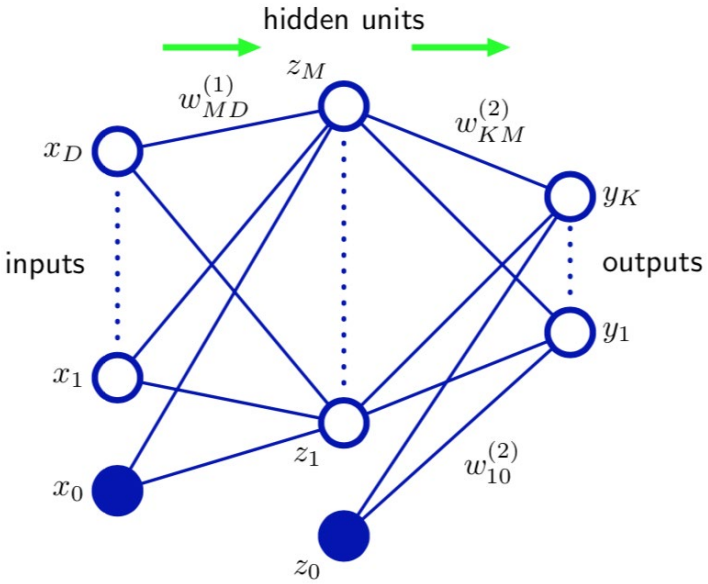
Figure below gives a high-level schematic of rule-based learning, classical machine learning and representation learning. Shaded boxes indicate components that are able to **learn from data**.



3.3.1 Feed-forward Network Functions

MLP network belongs to a **feed-forward network**. Let us consider a network diagram with one hidden layer below. **The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes**, in which the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. Green arrows denote the direction of information flow through the network during forward propagation.

The network model can be described as **a series of functional transformation**. First, we construct M linear combinations of the input variables $x_1, \cdots, x_D$ in the form:



$$u_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

where $j = 1, \cdots, M$ and the superscript **(1)** indicates that **the corresponding parameters are in the first 'layer' of the network**. The parameters $w_{ji}^{(1)}$ are the weights, the parameter $w_{j0}^{(1)}$ is the bias and $u_j^{(1)}$ are the activation potentials for the first layer. The equation is often expressed in a more compact form:

(3.1) $$u_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$

by absorbing the bias parameter and setting $x_0 = 1$.

The quantities $u_j^{(1)}$ are activation potentials (or simply activation). Each of them is then transformed using a differentiable, nonlinear activation function $h(.)$ such as sigmoid to give:

(3.2) $$z_j^{(1)} = h(u_j^{(1)})$$

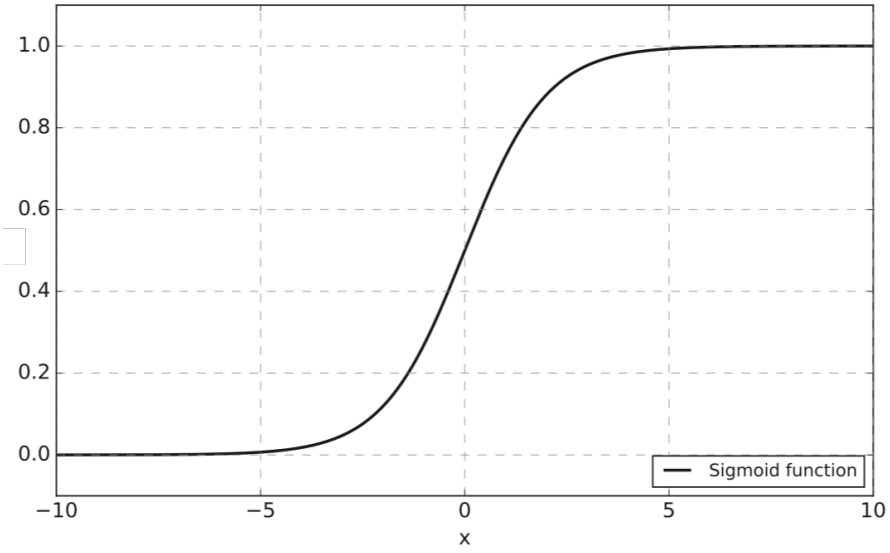
Remark: a sigmoid function is defined as:

$$g(x) = \frac{e^x}{1 + e^x}$$

Or sometimes express as:

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

A plot of this function is given in the figure below.



