Knowledge Distillation

Author: Kenneth Borup
Date created: 2020/09/01
Last modified: 2020/09/01
Description: Implementation of classical Knowledge Distillation.

Introduction to Knowledge Distillation

Knowledge Distillation is a procedure for model compression, in which a small (student) model is trained to match a large pre-trained (teacher) model. Knowledge is transferred from the teacher model to the student by minimizing a loss function, aimed at matching softened teacher logits as well as ground-truth labels.

The logits are softened by applying a "temperature" scaling function in the softmax, effectively smoothing out the probability distribution and revealing inter-class relationships learned by the teacher.

Reference:

- Hinton et al. (2015)

Setup

```
import os
import keras
from keras import layers
from keras import ops
import numpy as np
```

```
Start coding or generate with AI.
```

Construct `Distiller()` class

The custom `Distiller()` class, overrides the `Model` methods `compile`, `compute_loss`, and `call`. In order to use the distiller, we need:

- A trained teacher model
- A student model to train
- A student loss function on the difference between student predictions and ground-truth

- A distillation loss function, along with a `temperature`, on the difference between the soft student predictions and the soft teacher labels
- An `alpha` factor to weight the student and distillation loss
- An optimizer for the student and (optional) metrics to evaluate performance

In the `compute_loss` method, we perform a forward pass of both the teacher and student, calculate the loss with weighting of the `student_loss` and `distillation_loss` by `alpha` and `1 - alpha`, respectively. Note: only the student weights are updated.

```
class Distiller(keras.Model):
    def __init__(self, student, teacher):
        super().__init__()
        self.teacher = teacher
        self.student = student

    def compile(
        self,
        optimizer,
        metrics,
        student_loss_fn,
        distillation_loss_fn,
        alpha=0.1,
        temperature=3,
    ):
        """Configure the distiller.

        Args:
            optimizer: Keras optimizer for the student weights
            metrics: Keras metrics for evaluation
            student_loss_fn: Loss function of difference between student
                predictions and ground-truth
            distillation_loss_fn: Loss function of difference between soft
                student predictions and soft teacher predictions
            alpha: weight to student_loss_fn and 1-alpha to distillation_loss_fn
            temperature: Temperature for softening probability distributions.
                Larger temperature gives softer distributions.
        """
        super().compile(optimizer=optimizer, metrics=metrics)
        self.student_loss_fn = student_loss_fn
        self.distillation_loss_fn = distillation_loss_fn
        self.alpha = alpha
        self.temperature = temperature

    def compute_loss(
        self, x=None, y=None, y_pred=None, sample_weight=None, allow_empty=False
    ):
        teacher_pred = self.teacher(x, training=False)
        student_loss = self.student_loss_fn(y, y_pred)

        distillation_loss = self.distillation_loss_fn(
            ops.softmax(teacher_pred / self.temperature, axis=1),
            ops.softmax(y_pred / self.temperature, axis=1),
        ) * (self.temperature**2)

        loss = self.alpha * student_loss + (1 - self.alpha) * distillation_loss
        return loss

    def call(self, x):
        return self.student(x)
```

Create student and teacher models

Initialy, we create a teacher model and a smaller student model. Both models are convolutional neural networks and created using `Sequential()`, but could be any Keras model.

```
# Create the teacher
teacher = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(256, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.MaxPooling2D(pool_size=(2, 2), strides=(1, 1), padding="same"),
        layers.Conv2D(512, (3, 3), strides=(2, 2), padding="same"),
        layers.Flatten(),
        layers.Dense(10),
    ],
    name="teacher",
)

# Create the student
student = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(16, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.MaxPooling2D(pool_size=(2, 2), strides=(1, 1), padding="same"),
        layers.Conv2D(32, (3, 3), strides=(2, 2), padding="same"),
        layers.Flatten(),
        layers.Dense(10),
    ],
    name="student",
)

# Clone student for later comparison
student_scratch = keras.models.clone_model(student)
```

Prepare the dataset

The dataset used for training the teacher and distilling the teacher is MNIST, and the procedure would be equivalent for any other dataset, e.g. CIFAR-10, with a suitable choice of models. Both the student and teacher are trained on the training set and evaluated on the test set.

```
# Prepare the train and test dataset.
batch_size = 64
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Normalize data
x_train = x_train.astype("float32") / 255.0
x_train = np.reshape(x_train, (-1, 28, 28, 1))

x_test = x_test.astype("float32") / 255.0
x_test = np.reshape(x_test, (-1, 28, 28, 1))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mn
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Train the teacher

In knowledge distillation we assume that the teacher is trained and fixed. Thus, we start by training the teacher model on the training set in the usual way.

```
# Train teacher as usual
teacher.compile(
    optimizer=keras.optimizers.Adam(),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)

# Train and evaluate teacher on data.
teacher.fit(x_train, y_train, epochs=5)
teacher.evaluate(x_test, y_test)
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 588s 313ms/step - loss: 0.2375 - sparse_categorical_a
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 613s 308ms/step - loss: 0.0909 - sparse_categorical_a
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 634s 315ms/step - loss: 0.0740 - sparse_categorical_a
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 622s 315ms/step - loss: 0.0693 - sparse_categorical_a
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 610s 309ms/step - loss: 0.0621 - sparse_categorical_a
313/313 ━━━━━━━━━━━━━━━━━━━━ 25s 79ms/step - loss: 0.1070 - sparse_categorical_accur
[0.08782276511192322, 0.9771000146865845]
```

Distill teacher to student

We have already trained the teacher model, and we only need to initialize a `Distiller(student, teacher)` instance, `compile()` it with the desired losses, hyperparameters and optimizer, and

distill the teacher to the student.

```
# Initialize and compile distiller
distiller = Distiller(student=student, teacher=teacher)
distiller.compile(
    optimizer=keras.optimizers.Adam(),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
    student_loss_fn=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    distillation_loss_fn=keras.losses.KLDivergence(),
    alpha=0.1,
    temperature=10,
)

# Distill teacher to student
distiller.fit(x_train, y_train, epochs=3)

# Evaluate student on test dataset
distiller.evaluate(x_test, y_test)
```

```
Epoch 1/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 590s 313ms/step - loss: 1.5990 - sparse_categorical_a
Epoch 2/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 632s 319ms/step - loss: 0.0320 - sparse_categorical_a
Epoch 3/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 608s 311ms/step - loss: 0.0211 - sparse_categorical_a
313/313 ━━━━━━━━━━━━━━━━━━━━ 26s 81ms/step - loss: 0.0165 - sparse_categorical_accur
[0.014691012911498547, 0.9722999930381775]
```

Train student from scratch for comparison

We can also train an equivalent student model from scratch without the teacher, in order to