11

Supervised and unsupervised learning

This chapter covers

- Reviewing the basics of artificial intelligence, machine learning, and deep learning
- Understanding graph machine learning, graph embedding, and graph convolutional networks
- Understanding attention mechanisms
- Understanding self-organizing maps
- Solving optimization problems using supervised and unsupervised machine learning

Artificial intelligence (AI) is one of the fastest growing fields of technology, driven by advancements in computing power, access to vast amounts of data, breakthroughs in algorithms, and increased investment from both public and private sectors. AI aims to create intelligent systems or machines that can exhibit intelligent behavior, often by mimicking or drawing inspiration from biological intelligence. These systems can be designed to function autonomously or with some human guidance, and ideally, they can adapt to environments with diverse structures, observability levels, and dynamics. AI augments our intelligence by empowering us to analyze vast amounts of multidimensional, multimodal data and identify hidden patterns that would be difficult for humans to recognize. AI also supports our learning and decision-making by providing relevant insights and potential courses of action. AI encompasses various subfields, such as situation awareness (comprising perception, comprehension, and projection), knowledge representation, cognitive reasoning, machine learning, data analytics (covering descriptive, diagnostic, predictive, and prescriptive analytics), problem solving (involving constraint satisfaction and problem-solving using search and optimization), as well as digital and physical automation (such as conversational AI and robotics).

In this last part of the book, we will explore the convergence of two branches of AI: machine learning and optimization. Our focus will be on showcasing the practical applications of machine learning in tackling optimization problems. This chapter provides an overview of machine learning fundamentals as essential background knowledge, and then it delves into applications of supervised and unsupervised machine learning in handling optimization problems. Reinforcement learning will be covered in the next chapter.

11.1 A day in the life of AI-empowered daily routines

AI, and machine learning in particular, forms the foundation of many successful disruptive industries and has successfully delivered many commercial products that touch everybody's life every day. Starting at home, voice assistants eagerly await your commands, effortlessly controlling smart appliances and adjusting the smart thermostat to ensure comfort and convenience. Smart meters intelligently manage energy consumption, optimizing efficiency and reducing costs.

On the route to school or work, navigation apps with location intelligence guide the way, considering real-time traffic updates to provide the fastest and most efficient route. Shared mobility services offer flexible transportation options on demand, while advanced driver assistance systems enhance safety and convenience if you decide to drive. In the not-too-distant future, we will enjoy safe and entertaining self-driving vehicles as a third living space, after our homes and workplaces, with consumer-centric products and services.

Once at school or at the workplace, AI becomes an invaluable tool for personalization and to boost productivity. Personalized learning platforms cater to individual needs, adapting teaching methods and content to maximize understanding and retention. Summarization and grammar-checking algorithms aid in crafting flawless documents, while translation tools bridge language barriers effortlessly. Excel AI formula generators streamline complex calculations, saving time and effort. Human-like text generation enables natural and coherent writing, while audio, image, and video generation from text unlock creative possibilities. Optimization algorithms ensure optimal resource allocation and scheduling, maximizing efficiency in various scenarios, and handle different design, planning, and control problems.

During shopping, AI enhances the experience in numerous ways. Voice search enables hands-free exploration, while searching by images allows for effortless discovery of desired items. Semantic search understands context and intent, providing more accurate results. Recommendation engines offer personalized suggestions based on individual preferences and online shopping behavior, while last-mile or door-to-door delivery services ensure timely, transparent, and convenient package arrival.

In the realm of health, AI revolutionizes personalized healthcare, assisting with diagnosis, treatment planning, and rehabilitation. Lab automation speeds up testing processes, improving accuracy and efficiency. AI-driven drug discovery and delivery enable the development of innovative treatments and targeted therapies, transforming lives.

During leisure time, AI contributes to physical and mental well-being. Fitness planning apps tailor workout routines to individual goals and capabilities, providing personalized guidance and motivation. Trip planning tools recommend exciting destinations and itineraries, ensuring memorable experiences. AI-powered meditation apps offer customized relaxation experiences, soothing the mind and promoting mindfulness.

Machine learning, a prominent subfield of artificial intelligence, has played a pivotal role in bringing AI from the confines of high-tech research labs to the convenience of our daily lives.

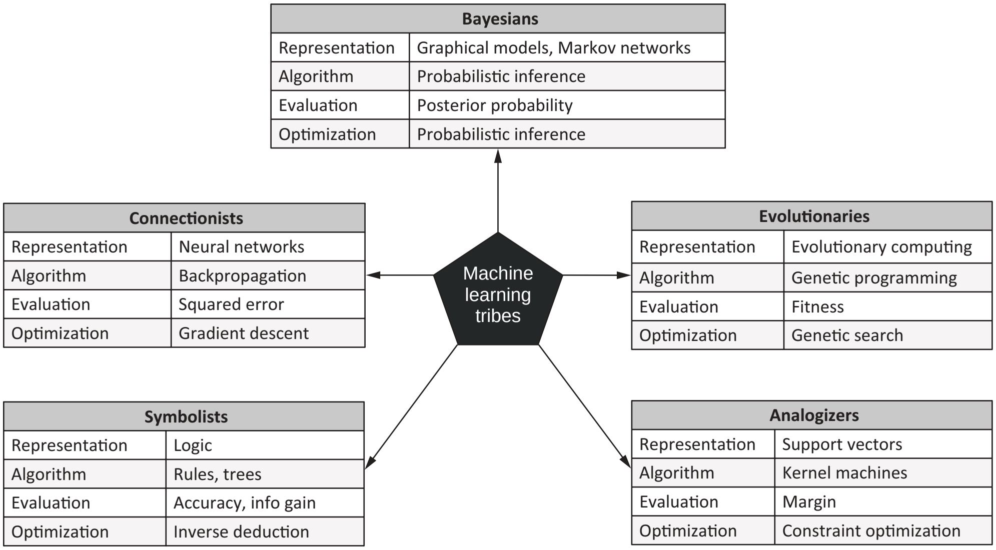
11.2 Demystifying machine learning

The goal of learning is to create an internal model or abstraction of the external world. More comprehensively, Stanislas Dehaene, in *How We Learn* [1], introduced seven key definitions of learning that lie at the heart of present-day machine learning algorithms:

- Learning is adjusting the parameters of a mental model.
- Learning is exploring a combinatorial explosion.
- Learning is minimizing errors.
- Learning is exploring the space of possibilities.
- Learning is optimizing a reward function.
- Learning is restricting search space.
- Learning is projecting a priori hypotheses.

Machine learning (ML) is a subfield of AI that endows an artificial system or process with the ability to learn from experience and observation without being explicitly programmed. Thomas Mitchell, in *Machine Learning*, defines ML as follows: “A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*” [2]. In his book *The Master Algorithm*, Pedro Domingos summarizes the ML schools of thought into five main schools [3], illustrated in figure 11.1:

- Bayesians with probabilistic inference as the master algorithm
- Symbolists with rules and trees as the main core algorithm within this paradigm
- Connectionists who use neural networks with backpropagation as a master algorithm
- Evolutionaries who rely on the evolutionary computing paradigm
- Analogizers who use mathematical techniques like support vector machines with different kernels



Machine learning tribes

Bayesians	
Representation	Graphical models, Markov networks
Algorithm	Probabilistic inference
Evaluation	Posterior probability
Optimization	Probabilistic inference

Connectionists	
Representation	Neural networks
Algorithm	Backpropagation
Evaluation	Squared error
Optimization	Gradient descent

Evolutionaries	
Representation	Evolutionary computing
Algorithm	Genetic programming
Evaluation	Fitness
Optimization	Genetic search

Symbolists	
Representation	Logic
Algorithm	Rules, trees
Evaluation	Accuracy, info gain
Optimization	Inverse deduction

Analogizers	
Representation	Support vectors
Algorithm	Kernel machines
Evaluation	Margin
Optimization	Constraint optimization

Figure 11.1 Different ML schools of thought according to Domingos' *The Master Algorithm*

Nowadays, connectionist learning approaches have attracted most of the attention, thanks to their perception and learning capabilities in several challenging domains. These statistical ML algorithms follow a bottom-up inductive reasoning paradigm (i.e., inferring general rules from a set of examples) to discover patterns from vast amounts of data.

The unreasonable effectiveness of data

Simple models and a lot of data trump more elaborate models based on less data [4]. This means that having a large amount of data to train simple models is often more effective than using complex models with only a small amount of data. For example, in self-driving vehicles, a simple model that has been trained on millions of hours of driving data can often be more effective in recognizing and reacting to diverse road situations than a more complex model trained on a smaller dataset. This is because the massive amount of data helps the simple model learn a wide range of patterns and scenarios, including adversarial and edge cases it might encounter, making it more adaptable and reliable in real-world driving conditions.

These connectionist learning or statistical ML approaches are based on the experimental findings that even very complex problems in artificial intelligence may be solved by simple statistical models trained on massive datasets [4]. Statistical ML is currently the most famous form of AI. The rapid advancement of this form of ML can be attributed primarily to the widespread availability of big data and open source tools, enhanced computational power such as AI accelerators, and substantial research and development funding from both public and private sectors.

Generally speaking, ML algorithms can be categorized into supervised, unsupervised, hybrid learning, and reinforcement learning algorithms, as illustrated in figure 11.2.

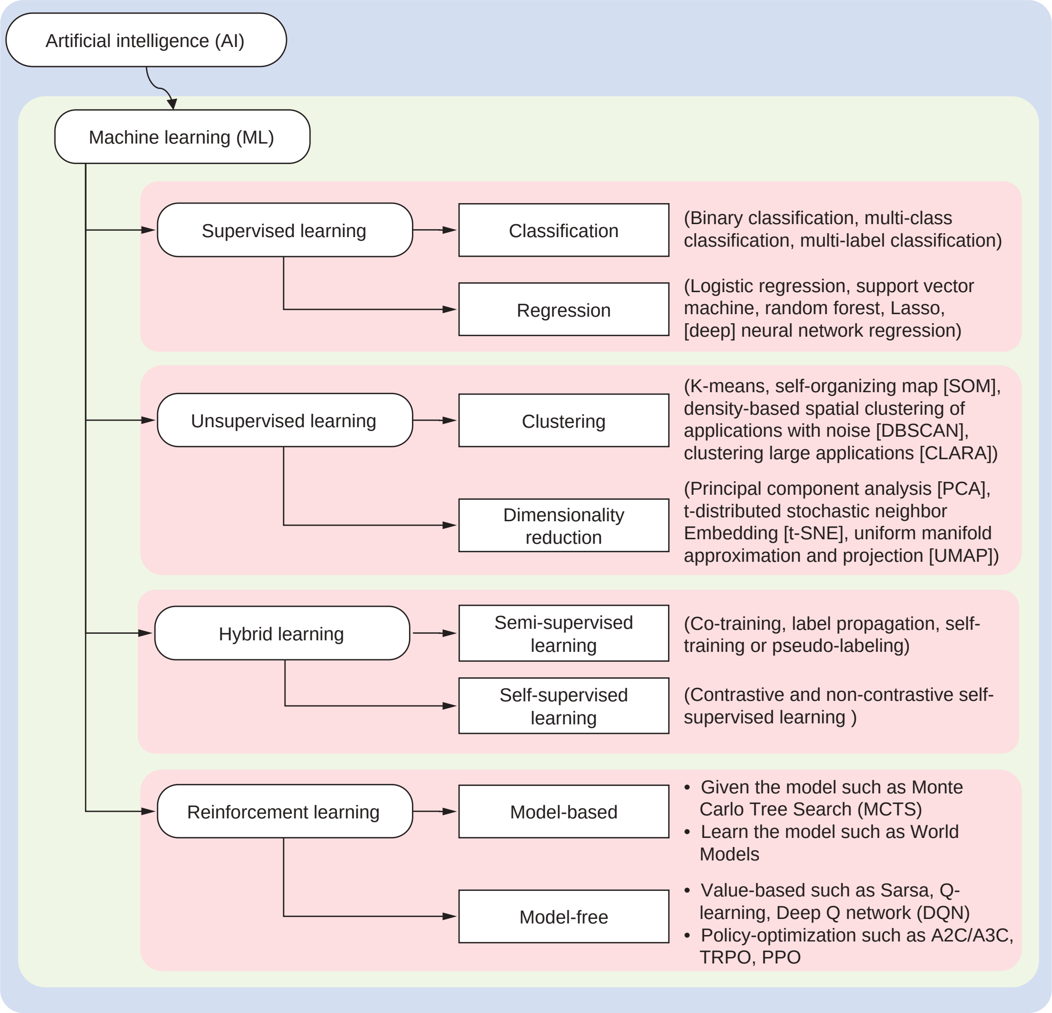

Artificial intelligence (AI)
Machine learning (ML)
Supervised learning
Classification
(Binary classification, multi-class classification, multi-label classification)
Regression
(Logistic regression, support vector machine, random forest, Lasso, [deep] neural network regression)
Unsupervised learning
Clustering
(K-means, self-organizing map [SOM], density-based spatial clustering of applications with noise [DBSCAN], clustering large applications [CLARA])
Dimensionality reduction
(Principal component analysis [PCA], t-distributed stochastic neighbor Embedding [t-SNE], uniform manifold approximation and projection [UMAP])
Hybrid learning
Semi-supervised learning
(Co-training, label propagation, self-training or pseudo-labeling)
Self-supervised learning
(Contrastive and non-contrastive self-supervised learning)
Reinforcement learning
Model-based
• Given the model such as Monte Carlo Tree Search (MCTS)
• Learn the model such as World Models
Model-free
• Value-based such as Sarsa, Q-learning, Deep Q network (DQN)
• Policy-optimization such as A2C/A3C, TRPO, PPO

Figure 11.2 ML taxonomy as a subfield of AI

- *Supervised learning*—This approach uses inductive inference to approximate mapping functions between data and known labels or classes. This mapping is learned using already labeled training data. *Classification* (predicting discrete or categorical values) and *regression* (predicting continuous values) are common tasks in supervised learning. For example, classification seeks a scoring function $f: X \times C \longrightarrow R$, where X represents the training data space and C represents the label or class space. This mapping can be learned using N training examples of the form $\{(x_{11}, x_{21}, \ldots, x_{m1}, c_1), (x_{12}, x_{22}, \ldots, x_{m2}, c_2), \ldots, (x_{1N}, x_{2N}, \ldots, x_{mN}, c_N)\}$, where x_i is the feature vector of the i-th example, m is number of features, and c_i is the corresponding class. The predicted class is the class that gives the highest score of f, i.e., $c(x) = \text{argmax}_c f(x,c)$. In the context of self-driving vehicles, supervised learning might be used to train a model to recognize traffic signs. The input data would be images of various traffic signs, and the correct output (the labels) would be the type of each sign. The trained model could then identify traffic signs correctly when driving. Feedforward neural networks (FNNs) or multilayer perceptrons (MLPs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short-term memory (LSTM) networks, and sequence-to-sequence (Seq2Seq) models are examples of common neural network architectures that are typically trained using supervised learning. Examples of solving combinatorial problems using supervised ML are provided in sections 11.6, 11.7, and 11.9.

- *Unsupervised learning*—This approach deals with unlabeled data through techniques like *clustering* and *dimensionality reduction*. In clustering, for example, n objects (each could be a vector of d features) are given, and the task is to group them based on certain similarity measures into c groups (clusters) in such a way that all objects in a single group have a “natural” relation to one another, and objects not in the same group are somehow different. For instance, unsupervised learning might be used in self-driving vehicles to cluster similar driving scenarios or environments. Using unsupervised learning, the car might learn to identify different types of intersections or roundabouts, even if no one has explicitly labeled the data with these categories. Autoencoders, k-means, density-based spatial clustering (DBSCAN), principal component analysis (PCA), and self-organizing maps (SOMs) are examples of unsupervised learning methods. SOM is explained in section 11.4. An example of a combinatorial problem using SOM is provided in section 11.8.
- *Hybrid learning*—This approach includes *semi-supervised learning and self-supervised learning* techniques. Semi-supervised learning is a mix of supervised and unsupervised learning where only a fraction of the input data is labeled with corresponding outputs. In this case, the training process uses the small amount of labeled data available and pseudo-labels the rest of the dataset—for example, training a self-driving vehicle's perception system with a limited set of labeled driving scenarios, then using a vast collection of unlabeled driving data to improve its ability to recognize and respond to various road conditions and obstacles. Self-supervised learning is an ML process where a model learns meaningful representations of the input data by using the inherent structure or relationships within the data itself. This is achieved by creating supervised learning tasks from the unlabeled data. For instance, a self-supervised model might be trained to predict the next word in a sentence based on the previous words or to reconstruct an image from a scrambled version. These learned representations can then be used for various

downstream tasks, such as image classification or object detection. In the context of self-driving vehicles, a perception system can be trained to identify essential features in unlabeled driving scenes, such as lane markings, pedestrians, and other vehicles. Then, the learned features are utilized as pseudo-labels to classify new driving scenes in a supervised manner, enabling the vehicle to make decisions based on its understanding of the road environment.

- *Reinforcement learning (RL)*—This approach learns from interactions through a feedback loop or by trial and error. A learning agent learns to make decisions by taking actions in an environment to maximize some notion of cumulative reward. For self-driving vehicles, reinforcement learning could be used in the decision-making process. For instance, the car might learn over time the best way to merge into traffic on a busy highway. It would receive positive rewards for successful merges and negative rewards for dangerous maneuvers or failed attempts. Over time, through trial and error and the desire to maximize the reward, the car would learn an optimal policy for merging into traffic. More details about RL are provided in the next chapter.

