Chapter 8 – Advanced Inference Techniques

**Target: 25 pages**

The refinement of inference methods is a cornerstone for deploying NLP systems that perform effectively in complex, real-world scenarios. Chapter 8 explores a range of advanced inference techniques designed to enhance pipeline functionality, elevate output precision, and achieve scalability using the Hugging Face Diffusers library. The chapter integrates theoretical perspectives with practical implementations, empowering readers to implement innovative approaches to achieve high-performance NLP solutions.

In this chapter, we will cover the following main topics:

1. Introduction to Inference Techniques
2. Pipeline Functionality Enhancements
3. Improving Inference Quality
4. Case Studies: Practical Applications of Advanced Inference Techniques

Learning Objectives

By the end of this chapter, readers will be able to:

1. **Implement Callbacks and Extensions:** Understand and apply callback functions and custom extensions to enhance AI pipelines, fostering dynamic and responsive model behavior.
2. **Deploy Distributed Inference Systems:** Get skills to implement distributed inference systems that efficiently process large-scale data across multiple computing resources.
3. **Refine Inference with Prompt Engineering:** Master prompt engineering to refine input queries and improve post-processing strategies, ensuring improved model outputs.
4. **Leverage Ensemble Methods:** Learn to combine predictions from multiple models using ensemble methods, enhancing prediction robustness and accuracy.
5. **Enhance Efficiency in Inference:** Develop strategies to improve inference speed and minimize computational overhead, ensuring scalable deployment of AI systems.

Enhancing Pipeline Functionality: Callbacks and Extensions

Enhancing pipeline functionality through callbacks and custom extensions sets up a foundation for adaptable and dynamic NLP systems, enabling real-time adjustments to meet specific use-case requirements. These capabilities are essential for achieving responsive interactions and streamlining complex processes, directly affecting the performance and scalability of advanced applications. These tools provide mechanisms for dynamic interaction within the inference process, enabling interventions that streamline outputs and support advanced customization.

Key Concepts

Adapting and extending pipeline functionality is a cornerstone of modern NLP systems. The integration of callbacks and extensions allows for a highly customizable framework that reacts dynamically to evolving requirements. These concepts form the foundation for developing intelligent, flexible models capable of addressing diverse challenges in real-world applications.

* **Callbacks:** Functions executed at specific stages of model training or inference. Callbacks help tasks like dynamic logging, halting processes based on conditions, or making real-time adjustments to parameters [1].
* **Extensions:** Tailored modifications to training loops or inference pipelines, expanding standard functionalities with novel features such as domain-specific preprocessing or customized loss functions [2].

Practical Applications

1. **Early Stopping in Training:** Callbacks can prevent overfitting by halting training when validation performance stagnates. For example, in a neural network model, an early stopping callback checks the validation loss and stops training if no improvement is seen over a specified number of epochs, ensuring the model does not overfit to training data.
2. **Dynamic Learning Rate Adjustment:** Extensions enable real-time learning rate modifications to adapt to model convergence patterns. Dynamic learning rate adjustments can ensure the model explores the best solutions during early training phases and fine-tunes its performance in later stages, enhancing convergence efficiency.

Example: Early Stopping Callback

Early stopping is a critical technique in training machine learning models, particularly neural networks, to prevent overfitting and conserve computational resources. By monitoring model performance during training, early stopping halts the training process when improvements in a specified metric cease, ensuring that the model generalizes well to unseen data. The code below proves how to implement this approach using the Keras library, highlighting its practical benefits in achieving efficient and robust training.

`python

from tensorflow.keras.callbacks import EarlyStopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, verbose=1)

model.fit(x\_train, y\_train, validation\_data=(x\_val, y\_val), callbacks=[early\_stopping])

`

This example uses the Keras EarlyStopping callback, a built-in mechanism for checking a specified performance metric during training and halting the training process when the metric stops improving. The goal is to avoid overfitting, which occurs when a model becomes too specialized in the training data and performs poorly on unseen data.