Q: what is the implication when we do Eq. (3.2) using a sigmoid function?
A:

These values $z_j^{(1)}$ are again linearly combined to give output unit activations:

$$u_k^{(2)} = \sum_{j=1}^{M} w_{kj}^{(2)} z_j^{(1)} + w_{k0}^{(2)}$$

where $k = 1, \cdots, K$ and K is the total number of outputs. Again the superscript $(\mathbf{2})$ indicates that **the corresponding parameters are in the second 'layer' of the network** and the transformation corresponds to the second layer of the network (in our case, the output layer). The equation is often expressed in a more compact form:

(3.3) $$u_k^{(2)} = \sum_{j=0}^{M} w_{kj}^{(2)} z_j^{(1)}$$

by absorbing the bias parameter and setting $z_0 = 1$.

Finally, the output unit activations are transformed using an appropriate activation function to give a set of network outputs y_k.

$$y_k = g(u_k^{(2)})$$

We can combine these various stages to give the overall network function that takes the form:

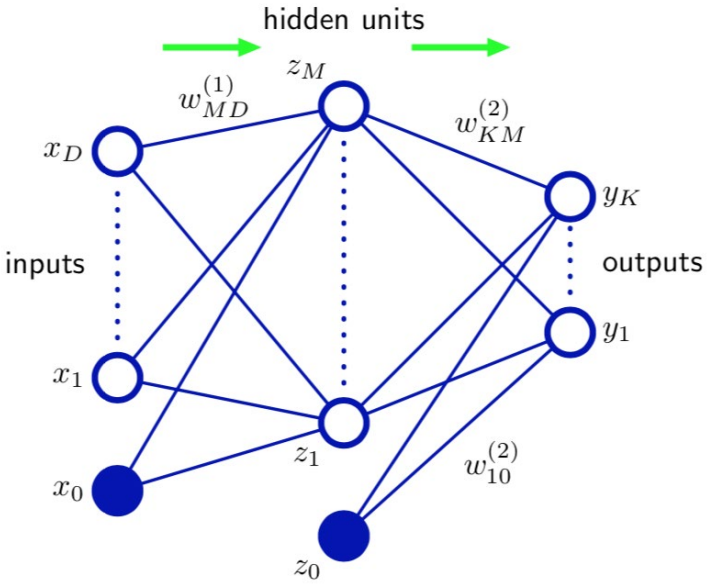
$$y_k(\mathbf{x}, \mathbf{w}) = g(\sum_{j=1}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}) + w_{k0}^{(2)})$$

Or in a more compact form:

(3.4) $$y_k(\mathbf{x}, \mathbf{w}) = g(\sum_{j=0}^{M} w_{kj}^{(2)} h(\sum_{i=0}^{D} w_{ji}^{(1)} x_i))$$

by absorbing the bias parameters and setting $x_0 = z_0 = 1$.

3.3.2 Network Training



So far, we have viewed neural networks as a general class of **parametric nonlinear functions** from a vector $\mathbf{x}$ of input variables $\mathbf{x} = [1 \quad x_1 \quad x_2 \quad \cdots \quad x_D]$ to a vector $\mathbf{y}$ of output variables $\mathbf{y} = [y_1 \quad y_2 \quad \cdots \quad y_K]$.

Q: where does the nonlinearity come from?
A:

A simple approach to determine the network parameters is to minimize a sum-of-squares error function. Given a training set comprising a set of input samples $\{\mathbf{x}_n\}$, where $n = 1, \cdots, N$, together with a corresponding set of **target** vectors $\{\mathbf{t}_n\}$, we minimize the error function:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} (\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n)^2 \tag{3.5}$$

where $E_n(\mathbf{w})$ is the error from an individual sample n and $E(\mathbf{w})$ is called **the error function** (or loss function, cost function). The goal of the learning algorithm is to determine a set of weights $\mathbf{w}$ that minimize $E(\mathbf{w})$.

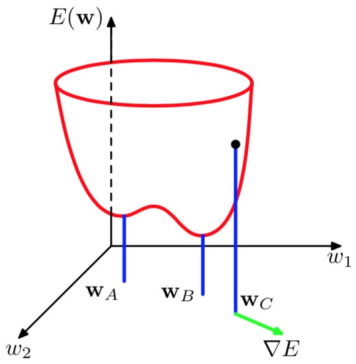
$E(\mathbf{w})$ is a smooth continuous function of $\mathbf{w}$. Thus, its smallest value will occur at a point in weight space such that the gradient of the error function vanishes, so that $\nabla E(\mathbf{w}) = 0$. Because there is clearly no hope of finding an analytical solution to the equation $\nabla E(\mathbf{w}) = 0$ we resort to iterative numerical procedures.