Deep learning (DL) is a subfield of ML concerned with learning underlying features in data using neural networks with many layers (hence "deep") enabling artificial systems to build complex concepts out of simpler concepts. DL enables learning discriminative features or representations and learning at different levels of abstraction. To achieve this, the network uses hierarchical feature learning and employs a handful of convolutional layers. DL revolutionizes the field of ML by reducing the need for extensive data preprocessing. DL models can automatically extract highly discriminative features from raw data, eliminating the need for hand-crafted feature engineering. This end-to-end learning process significantly reduces the reliance on human experts, as the model learns to extract meaningful representations and patterns directly from the input data.

Unlike traditional ML algorithms, DL models have the ability to directly consume and process various forms of structured and unstructured data, such as text, audio, images, video, and even graphs. Graph-structured data is particularly important in the field of combinatorial optimization due to its ability to capture and represent the relationships and constraints between elements in optimization problems. Geometric DL is a subfield of ML that combines graph theory with DL.

The following two sections address graph machine learning and self-organizing maps in more detail. They are essential background knowledge to the use cases described later in this chapter.

11.3 Machine learning with graphs

As explained in section 3.1, a graph is a nonlinear data structure composed of entities known as *vertices* (or nodes) and the relationships between them, known as *edges* (or *arcs* or *links*). Data coming from different domains can be nicely captured using a graph. Social media networks, for instance, employ graphs to depict connections between users and to analyze social interactions, which in turn drive content propagation and recommendations. Navigation applications use graphs to represent physical locations and the paths between them, enabling route calculations, real-time traffic updates, and estimated time of arrival (ETA) predictions. Recommender systems rely on graphs to model user–item interactions and preferences, thereby offering personalized recommendations. Search engines use web graphs, where web pages are nodes and hyperlinks are edges, to crawl and index the internet and facilitate efficient information retrieval. Knowledge graphs offer a structured representation of factual information, relationships, and entities, and they're used in diverse fields from digital assistants to enterprise data integration. Question-answering engines use graphs to understand and decompose complex questions and search for relevant answers in structured datasets. In the realm of chemistry, molecular structures can be viewed as graphs, where atoms are nodes and bonds are edges, supporting tasks like discovering compounds and predicting properties.

Graph-structured data is vital due to its power to model complex relationships and dependencies between entities in an intuitive, self-descriptive, intrinsically explainable, and natural way. Unlike traditional tabular data, graphs allow for the representation of networked relationships and complex interconnectedness between entities of interest, making them an excellent tool for modeling numerous real-world systems. Tabular data can be converted into graph-structured data—the specific definitions of nodes and edges would depend on what relationships you're interested in examining within the data. For example, in the context of a FIFA dataset, we can define nodes and edges based on the information available in this dataset:

- *Nodes*—Nodes represent entities of interest and could be the players, the clubs they play for, or their nationalities. Each of these entities could be a separate node in the graph. For example, Lionel Messi, Inter Miami, and Argentina could all be individual nodes in the graph.
- *Edges*—Edges represent the relationships between the nodes. For instance, an edge could connect a player to the club they play for, indicating that the player is part of that club. Another edge could connect a player to their nationality, showing that the player belongs to that country. So, for example, Lionel Messi could be connected to Inter Miami with an edge indicating that Messi plays for Inter Miami, and another edge could connect Lionel Messi to Argentina, indicating his nationality.

The next listing shows how to convert tabular data for 10 selected soccer players into a graph using NetworkX.

Listing 11.1 Converting tabular data to a graph

```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
```

```
data={'Player':['L. Messi','R. Lewandowski','C. Ronaldo','Neymar Jr','K.
➥ Mbappé','E.Haaland','H. Kane','Luka Modrić','L. Goretzka','M. Salah'],
➥    'Age':[36,34,38,22,24,35,29,37,28,31],
➥    'Nationality':['Argentina','Poland','Portugal','Brazil','France','Norway',
➥    'England','Croatia','Germany','Egypt'],
➥    'Club':['Inter Miami','Barcelona','Al-Nassr','Al-Hilal ','PSG','Manchester
➥    City','Tottenham Hotspur','Real Madrid','Bayern Munich','Liverpool'],
➥    'League':['Major League Soccer ','Spain Primera Division','Saudi Arabia
➥ League','Saudi Arabia League','French Ligue 1','English Premier
➥ League','English Premier League','Spain Primera Division','German 1.
➥ Bundesliga','English Premier League']}
df=pd.DataFrame.from_dict(data)
```

As a continuation of listing 11.1, we can create a NetworkX graph whose nodes represent the player name, club, and nationality and whose edges represent the semantic relationships between these nodes.

```
G = nx.Graph()
for index, row in df.iterrows():
    G.add_edge(row['Player'], row['Club'], relationship='plays_for')
for index, row in df.iterrows():
    G.add_edge(row['Player'], row['Nationality'], relationship='belongs_to')
pos = nx.kamada_kawai_layout(G)
plt.figure(figsize=(20, 14))
player_nodes = df['Player'].unique().tolist()
club_nodes = df['Club'].unique().tolist()
nationality_nodes = df['Nationality'].unique().tolist()
nx.draw_networkx_nodes(G, pos, nodelist=player_nodes, node_color='blue',
➥ label='Player Name', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=club_nodes, node_color='red', label='Club',
➥ node_shape='d')
nx.draw_networkx_nodes(G, pos, nodelist=nationality_nodes, node_color='gray',
➥ label='Nationality', node_shape='v')
nx.draw_networkx_edges(G, pos)
edge_labels = nx.get_edge_attributes(G, 'relationship')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=12)
nx.draw_networkx_labels(G, pos)
plt.legend(fontsize=13, loc='upper right')
plt.show()
```

Add nodes and edges for nationalities.

Create a new graph.

Add nodes and edges for clubs.

Create the layout.

Set the size of the figure.

Get lists of player, club, and nationality nodes.

Draw nodes in different colors.

Draw edges.

Draw edge labels.

Draw node labels.

Figure 11.3 shows the data for the 10 selected soccer players in a graph. This graph shows the entities of interest (player, club, and nationality) and their relationships. For example, L. Messi is a player who plays for Inter Miami and is from Argentina.

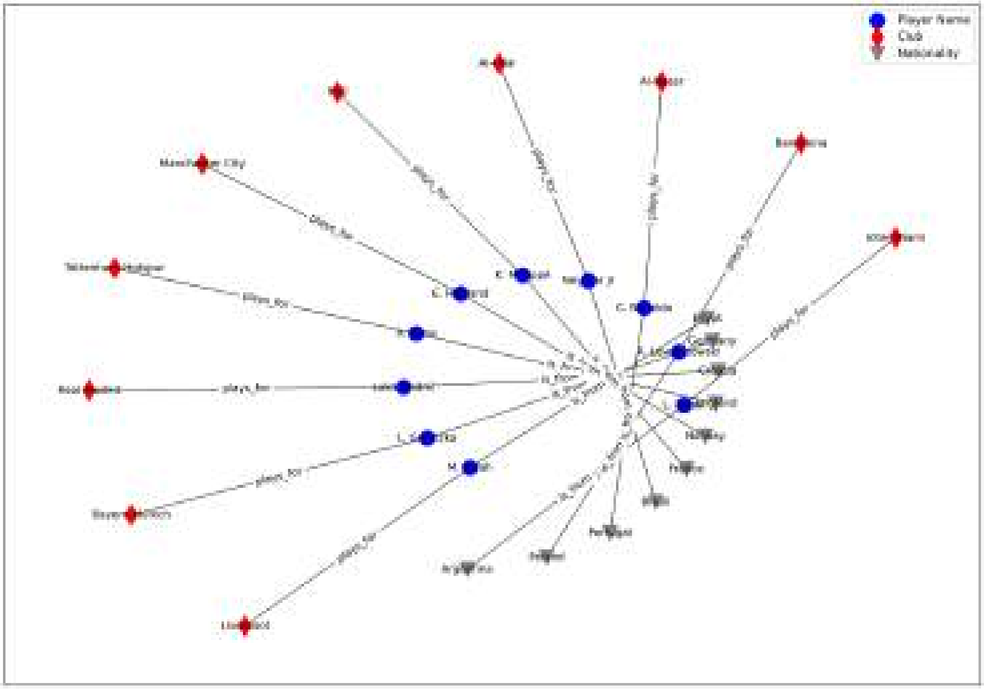


Figure 11.3 Graph-structured data for 10 selected soccer players

Graph data fundamentally differs from Euclidean data, as the concept of distance is not simply a matter of straight-line (Euclidean) distance between two points. In the case of a graph, what matters is the structure of the nodes and edges—whether two nodes are connected by an edge and how they are connected to other nodes in the graph. Table 11.1 summarizes the differences between Euclidean and non-Euclidean graph data.

Table 11.1 Euclidean data versus non-Euclidean graph data

Aspects	Euclidean data	Non-Euclidean graph data
Common data types	Numerical, text, audio, images, videos	Road networks, social networks, web pages, and molecular structures
Dimensionality	Can be 1D (e.g., numbers, text), 2D (e.g., images, heatmaps), or higher-dimensional (e.g., RGB-D images or depth maps, 3D point cloud data)	Large dimensionality (e.g., a Pinterest graph has 3 billion nodes and 18 billion edges)
Structure	Fixed structure (e.g., in the case of an image, the structure is embedded via pixel proximity)	Arbitrary structure (every node can have a different neural structure because the network neighborhood around it is different, as the model adapts to the data)

Table 11.1 **Euclidean data versus non-Euclidean graph data (*continued*)**

Aspects	Euclidean data	Non-Euclidean graph data
Spatial locality	Yes (i.e., data points that are close together in the input space are also likely to be close together in the output space).	No, "closeness" is determined by the graph structure, not spatial arrangement (i.e., two nodes that are "close" to each other might not necessarily have similar properties or features, such as in the case of a traffic light node and a crosswalk node).
Shift-invariance	Yes (i.e., data-inherent meaning is preserved when shifted; for instance, the concept of a cat in a picture does not change if the cat is in the top left corner or the bottom right corner of the image).	No (in a graph, there's no inherent meaning to the "position" of a node that can be "shifted").
Ordinality or hierarchy	Yes	No, graph data has "permutation invariance"—the specific ordering or labeling of nodes doesn't usually affect the underlying relationships and properties of the graph.
Shortest path between two points	A straight line	Is not necessarily a straight line
Examples of ML models	Convolutional neural networks (CNNs), long short-term memory (LSTM), and recurrent neural networks (RNNs)	Graph neural networks (GNNs), graph convolutional networks (GCNs), temporal graph networks (TGNs), spatial-temporal graph neural networks (STGNNs)

Geometric deep learning (GDL) is an umbrella term for emerging techniques seeking to extend (structured) deep neural models to handle non-Euclidean data with underlying geometric structures, such as graphs (networks of connected entities), point clouds (collections of 3D data points), molecules (chemical structures), and manifolds (curved, high-dimensional surfaces). Graph machine learning (GML) is a subfield of ML that focuses on developing algorithms and models capable of learning from graph-structured data. Graph embedding or representation learning is the first step in performing ML tasks such as node classification (predicting a category for each node), link prediction (forecasting connections between nodes), and community detection (identifying groups of interconnected nodes). The next subsection describes different graph embedding techniques.

11.3.1 Graph embedding

Graph embedding is a task that aims to learn a mapping from a discrete high-dimensional graph domain to a low-dimensional continuous domain. Through the process of graph embedding, graph nodes, edges, and their features are transformed into continuous vectors while preserving the structural information of the graph. For example, as shown in figure 11.4, an encoder, $ENC(v)$, maps node v from the input graph space G to a low-dimensional vector h_v in the embedding or latent space H based on the node's position in the graph, its local neighborhood structure, or its features, or some combination of the three.

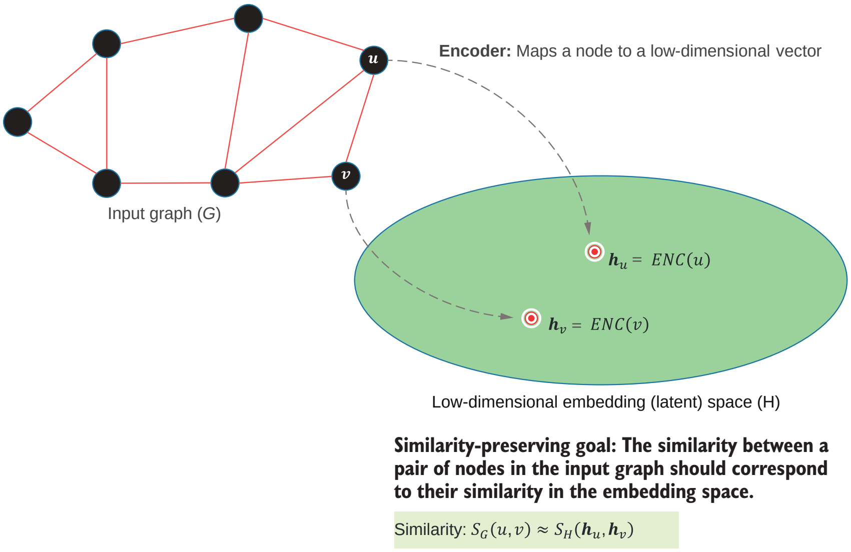

u
Encoder: Maps a node to a low-dimensional vector
v
Input graph (G)
$\boldsymbol{h}_u = ENC(u)$
$\boldsymbol{h}_v = ENC(v)$
Low-dimensional embedding (latent) space (H)
Similarity-preserving goal: The similarity between a pair of nodes in the input graph should correspond to their similarity in the embedding space.
Similarity: $S_G(u,v) \approx S_H(\boldsymbol{h}_u, \boldsymbol{h}_v)$

Figure 11.4 Graph embedding

This encoder needs to be optimized to minimize the difference between the similarity of a pair of nodes in the graph and their similarity in the embedding space. Nodes that are connected or nearby in the graph should be close in the embedded space. Conversely, nodes that are not connected or are far apart in the graph should be far apart in the embedded space. In a more generalized encoder/decoder architecture, a decoder is added to extract user-specified information from the low-dimensional embedding [5]. By jointly optimizing the encoder and decoder, the system learns to compress information about the graph structure into the low-dimensional embedding space.

There are various methods for graph embedding which can be broadly classified into transductive (shallow) embedding and inductive embedding:

- *Transductive embedding*—In the transductive learning paradigm, the model learns embeddings only for the nodes present in the graph during the training phase. The learned embeddings are specific to these nodes, and the model cannot generate embeddings for new nodes that weren't present during training. These methods are difficult to scale and are suitable for static graphs. Examples of transductive methods for graph embedding include random walk (e.g., node2vec and DeepWalk) and matrix factorization (e.g., graph factorization and HOPE).
- *Inductive embedding*—Inductive learning methods can generalize to unseen nodes or entire graphs that were not present during training. They do this by learning a function that generates the embedding of a node based on its features and the structure of its local neighborhood, which can be applied to any node, regardless of whether it was present during training or not. These methods are suitable for evolving graphs. Examples of inductive methods for graph embedding are graph neural networks (GNN) and graph convolutional networks (GCNs).

Appendix A contains examples of some of these methods. For more information, see Broadwater and Stillman's *Graph Neural Networks in Action* [6]. We'll focus on GCN, as it is the most relevant approach to the combinatorial optimization application presented in this chapter.

Transductive versus inductive learning

Transductive learning aims to learn from a specific set of data to a specific set of predictions without generalizing to new data. *Inductive learning* aims to learn general rules from observed training cases. These general rules can then be applied to new, unseen data.

The *convolution operation* forms the basis of representation learning in many structured data scenarios, enabling the automatic learning of meaningful features from raw data, thereby obviating the need for manual feature engineering. Convolution is a mathematical operation that takes two functions (input data and a kernel, filter, or feature detector) and measures their overlap or merges the two sets of information to produce a feature map. One critical aspect of convolution is its ability to respect and utilize the known structural relationships among data points, such as the positional associations among pixels, the temporal order of time points, or the edges linking nodes in a network. In traditional ML, convolutional neural networks (CNNs) employ the convolution operator as a key tool for identifying spatial patterns within images. This is made possible by the inherent grid-like structure of image data, which allows the model to slide filters over the image, exploit the spatial regularities, and extract features in a manner akin to pattern recognition.

However, in the realm of graph machine learning (GML), the situation changes considerably. The data in this context is non-Euclidean, as explained previously in table 11.1, meaning that it isn't arranged on a regular grid like pixels are in an image or points are on a 3D surface. Instead, it's represented in the form of a network or graph, which can capture complex relationships. Moreover, this data exhibits order invariance, implying that the output does not change with the rearrangement of nodes.

Unlike CNNs, which operate on a regular grid, GCNs are designed to work with data that's structured as a graph, which can represent a wide variety of irregular and complex structures. Each node is connected to its neighbors without any predefined pattern, and the convolution operation is applied to a node and its direct neighbors in the graph.

How does Google DeepMind predict the estimated time of arrival?

Have you ever wondered how Google Maps predicts the estimated time of arrival (ETA) when you're planning your trip? Google DeepMind uses a GML approach to do so. The traditional ML approach would be to break the route down into a number of road segments, predict the time to traverse each road segment using a feedforward neural network, and sum them up to get the ETA. However, the underlying assumption of feedforward NN is that the road segments are independent of each other. In reality, road segment traffic easily influences the ETA of neighboring road segments, so the samples are not independent.

(continued)

For instance, consider the situation where congestion on a minor road influences the traffic flow on a main road. When the model encompasses multiple junctions, it naturally develops the capacity to predict slowdowns at intersections, delays due to converging traffic, and the total time taken in stop-and-go traffic conditions. A better approach is to use GML to take the influence of the neighboring road segments into consideration.