The EarlyStopping class is initialized with three key parameters:

1. monitor='val\_loss': This parameter specifies the metric to check. In this case, the validation loss (val\_loss) is chosen. Validation loss is how well the model performs on a separate validation dataset, which is not used for training. Monitoring this metric helps ensure that the model generalizes well to new data.
2. patience=3: This parameter sets the number of epochs to wait after the last observed improvement in the monitored metric before halting training. A patience value of 3 means that if the validation loss does not improve for three consecutive epochs, the training process will stop. This avoids premature stopping while ensuring that unnecessary epochs are not run.
3. verbose=1: This parameter controls the verbosity of the output. A value of 1 ensures that the user receives a message showing when early stopping is triggered, providing insight into the training process.

The model.fit function is used to train the model on the training data (x\_train and y\_train) with validation performed on a separate validation dataset (x\_val and y\_val). The callbacks argument accepts a list of callback functions, including the EarlyStopping callback defined earlier. During training, Keras evaluates the validation loss at the end of each epoch. If no improvement is seen in validation loss for three consecutive epochs, the training process halts.

By incorporating this callback, practitioners can ensure that computational resources are used efficiently, avoiding unnecessary training epochs once the model has reached its best performance. Additionally, early stopping minimizes the risk of overfitting, enabling the development of models that perform well on both training and validation datasets. This technique is particularly useful in scenarios where training is computationally expensive or where achieving a balance between training time and model performance is critical.

About Keras:

Keras is an open-source deep learning library that provides a high-level interface for building, training, and evaluating neural networks [3]. Originally developed as an independent library, Keras is now fully integrated with TensorFlow, one of the most widely used frameworks for machine learning and deep learning.

Key Features of Keras

* **User-Friendly and Modular:**  
  Keras simplifies the process of designing deep learning models by providing a clear, concise API [4]. Its modular design allows users to easily define models, layers, and other components, making it an excellent choice for both beginners and experienced practitioners.
* **Support for Multiple Backends:**  
  Initially, Keras supported multiple computational backends, including TensorFlow, Theano, and Microsoft Cognitive Toolkit (CNTK). In its current form, Keras is tightly integrated with TensorFlow, leveraging TensorFlow's backend for high-performance computation.
* **Prebuilt Layers and Models:**  
  Keras includes a wide range of prebuilt layers (e.g., Dense, Convolutional, Recurrent) and models that can be quickly customized for specific tasks. This speeds up development and reduces the need for extensive coding.
* **Flexibility and Scalability:**  
  Keras supports both simple sequential models and more complex architectures with multiple branches, shared layers, and custom components. It is scalable, allowing users to train models on CPUs, GPUs, or distributed systems [5].
* **Extensive Ecosystem:**  
  Integrated with TensorFlow, Keras receives help from a robust ecosystem, including tools for visualization (TensorBoard), deployment (TensorFlow Lite, TensorFlow Serving), and model optimization.
* **Community and Documentation:**  
  With a large community and comprehensive documentation, Keras provides extensive resources, tutorials, and examples for learning and troubleshooting [6].

Role of Keras in EarlyStopping

In the context of early stopping, Keras simplifies the implementation by offering a ready-to-use EarlyStopping callback. This feature exemplifies Keras's focus on usability and efficiency, allowing practitioners to integrate advanced techniques into their workflows with minimal effort.

By using Keras, developers can focus more on designing and refining their models rather than dealing with the complexities of low-level computation. This makes it a preferred tool for rapid prototyping and deployment of deep learning applications.

Techniques for Distributed Inference

Large-scale NLP tasks need distributed inference methods to manage computationally intensive workloads. Distributed inference techniques divide tasks across multiple systems, ensuring efficient data processing and best use of available resources.

Large-scale NLP tasks often involve processing enormous datasets or deploying complex models that surpass the capabilities of single devices or conventional setups. Distributed inference addresses these challenges by dividing computational workloads across multiple systems, enabling efficient processing and best resource use. Below are the core strategies that underpin distributed inference:

Core Strategies

To efficiently manage the computational workload associated with distributed inference, two primary strategies are commonly employed: data parallelism and model parallelism. These approaches optimize the use of available hardware and ensure that large-scale tasks are processed in a timely and effective manner.

Data Parallelism:

Splits data into subsets distributed across processors. Each processor applies the same model to its subset, aggregating the results for final predictions [7].

Data parallelism focuses on dividing the dataset into smaller subsets and distributing these subsets across multiple processors. Each processor runs the same model independently on its allocated data portion, and the results are aggregated to produce the final predictions. This approach is particularly effective for large datasets where processing can be parallelized.

