Ungraded lab: Knowledge Distillation

Welcome, during this ungraded lab you are going to perform a model compression technique known as **knowledge distillation** in which a `student` model "learns" from a more complex model known as the `teacher`. In particular you will:

1. Define a `Distiller` class with the custom logic for the distillation process.
2. Train the `teacher` model which is a CNN that implements regularization via dropout.
3. Train a `student` model (a smaller version of the teacher without regularization) by using knowledge distillation.
4. Train another `student` model from scratch without distillation called `student_scratch`.
5. Compare the three students.

This notebook is based on [this](#) official Keras tutorial.

If you want a more theoretical approach to this topic be sure to check this paper [Hinton et al. (2015)](#).

Let's get started!

Imports

```
# For setting random seeds
import os
os.environ['PYTHONHASHSEED']=str(42)

# Libraries
import random
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds

# More random seed setup
tf.random.set_seed(42)
```

```
np.random.seed(42)
```

Prepare the data

For this lab you will use the [cats vs dogs](#) which is composed of many images of cats and dogs alongise their respective labels.

Begin by downloading the data:

```
# Define train/test splits
splits = ['train[:80%]', 'train[80%:90%]', 'train[90%:]']

# Download the dataset
(train_examples, validation_examples, test_examples), info = t

# Print useful information
num_examples = info.splits['train'].num_examples
num_classes = info.features['label'].num_classes

print(f"There are {num_examples} images for {num_classes} clas
```

Downloading and preparing dataset cats_vs_dogs/4.0.0 (download: 786.68 MiB, gene

Dl Completed...: 100% 1/1 [00:09<00:00, 9.75s/ url]

Dl Size...: 100% 786/786 [00:09<00:00, 84.72 MiB/s]

```
WARNING:absl:1738 images were corrupted and were skipped
Shuffling and writing examples to /root/tensorflow_datasets/cats_vs_dogs/4.0.0.i
```

100% 23261/23262 [00:02<00:00, 6899.65 examples/s]

Dataset cats_vs_dogs downloaded and prepared to /root/tensorflow_datasets/cats_v

```
There are 23262 images for 2 classes.
```

```
info
```

```
tfds.core.DatasetInfo(
    name='cats_vs_dogs',
    version=4.0.0,
    description='A large set of images of cats and dogs.There are 1738 corrupted
    homepage='https://www.microsoft.com/en-us/download/details.aspx?id=54765',
    features=FeaturesDict({
        'image': Image(shape=(None, None, 3), dtype=tf.uint8),
        'image/filename': Text(shape=(), dtype=tf.string),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    }),
    total_num_examples=23262,
```

```
        splits={
            'train': 23262,
        },
        supervised_keys=('image', 'label'),
        citation="""@Inproceedings (Conference){asirra-a-captcha-that-exploits-inter
        author = {Elson, Jeremy and Douceur, John (JD) and Howell, Jon and Saul, Jar
        title = {Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categ
        booktitle = {Proceedings of 14th ACM Conference on Computer and Communicatic
        year = {2007},
        month = {October},
        publisher = {Association for Computing Machinery, Inc.},
        url = {https://www.microsoft.com/en-us/research/publication/asirra-a-captcha
        edition = {Proceedings of 14th ACM Conference on Computer and Communications
        }""",
        redistribution_info=,
    )
```

Preprocess the data for training by normalizing pixel values, reshaping them and creating batches of data:

```
# Some global variables
pixels = 224
IMAGE_SIZE = (pixels, pixels)
BATCH_SIZE = 32

# Apply resizing and pixel normalization
def format_image(image, label):
    image = tf.image.resize(image, IMAGE_SIZE) / 255.0
    return  image, label

# Create batches of data
train_batches = train_examples.shuffle(num_examples // 4).map(
validation_batches = validation_examples.map(format_image).bat
test_batches = test_examples.map(format_image).batch(1)
```

Code the custom `Distiller` model

In order to implement the distillation process you will create a custom Keras model which you will name `Distiller`. In order to do this you need to override some of the vanilla methods of a `keras.Model` to include the custom logic for the knowledge distillation. You need to override these methods:

- `compile`: This model needs some extra parameters to be compiled such as the teacher and student losses, the alpha and the temperature.

- `train_step`: Controls how the model is trained. This will be where the actual knowledge distillation logic will be found. This method is what is called when you do `model.fit`.
- `test_step`: Controls the evaluation of the model. This method is what is called when you do `model.evaluate`.