In this case, the road network will first be converted into a graph where each road segment is represented as a node. If two road segments are connected to each other, their corresponding nodes will be connected by an edge in the graph. Graph embedding is then generated by GNN to map the node features and graph structures from a high-dimensional discrete graph space to a low-dimensional continuous latent space. Information is propagated and aggregated across the graph through a technique called *message passing*, where, at the end, the embedding vector for each node contains and encodes its own information as well as the network information from all its neighboring nodes, according to the degree of neighborhood. Adjacent nodes pass messages to each other. In the first pass, each node knows about its neighbor. In the second pass, every node knows about its neighbor's neighbors, and this information is encoded into the embedding, and so on. This allows us to represent the influence of the traffic in each of the neighboring road segments.

The accuracy of real time ETAs was improved by up to 50% in places like Berlin, Jakarta, São Paulo, Sydney, Tokyo, and Washington DC using this approach [7].

As illustrated in figure 11.5, given an input graph, which includes node features x_v and an adjacency matrix A, a GCN transforms the features of each node into a latent or embedding space H, while preserving the graph structure denoted by the adjacency matrix A. These latent vectors provide a rich representation of each node, making it possible to perform node classification independently.

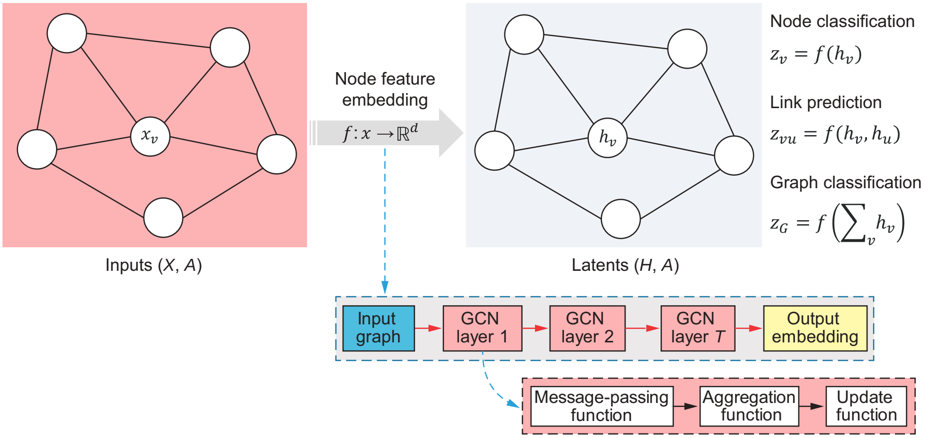


Figure 11.5 Graph embedding and node, link, and graph classification

Moreover, GCNs are also capable of predicting characteristics related to edges, such as whether a link exists between two nodes. Once node embeddings are generated, the likelihood of an edge between two nodes v and u can be predicted based on their embeddings h_v, h_u. A common approach is to compute a similarity measure (e.g., a dot product) between the embeddings of two nodes. This similarity can then be passed through a sigmoid function to predict the probability of an edge. The errors (loss) on predictions will be backpropagated and update the weights in neural networks.

Finally, GCNs enable classification at the level of the entire graph. This can be achieved by aggregating all the latent or embedding vectors (H) for all the nodes. The aggregation function used must be permutation invariant, meaning the output should remain the same regardless of the order of the nodes. Common examples of such functions are summation or averaging or maximizing. Once you've aggregated the latent vectors into a single representation, you can feed this representation into a module (e.g., a neural network layer) to predict an output for the whole graph. In essence, GCNs allow node-level, edge-level, and graph-level predictions.

To better understand how GCN works, let's consider a graph with five nodes, as shown in figure 11.6. For each node in the graph, the first step is to find the neighboring nodes. Let's assume we want to examine how the embedding for node 5 is generated. As you can see in the original graph (upper-left corner of figure 11.6), nodes 2 and 4 are neighbors of node 5. The second step is message-passing, which is the process of nodes sending, receiving, and aggregating messages from their neighbors to iteratively update their features. This allows GCNs to learn a representation for each node that captures both its own features and its context within the graph. The learned representations can then be used for downstream tasks like node classification, link prediction, or graph classification.

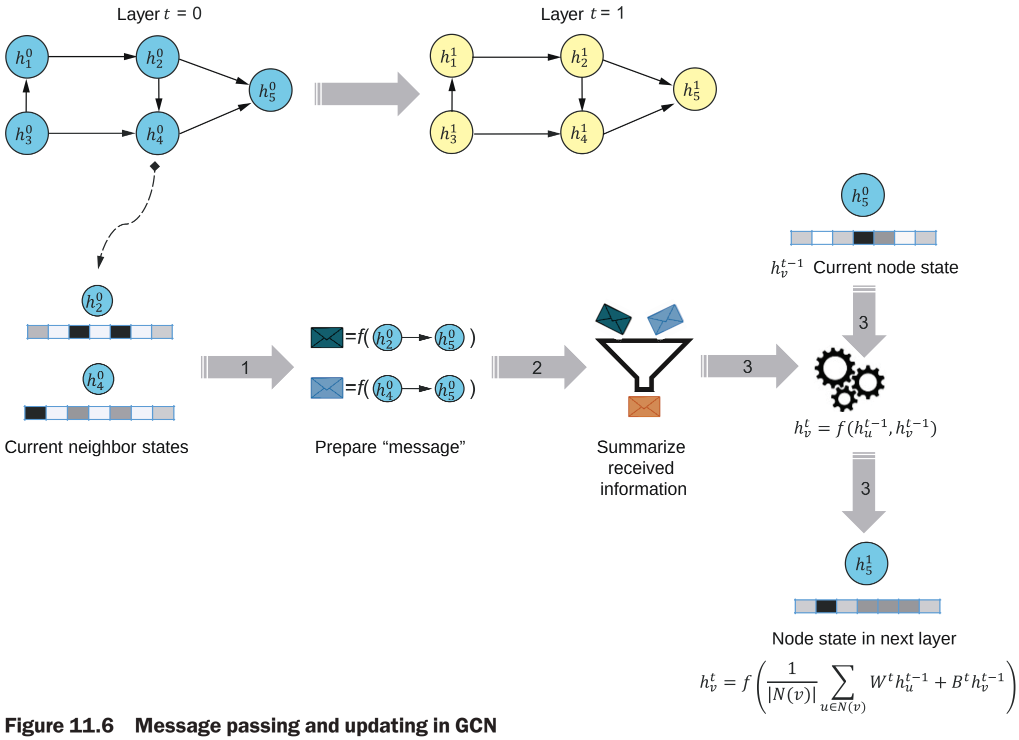
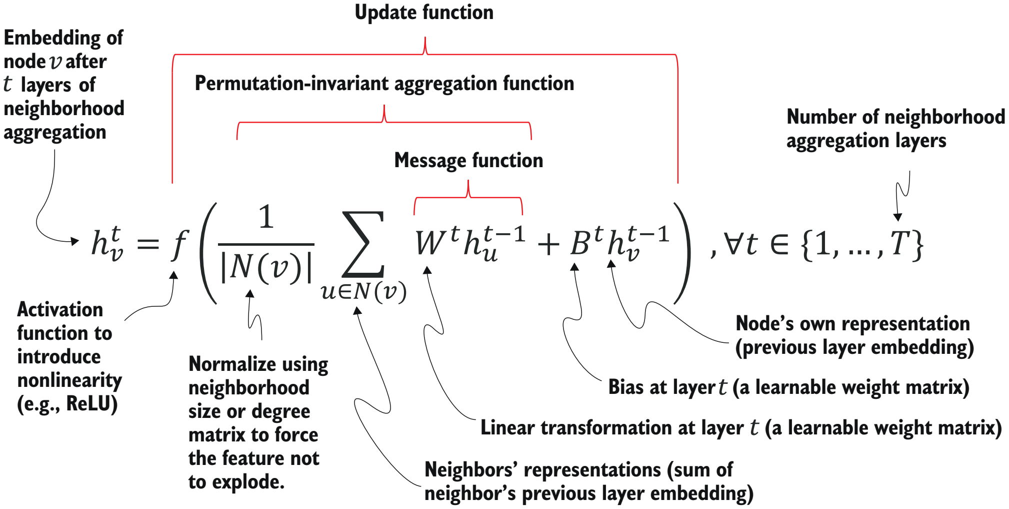


Figure 11.6 Message passing and updating in GCN

The embedding of node v after t layers of neighborhood aggregation considering $N(v)$ neighboring nodes is based on the formula shown in figure 11.7. The initial 0th layer embeddings h_v^0 are equal to node features x_v.



$$h_v^t = f\overbrace{\left(\overbrace{\frac{1}{|N(v)|}\sum_{u \in N(v)} \overbrace{W^t h_u^{t-1}}^{\text{Message function}}}^{\text{Permutation-invariant aggregation function}} + B^t h_v^{t-1}\right)}^{\text{Update function}}, \forall t \in \{1, \dots, T\}$$

Embedding of node v after t layers of neighborhood aggregation
Number of neighborhood aggregation layers
Activation function to introduce nonlinearity (e.g., ReLU)
Normalize using neighborhood size or degree matrix to force the feature not to explode.
Node's own representation (previous layer embedding)
Bias at layer t (a learnable weight matrix)
Linear transformation at layer t (a learnable weight matrix)
Neighbors' representations (sum of neighbor's previous layer embedding)

Figure 11.7 Embedding function in GCN

This formula is applied recursively to get another, better vector h at each time step, where h is the vector representation of the nodes in the latent space. The weight matrix is learned through training on given data. At the beginning, each node in the graph is aware only of its own initial features. In the first layer of the GCN, each node communicates with its immediate neighbors, aggregating its own features and receiving features from those neighbors. As we move to the second layer, each node again communicates with its neighbors. However, because the neighbors have already incorporated information from their own neighbors in the first layer, the original node now indirectly accesses information from two hops away in the graph—its neighbors' neighbors. As this process repeats through more layers in the GCN, information is propagated and aggregated across the graph. At the end, the embedding vector for each node contains and encodes its own information as well as the network information from all its neighboring nodes according to the degree of neighborhood, or its *k*-hop neighborhood, to create context embedding. The *k-hop neighborhood*, or neighborhood of radius *k*, of a node is a set of neighboring nodes at a distance less than or equal to *k*.

Listing 11.2 shows how to generate node embedding for the Cora dataset using GCN. The Cora dataset consists of 2,708 scientific publications classified into one of seven classes. The citation network consists of 5,429 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word in the dictionary. The dictionary consists of 1,433 unique words.

PyG (PyTorch Geometric) is used and can be installed as follows:

```
$conda install pytorch torchvision -c pytorch
$conda install torch_scatter
$conda install torch_sparse
$conda install torch_cluster
$conda install torch-spline-conv
$conda install torch_geometric
```

More information about PyG CUDA installation is available in the PyG documentation (https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html).

We'll start by importing the libraries we'll use.

Listing 11.2 Node embedding using GCN

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
import torch
import torch.nn.functional as F
from torch_geometric.datasets import Planetoid
from torch_geometric.nn import GCNConv
from torch_geometric.utils import to_networkx
```

PyG provides several datasets that can be loaded directly, such as KarateClub, Cora, Amazon, Reddit, etc. The Cora dataset is part of the Planetoid dataset and can be loaded as follows:

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```

As you can see in the following code, the GCN model is defined with two `GCNConv` layers (`GCNConv`) and a `torch.nn.Dropout` layer. `GCNConv` is a graph convolution layer, and `torch.nn.Dropout` is a dropout layer, which randomly zeroes some of the elements of the input tensor with probability 0.5 during training as a simple way to prevent overfitting.

The `forward` function defines the forward pass of the model. It takes a data object as input, representing the graph, and the features of the nodes and the adjacency list of the graph are extracted from the input data. The node features (`x`) are passed through the first GCN layer `conv1`, a `relu` activation function, a dropout layer, and finally the second GCN layer `conv2`. The adjacency list, `edge_index`, is required for the convolution operation in the GCN layers. The output of the network is then returned:

```
class GCN(torch.nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)
        self.dropout = torch.nn.Dropout(0.5)
    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return x
```

As a continuation of listing 11.2, the following code snippet trains the GCN model on a single graph and extracts the node embedding from the trained model. The `model` is trained for 200 epochs. Its gradients are first zeroed, then the forward pass is computed, and the negative log-likelihood loss is calculated on the training nodes (those marked by `data.train_mask`). The backward pass is then computed to get the gradients, and the optimizer performs a step to update the model parameters. The model is set to evaluation mode and is run on the graph again to obtain the final node embeddings:

Create an instance of the GCN model, and move it to the chosen device.

If CUDA is available, the code uses the GPU; otherwise, it will use the CPU.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN().to(device)
data = dataset[0].to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()

model.eval()
embeddings_pyg = model(data).detach().cpu().numpy()
```

Load the first graph in the dataset, and move it to the device.

Use the Adam optimizer with a learning rate of 0.01 and weight decay (a form of regularization) of 0.0005.

Train the model for 200 epochs.

Set evaluation mode.

Obtain the final node embeddings.

The `.detach()` function is used to detach the output from the computational graph and returns a new tensor that doesn't require a gradient. The embeddings are then moved from the GPU (if they were on the GPU) to the CPU. This is done to make the data accessible for further processing, such as converting it to a NumPy array. The generated embedding has a size of (2708, 7), where the number of nodes is 2,708 and the number of classes or subjects is 7. Dimensionality reduction using principle component analysis (PCA) is applied to visualize the embedding in 2D as shown in figure 11.8.

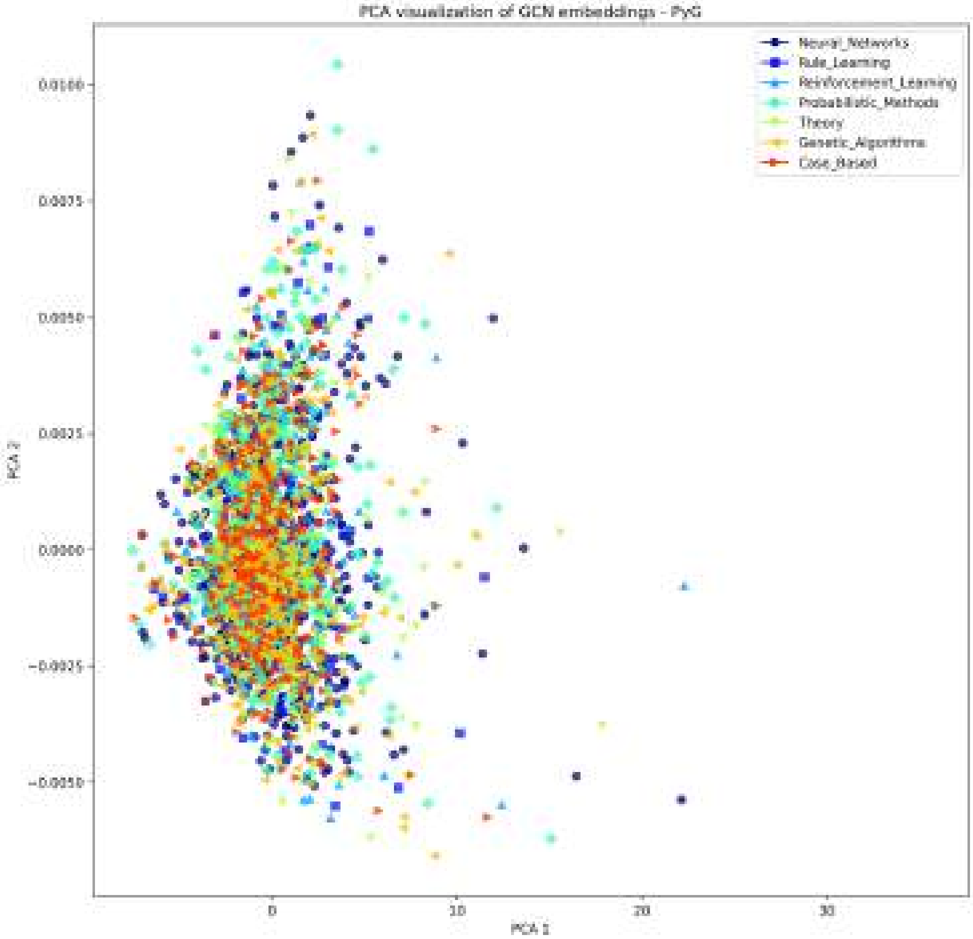


Figure 11.8 Node embedding using GCN in PyG

As you can see, the node embedding makes the nodes belonging to the same classes cluster together. This means increased discrimination power of the features, which results in more accurate predictions.

The complete version of listing 11.2 available in the book's GitHub repo also shows how to generate node embedding using the GCN available in StellarGraph. StellarGraph is a Python library for ML on graphs and networks.

11.3.2 Attention mechanisms

As you saw in figure 11.7, the embedding function in GCN consists of message passing, aggregation, and update functions. The message passing function mainly integrates messages from the node's neighbors based on a learnable weight matrix *W*. This weight matrix does not reflect the degree of importance of neighboring nodes. The convolution operation applies the same learned weights to all neighbors of a node as a linear transformation, without explicitly accounting for their importance or relevance. This might not be ideal because some segments may need more attention than others.

The concept of "attention" in DL essentially permits the model to selectively concentrate on specific segments of the input data as it produces the output sequence. This mechanism ensures that context is maintained and propagated from the initial stages to the end. It also allows the model to dynamically allocate its resources by focusing on the most important parts of the input at each time step. In a broad sense, attention in DL can be visualized as a vector consisting of importance or relevance scores. These scores help quantify the relationship or association between a node in a graph and all other nodes in the graph.

Attention is all you need

The groundbreaking paper "Attention Is All You Need" [8] proposes a new Transformer model for processing sequential data like text. In the world of language processing and translation, models usually read an entire sentence or document word by word, in order (like we do when we read a book), and then make predictions based on that. These models have some difficulties understanding long sentences and recalling information from far away in the text. In the case of long sequences, there is a high probability that the initial context will be lost by the end of the sequence. This is called the *forgetting problem*.