For example, in a distributed text classification task, the dataset is split into chunks, and each chunk is sent to a separate GPU or CPU. After processing, the individual results are combined to form a unified output. Frameworks like TensorFlow and PyTorch facilitate data parallelism by providing built-in tools for synchronizing models and sharing gradients across devices.

The key advantage of data parallelism is its scalability. As datasets grow larger, additional processors can be added to maintain high performance. However, the synchronization overhead of sharing model gradients between devices can become a bottleneck, particularly in scenarios with a high number of distributed nodes.

Code Snippet: Data Parallelism with TensorFlow

`python

import tensorflow as tf

strategy = tf.distribute.MirroredStrategy()

with strategy.scope():

model = tf.keras.Sequential([

tf.keras.layers.Embedding(input\_dim=10000, output\_dim=256),

tf.keras.layers.GlobalAveragePooling1D(),

tf.keras.layers.Dense(1, activation='sigmoid')

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5, batch\_size=512)

`

This example demonstrates TensorFlow’s MirroredStrategy, where data parallelism is achieved by splitting the dataset across multiple GPUs. Each GPU trains the model on its subset of data, and the gradients are synchronized after each batch.

Model Parallelism:

Divides large models across devices, enabling the use of limited memory resources while processing complex tasks efficiently.

Model parallelism, on the other hand, divides a large model into smaller components and distributes these components across multiple devices. Each device is responsible for processing a specific part of the model, and intermediate results are passed between devices. This approach is ideal for scenarios where the model’s memory requirements exceed the capacity of a single device, such as with transformer-based architectures like GPT or BERT.

For instance, in a transformer model, the encoder layers could be assigned to one GPU, while the decoder layers are handled by another. Intermediate computations are transferred between devices during the forward and backward passes. Model parallelism enables the use of high-capacity models on limited hardware but requires careful partitioning to minimize communication overhead between devices.

Example: Model Parallelism

`python

import torch

from torch import nn c

lass Model(nn.Module):

def \_\_init\_\_(self):

super(Model, self).\_\_init\_\_()

self.layer1 = nn.Linear(1000, 500).to('cuda:0') # Assign to GPU 0

self.layer2 = nn.Linear(500, 10).to('cuda:1') # Assign to GPU 1

def forward(self, x):

x = self.layer1(x.to('cuda:0'))

x = self.layer2(x.to('cuda:1'))

return x

model = Model() input\_data = torch.randn(64, 1000) output = model(input\_data)

`

In this example, the model's layers are split across two GPUs, and the data is transferred between devices during the forward pass. This ensures that no single GPU is overloaded with memory-intensive tasks, enabling the training of large models on hardware with limited capacity.

Challenges and Trade-Offs

Both data parallelism and model parallelism come with challenges. Data parallelism requires efficient gradient synchronization, while model parallelism demands optimized communication pathways to reduce latency. Combining these methods, known as hybrid parallelism, can further enhance performance by leveraging the strengths of both strategies.

By implementing data and model parallelism effectively, developers can overcome the limitations of single-device setups, paving the way for scalable and efficient NLP solutions capable of handling the demands of modern applications.

Example: Distributed Text Classification

This example illustrates how distributed inference can be implemented to manage large-scale text classification tasks, improving computational efficiency across multiple devices.

`python

import tensorflow as tf

strategy = tf.distribute.MirroredStrategy()

with strategy.scope():

model = tf.keras.Sequential([

tf.keras.layers.Embedding(input\_dim=10000, output\_dim=256),

tf.keras.layers.GlobalAveragePooling1D(),

tf.keras.layers.Dense(1, activation='sigmoid')

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5, batch\_size=512)

`

This code shows data parallelism using TensorFlow’s MirroredStrategy, which enables distributed training across GPUs.

Improving Inference Quality: Prompt Engineering and Post-Processing Techniques

Inference output quality significantly depends on well-crafted input prompts and robust post-processing methods. Thoughtful prompt engineering guides model behavior, while post-processing refines outputs for specific applications.

Techniques

Improving inference speed is essential for deploying NLP models in real-time applications. By reducing latency and computational overhead, these techniques ensure models can deliver the right results quickly, meeting the demands of interactive and large-scale systems.