To learn more about customizing models check out the official docs.

```
class Distiller(keras.Model):

    # Needs both the student and teacher models to create an ins
    def __init__(self, student, teacher):
        super(Distiller, self).__init__()
        self.teacher = teacher
        self.student = student


    # Will be used when calling model.compile()
    def compile(self, optimizer, metrics, student_loss_fn,
                distillation_loss_fn, alpha, temperature):

        # Compile using the optimizer and metrics
        super(Distiller, self).compile(optimizer=optimizer, metr

        # Add the other params to the instance
        self.student_loss_fn = student_loss_fn
        self.distillation_loss_fn = distillation_loss_fn
        self.alpha = alpha
        self.temperature = temperature


    # Will be used when calling model.fit()
    def train_step(self, data):
        # Data is expected to be a tuple of (features, labels)
        x, y = data

        # Vanilla forward pass of the teacher
        # Note that the teacher is NOT trained
        teacher_predictions = self.teacher(x, training=False)

        # Use GradientTape to save gradients
        with tf.GradientTape() as tape:
```

```
            # Vanilla forward pass of the student
            student_predictions = self.student(x, training=True)

            # Compute vanilla student loss
            student_loss = self.student_loss_fn(y, student_predi

            # Compute distillation loss
            # Should be KL divergence between logits softened by
            distillation_loss = self.distillation_loss_fn(
                tf.nn.softmax(teacher_predictions / self.tempera
                tf.nn.softmax(student_predictions / self.tempera

            # Compute loss by weighting the two previous losses
            loss = self.alpha * student_loss + (1 - self.alpha)

        # Use tape to calculate gradients for student
        trainable_vars = self.student.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)

        # Update student weights
        # Note that this done ONLY for the student
        self.optimizer.apply_gradients(zip(gradients, trainable_

        # Update the metrics
        self.compiled_metrics.update_state(y, student_prediction

        # Return a performance dictionary
        # You will see this being outputted during training
        results = {m.name: m.result() for m in self.metrics}
        results.update({"student_loss": student_loss, "distillat
        return results


    # Will be used when calling model.evaluate()
    def test_step(self, data):
        # Data is expected to be a tuple of (features, labels)
        x, y = data

        # Use student to make predictions
        # Notice that the training param is set to False
        y_prediction = self.student(x, training=False)

```

```python
        # Calculate student's vanilla loss
        student_loss = self.student_loss_fn(y, y_prediction)

        # Update the metrics
        self.compiled_metrics.update_state(y, y_prediction)

        # Return a performance dictionary
        # You will see this being outputted during inference
        results = {m.name: m.result() for m in self.metrics}
        results.update({"student_loss": student_loss})
        return results
```

Teacher and student models

For the models you will use a standard CNN architecture that implements regularization via some dropout layers (in the case of the teacher), but it could be any Keras model.

Define the `create_model` functions to create models with the desired architecture using Keras' Sequential Model.

Notice that `create_small_model` returns a simplified version of the model (in terms of number of layers and absence of regularization) that `create_big_model` returns:

```python
# Teacher model
def create_big_model():
  tf.random.set_seed(42)
  model = keras.models.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', input_s
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.2),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(128, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.5),
    keras.layers.Flatten(),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(2)
  ])

```

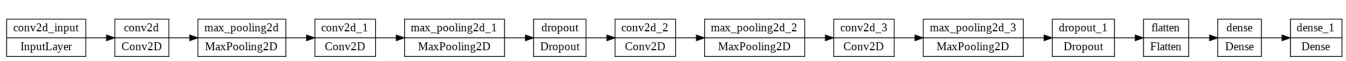
```
  return model



# Student model
def create_small_model():
  tf.random.set_seed(42)
  model = keras.models.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', input_s
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(2)
  ])

  return model
```

There are two important things to notice:

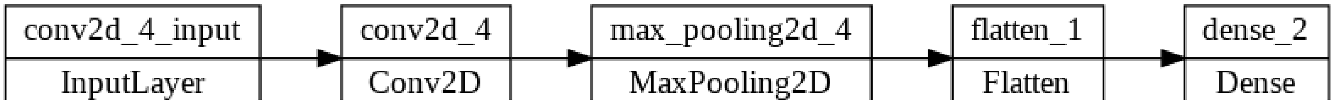
- The last layer does not have an softmax activation because the raw logits are needed for the knowledge distillation.
- Regularization via dropout layers will be applied to the teacher but NOT to the student. This is because the student should be able to learn this regularization through the distillation process.