The authors of the paper propose a different way of handling this task. Instead of reading everything in order, their model focuses on different parts of the input at different times, almost like it's jumping around the text. This is what they refer to as "attention." The attention mechanism allows the model to dynamically prioritize which parts of the input are most relevant for each word it's trying to predict, making it more effective at understanding context and reducing confusion arising from long sentences or complex phrases. For more details, see "The Annotated Transformer" [9].

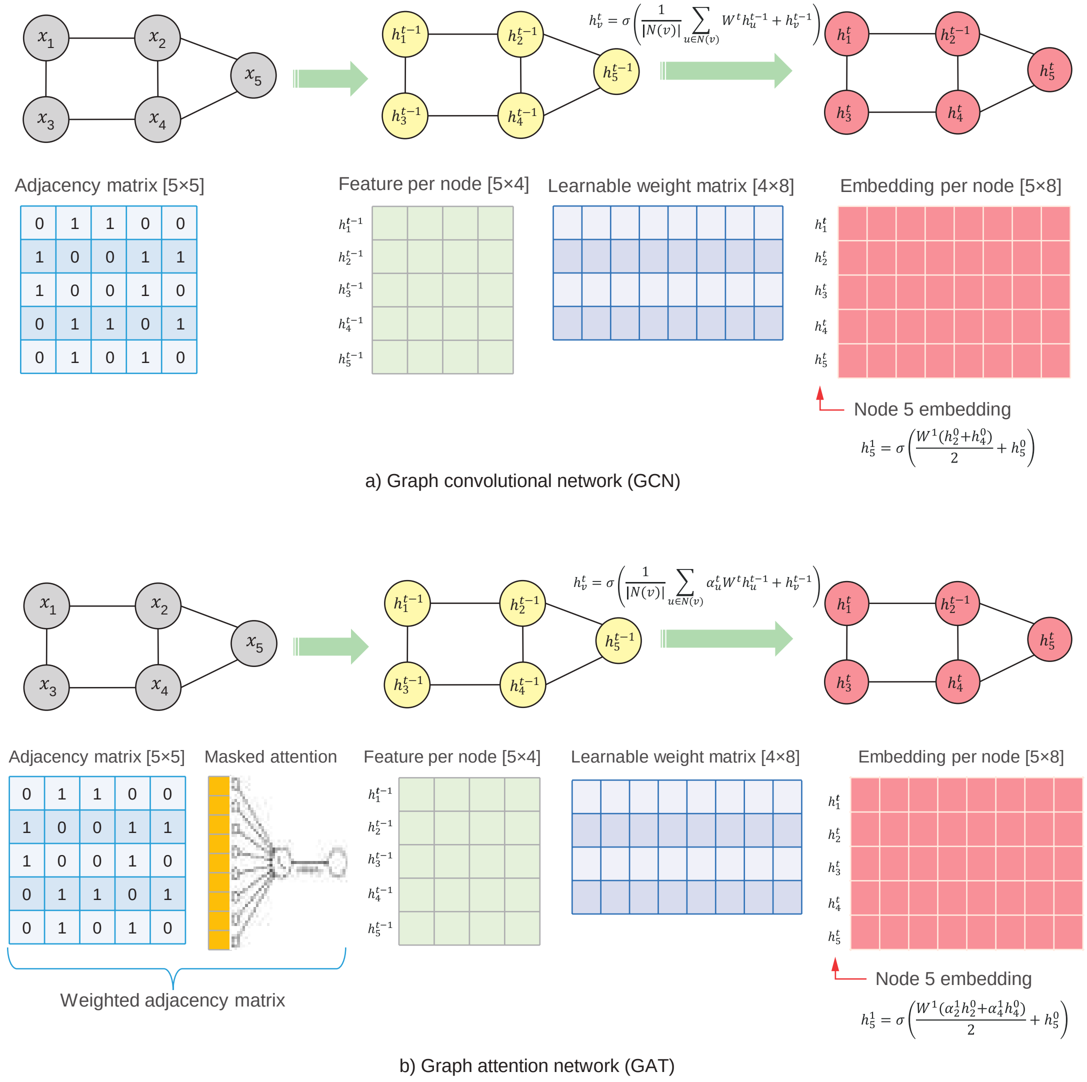
Figure 11.9b shows a graph attention network (GAT), where a weighting factor or attention coefficient α is added to the embedding equation to reflect the importance of the neighboring nodes. GAT uses a weighted adjacency matrix instead of nonweighted adjacency matrix used in case of GCN (figure 11.9a). An attentional mechanism a is used to compute unnormalized coefficients e_{vu} across pairs of nodes v and u based on their features:

$$e_{vu} = a(h_v, h_u) \tag{11.1}$$

An example of this attentional mechanism can be dot-product attention that measures the similarity or alignment between the features of the two nodes, providing a quantitative indication of how much attention node v should give to node u. Other mechanisms may involve learned attention weights, nonlinear transformations, or more complex interactions between node features. Following the graph structure, node v can attend over nodes in its neighborhood only $i \in N_v$.

Attention coefficients are typically normalized using the softmax function so that they are comparable, irrespective of the scale or distribution of raw scores in different neighborhoods or contexts. Note that in figure 11.9b, for simplicity, the attention coefficients α_{vu} are denoted as α_u.

$$\alpha_{vu} = \frac{exp(e_{vu})}{\sum_{i \in N(v)} exp(e_{vi})} \tag{11.2}$$



0	1	1	0	0
1	0	0	1	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0

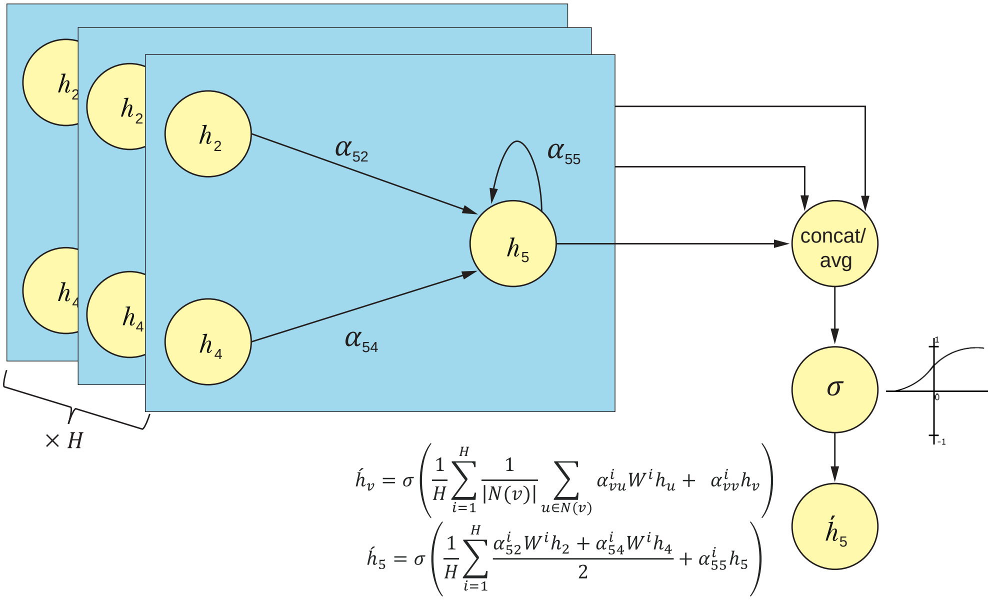
a) Graph convolutional network (GCN)

0	1	1	0	0
1	0	0	1	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0

b) Graph attention network (GAT)

Figure 11.9 Graph convolutional network (GCN) vs. graph attention network (GAT)

Multi-head attention is a key component in GATs and also in the Transformer model discussed in the “Attention Is All You Need” paper. In a multi-head attention mechanism, the model has multiple sets of attention weights. Each set (or “head”) can learn to pay attention to different parts of the input. Instead of having just one focus of attention, the model can have multiple focuses, allowing it to capture different types of relationships and patterns in the data. In the context of GATs, a multi-head attention mechanism allows each node in the graph to focus on different neighboring nodes in different ways, as shown in figure 11.10.



$$\acute{h}_v = \sigma\left(\frac{1}{H}\sum_{i=1}^{H}\frac{1}{|N(v)|}\sum_{u\in N(v)}\alpha_{vu}^{i}W^{i}h_u + \alpha_{vv}^{i}h_v\right)$$

$$\acute{h}_5 = \sigma\left(\frac{1}{H}\sum_{i=1}^{H}\frac{\alpha_{52}^{i}W^{i}h_2+\alpha_{54}^{i}W^{i}h_4}{2} + \alpha_{55}^{i}h_5\right)$$

Figure 11.10 Multi-head attention with H = 3 heads by node 5. α_{52}, α_{54}, and α_{55} are the attention coefficients between the nodes. The aggregated features from each head are averaged to obtain the final embedding of the node.

Once the multiple heads have performed their respective attention operations, their results are typically averaged. This process condenses the diverse perspectives captured by the multiple attention heads into a single output. After the results of the multi-head attention operation are combined, a final nonlinearity is then applied. This step typically involves the use of a softmax function or logistic sigmoid function, especially in classification problems. These functions serve to translate the model’s final outputs into probabilities, making the output easier to interpret and more useful for prediction tasks.

11.3.3 Pointer networks

Sequential ML involves dealing with data where the order of observations matters, such as time series data, sentences, or permutations. Sequential ML tasks can be classified based on the number of inputs and outputs, as shown in table 11.2. A *sequence-to-sequence* (seq2seq) model takes a sequence of items and outputs another sequence of items. Recurrent neural networks (RNN) and long short-term memory (LSTM) have been established as state-of-the-art approaches in seq2seq modeling.

Table 11.2 Sequential ML

Task	Example
One-to-one	Image classification. We provide a single image as input, and the model outputs the classification or category, like “dog” or “cat,” as a single output.
One-to-many	Image captioning. We input a single image into the model, and it generates a sequence of words describing that image.
Many-to-one	Sentiment analysis. We input a sequence of words (like a sentence or a tweet), and the model outputs a single sentiment score (like “positive," “negative,” or “neutral”).
Many-to-many (type 1)	Sequence input and sequence output, like in the case of named entity recognition (NER). We input a sentence (a sequence of words), and the model outputs the recognized entity, such as a person, organization, location, etc.
Many-to-many (type 2), known as a synchronized sequence model	Synced sequence input and output. The model takes a sequence of inputs but doesn’t output anything until the entire sequence has been read. Then it outputs a sequence. An example of this is video classification, where the model takes a sequence of video frames as input and then outputs a sequence of labels for those frames.

In discrete combinatorial optimization problems like the travelling salesman problem, sorting tasks, or the convex hull problem, both the input and output data are sequential. However, traditional seq2seq models struggle to solve these problems effectively. This is primarily because the discrete categories of output elements are not predetermined. Instead, they are contingent on the variable size of the input (for instance, the output dictionary is dependent on the input length). The *pointer network* (Ptr-Net) model [10] addresses this problem by utilizing attention as a mechanism to point to or select a member of the input sequence for the output. This model not only enhances performance over the conventional seq2seq model equipped with input attention, but it also enables us to generalize to output dictionaries of variable sizes.

While traditional attention mechanisms distribute attention over the input sequence to generate an output element, Ptr-Net instead uses attention as a pointer. This pointer is used to select an element from the input sequence to be included in the output sequence. Let’s consider the convex hull problem as an example of a discrete combinatorial optimization problem. A convex hull is a geometric shape, specifically a polygon, that fully encompasses a given set of points. It achieves this by optimizing two distinct parameters: it maximizes the area that the shape covers, while simultaneously minimizing the boundary or circumference of the shape, as illustrated in figure 11.11. To understand this concept, it can be useful to imagine stretching a rubber band around the extreme points or vertices of the set. When you release the rubber band, it automatically encompasses the entire set in the smallest perimeter possible, and this is essentially what a convex hull does.

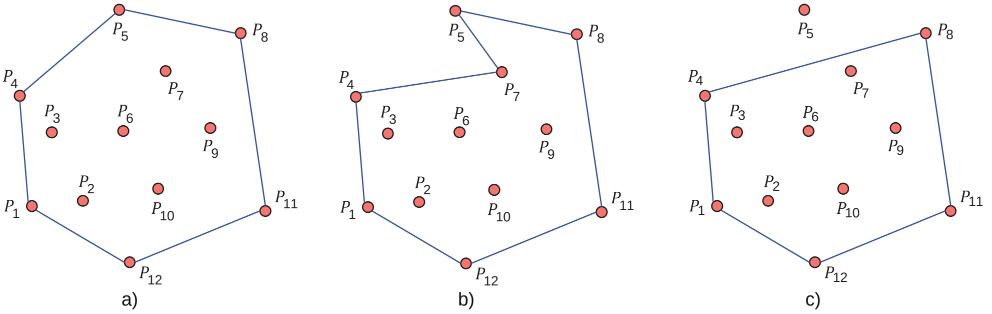


Figure 11.11 The convex hull problem. a) A valid convex hull that encloses all points while maximizing the area and minimizing the circumference. Note that the number of points included in the output sequence of the polygon may be smaller than the number of given points. b) An invalid convex hull, as the circumference is not minimized. c) An invalid convex hull, as not all the points are enclosed.

Convex hulls have a multitude of applications across a variety of disciplines. For example, in the field of image recognition, convex hulls can help determine the shape and boundary of objects within an image. Similarly, in robotics, they can assist in obstacle detection and navigation by defining the "reachable" space around a robot.

The problem of finding or computing a convex hull, given a set of points, has been addressed through various algorithms. For example, the Graham scan algorithm sorts the points according to their angle with the point at the bottom of the hull and then processes them to find the convex hull [11]. The Jarvis march (or the gift wrapping algorithm) starts with the leftmost point and wraps the remaining points like wrapping a gift [12]. The quickhull algorithm finds the convex hull of a point set by recursively dividing the set into subsets, selecting the point farthest from the line between two extreme points, and eliminating points within the formed triangles until the hull's vertices are identified [13].

As shown in figure 11.12, Ptr-Net takes as input a planar set of points $P=\{P_1, P_2, \ldots, P_n\}$ with n elements each, where $P_j = (x_j, y_j)$ are the Cartesian coordinates of the points. The outputs $C_P=\{C_1, C_2,\ldots, C_{m(P)}\}$ are sequences representing the solution associated with the point set P. In this figure, Ptr-Net estimates the output sequence [1 4 2] from the input data points [1 2 3 4]. This output sequence represents the convex hull that includes all the input points with maximum area and minimum circumference. As can be seen, the convex hull is formed by connecting P_1, P_2, and P_4. The third point P_3 is inside this convex hull.

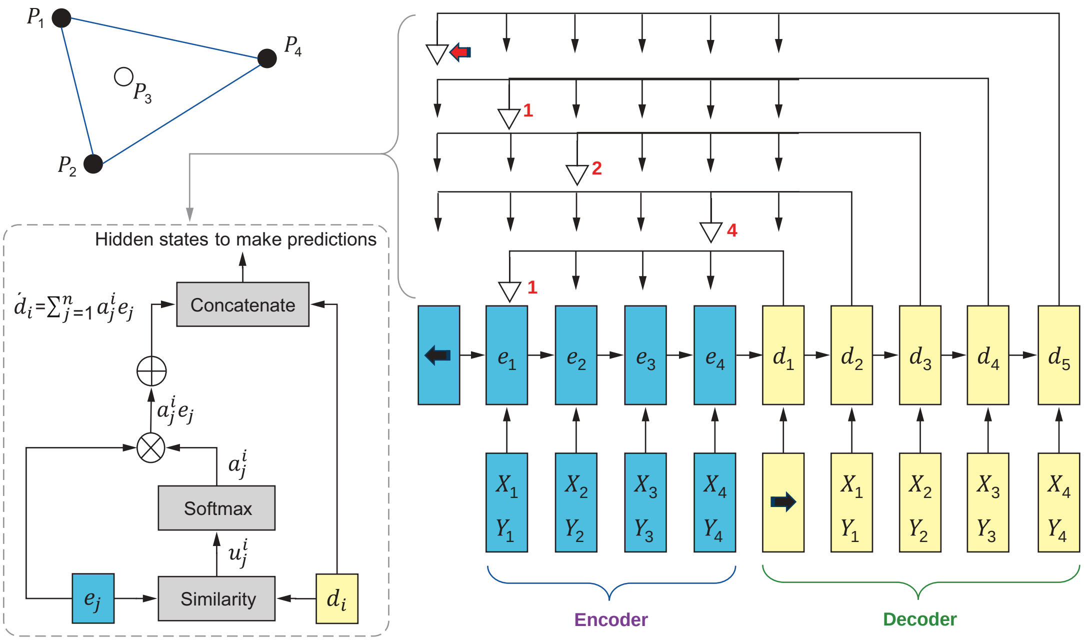


Figure 11.12 Pointer network (Prt-Net) estimating the output sequence [1 4 2] from the input data points [1 2 3 4]

Ptr-Net consists of three main components:

- *Encoder*—The encoder is a recurrent neural network (RNN), often implemented with long short-term memory (LSTM) units or gated recurrent units (GRUs). The encoder's purpose is to process the input sequence, converting each input element into a corresponding hidden state. These hidden states (e_1,..., e_n) encapsulate the context-dependent representation of the elements in the input sequence.
- *Decoder*—Like the encoder, the decoder is also an RNN. It's responsible for generating the output sequence (d_1,..., d_m). For each output step, it takes the previous output and its own hidden state as inputs.
- *Attention mechanism (pointer)*—The attention mechanism in a Ptr-Net operates as a pointer. It computes a distribution over the hidden states output by the encoder, indicating where to "point" in the input sequence for each output step. Essentially, it decides which of the inputs should be the next output. The attention mechanism is a softmax function over the learned attention scores, which gives a probability distribution over the input sequence, signifying the likeliness of each element being pointed at.