* **Prompt Engineering:** Carefully designed prompts improve model performance by providing context and guiding response generation [8].
* **Post-Processing:** Techniques like text normalization, confidence filtering, and domain-specific adjustments enhance output usability.

Example: Prompt Engineering for Chatbots

This example shows the use of prompt engineering to tailor a model’s responses to user queries by structuring input prompts.

`python

def generate\_prompt(user\_input):

return f"The user says: '{user\_input}'. Respond with a helpful reply."

response = model.generate(generate\_prompt("I need help resetting my password."))

`

This example displays prompt engineering by structuring user input to guide model responses effectively.

Using Ensemble Methods for Better Inference Results

Ensemble methods combine multiple models to improve prediction accuracy and reliability. By using diverse algorithms, ensemble approaches reduce errors and enhance robustness.

Key Approaches

Refining inference often requires advanced ensemble techniques to address the complexities of diverse NLP tasks. These methods, by combining multiple models or approaches, improve accuracy, reliability, and adaptability, making them indispensable in scenarios where robustness and precision are critical.

Bagging:

Combines predictions through majority voting or averaging, improving generalization [9].

Bagging, or Bootstrap Aggregating, is a powerful ensemble learning technique that reduces variance and prevents overfitting by combining the outputs of multiple models. The approach involves training different base models, often weak learners like decision trees, on different subsets of the training dataset. These subsets are created using bootstrapping, a method that generates random samples with replacement. The predictions of these models are then aggregated, either by majority voting for classification tasks or by averaging for regression problems.

**Why it Works:** Bagging uses diversity among models. Because each model is trained on a unique subset of data, it captures distinct aspects of the dataset, and aggregating their outputs leads to more robust and generalized predictions.

**Example:** Random Forests are a prime application of bagging, where multiple decision trees are combined to make a collective prediction. In NLP, bagging can be applied to tasks like sentiment analysis by training multiple models on diverse subsets of user-generated content and combining their predictions to account for variations in text.

**Real-World Scenario:** In fraud detection systems, bagging helps reduce the impact of noisy or outlier data by allowing multiple models to collectively find suspicious transactions, thereby increasing reliability.

Boosting:

Sequentially adjusts model weights to address difficult predictions.

Boosting takes a sequential approach to ensemble learning, where models are trained iteratively, and each new model focuses on correcting the errors of its predecessor. The ensemble assigns higher weights to misclassified instances during training, ensuring that the next models pay more attention to difficult cases.

**How It Works:** Boosting combines the predictions of all models in the ensemble, giving higher importance to more correct models. Techniques like AdaBoost and Gradient Boosting build models sequentially, improving for residual errors in each iteration.

**Advanced Techniques:** Gradient Boosting, a popular variant, constructs an additive model in a stage-wise manner, where new trees predict the residuals of earlier trees. In NLP, this can improve text classification by learning nuanced patterns in complex datasets.

**Example:** XGBoost, a widely used boosting framework, excels in tasks requiring high predictive accuracy. In NLP, boosting can refine named entity recognition (NER) by focusing on edge cases like rare or ambiguous entity mentions.

**Real-World Scenario:** In search engines, boosting enhances ranking algorithms by iteratively refining results, ensuring that the most relevant pages appear at the top even when search queries are ambiguous.

Stacking:

Uses a meta-model to aggregate outputs from diverse models, achieving superior performance.

Stacking, or Stacked Generalization, is an advanced ensemble technique that combines the predictions of multiple models through a meta-model, which learns how to best aggregate the individual model outputs. Unlike bagging and boosting, stacking uses diverse base models, each contributing unique strengths to the ensemble.

**How It Works:** The base models make predictions, which are then used as inputs for the meta-model. This secondary model, often a logistic regression or neural network, learns how to weigh and combine these predictions to maximize overall accuracy.

**Example:** In NLP, stacking can improve machine translation systems by combining outputs from models trained on different linguistic features, such as syntax, semantics, and contextual embeddings. The meta-model integrates these complementary predictions for superior translations.

**Implementation:** A stacking ensemble might include BERT for semantic understanding, a CNN for extracting local patterns, and an LSTM for capturing sequential dependencies. The meta-model evaluates the outputs of these diverse architectures to make a final prediction.

