Implementando e Treinando Modelos de Machine Learning em Java: Conceitos Avançados e Códigos Práticos com Outputs Detalhados

Com o uso crescente de bibliotecas de Machine Learning (ML) em Java, este artigo apresenta um exemplo detalhado de como criar e treinar modelos em Java utilizando a **Deep Java Library (DJL)**.

Vamos também apresentar os **outputs** do treinamento em diferentes estágios para que o desenvolvedor entenda como cada etapa do processo impacta os resultados.

Versão de Java

Java 11 ou superior é recomendado, devido à sua robustez no gerenciamento de memória, maior eficiência em operações paralelas e total compatibilidade com bibliotecas modernas como DJL.

Bibliotecas Utilizadas

1. **DJL**: Biblioteca de código aberto que oferece integração com TensorFlow, PyTorch e Apache MXNet.
    - **Documentação**: Deep Java Library Documentation
2. **TensorFlow com DJL**: Usado como backend para deep learning, fornecendo operações eficientes em CPU e GPU.
    - **Documentação**: TensorFlow Documentation

Estrutura do Código

A seguir, é apresentado um exemplo de treinamento utilizando a DJL e TensorFlow para classificar imagens do dataset MNIST. Adicionamos **saídas intermediárias** com System.out.println() para observar o progresso do treinamento.

Dependências Maven

xml

```
<dependency>
    <groupId>ai.djl.tensorflow</groupId>
    <artifactId>tensorflow-engine</artifactId>
    <version>0.22.0</version>
</dependency>
<dependency>
    <groupId>ai.djl.tensorflow</groupId>
    <artifactId>tensorflow-model-zoo</artifactId>
    <version>0.22.0</version>
</dependency>
<dependency>
    <groupId>ai.djl.api</groupId>
    <artifactId>djl-api</artifactId>
    <version>0.22.0</version>
</dependency>
<dependency>
    <groupId>ai.djl.basicdataset</groupId>
    <artifactId>basic-dataset</artifactId>
    <version>0.22.0</version>
</dependency>
```

Código: Treinamento e Output de um Modelo MLP

O código a seguir foi modificado para incluir **outputs detalhados** que imprimem o estado atual do treinamento e as métricas de desempenho em cada época.

java

```
import ai.djl.Model;
import ai.djl.ModelException;
import ai.djl.basicdataset.cv.classification.Mnist;
import ai.djl.basicmodelzoo.basic.Mlp;
import ai.djl.metric.Metrics;
import ai.djl.ndarray.NDManager;
import ai.djl.ndarray.types.Shape;
import ai.djl.training.DefaultTrainingConfig;
import ai.djl.training.Trainer;
import ai.djl.training.listener.TrainingListener;
import ai.djl.training.loss.Loss;
import ai.djl.training.optimizer.Optimizer;
import ai.djl.training.dataset.Batch;
import ai.djl.training.util.ProgressBar;

import java.io.IOException;

public class NeuralNetworkExample {
    public static void main(String[] args) throws IOException, ModelException {
        // Gerenciador de NDArrays (tensores)
        try (NDManager manager = NDManager.newBaseManager()) {

            // Carregar o dataset MNIST
            Mnist dataset = Mnist.builder()
                    .optUsage(Mnist.Usage.TRAIN)
                    .setSampling(32, true)
```

```
                .build();
        dataset.prepare(new ProgressBar());

        // Definir o modelo: Multi-Layer Perceptron (MLP)
        try (Model model = Model.newInstance("mlp")) {
            model.setBlock(new Mlp(28 * 28, 10, new int[]{128, 64})); // MLP com duas camadas
ocultas

            // Configuração do treinamento
            DefaultTrainingConfig config = new
DefaultTrainingConfig(Loss.softmaxCrossEntropyLoss())
                    .optOptimizer(Optimizer.sgd().setLearningRate(0.01f).build())
                    .addTrainingListeners(TrainingListener.Defaults.logging());

            // Inicializar o trainer
            try (Trainer trainer = model.newTrainer(config)) {
                trainer.setMetrics(new Metrics());
                trainer.initialize(new Shape(32, 28 * 28)); // Batch size de 32, input de 28x28 pixels

                System.out.println("Iniciando o treinamento...");

                // Treinamento por 10 épocas
                for (int epoch = 0; epoch < 10; epoch++) {
                    System.out.println("Época " + (epoch + 1) + " de 10");

                    int batchIndex = 0;
                    for (Batch batch : trainer.iterateDataset(dataset)) {
                        trainer.trainBatch(batch); // Realiza a retropropagação
                        trainer.step(); // Atualiza os parâmetros
                        batch.close(); // Libera memória do batch

                        batchIndex++;
                        // Imprimir a cada 100 batches
                        if (batchIndex % 100 == 0) {
                            System.out.println("Batch " + batchIndex + " processado.");
                        }
                    }

                    // Após cada época, imprimir as métricas
                    System.out.println("Métricas após época " + (epoch + 1) + ": " +
trainer.getMetrics().toString());
                }

                // Avaliação do modelo após o treinamento
                Batch testBatch = trainer.iterateDataset(dataset).iterator().next();
                System.out.println("Avaliando o modelo...");
                System.out.println(trainer.evaluateBatch(testBatch));
            }
        }
    }
}
```

Output

1. **Inicialização**: A fase inicial imprime o início do treinamento.

Iniciando o treinamento..

2. **Treinamento por Época**: Para cada época, o número da época atual será impresso.

Época 1 de 10

3. **Processamento por Batch**: A cada 100 batches, o progresso será relatado:

Batch 100 processado.
Batch 200 processado.
...

4. **Métricas ao final de cada época**: As métricas acumuladas são impressas ao fim de cada época, como por exemplo:

yaml
Métricas após época 1: {TrainLoss: 0.56, TrainAccuracy: 87.32%}
Métricas após época 2: {TrainLoss: 0.43, TrainAccuracy: 90.14%}

Esses valores são fictícios e irão variar com o treinamento real.

5. **Avaliação Final**: Ao final do treinamento, o modelo será avaliado, e a precisão final ou outras métricas serão mostradas:

yaml
Avaliando o modelo...
{TestLoss: 0.32, TestAccuracy: 92.68%}

Explicação dos Outputs

1. **Métricas de Treinamento**: As métricas de perda (**TrainLoss**) e acurácia (**TrainAccuracy**) mostram a performance do modelo nos dados de treinamento ao final de cada época.
2. **Avaliando o Modelo**: O método evaluateBatch() retorna métricas de desempenho, como a perda e a acurácia nos dados de teste. Isso é importante para verificar se o modelo está superajustando aos dados de treinamento ou generalizando bem.

Conclusão

Este artigo apresentou um fluxo detalhado de como construir, treinar e avaliar um modelo de Machine Learning em Java utilizando a **Deep Java Library (DJL)** com backend TensorFlow.

Adicionamos outputs explicativos no código para permitir que os desenvolvedores monitorem e entendam cada passo do processo de treinamento e seus impactos no modelo final.

Referências

- Deep Java Library Documentation
- TensorFlow Documentation
- Apache Spark MLlib Documentation
- Weka Documentation

EducaCiência FastCode para a comunidade