The attention vector at each output time i is computed using the following equations:

$$u_j^i = v^T \tanh\left(W_1 e_j + W_2 d_i\right) \quad j \in (1, \ldots, n) \qquad \mathbf{11.3}$$

$$a_j^i = softmax\left(u_j^i\right) \quad j \in (1, \ldots, n) \tag{11.4}$$

$$d_i' = \sum_{j=1}^{n} a_j^i e_j \tag{11.5}$$

where

- u_j is the attention vector or alignment score that represents the similarity between the decoder and encoder hidden states. v, W_1, and W_2 are learnable parameters of the model. If the same hidden dimensionality is used for the encoder and decoder (typically 512), v is a vector, and W_1 and W_2 are square matrices.
- a_j is the attention mask over the input or weights computed by applying the softmax operation to the alignment scores.
- $d_i^{'}$ is the context vector that is fed into the decoder at each time step. In other words, d_i and $d_i^{'}$ are concatenated and used as the hidden states from which the predictions are made. This weighted sum of all the encoder hidden states allows the decoder to flexibly focus the attention on the most relevant parts of the input sequence.

Ptr-Net can process variable-length sequences and solve complex combinatorial problems, especially those involving sorting or ordering tasks, where the output is a permutation of the input, as you will see in section 11.9.

11.4 Self-organizing maps

The *self-organizing map* (SOM), also known as a *self-organizing feature map* (SOFM) or *Kohonen map*, is a type of artificial neural network (ANN) that is trained with unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a *map*. SOMs are distinguished from traditional ANNs by the nature of their learning process, known as *competitive learning*. In such algorithms, processing elements or neurons compete for the right to respond to a subset of the input data. The degree to which an output neuron is activated is amplified as the similarity between the neuron's weight vector and the input grows. The similarity between the weight vector and the input, leading to neuron activation, is commonly gauged through the calculation of Euclidean distance. The output unit that demonstrates the highest level of activation, or equivalently the shortest distance, in response to a specific input is deemed the best matching unit (BMU) or the "winning" neuron, as illustrated in figure 11.13. This winner is then drawn incrementally closer to the input data point by adjusting its weight.

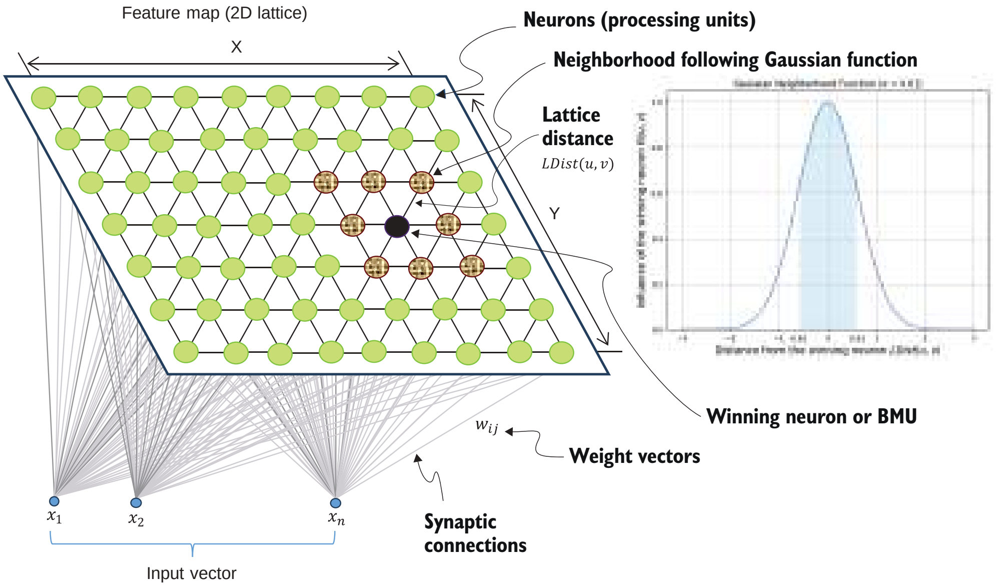


Figure 11.13 Self-organizing map (SOM) with a Gaussian neighborhood function

A key characteristic of SOM is the concept of a *neighborhood function*, which ensures that not only the winning neuron but also its neighbors learn from each new input, creating clusters of similar data. This allows the network to preserve the topological properties of the input space. Equation 11.6 shows an example of a neighborhood function:

$$\theta(u, v) = exp\left(-\frac{LDist(u, v)^2}{2\sigma^2}\right) \tag{11.6}$$

where v is the index of the node in the map, u is the index of the winning neuron, $LDist(u,v)$ represents the lattice distance between u and v, and σ is the bandwidth of the Gaussian kernel. In SOMs, σ represents the radius or width of the neighborhood and determines how far the influence of the winning neuron extends to its neighbors during the weight update phase. A large σ means a broader neighborhood is affected. On the other hand, a small σ means that fewer neighboring neurons are influenced. When σ is set to an extremely small value, the neighborhood effectively shrinks to include only the winning neuron itself. This means that only the winning neuron's weights are significantly updated in response to the input, while the weights of the other neurons are barely or not at all affected. This behavior, where only the winning neuron is updated, is referred to as "winner take all" learning.

Algorithm 11.1 shows the steps of SOM, assuming that D_t is a target input data vector, W_v is the current weight vector of node v, $\theta(u,v,s)$ is the neighborhood function that represents the restraint due to the distance from the winning neuron, and α is a learning rate where $\alpha \in (0,1)$.

Algorithm 11.1 Self-organizing map (SOM)

```
Randomly initialize the weights of each neuron
For each step s=1 to iteration limit:
    Randomly pick an input vector from the dataset
        Traverse each node in the map
            Calculate Euclidean distance as a similarity measure
            Determine the node that produces the smallest distance (winning neuron)
        Adapt the weights of each neuron v according to the following rule
        $W_v(s+1)=W_v(s)+ \alpha(s).\theta(u,v,s).\|D_t-W_v(s)\|$
```

SOMs were initially used as a dimensionality reduction method for data visualization and clustering tasks. For example, the neural phonetic typewriter was one of the early applications of Kohonen's SOM algorithm. It was a system where spoken phonemes (the smallest unit of speech that can distinguish one word from another) were recognized and converted into symbols. When someone spoke into the system, the SOM would classify the input phoneme and type the corresponding symbol. SOMs can be applied to different problems such as feature extraction, adaptive control, and travelling salesman problems (see section 11.8).

SOMs offer a significant advantage in that they preserve the relative distances between points as calculated within the input space. Points that are close in the input space are mapped onto neighboring units within the SOM, making SOMs effective tools for analyzing clusters within high-dimensional data. When using techniques like principal component analysis (PCA) to handle high-dimensional data, data loss may occur when reducing the dimensions to two. If the data contains numerous dimensions and if each dimension carries valuable information, then SOMs can be superior to PCA for dimensionality reduction purposes. Beyond this, SOMs also possess the ability to generalize. Through this process, the network can identify or categorize input data that it has not previously encountered. This new input is associated with a specific unit on the map and is thus mapped accordingly.

The previous sections have offered a fundamental foundation in ML, equipping you with essential background knowledge. The upcoming sections will delve deeply into the practical applications of supervised and unsupervised ML in tackling optimization problems.

11.5 Machine learning for optimization problems

The utilization of ML techniques to tackle combinatorial optimization problems represents an emergent and exciting field of study. *Neural combinatorial optimization* refers to the application of ML and neural network models, specifically seq2seq supervised models, unsupervised models, and reinforcement learning, to solve combinatorial optimization problems. Within this context, the application of ML to combinatorial optimization has been comprehensively described by Yoshua Bengio and his co-authors [14]. The authors depict three distinctive methods for harnessing ML for combinatorial optimization (see figure 11.14):

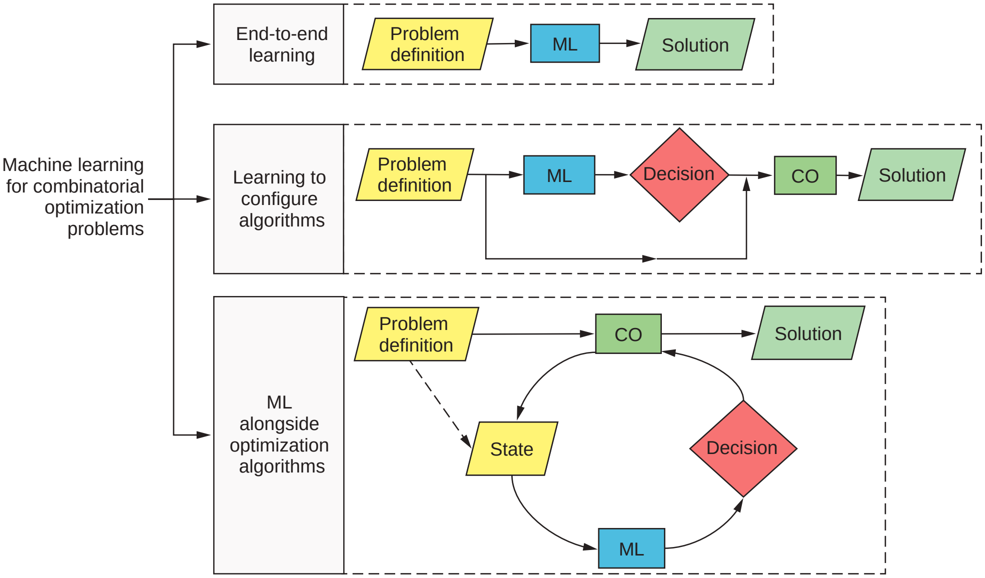


Figure 11.14 Machine learning (ML) for combinatorial optimization (CO) problems

- *End-to-end learning*—To use ML to address optimization problems, we need to instruct the ML model to formulate solutions directly from the input instance. An example of this approach is Ptr-Net, which is trained on m points and validated on n points for a Euclidean planar symmetric TSP [10]. Examples of solving combinatorial optimization problems using end-to-end learning are provided in sections 11.6, 11.7, and 11.9.
- *Learning to configure algorithms*—The second method involves applying an ML model to enhance a combinatorial optimization algorithm with pertinent information. In this regard, ML can offer a parameterization of the algorithm. Examples of such parameters comprise, but are not restricted to, learning rate or step size in gradient descent methodologies; initial temperature or cooling schedule in simulated annealing; standard deviation of Gaussian mutation or selective crossover in genetic algorithms; inertia weight or cognitive and social acceleration coefficients in particle swarm optimization (PSO); or rate of evaporation, influence of pheromone deposition, or influence of the desirability of state transition in ant colony optimization (ACO).
- *ML in conjunction with optimization algorithms*—The third method calls for a combinatorial optimization algorithm to repetitively consult the same ML model for decision-making purposes. The ML model accepts as input the current state of the algorithm, which could encompass the problem definition. The fundamental distinction between this approach and the other two lies in the repeated utilization of the same ML model by the combinatorial optimization algorithm to

make identical kinds of decisions, approximately as many times as the total number of iterations of the algorithm. An example of this approach is DL-assisted heuristic tree search (DLTS), which consists of a heuristic tree search in which decisions about which branches to explore and how to bound nodes are made by deep neural networks (DNNs) [15].

Another intriguing research paper by Vesselinova et al. delves into some pertinent questions concerning the intersection of ML and combinatorial optimization [16]. Specifically, the paper investigates the following questions:

- Can ML techniques be utilized to automate the process of learning heuristics for combinatorial optimization tasks and, as a result, solve these problems more efficiently?
- What essential ML methods have been employed to tackle these real-world problems?
- How applicable are these methods to practical domains?

This paper offers a thorough survey of various applications of supervised and reinforcement learning strategies in tackling optimization problems. The authors analyze these learning approaches by examining their application to a range of optimization problems:

- The knapsack problem (KP), where the goal is to maximize the total value of items chosen without exceeding the capacity of the knapsack
- The maximal clique (MC) and maximal independent set (MIS) problems, which both involve identifying subsets of a graph with specific properties
- The maximum coverage problem (MCP), which requires selecting a subset of items to maximize coverage
- The maximum cut (MaxCut) and minimum vertex cover (MVC) problems, which involve partitioning a graph in particular ways

In addition, the paper discusses the application of ML approaches to the satisfiability problem (SAT), which is a decision problem involving Boolean logic; the classic TSP, which requires finding the shortest possible route that visits a given set of cities and returns to the origin city; and the vehicular routing problem (VRP), which is a generalized version of TSP where multiple "salesmen" (vehicles) are allowed. More information about benchmark optimization problems is provided in appendix B.

Optimization by prompting (OPRO) is described in Chengrun et al.'s "Large Language Models as Optimizers" article as a simple and effective approach to using large language models (LLMs) as optimizers, where the optimization task is described in natural language [17]. Additional examples showcasing the use of ML in addressing optimization problems can be accessed through the AI for Smart Mobility publication hub (https://medium.com/ai4sm). To stimulate further exploration and draw more researchers into this emerging domain, a competition named Machine Learning for Combinatorial Optimization (ML4CO) was organized as part of the Neural Information Processing Systems (NeurIPS) conference. The competition posed a unique proposition for participants, requiring them to devise ML models or algorithms targeted at resolving three separate challenges. Each of these challenges mirrors a specific control task that commonly emerges in conventional optimization solvers. This competition provides a platform where researchers can explore and test novel ML strategies, contributing to the advancement of the field of combinatorial optimization.

11.6 Solving function optimization using supervised machine learning

Amortized optimization, or *learning to optimize*, is an approach where ML models are used to rapidly predict the solutions to an optimization problem. Amortized optimization methods try to learn the mapping between the decision variable space and the optimal or near-optimal solution space. The learned model can be used to predict the optimal value of an objective function, enabling fast solvers. The computation cost of the optimization process is spread out between learning and inferencing. This is the reason for the name “amortized optimization,” as the word “amortization” generally refers to spreading out costs.

B. Amos shows several examples of how to use amortized optimization to solve optimization problems in his tutorial [18]. For example, a supervised ML approach can learn to solve optimization problems over spheres. Here the objective is to find the extreme values of a function defined on the earth or other space that can be approximated with a sphere of the form

$$y^*(x) \in \underset{y \in S^2}{\arg\min} f(y, x) \qquad \textbf{11.7}$$

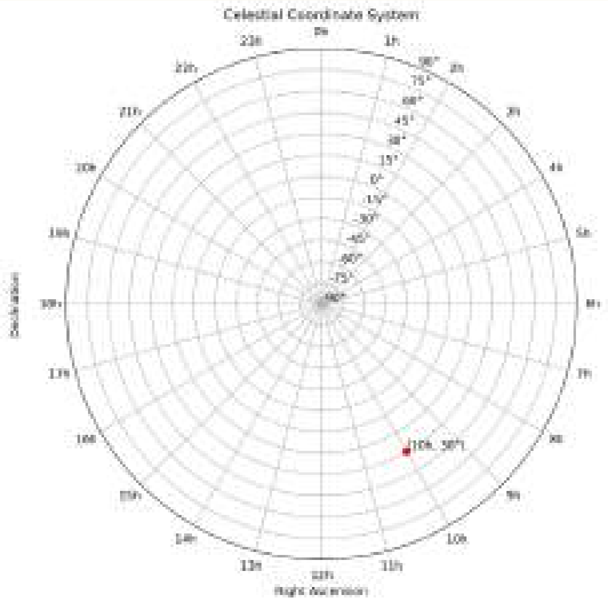
where S^2 is the surface of the unit 2-sphere embedded in real-number space R^3 as $S^2 := \{y \in R^3 \mid ||y||_2 = 1\}$, and x is some parameterization of the function $f: S^2 \times X \rightarrow R$. $||y||_2$ refers to the Euclidean norm (also known as the *L2 norm* or *2-norm*) of a vector y. More details about the amortization objective function are available in Amos’s “Tutorial on amortized optimization for learning to optimize over continuous domains” [18].

Listing 11.3 shows the steps for applying amortized optimization based on supervised learning to solve the problem of finding the extreme values of a function defined on the earth or other spaces. We’ll start by defining two conversion functions, `celestial_to_euclidean()` and `euclidean_to_celestial()`, that convert between celestial coordinates (right ascension, `ra`, and declination, `dec`) and Euclidean coordinates (`x, y, z`).

The celestial coordinate system

The *astronomical* or *celestial coordinate system* is a reference system used to specify the positions of objects in the sky, such as satellites, stars, planets, galaxies, and other celestial bodies. There are several celestial coordinate systems, with the most common being the equatorial system. In the equatorial system, right ascension (RA) and declination (Dec) are the two numbers used to fix the location of an object in the sky. These coordinates are analogous to the latitude and longitude used in earth's geographic coordinate system.

As shown in the following figure, RA is measured in hours, minutes, and seconds (h:m:s), and it is analogous to longitude in earth's coordinate system. RA is the angular distance of an object measured eastward along the celestial equator from the vernal equinox (the point where the sun crosses the celestial equator during the March equinox). The celestial equator is an imaginary great circle on the celestial sphere, lying in the same plane as earth's equator. Dec is measured in degrees and represents the angular distance of an object north or south of the celestial equator. It is analogous to latitude in earth's coordinate system.



Celestial coordinate system with an example point with a right ascension of 10 hours and a declination of 30 degrees

Positive declination is used for objects above the celestial equator, and negative declination is used for objects below the celestial equator.

The `sphere_dist(x, y)` function calculates the Riemannian distance (the great-circle distance) between two points on the sphere in the Euclidean space. This distance represents the shortest (geodesic) path between two points on the surface of a sphere, measured along the surface rather than through the interior of the sphere. The function asserts that the input vectors are two-dimensional. Then it calculates the dot product of *x* and *y* and returns the arccosine of the result, which corresponds to the angle between *x* and *y*.