The gradient descent method is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, **one takes steps proportional to the negative of the gradient**. In our context, the weight update formula used by the gradient descent method can be written as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla E(\mathbf{w})$$

where $0 < \eta < 1$ is a constant that defines the learning rate.

The geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space is illustrated in figure below. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.

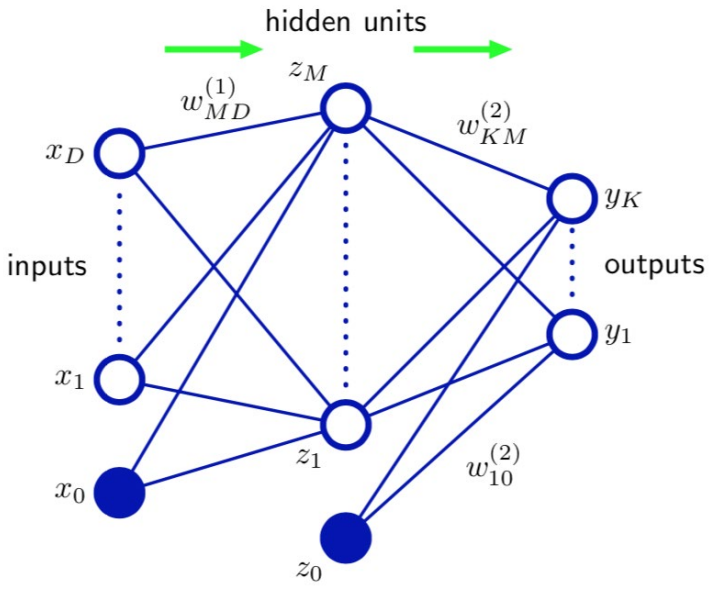


2.3.3 Theoretical Minimum: Backpropagation Algorithm

Motivation

The gradient descent method can be used to learn the weights of the output and hidden nodes of a neural network. For hidden nodes, the computation is not trivial.

Q: Why it is non-trivial?
A:



Remark: This coupling of parameters between layers can make the math quite messy (primarily as a result of using the product rule, discussed below), and if not implemented cleverly, can make the final gradient descent calculations slow. **Backpropagation addresses both of these issues by simplifying the mathematics of gradient descent, while also facilitating its efficient calculation.**

Evaluation of error-function derivatives

We now derive the backpropagation algorithm for a general network having **arbitrary** feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function.

Recall the error function:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

where $E_n(\mathbf{w})$ is the error from an individual sample n in a training set with N samples.

Here we shall consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such sample in the error function. This may be used directly for on-line training and optimization, or the results can be accumulated over the training set in the case of batch methods.

Consider the evaluation of the derivative of E_n with respect to a weight $w_{ji}^{(k)}$. The derivation of the backpropagation algorithm begins by applying the chain rule to the error function partial derivative:

(3.6)
$$\frac{\partial E_n}{\partial w_{ji}^{(k)}} = \frac{\partial E_n}{\partial u_j^{(k)}} \frac{\partial u_j^{(k)}}{\partial w_{ji}^{(k)}}$$

where $u_j^{(k)}$ is the activation (product-sum plus bias) of node j in layer k before it is passed to the nonlinear activation function (in most of the cases, the sigmoid function) to generate the output.

Q: Explain the meaning of Equation (3.6).
A:

The first term in Equation (3.6) is usually called the **sensitivity** for layer k, which is the sensitivity (gradient) of the error E_n with respect to the activation $u_j^{(k)}$. It is denoted:

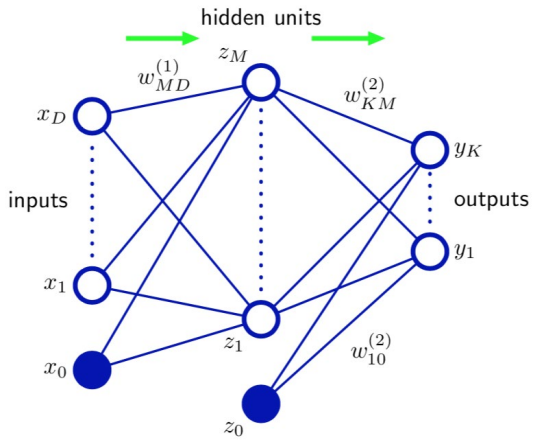
(3.7)
$$\delta_j^{(k)} \equiv \frac{\partial E_n}{\partial u_j^{(k)}}$$