Remember that the student model can be thought of as a simplified (or compressed) version of the teacher model.

```
# Create the teacher
teacher = create_big_model()

# Plot architecture
keras.utils.plot_model(teacher, rankdir="LR")
```




```
# Create the student
student = create_small_model()

# Plot architecture
keras.utils.plot_model(student, rankdir="LR")
```



Check the actual difference in number of trainable parameters (weights and biases) between both models:

```
# Calculates number of trainable params for a given model
def num_trainable_params(model):
  return np.sum([np.prod(v.get_shape()) for v in model.trainab


student_params = num_trainable_params(student)
teacher_params = num_trainable_params(teacher)

print(f"Teacher model has: {teacher_params} trainable params.\
print(f"Student model has: {student_params} trainable params.\
print(f"Teacher model is roughly {teacher_params//student_para
```

```
Teacher model has: 9568898 trainable params.

Student model has: 789442 trainable params.

Teacher model is roughly 12 times bigger than the student model.
```

Train the teacher

In knowledge distillation it is assumed that the teacher has already been trained so the natural first step is to train the teacher. You will do so for a total of 8 epochs:

```
# Compile the teacher model
teacher.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_lo
    optimizer=keras.optimizers.Adam(),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
)

# Fit the model and save the training history (will take from
teacher_history = teacher.fit(train_batches, epochs=8, validat
```

```
Epoch 1/8
582/582 [==============================] - 41s 48ms/step - loss: 0.6794 - sparse
Epoch 2/8
582/582 [==============================] - 31s 46ms/step - loss: 0.5932 - sparse
Epoch 3/8
```

```
582/582 [==============================] - 31s 47ms/step - loss: 0.5168 - sparse
Epoch 4/8
582/582 [==============================] - 31s 46ms/step - loss: 0.4571 - sparse
Epoch 5/8
582/582 [==============================] - 31s 46ms/step - loss: 0.3968 - sparse
Epoch 6/8
582/582 [==============================] - 31s 46ms/step - loss: 0.3473 - sparse
Epoch 7/8
582/582 [==============================] - 31s 46ms/step - loss: 0.3106 - sparse
Epoch 8/8
582/582 [==============================] - 31s 46ms/step - loss: 0.2628 - sparse
```

Train a student from scratch for reference

In order to assess the effectiveness of the distillation process, train a model that is equivalent to the student but without doing knowledge distillation. Notice that the training is done for only 5 epochs:

```
# Create student_scratch model with the same characteristics a
student_scratch = create_small_model()

# Compile it
student_scratch.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_lo
    optimizer=keras.optimizers.Adam(),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
)

# Train and evaluate student trained from scratch (will take a
student_scratch_history = student_scratch.fit(train_batches, e
```

```
Epoch 1/5
582/582 [==============================] - 29s 42ms/step - loss: 0.7741 - sparse
Epoch 2/5
582/582 [==============================] - 28s 42ms/step - loss: 0.4950 - sparse
Epoch 3/5
582/582 [==============================] - 28s 42ms/step - loss: 0.3865 - sparse
Epoch 4/5
582/582 [==============================] - 28s 42ms/step - loss: 0.2919 - sparse
Epoch 5/5
582/582 [==============================] - 28s 42ms/step - loss: 0.1997 - sparse
```

Knowledge Distillation

To perform the knowledge distillation process you will use the custom model you previously coded. To do so, begin by creating an instance of the `Distiller` class and passing in the student and teacher models. Then compile it with the appropiate parameters and train it!

The two student models are trained for only 5 epochs unlike the teacher that was trained for 8. This is done to showcase that the knowledge distillation allows for quicker training times as the student learns from an already trained model

```
# Create Distiller instance
distiller = Distiller(student=student, teacher=teacher)

# Compile Distiller model
distiller.compile(
    student_loss_fn=keras.losses.SparseCategoricalCrossentropy
    optimizer=keras.optimizers.Adam(),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
    distillation_loss_fn=keras.losses.KLDivergence(),
    alpha=0.05,
    temperature=5,
)

# Distill knowledge from teacher to student (will take around
distiller_history = distiller.fit(train_batches, epochs=5, val
```

```
Epoch 1/5
582/582 [==============================] - 30s 44ms/step - sparse_categorical_ac
Epoch 2/5
582/582 [==============================] - 29s 43ms/step - sparse_categorical_ac
Epoch 3/5
582/582 [==============================] - 29s 44ms/step - sparse_categorical_ac
Epoch 4/5
582/582 [==============================] - 29s 44ms/step - sparse_categorical_ac
Epoch 5/5
582/582 [==============================] - 29s 44ms/step - sparse_categorical_ac
```