**Real-World Scenario:** In recommendation systems, stacking aggregates predictions from collaborative filtering, content-based filtering, and deep learning models to provide personalized suggestions that adapt to user preferences and behaviors.

Example: Sentiment Analysis Ensemble

This example shows how ensemble methods can improve the accuracy of sentiment analysis by combining predictions from multiple models.

`python

from sklearn.ensemble import VotingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

model1 = LogisticRegression()

model2 = SVC(probability=True)

model3 = DecisionTreeClassifier()

ensemble\_model = VotingClassifier(

estimators=[('lr', model1), ('svm', model2), ('dt', model3)],

voting='soft'

)

ensemble\_model.fit(X\_train, y\_train)

accuracy = ensemble\_model.score(X\_test, y\_test)

print(f"Ensemble accuracy: {accuracy:.2f}")

`

This code shows an ensemble approach to sentiment analysis using the VotingClassifier from the scikit-learn library. The ensemble model combines the predictions of three base models: logistic regression, a support vector classifier (SVC), and a decision tree classifier. Each of these base models contributes its unique strengths to improve the overall prediction accuracy of the sentiment analysis task.

The code begins by importing the necessary libraries and defining the three base models. The logistic regression model (model1) is particularly effective for binary classification tasks due to its simplicity and ability to manage linearly separable data. The support vector classifier (model2) is configured to output probability estimates, making it suitable for integration in the ensemble. The decision tree classifier (model3) is included for its interpretability and ability to capture non-linear relationships in the data.

Next, the VotingClassifier is instantiated with the three models as its components. The voting='soft' parameter specifies that the ensemble will use probability-weighted voting. In this method, each model's confidence in its predictions is considered, allowing the ensemble to use the models' strengths in areas where they excel. For instance, if logistic regression is more dependable in certain patterns, its higher confidence in those predictions will carry more weight in the final decision.

The fit method is then used to train the ensemble model on the training dataset (X\_train, y\_train). During this process, each base model is trained independently on the same dataset, and their predictions are later aggregated by the ensemble. The ensemble's ability to integrate diverse decision boundaries and use complementary strengths of its base models often results in better generalization and robustness than any single model.

Finally, the score method evaluates the ensemble's accuracy on the test dataset (X\_test, y\_test), providing a quantitative measure of its performance. The code concludes by printing the accuracy score, offering a sign of how well the ensemble model predicts sentiment labels compared to the ground truth.

This ensemble approach shows the practicality of combining multiple classifiers to improve predictive performance in NLP tasks like sentiment analysis. By using the strengths of diverse models, the ensemble mitigates individual model weaknesses and enhances overall robustness and reliability.

Improving Inference Speed and Efficiency

Inference speed and computational efficiency are critical for real-time NLP applications. Optimization strategies reduce latency while ensuring accuracy, enabling seamless deployment in resource-constrained environments.

Techniques

Improving inference speed and efficiency is a crucial step for deploying NLP models in real-world applications. These optimizations not only reduce latency but also ensure that models work effectively in resource-constrained environments, such as edge devices or systems with limited computational power. Below are key techniques that streamline inference processes and enable scalable, high-performance applications.

Model Simplification:

Pruning or distillation reduces model complexity without compromising performance.

Simplification techniques like pruning and distillation aim to reduce the size and computational requirements of models while keeping performance. Pruning eliminates less significant parameters or connections in the model, resulting in a more compact structure. For example, in a neural network, weights with values close to zero can be removed without significantly affecting the overall accuracy. This reduction in size decreases memory usage and speeds up computation. Distillation, on the other hand, involves training a smaller "student" model to replicate the behavior of a larger "teacher" model. By transferring knowledge from the teacher to the student, distillation achieves comparable performance with fewer parameters. These techniques are particularly valuable for deploying models on devices with limited memory and processing capabilities, such as smartphones or IoT devices.

Quantization:

Converts model weights to lower precision, accelerating computation and reducing memory usage [10].