Listing 11.3 Solving a function optimization problem using supervised learning

```
import torch
from torch import nn
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

def celestial_to_euclidean(ra, dec):
    x = np.cos(dec)*np.cos(ra)
    y = np.cos(dec)*np.sin(ra)
    z = np.sin(dec)
    return x, y, z
```

Convert from celestial coordinates to Euclidean coordinates.

```
def euclidean_to_celestial(x, y, z):
    sindec = z
    cosdec = (x*x + y*y).sqrt()
    sinra = y / cosdec
    cosra = x / cosdec
    ra = torch.atan2(sinra, cosra)
    dec = torch.atan2(sindec, cosdec)
    return ra, dec
```

Convert from Euclidean coordinates to celestial coordinates.

```
def sphere_dist(x,y):
    if x.ndim == 1:
        x = x.unsqueeze(0)
    if y.ndim == 1:
        y = y.unsqueeze(0)
    assert x.ndim == y.ndim == 2
    inner = (x*y).sum(-1)
    return torch.arccos(inner)
```

Calculate the Riemannian distance between two points on the sphere.

We then define a `c-convex` class as a subclass of `nn.Module`, which makes it a trainable model in PyTorch. Cohen and his co-authors defined *c-convex* in their "Riemannian convex potential maps" article as a synthetic class of optimization problems defined on the sphere [19]. The `c-convex` class models a c-convex function on the sphere with `n_components` components that we can sample data from for training. The `gamma` parameter controls the aggregation of the components of the function, and `seed` is used to initialize the random number generator for reproducibility. It also generates random parameters `ys` (which are unit vectors in the 3D space) and `alphas` (which are scalars between 0 and 0.7) for each component of the c-convex function. The parameters are concatenated into a single `params` vector. The `forward(xyz)` method calculates the value of the c-convex function at the point `xyz`:

```
class c_convex(nn.Module):
    def __init__(self, n_components=4, gamma=0.5, seed=None):
        super().__init__()
        self.n_components = n_components
        self.gamma = gamma
```

Define a c-convex function.

Sample random parameters.

```
        if seed is not None:
            torch.manual_seed(seed)
        self.ys = torch.randn(n_components, 3)
        self.ys = self.ys / torch.norm(self.ys, 2, dim=-1, keepdim=True)
        self.alphas = .7*torch.rand(self.n_components)
        self.params = torch.cat((self.ys.view(-1), self.alphas.view(-1)))

    def forward(self, xyz):
        cs = []
        for y, alpha in zip(self.ys, self.alphas):
            ci = 0.5*sphere_dist(y, xyz)**2 + alpha
            cs.append(ci)
        cs = torch.stack(cs)
        if self.gamma == None or self.gamma == 0.:
            z = cs.min(dim=0).values
        else:
            z = -self.gamma*(-cs/self.gamma).logsumexp(dim=0)
        return z
```

Computes the output of the c-convex function given input coordinates xyz on the sphere.

As a continuation of the preceding code, we define an amortized model, which takes a parameter vector as input and outputs a 3D vector representing a point on the sphere. The amortized model uses a neural network to learn a mapping from the parameter space to the 3D space of points on the sphere. The code also initializes a list of `c_convex` objects with different seeds and sets the number of parameters for the amortized model:

Create a list of integers representing different seeds.

```
seeds = [8,9,2,31,4,20,16,7]
fs = [c_convex(seed=i) for i in seeds]
n_params = len(fs[0].params)
```

Create an fs list that contains different instances of the c_convex class.

Set the number of parameters in the first c_convex object (fs[0]).

The amortized model is represented as `nn.Module` in the following code. The neural network is defined as a feedforward neural network or a multilayer perceptron that consists of three fully connected (linear) layers with ReLU activation functions:

```
class AmortizedModel(nn.Module):
    def __init__(self, n_params):
        super().__init__()
        self.base = nn.Sequential(

            nn.Linear(n_params, n_hidden),
            nn.ReLU(inplace=True),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(inplace=True),
            nn.Linear(n_hidden, 3)
        )
```

Number of parameters in the c-convex function that will be used as input to the neural network

Define the layers of the neural network in sequence.

```
    def forward(self, p):
        squeeze = p.ndim == 1
        if squeeze:
            p = p.unsqueeze(0)
        assert p.ndim == 2
```

Define the forward pass of the amortized model, which maps the input p (parameter vector) to a point on the sphere.

```
        z = self.base(p)
        z = z / z.norm(dim=-1, keepdim=True)
        if squeeze:
            z = z.squeeze(0)
        return z
```

We can now train the amortized model to learn a mapping from parameter vectors to points on the sphere. It uses a list of c_convex functions (fs) with different random seeds to generate training data. The amortized model is trained using an Adam optimizer, and its progress is visualized using a tqdm progress bar. The resulting output points on the sphere are stored in a tensor xs:

```
n_hidden = 128
torch.manual_seed(0)
model = AmortizedModel(n_params=n_params)
opt = torch.optim.Adam(model.parameters(), lr=5e-4)

xs = []
num_iterations = 100

pbar = tqdm(range(num_iterations), desc="Training Progress")

for i in pbar:
    losses = []
    xis = []
    for f in fs:
        pred_opt = model(f.params)
        xis.append(pred_opt)
        losses.append(f(pred_opt))
    with torch.no_grad():
        xis = torch.stack(xis)
        xs.append(xis)
    loss = sum(losses)

    opt.zero_grad()
    loss.backward()
    opt.step()

    pbar.set_postfix({"Loss": loss.item()})

xs = torch.stack(xs, dim=1)
```

Set the number of hidden units for the AmortizedModel neural network.

Set the random seed to ensure the reproducibility of the training process.

Create an instance of the AmortizedModel.

Create an Adam optimizer to update the parameters with a learning rate of 0.0005.

Store the output points on the sphere for each iteration of training.

Training loop

Store the losses for each c_convex function and the corresponding output points on the sphere (xis).

Iterate over each c_convex function (f) in the list fs.

After training is complete, all the predicted output points on the sphere are stacked along a new dimension, resulting in a tensor xs with the following shape: number of iterations, number of c_convex functions, 3. Each element in this tensor represents a point on the sphere predicted by the amortized model at different stages of training. It generates a visual representation of the training progress for the amortized model and c_convex functions, as shown in figure 11.15.

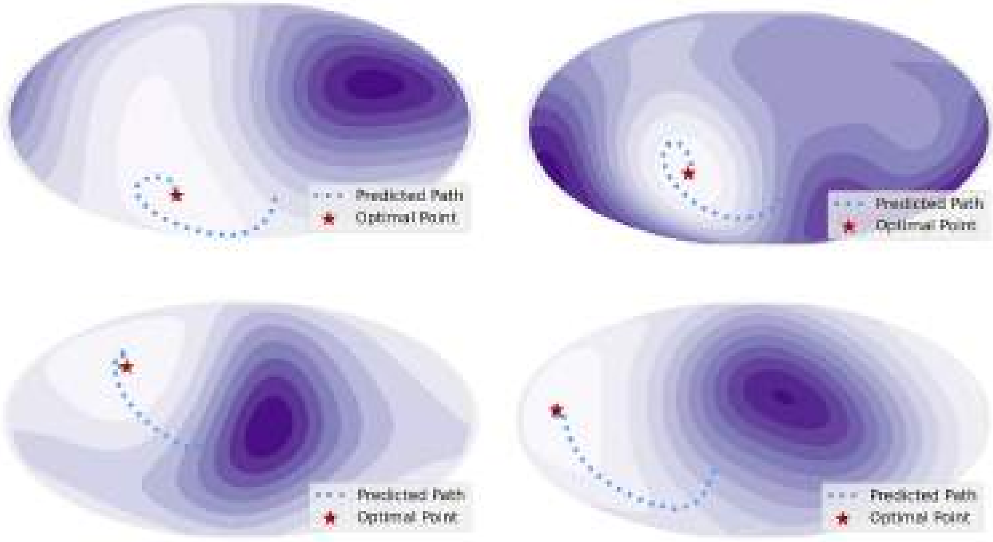


Figure 11.15 Examples of output from the trained amortized model

The complete version of listing 11.3 is available in the book's GitHub repo. It creates a grid of celestial coordinates, evaluates the `c_convex` functions and the amortized model on this grid, and then plots contour maps of the functions, the predicted paths, and the optimal points on the sphere. The optimal points are the points that give minimum loss, given that supervised learning is used to train the amortized model.

11.7 Solving TSP using supervised graph machine learning

Joshi, Laurent, and Bresson, in their "Graph Neural Networks for the Travelling Salesman Problem" article [20], proposed a generic end-to-end pipeline to tackle combinatorial optimization problems such as the traveling salesman problem (TSP), vehicle routing problem (VRP), satisfiability problem (SAT), maximum cut (MaxCut), and maximal independent set (MIS). Figure 11.16 shows the steps of solving TSP using ML.

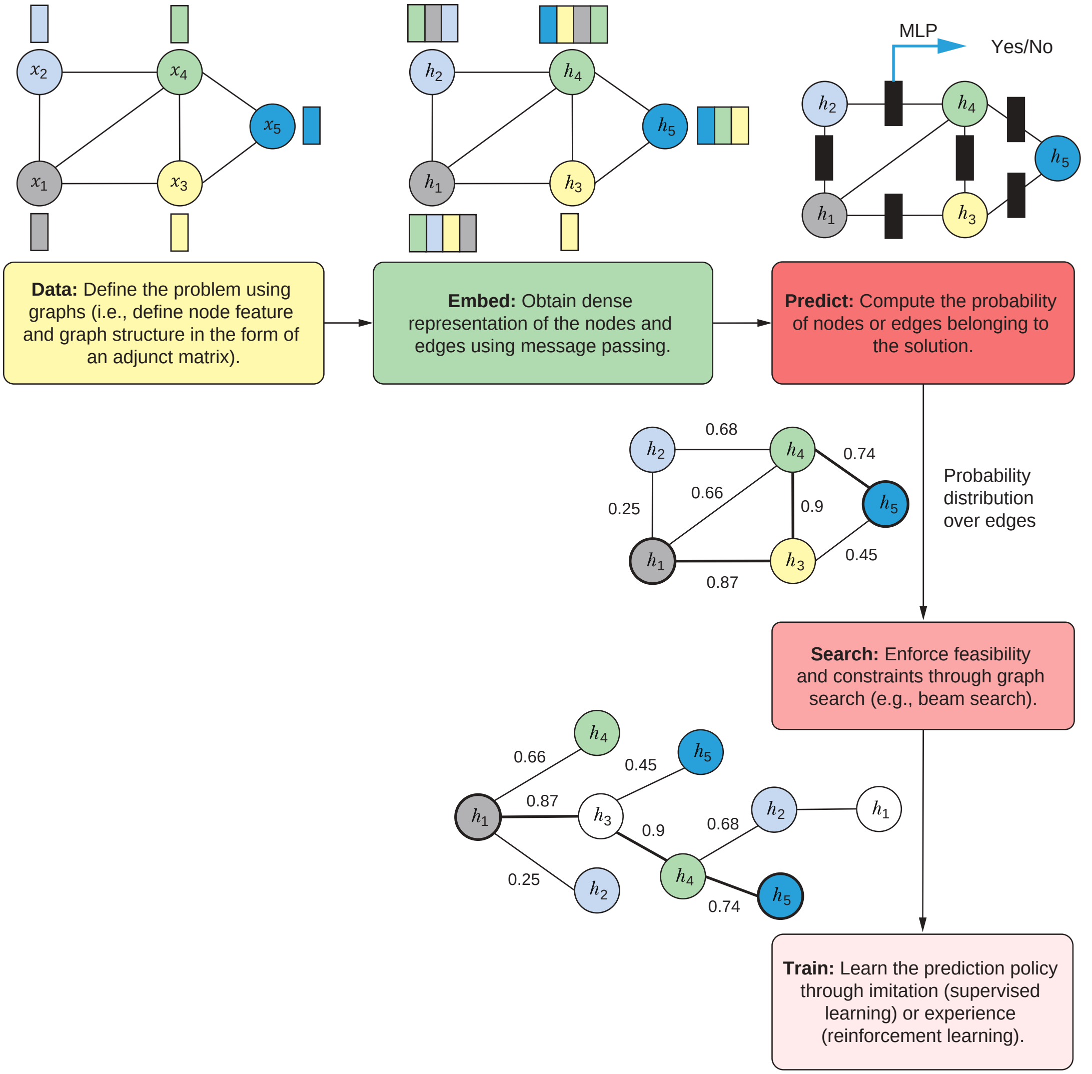

x_2
x_4
x_5
x_1
x_3
h_2
h_4
h_5
h_1
h_3
MLP
Yes/No
Data: Define the problem using graphs (i.e., define node feature and graph structure in the form of an adjunct matrix).
Embed: Obtain dense representation of the nodes and edges using message passing.
Predict: Compute the probability of nodes or edges belonging to the solution.
0.68
0.74
0.66
0.25
0.9
0.45
0.87
Probability distribution over edges
Search: Enforce feasibility and constraints through graph search (e.g., beam search).
0.66
0.45
0.87
0.9
0.68
0.25
0.74
Train: Learn the prediction policy through imitation (supervised learning) or experience (reinforcement learning).

Figure 11.16 End-to-end pipeline for combinatorial optimization problems

Following this approach, we start by defining the graph problem in the form of node features and an adjacency matrix between the nodes. A low-dimensional graph embedding is then generated using GNN or GCN, based on the message-passing approach. The probability of nodes or edges belonging to the solution is predicted using multilayer perceptrons (MLPs). A graph search, such as beam search (see chapter 4), is then applied to search the graph with the probability distribution over the edge to find a feasible candidate solution. Learning by imitation (supervised learning) and learning by exploration (reinforcement learning) are applied. Supervised learning minimizes the loss between optimal solutions (obtained by a well-known solver such as Concorde in the case of TSP) and the model's prediction. The reinforcement learning approach uses a policy gradient to minimize the length of the tour predicted by the model at the end of decoding. Reinforcement learning is discussed in the next chapter.

Training an ML model from scratch and applying it to solve TSP requires a substantial amount of code and data preprocessing. Listing 11.4 shows how you can use the pretrained models to solve different instances of TSP. We start by importing the libraries and modules we'll use. These libraries provide functionality for handling data, performing computations, visualization, and optimization. The Gurobi library is used to eliminate subtours during optimization and to calculate the reduce costs for a set of points (see appendix A). We set the `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` environment variables to control the GPU device visibility.

Listing 11.4 Solving TSP using supervised ML

```
import os
import math
import itertools
import numpy as np
import networkx as nx
from scipy.spatial.distance import pdist, squareform
import seaborn as sns
import matplotlib.pyplot as plt

import torch
from torch.utils.data import DataLoader
from torch.nn import DataParallel

from learning_tsp.problems.tsp.problem_tsp import TSP
from learning_tsp.utils import load_model, move_to

from gurobipy import *

os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

As a continuation, the following `opts` class contains several class-level attributes that define the following options and configurations:

- dataset path—The TSP dataset available in the book's GitHub repo.
- batch size—This determines the number of TSP instances (problems) processed simultaneously during training or evaluation. It specifies how many TSP instances are grouped together and processed in parallel.
- number of samples—This is the number of samples per TSP size.
- neighbors—This is used in the TSP data processing pipeline to specify the proportion (percentage) of nearest neighbors to consider for graph sparsification. It controls the connectivity of the TSP graph by selecting a subset of the nearest neighbors for each node.
- knn strategy—This is the strategy used to determine the number of nearest neighbors when performing graph sparsification. In the code, the 'percentage' value indicates that the number of nearest neighbors is determined by the neighbors parameter, which specifies the percentage of neighbors to consider.
- model—This is the path for the pretrained ML model. The model used is a pretrained GNN model available in the book's GitHub repo.
- use_cuda—This checks if CUDA is available on the system. CUDA is a parallel computing platform and programming model that allows for efficient execution of computations on NVIDIA GPUs. torch.cuda.is_available() returns a Boolean value (true or false) indicating whether CUDA is available or not. If CUDA is available, that means a compatible NVIDIA GPU is present on the system and can be utilized for accelerated computations.
- device—This is the device to be used for computations:

```
class opts:
    dataset_path = "learning_tsp/data/tsp20-50_concorde.txt"
    batch_size = 16
    num_samples = 1280

    neighbors = 0.20
    knn_strat = 'percentage'

    model =
➥ "learning_tsp/pretrained/tspsl_20-50/sl-ar-var-20pnn-gnn-
max_20200308T172931"

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda:0" if use_cuda else "cpu")
```

The next step is to create a dataset object using the TSP class with the following parameters:

- filename—The path or filename of the dataset to be used, specified by opts .dataset_path
- batch_size—The number of samples to include in each batch, specified by opts.batch_size

- num_samples—The total number of samples to include in the dataset, specified by opts.num_samples
- neighbors—The value representing the number of nearest neighbors for graph sparsification, specified by opts.neighbors
- knn_strat—The strategy for selecting nearest neighbors ('percentage' or None), specified by opts.knn_strat
- supervised—A Boolean value indicating whether the dataset is used for supervised learning, set to True

The make_dataset method creates an instance of the TSP dataset class and initializes it with the provided arguments, returning the dataset object:

```
dataset = TSP.make_dataset(
    filename=opts.dataset_path, batch_size=opts.batch_size,
➥ num_samples=opts.num_samples,
➥ neighbors=opts.neighbors, knn_strat=opts.knn_strat, supervised=True
)
```

The following line creates a data loader object that enables convenient iteration over the dataset in batches, which is useful for processing the data during evaluation. The dataset object created in the previous line will be used as the source of the data. You can provide other optional arguments to customize the behavior of the data loader, such as shuffle (to shuffle the data) and num_workers (to specify the number of worker processes for data loading):