The second term in Equation (3.6) can be calculated by utilizing Equation (3.1) for $u_j^{(k)}$ above:

(3.8)
$$\frac{\partial u_j^{(k)}}{\partial w_{ji}^{(k)}} = \frac{\partial}{\partial w_{ji}^{(k)}} \left(\sum_{i=0}^{R^{(k-1)}} w_{ji}^{(k)} o_i^{(k-1)} \right) = o_i^{(k-1)}$$

where $R^{(k)}$ is the number of nodes in layer k and $o_i^{(k)}$ is (the output for) node i in layer k.

Q: What are the $R^{(k-1)}$ and $o_i^{(k-1)}$ in Equation (3.8) related to the following simple network diagram?



A:

Thus, the partial derivative of the error function E_n with respect to a weight $w_{ji}^{(k)}$ is:

$$\frac{\partial E_n}{\partial w_{ji}^{(k)}} = \delta_j^{(k)} o_i^{(k-1)} \tag{3.9}$$

Remark: the partial derivative of a weight is a product of the sensitivity term $\delta_j^{(k)}$ at node j in layer k, and the output $o_i^{(k-1)}$ of node i in layer $k-1$. This makes intuitive sense since the weight $w_{ji}^{(k)}$ connects the output of node i in layer $k-1$ to the input of node j in layer k in the network graph.

The Output Layer

Starting from the final output layer, backpropagation attempts to define the value $\delta_j^{(m)}$, where m is the final layer. Expressing the error function $E_n(\mathbf{w})$ of the node j in terms of the value $u_j^{(m)}$ gives

$$E_n(\mathbf{w}) = \frac{1}{2}(y_j - t_j)^2 = \frac{1}{2}\left(g(u_j^{(m)}) - t_j\right)^2 \tag{3.10}$$

where $g(.)$ is the activation function for the output layer. Thus, recalling Equation (3.7) and applying the partial derivative and using the chained rule gives

$$\delta_j^{(m)} = \frac{\partial E_n}{\partial u_j^{(m)}} = \left(y_j - t_j\right) g'(u_j^{(m)}) \tag{3.11}$$

Remark: $g'(u_j^{(m)})$ indicates that the activation function for the backpropagation needs to be **differentiable**.

Putting it all together, the partial derivative of the error function E_n with respect to a weight $w_{ji}^{(m)}$ in the final layer is:

$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \left(y_j - t_j\right) g'(u_j^{(m)}) o_i^{(k-1)} \tag{3.12}$$

The Hidden Layer

Now the question arises of how to calculate the partial derivatives of layers other than the output layer. Luckily, the chain rule for multivariate functions comes to the rescue again. Observe the following equation for the error term $\delta_j^{(k)}$ in layer $1 \leq k < m$:

$$\delta_j^{(k)} = \frac{\partial E_n}{\partial u_j^{(k)}} = \sum_{l=1}^{R^{(k+1)}} \frac{\partial E_n}{\partial u_l^{(k+1)}} \frac{\partial u_l^{(k+1)}}{\partial u_j^{(k)}} \tag{3.13}$$

where l ranges from 1 to $R^{(k+1)}$ (the number of nodes in the next layer). Note that, because the bias input $o_0^{(k)}$ corresponding to $w_{j0}^{(k+1)}$ is fixed to 1, its value is not dependent on the outputs of previous layers, and thus l does not take on the value 0.

Plugging in the error term $\delta_l^{(k+1)}$ gives the following equation:

$$\delta_j^{(k)} = \sum_{l=1}^{R^{(k+1)}} \delta_l^{(k+1)} \frac{\partial u_l^{(k+1)}}{\partial u_j^{(k)}} \tag{3.14}$$

Recalling the definition of $u_l^{(k+1)}$

(3.15) $$u_l^{(k+1)} = \sum_{j=1}^{R^{(k)}} w_{lj}^{(k+1)} h(u_j^{(k)})$$

where $h(.)$ is the activation function for the hidden layer. Thus,

(3.16) $$\frac{\partial u_l^{(k+1)}}{\partial u_j^{(k)}} = w_{lj}^{(k+1)} h'(u_j^{(k)})$$

Plugging (3.16) into the (3.14) yields a final equation for the error term $\delta_j^{(k)}$ in the hidden layers, called **the backpropagation formula**:

(3.17) $$\delta_j^{(k)} = \sum_{l=1}^{R^{(k+1)}} \delta_l^{(k+1)} w_{lj}^{(k+1)} h'(u_j^{(k)}) = h'(u_j^{(k)}) \sum_{l=1}^{R^{(k+1)}} w_{jl}^{(k+1)} \delta_l^{(k+1)}$$

Now consider the evaluation of the derivative of E_n with respect to a weight $w_{ji}^{(k)}$ in the hidden layer:

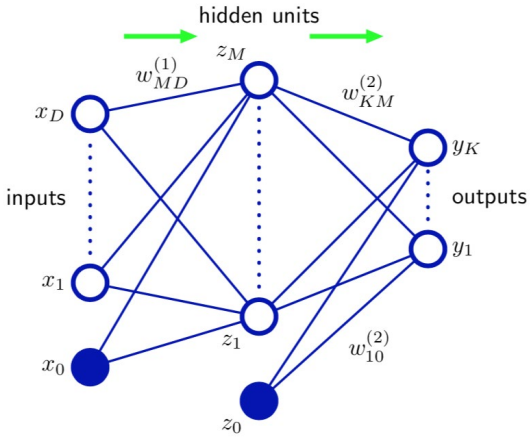
(3.18) $$\frac{\partial E_n}{\partial w_{ji}^{(k)}} = \frac{\partial E_n}{\partial u_j^{(k)}} \frac{\partial u_j^{(k)}}{\partial w_{ji}^{(k)}} = \delta_j^{(k)} \frac{\partial u_j^{(k)}}{\partial w_{ji}^{(k)}}$$

Q: what is $\frac{\partial u_j^{(k)}}{\partial w_{ji}^{(k)}}$?

A:

Remark: the calculation of the sensitivity term $\delta_j^{(k)}$ in the hidden layer is dependent on the values of sensitivity terms $\delta_l^{(k+1)}$ in the next layer. Thus, **computation of the sensitivity terms will proceed backwards from the output layer down to the input layer**. This is where **backpropagation** gets its name.

Simple Conceptual Example



Let us consider a simple two-layer network of the form illustrated in figure above, together with a sum-of-squares error, in which the output units have linear activation functions, so that $y_k = u_k$, while the hidden units have another famous logistic sigmoid activation functions given by:

$$h(u) = \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$

A useful feature of this function is that its derivative can be expressed in a particular simple form:

$$h'(u) = 1 - h(u)^2$$

We also consider a standard sum-of-squares error function, so that for an individual sample *n* the error is given by (3.10):

$$E_n(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$$

Q: what is the sensitivity $\delta_k^{(2)}$ of each output unit?
A:

Q: what is the sensitivity $\delta_j^{(1)}$ of each hidden unit?

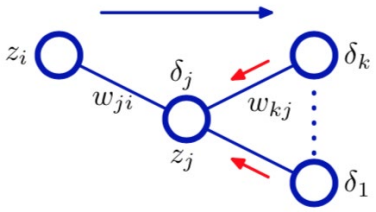
A:

Q: what are $\frac{\partial E_n}{\partial w_{ji}^{(1)}}$ and $\frac{\partial E_n}{\partial w_{kj}^{(2)}}$?

A:

Discussion: Backpropagation as Backwards Computation

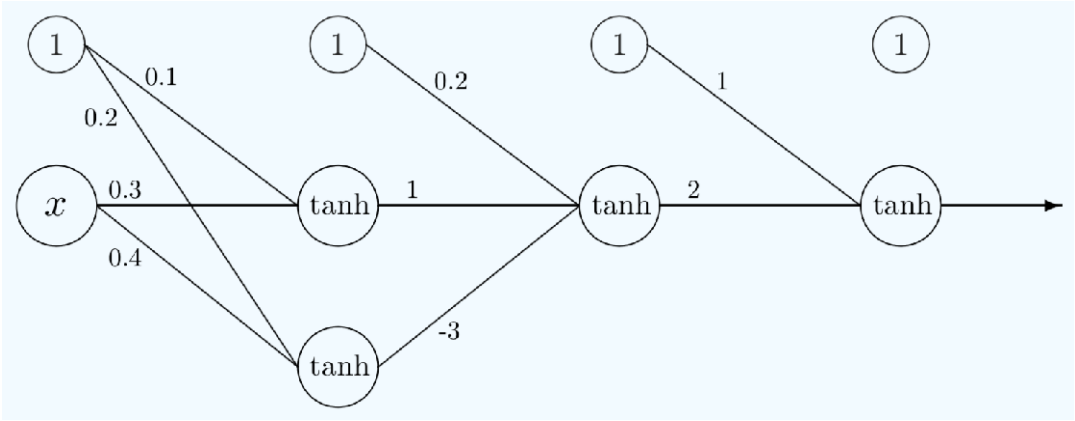
Equation (3.17) is where backpropagation gets its name. Namely, the sensitivity δ_j^k at layer k is dependent on the sensitivity δ_l^{k+1} at the next layer $k+1$. Thus, sensitivity (gradient of error with respect to activation) flows backward, from the last layer to the first layer. All that is needed is to compute the first sensitivity terms based on the computed output y_j and target output t_j. Then, the sensitivity terms for the previous layer are computed by performing a product sum (weighted by w_{jl}^{k+1}) of the sensitivity terms for the next layer and scaling it by $h'(u_j^k)$, repeated until the input layer is reached.