Comparing the models

To compare the models you can check the `sparse_categorical_accuracy` of each one on the test set:

```
# Compute accuracies
student_scratch_acc = student_scratch.evaluate(test_batches, r
distiller_acc = distiller.evaluate(test_batches, return_dict=T
teacher_acc = teacher.evaluate(test_batches, return_dict=True)

```

```
# Print results
print(f"\n\nTeacher achieved a sparse_categorical_accuracy of
print(f"Student with knowledge distillation achieved a sparse_
print(f"Student without knowledge distillation achieved a spar
```

```
2326/2326 [==============================] - 7s 3ms/step - loss: 0.7573 - sparse
2326/2326 [==============================] - 7s 3ms/step - sparse_categorical_ac
2326/2326 [==============================] - 8s 3ms/step - loss: 0.3197 - sparse


Teacher achieved a sparse_categorical_accuracy of 86.41%.

Student with knowledge distillation achieved a sparse_categorical_accuracy of 73

Student without knowledge distillation achieved a sparse_categorical_accuracy of
```

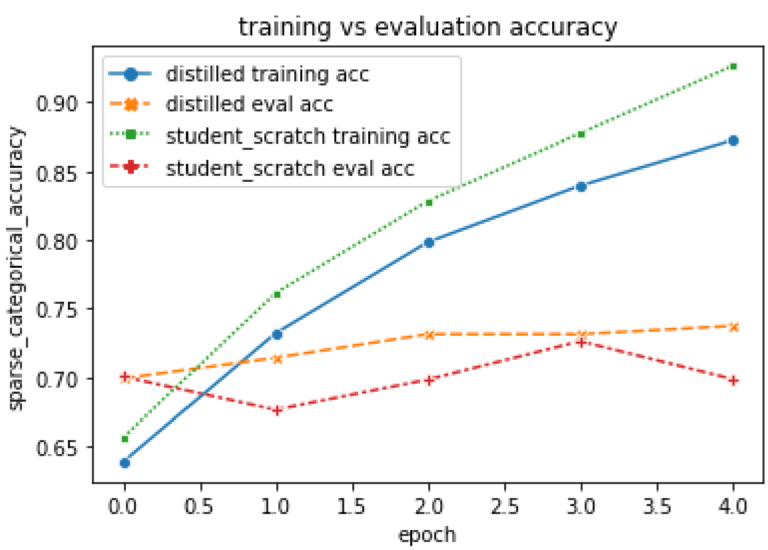
The teacher model yields a higger accuracy than the two student models. This is expected since it was trained for more epochs while using a bigger architecture.

Notice that the student without distillation was outperfomed by the student with knowledge distillation.

Since you saved the training history of each model you can create a plot for a better comparison of the two student models.

```
# Get relevant metrics from a history
def get_metrics(history):
  history = history.history
  acc = history['sparse_categorical_accuracy']
  val_acc = history['val_sparse_categorical_accuracy']
  return acc, val_acc


# Plot training and evaluation metrics given a dict of histori
def plot_train_eval(history_dict):

  metric_dict = {}

  for k, v in history_dict.items():
    acc, val_acc= get_metrics(v)
    metric_dict[f'{k} training acc'] = acc
    metric_dict[f'{k} eval acc'] = val_acc

  acc_plot = pd.DataFrame(metric_dict)
```

```
    acc_plot = sns.lineplot(data=acc_plot, markers=True)
    acc_plot.set_title('training vs evaluation accuracy')
    acc_plot.set_xlabel('epoch')
    acc_plot.set_ylabel('sparse_categorical_accuracy')
    plt.show()


# Plot for comparing the two student models
plot_train_eval({
    "distilled": distiller_history,
    "student_scratch": student_scratch_history,
})
```



This plot is very interesting because it shows that the distilled version outperformed the unmodified one in almost all of the epochs when using the evaluation set. Alongside this, the student without distillation yields a bigger training accuracy, which is a sign that it is overfitting more than the distilled model. **This hints that the distilled model was able to learn from the regularization that the teacher implemented!** Pretty cool, right?

Congratulations on finishing this ungraded lab! Now you should have a clearer understanding of what Knowledge Distillation is and how it can be implemented using Tensorflow and Keras.

This process is widely used for model compression and has proven to perform really well. In fact you might have heard about `DistilBert`, which is a smaller, faster, cheaper and lighter of BERT.

Keep it up!

0s completed at 4:29 AM