```
dataloader = DataLoader(dataset, batch_size=opts.batch_size, shuffle=False,
➥ num_workers=0)
```

We can now load the trained model and assign it to the model variable. If the model is wrapped in torch.nn.DataParallel, it extracts the underlying module by accessing model.module. DataParallel is a PyTorch wrapper that allows for parallel execution of models on multiple GPUs. If the model is indeed an instance of DataParallel, it extracts the underlying model module by accessing the module attribute. This step is necessary to ensure consistent behavior when accessing model attributes and methods. The decode type of the model is then set to "greedy". This means that during inference or evaluation, the model should use a greedy decoding strategy to generate output predictions:

```
model, model_args = load_model(opts.model, extra_logging=True)    ◄ Load a pretrained model.
model.to(opts.device)

if isinstance(model, DataParallel):    ◄ Extract the underlying module.
    model = model.module

model.set_decode_type("greedy")    ◄ Set the decoding type of the model to "greedy".

model.eval()    ◄ Set the model's mode to evaluation.
```

The complete version of listing 11.4, including the visualization code, is available in the book's GitHub repo. Figure 11.17 shows the output produced by the pretrained ML model for the TSP50 instance.

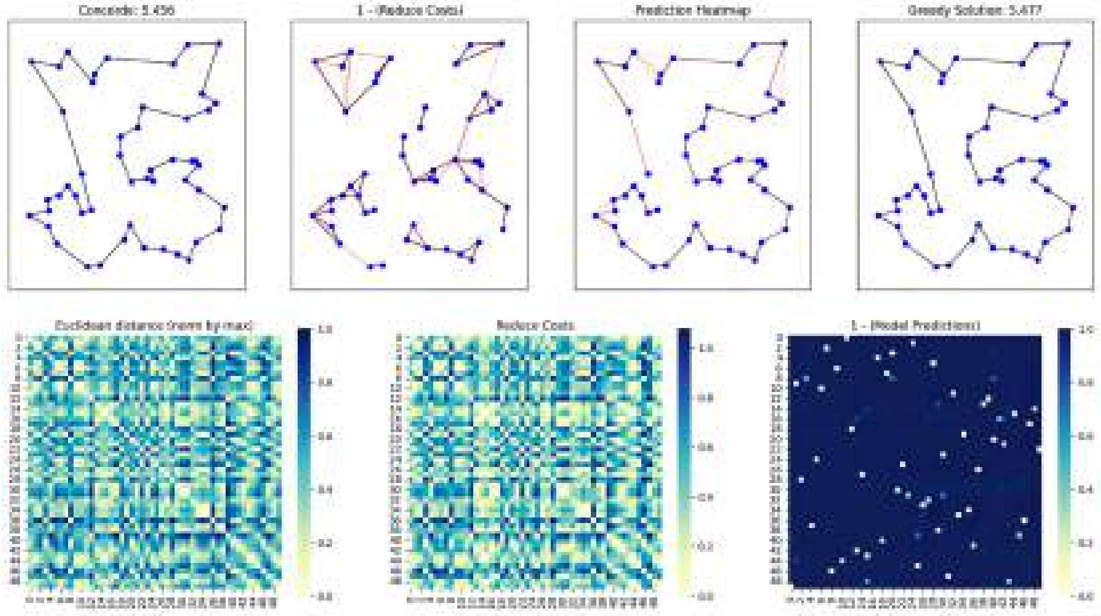


Figure 11.17 The TSP50 solution using a pretrained ML model

The figure shows the following seven plots related to the TSP instance and the model's predictions:

- *Concorde*—The plot in the upper-left corner shows the ground truth solution generated by the Concorde solver, which is an efficient implementation of the branch-and-cut algorithm for solving TSP instances to optimality. It shows the nodes of the TSP problem as circles connected by edges, representing the optimal tour calculated by Concorde. The title of the plot indicates the length (cost) of the tour obtained from Concorde.
- *1 - (Reduce Costs)*—The second plot contains the shortest subtour and shows the reduced costs for the points in these subtours using the Gurobi optimization library. It displays the edges of the TSP as red lines, with the edge color indicating the reduced cost value.
- *Prediction Heatmap*—The third plot presents a heatmap visualization of the model's predictions for the TSP problem. It uses a color scale to represent the prediction probabilities of edges, with higher probabilities shown in darker shades.

- *Greedy Solution*—The fourth plot illustrates the solution generated by the ML model using a greedy decoding strategy. It displays the nodes of the TSP problem connected by edges, representing the tour obtained from the model. The title of the plot shows the length (cost) of the tour calculated by the model.
- *Euclidean Distance (norm by max)*—The lower-left plot is a heatmap visualization of the Euclidean distances between nodes in the TSP problem. It uses a color scale to represent the distances, with lighter shades indicating smaller distances.
- *Reduce Costs*—The lower-middle plot is a heatmap representation of the reduced costs of edges in the TSP problem. It shows the reduced costs as a color scale, with lower values displayed in lighter shades.
- *1 - (Model Predictions)*—The lower-right plot presents a heatmap visualization of the model's predictions for the TSP problem, similar to the third plot. However, in this case, the heatmap displays "1 - (Model Predictions)" by subtracting the model's prediction probabilities from 1. Darker shades represent lower probabilities, indicating stronger confidence in the edge selection.

This example demonstrated how we can employ a pretrained GNN model for solving TSP. Figure 11.17 displays the model's solution alongside the Concorde TSP solver's results for a TSP instance comprising 50 points of interest. More information and complete code, including model training steps, are available in "Learning the Travelling Salesperson Problem Requires Rethinking Generalization" GitHub repo [21].

11.8 Solving TSP using unsupervised machine learning

As an example of an unsupervised ML approach, listing 11.5 shows how we can solve TSP using self-organizing maps (SOMs). We start by importing the libraries we'll use. Some helper functions are imported from the som-tsp implementation described in Vicente's blog post [22] to read the TSP instance, get the neighborhood, get the route, select the closest candidate, and calculate the route distance and plot the route. We read the TSP instance from the provided URL and obtain the cities and normalize their coordinates to a range of [0, 1].

Listing 11.5 Solving TSP using unsupervised learning

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
import requests
import os
from tqdm import tqdm

from som_tsp.helper import read_tsp, normalize, get_neighborhood, get_route,
➥ select_closest, route_distance, plot_network, plot_route

url = 'https://raw.githubusercontent.com/Optimization-Algorithms-Book/Code-
➥Listings/256207c4a8badc0977286c48a6e1cfd33237a51d/Appendix%20B/data/TSP/'
```

Define the URL where the TSP instances are located.

```
tsp='qa194.tsp'

response = requests.get(url+tsp)
response.raise_for_status()
problem_text = response.text
with open(tsp, 'w') as file:
    file.write(problem_text)

problem = read_tsp(tsp)

cities = problem.copy()
cities[['x', 'y']] = normalize(cities[['x', 'y']])
```

TSP instance

Download the file if it does not exist.

Read the TSP problem.

Obtain the normalized set of cities (with coordinates in [0,1]).

We can now set up various parameters and initialize a network of neurons for the SOM:

```
number_of_neurons = cities.shape[0] * 8

iterations = 12000

learning_rate=0.8

network = np.random.rand(number_of_neurons, 2)
```

The population size is 8 times the number of cities.

Set the number of iterations.

Set the learning rate.

Generate an adequate network of neurons.

As a continuation, the following code snippet implements the training loop for SOM. This loop iterates over the specified number of training iterations using `tqdm` to show a progress bar:

```
route_lengths = []
paths_x = []
paths_y = []

for i in tqdm(range(iterations)):
    if not i % 100:
        print('\t> Iteration {}/{}'.format(i, iterations), end="\r")

    city = cities.sample(1)[['x', 'y']].values
    winner_idx = select_closest(network, city)

    gaussian = get_neighborhood(winner_idx, number_of_neurons // 10,
➥  network.shape[0])

    network += gaussian[:, np.newaxis] * learning_rate * (city - network)

    paths_x.append(network[:, 0].copy())
    paths_y.append(network[:, 1].copy())

    learning_rate = learning_rate * 0.99997
    number_of_neurons = number_of_neurons * 0.9997

    if not i % 1000:
        plot_network(cities, network, name='diagrams/{:05d}.png'.format(i))
```

Store the lengths of the TSP routes during the SOM training iterations.

Store the x and y coordinates of the neurons in the network during the training iterations.

Training loop

Print only if the current iteration index is a multiple of 100.

Choose a random city.

Find the index of the neuron (winner) in the SOM network that is closest to the randomly chosen city.

Generate a filter that applies changes to the winner's gaussian.

Update the network's weights.

Append the current coordinates to the paths.

Decay the learning rate and the neighborhood radius n at each iteration to gradually reduce the influence of the Gaussian filter over time.

Check for the plotting interval.

```
        if number_of_neurons < 1:
            print('Radius has completely decayed, finishing execution',
                  ➥ 'at {} iterations'.format(i))
            break
        if learning_rate < 0.001:
            print('Learning rate has completely decayed, finishing execution',
                  ➥ 'at {} iterations'.format(i))
            break

        route = get_route(cities, network)
        problem = problem.reindex(route)
        distance = route_distance(problem)
        route_lengths.append(distance)
    else:
        print('Completed {} iterations.'.format(iterations))
```

Check if any parameter has completely decayed.

Calculate distance, and store it in the route_lengths list.

Indicate that the specified number of training iterations has been completed.

The following code snippet plots the route length in each iteration.

```
plt.figure(figsize=(8, 6))
plt.plot(range(len(route_lengths)), route_lengths, label='Route Length')
plt.xlabel('Iterations')
plt.ylabel('Route Length')
plt.title('Route Length per Iteration')
plt.grid(True)
plt.show()
```

Figure 11.18 shows the route length per iteration. The final route length is 9,816, and the optimal length for the Qatar TSP instance used, `qa194.tsp`, is 9,352.

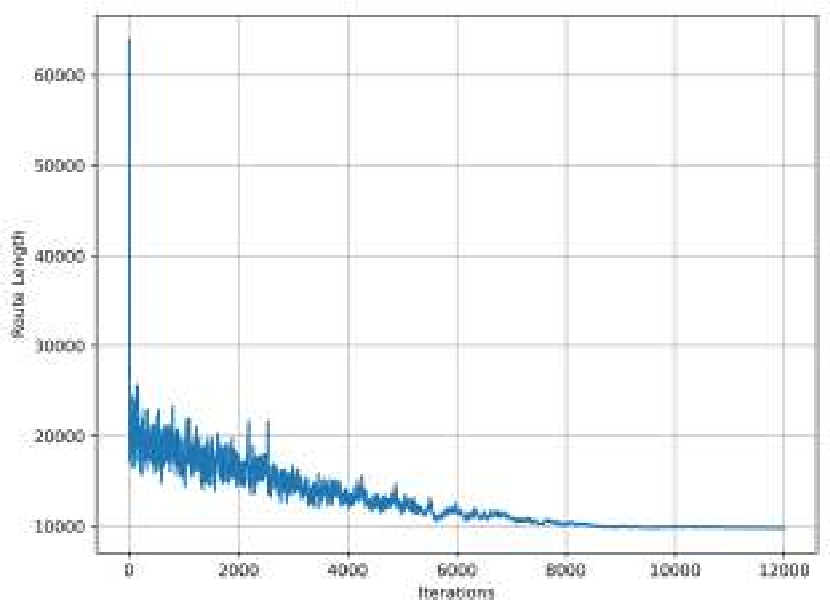


Figure 11.18 Route length per iteration of SOM for the Qatar TSP. The final route length is 9,816, and the optimal solution is 9,352.

The complete version of listing 11.5 is available in the book's GitHub repo, and it contains an implementation based on MiniSom. MiniSom is a minimalistic and Numpy-based implementation of SOM. You can install this library using `!pip install minisom`. However, the route obtained by MiniSom is 11,844.47, which is far from the optimal length of 9,352 for this TSP instance. To improve the result, you can experiment with the provided code and try to tune SOM parameters such as the number of neurons, the sigma, the learning rate, and the number of iterations.

11.9 Finding a convex hull

Ptr-Net can be used to tackle the convex hull problem using a supervised learning approach, as described by Vinyals and his co-authors in their "Pointer networks" article [10]. Ptr-Net has two key components: an encoder and a decoder, as illustrated in figure 11.19.

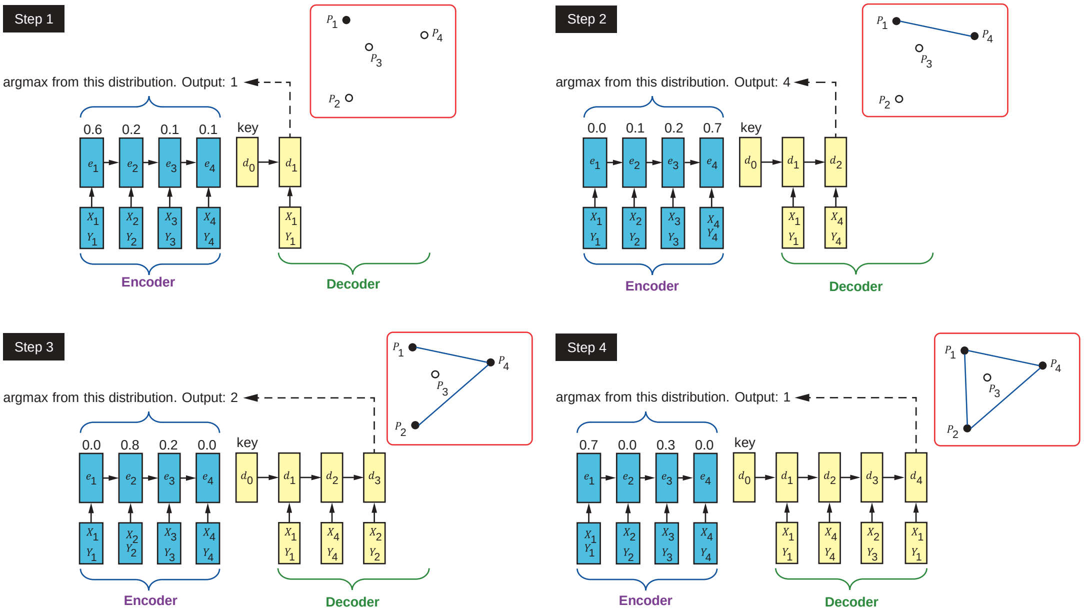

Step 1
argmax from this distribution. Output: 1
0.6 0.2 0.1 0.1
key
e_1 e_2 e_3 e_4 d_0 d_1
X_1 Y_1 X_2 Y_2 X_3 Y_3 X_4 Y_4 X_1 Y_1
Encoder
Decoder
P_1 P_4 P_3 P_2
Step 2
argmax from this distribution. Output: 4
0.0 0.1 0.2 0.7
key
e_1 e_2 e_3 e_4 d_0 d_1 d_2
X_1 Y_1 X_2 Y_2 X_3 Y_3 X_4 Y_4 X_1 Y_1 X_4 Y_4
Encoder
Decoder
P_1 P_4 P_3 P_2
Step 3
argmax from this distribution. Output: 2
0.0 0.8 0.2 0.0
key
e_1 e_2 e_3 e_4 d_0 d_1 d_2 d_3
X_1 Y_1 X_2 Y_2 X_3 Y_3 X_4 Y_4 X_1 Y_1 X_4 Y_4 X_2 Y_2
Encoder
Decoder
P_1 P_4 P_3 P_2
Step 4
argmax from this distribution. Output: 1
0.7 0.0 0.3 0.0
key
e_1 e_2 e_3 e_4 d_0 d_1 d_2 d_3 d_4
X_1 Y_1 X_2 Y_2 X_3 Y_3 X_4 Y_4 X_1 Y_1 X_4 Y_4 X_2 Y_3 X_1 Y_1
Encoder
Decoder
P_1 P_4 P_3 P_2

Figure 11.19 Solving the convex hull problem using Ptr-Net. The output in each step is a pointer to the input that maximizes the probability distribution.

The encoder, a recurrent neural network (RNN), converts the raw input sequence. In this case, it coordinates delineating the points for which we want to determine the convex hull into a more manageable representation.

This encoded vector is then passed on to the decoder. The vector acts as the modulator for a content-based attention mechanism, which is applied over the inputs. The content-based attention mechanism can be likened to a spotlight that highlights different segments of the input data at varying times, focusing on the most pertinent parts of the task at hand.

The output of this attention mechanism is a softmax distribution with a dictionary size equal to the length of the input. This softmax distribution gives probabilities to every point in the input sequence. This setup allows Ptr-Net to probabilistically decide at each step which point should be added next to the convex hull. This is determined based on the current state of the input and the network's internal state. The training process is repeated until the network has made a decision for every point, yielding a complete resolution to the convex hull problem.

Listing 11.6 shows the steps for solving convex hull problem using pointer networks. We start by importing several necessary libraries and modules, such as torch, numpy, and matplotlib. The three helper classes `Data`, `ptr_net`, and `Disp` are imported based on the implementations provided in McGough's "Pointer Networks with Transformers" article [23]. They contain functions for generating training and validation data, defining the pointer network architecture, and visualizing the results. This code generates two datasets for training and validation respectively. These datasets consist of random 2D points, where the number of points in each sample (the convex hull problem's input) varies between `min_samples` and `max_samples`. `Scatter2DDataset` is a custom dataset class used to generate these random 2D point datasets.