This backwards propagation of errors is very similar to the forward computation that calculates the neural network's output. Thus, **calculating the output is often called the forward phase while calculating the error terms and derivatives is often called the backward phase**. While going in the forward direction, the inputs are repeatedly recombined from the first layer to the last by product sums dependent on the weights w_{ij}^k and transformed by nonlinear activation functions $h(.)$ and $g(.)$. In the backward direction, the "inputs" are the final layer's error terms, which are repeatedly recombined

from the last layer to the first by product sums dependent on the weights w_{jl}^{k+1} and transformed by nonlinear scaling factors $h'(.)$ and $g'(.)$.

Example: Let us consider a very simple network with a single input $x = 2$ and a single output $y = 1$.

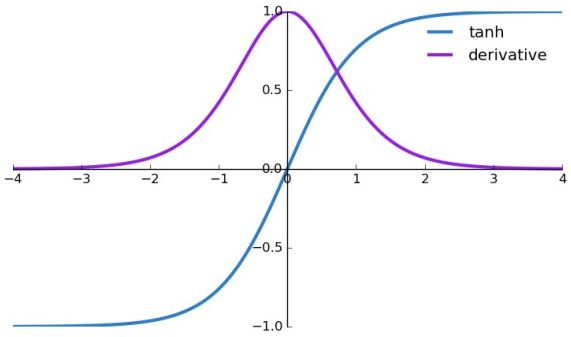


Forward propagation gives:

$\mathbf{x}^{(0)}$	$\mathbf{u}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{u}^{(2)}$	$\mathbf{z}^{(2)}$	$\mathbf{u}^{(3)}$	$\mathbf{y}^{(3)}$
$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}0.7\\1\end{bmatrix}$	$\begin{bmatrix}1\\0.6\\0.76\end{bmatrix}$	$[-1.48]$	$\begin{bmatrix}1\\-0.90\end{bmatrix}$	$[-0.8]$	-0.66

For example $u_1^{(1)} = 0.1 * 1 + 0.3 * 2 = 0.7$.

Recall the derivative of $\tanh(x) = 1 - \tanh^2(x)$



Backpropagation gives:

$\boldsymbol{\delta}^{(3)}$	$\boldsymbol{\delta}^{(2)}$	$\boldsymbol{\delta}^{(1)}$
$[-0.927]$	$[-0.347]$	$\begin{bmatrix}-0.220\\0.438\end{bmatrix}$

For example, $\boldsymbol{\delta}^{(2)} = [(1-0.9^2)\cdot 2\cdot(-0.927)] = [-0.347]$.

It is now simple matter to use Eq. (3.19) to compute the partial derivatives that are needed for the gradient:

$$\frac{\partial E_n}{\partial \mathrm{W}^{(1)}} = \mathbf{x}^{(0)}\left(\boldsymbol{\delta}^{(1)}\right)^T = \begin{bmatrix}-0.22 & 0.44\\-0.44 & 0.88\end{bmatrix}$$

$$\frac{\partial E_n}{\partial \mathrm{W}^{(2)}} = \mathbf{z}^{(1)}\left(\boldsymbol{\delta}^{(2)}\right)^T = \begin{bmatrix}-0.35\\-0.21\\-0.26\end{bmatrix}$$

$$\frac{\partial E_n}{\partial \mathrm{W}^{(3)}} = \mathbf{z}^{(2)}\left(\boldsymbol{\delta}^{(3)}\right)^T = \begin{bmatrix}-0.93\\0.84\end{bmatrix}$$

3.4 Implementation of ANN

In this subsection, we will now implement an MLP from scratch to classify the images in the MNIST dataset. To keep things simple, we will implement an MLP with only one hidden layer. Since the approach may seem a little bit complicated at first, you are encouraged to download the sample code for this chapter from course website so that you can view this MLP implementation annotated with comments and syntax highlighting for better readability.