Quantization converts model weights and activations from higher-precision data types (such as 32-bit floating-point) to lower-precision formats (such as 8-bit integers). This transformation drastically reduces the memory footprint of the model and accelerates computation, as lower-precision arithmetic operations require fewer resources. For instance, TensorFlow Lite provides tools for post-training quantization, enabling developers to improve pre-trained models for deployment. Quantization is especially helpful in environments like embedded systems or mobile applications, where both storage and computation power are limited. Despite the reduction in precision, carefully applied quantization supports near-original accuracy, making it a preferred choice for improving large-scale NLP models for real-time applications.

Example: Model Quantization

Quantization is a powerful optimization technique that reduces the precision of model weights and activations, enabling faster and more efficient inference. This approach is particularly useful when deploying models on resource-constrained devices, such as mobile phones, IoT devices, or embedded systems. The example below proves the process of converting a high-precision model to a quantized format using TensorFlow Lite, making it suitable for real-time applications with limited computational power.

`pyton

import tensorflow as tf

converter = tf.lite.TFLiteConverter.from\_keras\_model(model)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

quantized\_model = converter.convert()

with open('quantized\_model.tflite', 'wb') as f:

f.write(quantized\_model)

This example demonstrates TensorFlow Lite quantization, enabling faster inference on edge devices.

`

The provided code snippet illustrates how to perform model quantization using TensorFlow Lite. Quantization reduces the memory footprint and computational requirements of a machine learning model by converting its parameters from a high-precision data type (e.g., 32-bit floating-point) to a lower-precision one (e.g., 8-bit integers). This optimization improves inference speed while keeping acceptable accuracy levels.

The process begins by importing the TensorFlow library. The TFLiteConverter class is used to start the conversion process from a standard TensorFlow model into a TensorFlow Lite model, which is specifically designed for lightweight, on-device inference.

The from\_keras\_model function takes the trained TensorFlow model as an input. This function creates a converter object that manages the transformation of the model to a format suitable for TensorFlow Lite. The optimizations parameter is then set to [tf.lite.Optimize.DEFAULT], which applies default optimization strategies, including quantization. This step enables the model to be converted into a quantized format, reducing its precision while ensuring efficiency and preserving accuracy as much as possible.

The convert method executes the conversion process, creating a quantized TensorFlow Lite model. The resulting model is stored in memory and ready for deployment. The final part of the code saves the quantized model to a .tflite file using standard file I/O operations. The with open block open a file in write-binary mode and writes the quantized model to it. This file can then be loaded on compatible edge devices for inference.

This example proves the simplicity and efficiency of TensorFlow Lite quantization, highlighting how developers can adapt complex models for deployment in environments with constrained resources. By reducing precision without compromising performance, quantization helps faster computations and reduced energy consumption, making it an essential technique for modern AI applications.

Conclusion: Advanced Inference Techniques

Chapter 8 examined advanced inference strategies integral to NLP applications. Readers learned how to:

* Enhance pipeline functionality using callbacks and extensions.
* Implement distributed inference for scalable tasks.
* Refine outputs through prompt engineering and post-processing.
* Leverage ensemble methods for robust predictions.
* Improve inference speed and efficiency through quantization and hardware acceleration.

In the next Chapter: Emerging Trends and Technologies

The next chapter delves into emerging NLP trends, including integrating AI with advanced technologies, addressing ethical considerations, and exploring innovations in language understanding. This transition equips readers to engage with the next wave of advancements in NLP.

References

|  |  |
| --- | --- |
| [1] | R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction, MIT Press, 2018. |
| [2] | I. Goodfellow, Y. Bengio and A. Courville, Deep learning, MIT Press, 2016. |
| [3] | Keras Official Documentation, "Keras: The Python Deep Learning Library," [Online]. Available: https://keras.io. [Accessed 04 January 2025]. |
| [4] | TensorFlow Official Documentation on Keras, "Keras API Overview," [Online]. Available: https://www.tensorflow.org/guide/keras. [Accessed 04 January 2025]. |
| [5] | I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016. |
| [6] | F. Chollet, "Keras," *GitHub Repository,* 2015. |
| [7] | J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters.," *Communications of the ACM,* vol. 51, no. 1, p. 107–113, 2008. |
| [8] | C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 4, no. 1, pp. 1, 43, 2020. |
| [9] | L. Breiman, "Bagging Predictors," *Machine Learning,* vol. 24, no. 2, p. 123–140, 1996. |
| [10] | B. Jacob and e. al., Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference., MIT Pres, 2016. |