Listing 11.6 Solving a convex hull problem using pointer networks

```
import numpy as np
import torch
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull

from ptrnets.Data import display_points_with_hull, cyclic_permute,
➥ Scatter2DDataset,Disp_results
from ptrnets.ptr_net import ConvexNet, AverageMeter, masked_accuracy,
➥ calculate_hull_overlap

min_samples = 5
max_samples = 50
n_rows_train = 100000
n_rows_val = 1000

torch.random.manual_seed(231)
train_dataset = Scatter2DDataset(n_rows_train, min_samples, max_samples)
val_dataset = Scatter2DDataset(n_rows_val, min_samples, max_samples)
```

Running this code generates 100,000 training points and 1,000 validation points. We can then set the parameters of the pointer network. These parameters include a `TOKENS` dictionary containing the following tokens:

- `<eos>`—End-of-sequence token with the index 0
- `c_inputs`—Number of input features for the model
- `c_embed`—Number of embedding dimensions

- c_hidden—Number of hidden units in the model
- n_heads—Number of attention heads in the multi-head self-attention mechanism
- n_layers—Number of layers in the model
- dropout—Dropout probability, which is used for regularization
- use_cuda—A Boolean flag indicating whether to use CUDA (GPU) if available or CPU
- n_workers—Number of worker threads for data loading in DataLoader

The training parameters include n_epochs (number of training epochs), batch_size (batch size used during training), lr (learning rate for the optimizer), and log_interval (interval for logging training progress). The code checks if CUDA (GPU) is available and sets the device variable accordingly:

```
TOKENS = {'<eos>': 0 }
c_inputs = 2 + len(TOKENS)
c_embed = 16
c_hidden = 16
n_heads = 4
n_layers = 3
dropout = 0.0
use_cuda = True
n_workers = 2
n_epochs = 5
batch_size = 16
lr = 1e-3
log_interval = 500
device = torch.device("cuda" if torch.cuda.is_available() and use_cuda else
"cpu")
```

As a continuation, we load the training and validation data with the specified batch_size and num_workers:

```
train_loader = DataLoader(train_dataset, batch_size=batch_size,
➥   num_workers=n_workers)

val_loader = DataLoader(val_dataset, batch_size=batch_size,
➥   num_workers=n_workers)
```

The ConvexNet model is a Ptr-Net model that is implemented as a transformer architecture with an encoder and decoder that use nn.TransformerEncoderLayer and apply multi-head self-attention. The complete code is available in the ptr_net.py class, available in the book's GitHub repo. This model is initialized with the predefined hyperparameters. The AverageMeter class is used for keeping track of the average loss and accuracy during training and validation:

Create a ConvexNet model.

Use the Adam optimizer for training the model.

```
model = ConvexNet(c_inputs=c_inputs, c_embed=c_embed, n_heads=n_heads,
➥   n_layers=n_layers, dropout=dropout, c_hidden=c_hidden).to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```
criterion = torch.nn.NLLLoss(ignore_index=TOKENS['<eos>'])

train_loss = AverageMeter()
train_accuracy = AverageMeter()
val_loss = AverageMeter()
val_accuracy = AverageMeter()
```

Use negative log-likelihood loss as the loss function.

Keep track of the average loss and accuracy during training and validation.

We can now perform the training and evaluation loop for a model (`ConvexNet`) using PyTorch. The model is being trained on the `train_loader` dataset with known labels and evaluated on the `val_loader` dataset:

```
for epoch in range(n_epochs):
  model.train()
  for bat, (batch_data, batch_labels, batch_lengths) in enumerate(train_
loader):
    batch_data = batch_data.to(device)
    batch_labels = batch_labels.to(device)
    batch_lengths = batch_lengths.to(device)

    optimizer.zero_grad()
    log_pointer_scores, pointer_argmaxs = model(batch_data, batch_lengths,
➥     batch_labels=batch_labels)
    loss = criterion(log_pointer_scores.view(-1, log_pointer_scores.
➥ shape[-1]), batch_labels.reshape(-1))

    assert not np.isnan(loss.item()), 'Model diverged with loss = NaN'

    loss.backward()
    optimizer.step()

    train_loss.update(loss.item(), batch_data.size(0))
    mask = batch_labels != TOKENS['<eos>']
    acc = masked_accuracy(pointer_argmaxs, batch_labels, mask).item()
    train_accuracy.update(acc, mask.int().sum().item())

    if bat % log_interval == 0:
      print(f'Epoch {epoch}: '
            f'Train [{bat * len(batch_data):9d}/{len(train_dataset):9d} '
            f'Loss: {train_loss.avg:.6f}\tAccuracy: {train_accuracy.
avg:3.4%}')
```

Train the model.

Iterate over batches of training data.

Set the model's parameters' gradients to zero to avoid accumulation from previous batches.

Calculate the loss.

A safeguard check to ensure that the loss value during the training process is not a NaN.

Perform a backward pass and optimization step.

Update training loss and accuracy.

Print the training progress.

As a continuation, the trained model (`model`) is evaluated on a validation dataset (`val_dataset`) to calculate the validation loss, accuracy, and overlap between the convex hull of the input data and the predicted pointer sequences. We start by setting the model to evaluation mode, where the model's parameters are frozen and the batch normalization or dropout layers behave differently than during training. The code then iterates through the validation dataset using the `val_loader`, which provides batches of data (`batch_data`), ground truth labels (`batch_labels`), and the lengths of each sequence (`batch_lengths`):

```
model.eval()
 hull_overlaps = []
 for bat, (batch_data, batch_labels, batch_lengths)
 ➥ in enumerate(val_loader):
  batch_data = batch_data.to(device)
  batch_labels = batch_labels.to(device)
  batch_lengths = batch_lengths.to(device)

  log_pointer_scores, pointer_argmaxs = model(batch_data, batch_lengths,
➥    batch_labels=batch_labels)
  loss = criterion(log_pointer_scores.view(-1, log_pointer_scores.
➥ shape[-1]),batch_labels.reshape(-1))

  assert not np.isnan(loss.item()), 'Model diverged with loss = NaN'

  val_loss.update(loss.item(), batch_data.size(0))
  mask = batch_labels != TOKENS['<eos>']
  acc = masked_accuracy(pointer_argmaxs, batch_labels, mask).item()
  val_accuracy.update(acc, mask.int().sum().item())

for data, length, ptr in zip(batch_data.cpu(), batch_lengths.cpu(),
➥     pointer_argmaxs.cpu()):
   hull_overlaps.append(calculate_hull_overlap(data, length, ptr))

 print(f'Epoch {epoch}: Val\tLoss: {val_loss.avg:.6f} '
    f'\tAccuracy: {val_accuracy.avg:3.4%} '
    f'\tOverlap: {np.mean(hull_overlaps):3.4%}')
 train_loss.reset()
 train_accuracy.reset()
 val_loss.reset()
 val_accuracy.reset()
```

Initialize an empty list to store the overlap values between the convex hull of the input data and the predicted pointer sequences.

Set the model to evaluation mode.

Iterate through the validation dataset.

Produce pointer scores and argmax predictions.

Calculate the validation loss.

Ignore the loss contribution from positions where the <eos> token is present in batch_labels.

Update the validation loss.

Calculate the masked accuracy.

Update the validation accuracy.

Print the epoch-wise validation loss, accuracy, and mean overlap.

Reset the metrics.

Calculate the overlap between the convex hull of the input data and the predicted pointer sequences.

Iterate through each batch's data, lengths, and pointer argmax predictions.

You can display the results of training and validation losses and accuracies using the `Disp_results` helper function:

```
Disp_results(train_loss, train_accuracy, val_loss, val_accuracy, n_epochs)
```

The preceding line of code will generate output like the following:

```
Best Scores:
train_loss: 0.0897 (ep: 9)
train_accuracy 96.61% (ep: 9)
val_loss: 0.0937 (ep: 7)
val_accuracy: 96.54% (ep: 7)
```

After model training and validation, we can test the model. The following test function will evaluate a trained model (`model`) on a test dataset. The function evaluates the model's accuracy and overlap with the convex hull for different test sample sizes. This test function takes as inputs the model, the number of test samples, and the number of points per sample. The code performs the test for different numbers of points per sample (`i`) by iterating from 5 to 45 in steps of 5. The `AverageMeter` class is used to keep track of average loss and accuracy during testing:

Set the number of test samples to be generated for each test.

```
n_rows_test = 1000
```

Test function

```
def test(model, n_rows_test, n_per_row):
  test_dataset = Scatter2DDataset(n_rows_test, n_per_row, n_per_row)
  test_loader = DataLoader(test_dataset, batch_size=batch_size,
   ➥ num_workers=n_workers)
```

Generate the test dataset.

```
  test_accuracy = AverageMeter()
  hull_overlaps = []
  model.eval()
```

Iterate through the batches of test data.

```
  for _, (batch_data, batch_labels, batch_lengths) in enumerate(test_loader):
    batch_data = batch_data.to(device)
    batch_labels = batch_labels.to(device)
    batch_lengths = batch_lengths.to(device)

    _, pointer_argmaxs = model(batch_data, batch_lengths)
```

Track the loss and accuracy.

```
    val_loss.update(loss.item(), batch_data.size(0))
    mask = batch_labels != TOKENS['<eos>']
    acc = masked_accuracy(pointer_argmaxs, batch_labels, mask).item()
    test_accuracy.update(acc, mask.int().sum().item())
```

Update the overlap between the convex hull and predicted pointer sequences.

```
    for data, length, ptr in zip(batch_data.cpu(), batch_lengths.cpu(),
       ➥ pointer_argmaxs.cpu()):
      hull_overlaps.append(calculate_hull_overlap(data, length, ptr))
```

Print the accuracy and overlap.

```
  print(f'# Test Samples: {n_per_row:3d}\t '
        f'\tAccuracy: {test_accuracy.avg:3.1%} '
        f'\tOverlap: {np.mean(hull_overlaps):3.1%}')
```

Iterate and print results for different sample sizes.

```
for i in range(5,50,5):
  test(model, n_rows_test, i)
```

This code will produce output like the following:

```
# Test Samples:   5     Accuracy: 54.8%     Overlap: 43.7%
# Test Samples:  10     Accuracy: 72.1%     Overlap: 79.1%
# Test Samples:  15     Accuracy: 79.0%     Overlap: 90.1%
# Test Samples:  20     Accuracy: 84.8%     Overlap: 92.7%
# Test Samples:  25     Accuracy: 80.6%     Overlap: 92.3%
# Test Samples:  30     Accuracy: 80.3%     Overlap: 91.6%
# Test Samples:  35     Accuracy: 77.8%     Overlap: 91.9%
# Test Samples:  40     Accuracy: 75.8%     Overlap: 92.1%
# Test Samples:  45     Accuracy: 72.4%     Overlap: 90.4%
```

Let's now test the trained model and see how well this model generalizes to new unseen data. We'll use a dataset with 50 points to test the trained and validated model and calculate the convex hull overlap between the predicted hull and the ground truth hull obtained by SciPy. We pass the batch of input data and its lengths through the model to obtain the predicted scores (`log_pointer_scores`) and the argmax indices (`pointer_argmaxs`) of the pointer network. The ground truth is the convex hull obtained using the `ConvexHull` function from `scipy.spatial`:

```
idx = 0
n_per_row = 50

test_dataset = Scatter2DDataset(n_rows_test, n_per_row, n_per_row)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
➥   num_workers=n_workers)
batch_data, batch_labels, batch_lengths = next(iter(test_loader))
print(batch_data.shape,batch_lengths.shape)
log_pointer_scores, pointer_argmaxs = model(batch_data.to(device),
➥   batch_lengths.to(device))
pred_hull_idxs = pointer_argmaxs[idx].cpu()
pred_hull_idxs = pred_hull_idxs[pred_hull_idxs > 1] - 2
points = batch_data[idx, 2:batch_lengths[idx], :2]
points1 = batch_data[idx, 1:batch_lengths[idx], :2]
print(points.shape,)
true_hull_idxs = ConvexHull(points).vertices.tolist()
true_hull_idxs = cyclic_permute(true_hull_idxs, np.argmin(true_hull_idxs))

overlap = calculate_hull_overlap(batch_data[idx].cpu(), batch_lengths[idx].
cpu(),
➥   pointer_argmaxs[idx].cpu())

print(f'Predicted: {pred_hull_idxs.tolist()}')
print(f'True:      {true_hull_idxs}')
print(f'Hull overlap: {overlap:3.2%}')
```

Extract the predicted argmax indices for the selected sample from the batch.

Obtain the predicted scores and the argmax indices of the pointer network.

Set the number of points in each sample.

Load the first batch of data from the test dataset.

Create a test dataset.

Extract and print the 2D points for the selected sample from the batch.

Filter out the special tokens (e.g., <eos>) and adjust the indices for indexing the points correctly.

Ground truth convex hull

Calculate the convex hull overlap.

Print the list of predicted convex hull indices, convex hull indices, and overlap percentage.

Running the code will produce output like the following. You can run the preceding code snippets multiple times to get a high percentage of hull overlap:

```
torch.Size([16, 51, 3]) torch.Size([16])
torch.Size([49, 2])
Predicted: [0, 3, 5, 31, 45, 47, 48, 40, 10]
True:      [0, 3, 5, 31, 45, 47, 48, 40, 10]
Hull overlap: 100.00%
```

The following code snippet can be used to visualize the convex hull generated by the pointer network (`ConvexNet`) in comparison with the convex hull generated by `scipy.spatial` as a ground truth:

```
plt.rcParams['figure.figsize'] = (10, 6)
plt.subplot(1, 2, 1)
```

Set the default figure size, and create the first subplot.

```
true_hull_idxs = ConvexHull(points).vertices.tolist()
display_points_with_hull(points, true_hull_idxs)
_ = plt.title('SciPy Convex Hull')

plt.subplot(1, 2, 2)
display_points_with_hull(points, pred_hull_idxs)
_ = plt.title('ConvexNet Convex Hull')
```

Display the points and their convex hull in the first subplot.

Create a second subplot.

Display the points and the convex hull generated by ConvexNet.

Compute the convex hull of a set of points (points) using the ConvexHull function from scipy.spatial.

Figure 11.20 shows the convex hulls generated by SciPy and ConvexNet. These convex hulls are identical in some instances (i.e., hull overlap = 100.00%), yet achieving this consistency requires proper training and careful tuning of the ConvexNet parameters.

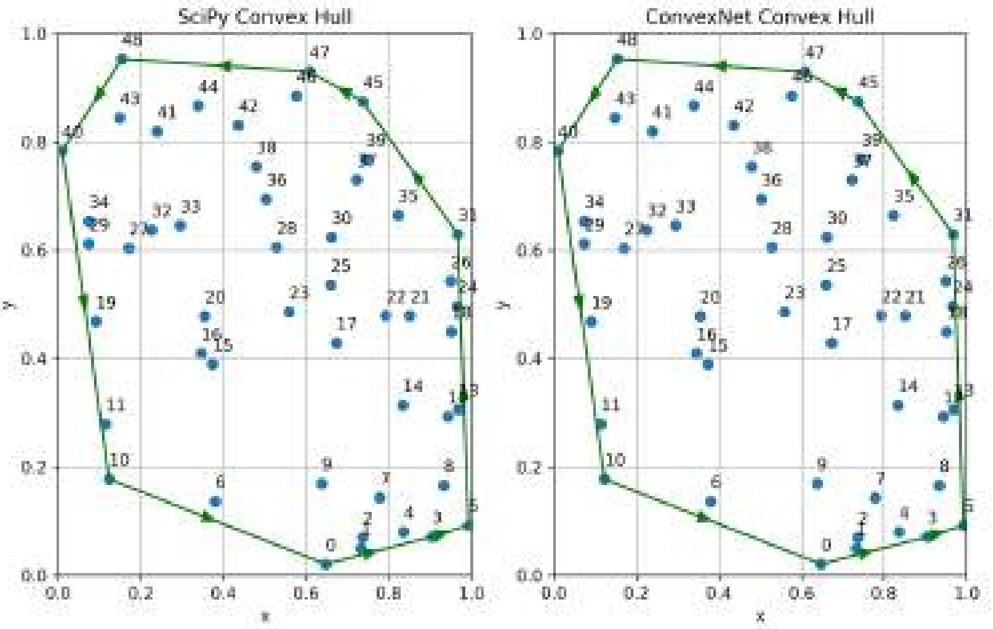


Figure 11.20 Convex hulls generated by SciPy and Ptr-Net for 50 points

This chapter has offered a fundamental foundation in ML and discussed the applications of supervised and unsupervised ML in handling optimization problems. The next chapter will focus on reinforcement learning and will delve deeply into its practical applications in tackling optimization problems.

Summary

- Machine learning (ML), a branch of artificial intelligence (AI), grants an artificial system or process the capacity to learn from experiences and observations, rather than through explicit programming.
- Deep learning (DL) is a subset of ML that is focused on the detection of inherent features within data by employing deep neural networks. This allows artificial systems to form intricate concepts from simpler ones.
- Geometric deep learning (GDL) extends (structured) deep neural models to handle non-Euclidean data with underlying geometric structures, such as graphs, point clouds, and manifolds.
- Graph machine learning (GML) is a subfield of ML that focuses on developing algorithms and models capable of learning from graph-structured data.
- Graph embedding represents the process of creating a conversion from the discrete, high-dimensional graph domain to a lower-dimensional continuous domain.
- The attention mechanism allows a model to selectively focus on certain portions of the input data while it is in the process of generating the output sequence.
- The pointer network (Ptr-Net) is a variation of the sequence-to-sequence model with attention designed to deal with variable-sized input data sequences.
- A self-organizing map (SOM), also known as a Kohonen map, is a type of artificial neural network (ANN) used for unsupervised learning. SOMs differ from other types of ANNs, as they apply competitive learning rather than error-correction learning (such as backpropagation with gradient descent).
- Neural combinatorial optimization refers to the application of ML to solve combinatorial optimization problems.
- Harnessing ML for combinatorial optimization can be achieved through three main methods: end-to-end learning where the model directly formulates solutions, using ML to configure and improve optimization algorithms, and integrating ML with optimization algorithms where the model continuously guides the optimization algorithm based on its current state.