If you are not running the code from the accompanying Jupyter Notebook file copy the NeuralNetMLP code from this chapter into a Python script file in your current working directory (for example, neuralnet.py), which you can then import into your current Python session via the following command:

```
from neuralnet import NeuralNetMLP
```

The code will contain parts that we have not talked about yet, such as the backpropagation algorithm, but most of the code should look familiar to you based on our lab 3 and the discussion of forward propagation in earlier sections.

Do not worry if not all the code makes immediate sense to you; we will follow up on certain parts later in this chapter. However, going over the code at this stage can make it easier to follow the theory later.

The following is the implementation of an MLP:

```
import numpy as np
import sys
class NeuralNetMLP(object):
    """ Feedforward neural network / Multi-layer perceptron classifier.


    Parameters
    ------------
    n_hidden : int (default: 30)
        Number of hidden units.
    l2 : float (default: 0.)
        Lambda value for L2-regularization.
        No regularization if l2=0. (default)
    epochs : int (default: 100)
        Number of passes over the training set.
    eta : float (default: 0.001)
        Learning rate.
    shuffle : bool (default: True)
        Shuffles training data every epoch
        if True to prevent circles.
    minibatch_size : int (default: 1)
        Number of training examples per minibatch.
    seed : int (default: None)
        Random seed for initializing weights and shuffling.

    Attributes
    -----------
    eval_ : dict
        Dictionary collecting the cost, training accuracy,
        and validation accuracy for each epoch during training.

    """
    def __init__(self, n_hidden=30,
                 l2=0., epochs=100, eta=0.001,
                 shuffle=True, minibatch_size=1, seed=None):

        self.random = np.random.RandomState(seed)
        self.n_hidden = n_hidden
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.shuffle = shuffle
        self.minibatch_size = minibatch_size

    def _onehot(self, y, n_classes):
        """Encode labels into one-hot representation

        Parameters
        ------------
        y : array, shape = [n_examples]
            Target values.

        Returns
```

```
        -----------
        onehot : array, shape = (n_examples, n_labels)

        """
        onehot = np.zeros((n_classes, y.shape[0]))
        for idx, val in enumerate(y.astype(int)):
            onehot[val, idx] = 1.
        return onehot.T

    def _sigmoid(self, z):
        """Compute logistic function (sigmoid)"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def _forward(self, X):
        """Compute forward propagation step"""

        # step 1: net input of hidden layer
        # [n_examples, n_features] dot [n_features, n_hidden]
        # -> [n_examples, n_hidden]
        z_h = np.dot(X, self.w_h) + self.b_h

        # step 2: activation of hidden layer
        a_h = self._sigmoid(z_h)

        # step 3: net input of output layer
        # [n_examples, n_hidden] dot [n_hidden, n_classlabels]
        # -> [n_examples, n_classlabels]

        z_out = np.dot(a_h, self.w_out) + self.b_out
        # step 4: activation output layer
        a_out = self._sigmoid(z_out)

        return z_h, a_h, z_out, a_out

    def _compute_cost(self, y_enc, output):
        """Compute cost function.

        Parameters
        ----------
        y_enc : array, shape = (n_examples, n_labels)
            one-hot encoded class labels.
        output : array, shape = [n_examples, n_output_units]
            Activation of the output layer (forward propagation)

        Returns
        ---------
        cost : float
            Regularized cost

        """
        L2_term = (self.l2 *
                   (np.sum(self.w_h ** 2.) +
                    np.sum(self.w_out ** 2.)))

        term1 = -y_enc * (np.log(output))
        term2 = (1. - y_enc) * np.log(1. - output)
```

```
        cost = np.sum(term1 - term2) + L2_term
        return cost

    def predict(self, X):
        """Predict class labels

        Parameters
        -----------
        X : array, shape = [n_examples, n_features]
            Input layer with original features.

        Returns:
        ----------
        y_pred : array, shape = [n_examples]
            Predicted class labels.

        """
        z_h, a_h, z_out, a_out = self._forward(X)
        y_pred = np.argmax(z_out, axis=1)
        return y_pred

    def fit(self, X_train, y_train, X_valid, y_valid):
        """ Learn weights from training data.

        Parameters
        -----------
        X_train : array, shape = [n_examples, n_features]
            Input layer with original features.
        y_train : array, shape = [n_examples]
            Target class labels.
        X_valid : array, shape = [n_examples, n_features]
            Sample features for validation during training
        y_valid : array, shape = [n_examples]
            Sample labels for validation during training

        Returns:
        ----------
        self

        """
        n_output = np.unique(y_train).shape[0] # no. of class
                                               #labels
        n_features = X_train.shape[1]

        ########################
        # Weight initialization
        ########################

        # weights for input -> hidden
        self.b_h = np.zeros(self.n_hidden)
        self.w_h = self.random.normal(loc=0.0, scale=0.1,
                                      size=(n_features,
                                            self.n_hidden))

        # weights for hidden -> output
        self.b_out = np.zeros(n_output)
```

```
        self.w_out = self.random.normal(loc=0.0, scale=0.1,
                                        size=(self.n_hidden,
                                              n_output))

        epoch_strlen = len(str(self.epochs)) # for progr. format.
        self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': \
                    []}

        y_train_enc = self._onehot(y_train, n_output)

        # iterate over training epochs
        for i in range(self.epochs):

            # iterate over minibatches
            indices = np.arange(X_train.shape[0])

            if self.shuffle:
                self.random.shuffle(indices)

            for start_idx in range(0, indices.shape[0] -\
                                   self.minibatch_size +\
                                   1, self.minibatch_size):
                batch_idx = indices[start_idx:start_idx +\
                                    self.minibatch_size]

                # forward propagation
                z_h, a_h, z_out, a_out = \
                    self._forward(X_train[batch_idx])

                ###################
                # Backpropagation
                ###################
                You need to finish this part in Lab 4

                # Regularization and weight updates
                delta_w_h = (grad_w_h + self.l2*self.w_h)
                delta_b_h = grad_b_h # bias is not regularized
                self.w_h -= self.eta * delta_w_h
                self.b_h -= self.eta * delta_b_h

                delta_w_out = (grad_w_out + self.l2*self.w_out)
                delta_b_out = grad_b_out # bias is not regularized
                self.w_out -= self.eta * delta_w_out
                self.b_out -= self.eta * delta_b_out

            #############
            # Evaluation
            #############

            # Evaluation after each epoch during training
            z_h, a_h, z_out, a_out = self._forward(X_train)

            cost = self._compute_cost(y_enc=y_train_enc,
                                      output=a_out)

            y_train_pred = self.predict(X_train)
```

```
            y_valid_pred = self.predict(X_valid)

            train_acc = ((np.sum(y_train ==
                                 y_train_pred)).astype(np.float) /
                         X_train.shape[0])
            valid_acc = ((np.sum(y_valid ==
                                 y_valid_pred)).astype(np.float) /
                         X_valid.shape[0])

            sys.stderr.write('\r%0*d/%d | Cost: %.2f '
                             '| Train/Valid Acc.: %.2f%%/%.2f%% '
                             %
                             (epoch_strlen, i+1, self.epochs,
                              cost,
                              train_acc*100, valid_acc*100))
            sys.stderr.flush()

            self.eval_['cost'].append(cost)
            self.eval_['train_acc'].append(train_acc)
            self.eval_['valid_acc'].append(valid_acc)

        return self
```

As you may have noticed from the preceding code, we implemented the fit method so that it takes four input arguments: training images, training labels, validation images, and validation labels. In NN training, it is useful to compare training and validation accuracy, which helps us judge whether the network model performs well, given the architecture and hyperparameters. For example, if we observe a low training and validation accuracy, there is likely an issue with the training dataset, or the hyperparameters settings are not ideal. A relatively large gap between the training and the validation accuracy indicated that the model is likely overfitting the training dataset so that we want to reduce the number of parameters in the model or increase the regularization strength. If both the training and validation accuracies are high, the model is likely to generalize well to new data, for example, the test dataset, which we use for the final model evaluation.

In general, training (deep) NNs is relatively expensive compared with the other models we've discussed so far. Thus, we want to stop it early in certain circumstances and start over with different hyperparameter settings. On the other hand, if we find that it increasingly tends to overfit the training data (noticeable by an increasing gap between training and validation dataset performance), we may want to stop the training early as well. In our NeuralNetMLP implementation, we also defined an eval_ attribute that collects the cost, training, and validation accuracy for each epoch so that we can visualize the results using Matplotlib.