Deep Learning

Lecture Topic:
Best Practices for the Real World

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand hyperparameter tuning

2. Understand mixed-precision training

3. Understand how to train Keras on multiple GPUs or on a TPU

Roadmap of this lecture:

1. hyperparameter tuning

2. Mixed-precision training

3. Train Keras on multiple GPUs or on a TPU

Hyper-parameter Optimization

When building a deep learning model, you have to make many seemingly arbitrary decisions: How many layers should you stack? How many units or filters should go in each layer? Should you use `relu` as activation, or a different function? Should you use `BatchNormalization` after a given layer? How much dropout should you use? And so on. These architecture-level parameters are called *hyperparameters* to distinguish them from the *parameters* of a model, which are trained via backpropagation.

Hyper-parameter Optimization: Challenges

- The hyperparameter space is typically made up of discrete decisions and thus isn't continuous or differentiable. Hence, you typically can't do gradient descent in hyperparameter space. Instead, you must rely on gradient-free optimization techniques, which naturally are far less efficient than gradient descent.
- Computing the feedback signal of this optimization process (does this set of hyperparameters lead to a high-performing model on this task?) can be extremely expensive: it requires creating and training a new model from scratch on your dataset.
- The feedback signal may be noisy: if a training run performs 0.2% better, is that because of a better model configuration, or because you got lucky with the initial weight values?

Use KerasTuner for Hyper-parameter Optimization

Let's start by installing KerasTuner:

```
!pip install keras-tuner -q
```

KerasTuner lets you replace hard-coded hyperparameter values, such as `units=32`, with a range of possible choices, such as `Int(name="units", min_value=16, max_value=64, step=16)`. This set of choices in a given model is called the *search space* of the hyperparameter tuning process.

Use KerasTuner for Hyper-parameter Optimization

To specify a search space, define a model-building function (see the next listing). It takes an hp argument, from which you can sample hyperparameter ranges, and it returns a compiled Keras model.

Listing 13.1 A KerasTuner model-building function

```
from tensorflow import keras
from tensorflow.keras import layers

def build_model(hp):
    units = hp.Int(name="units", min_value=16, max_value=64, step=16)
    model = keras.Sequential([
        layers.Dense(units, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
    model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
return model
```

Sample hyperparameter values from the hp object. After sampling, these values (such as the "units" variable here) are just regular Python constants.

Different kinds of hyperparameters are available: Int, Float, Boolean, Choice.

The function returns a compiled model.

Use KerasTuner for Hyper-parameter Optimization

To specify a search space, define a model-building function (see the next listing). It takes an `hp` argument, from which you can sample hyperparameter ranges, and it returns a compiled Keras model.

Listing 13.1 A KerasTuner model-building function

```
from tensorflow import keras
from tensorflow.keras import layers

def build_model(hp):
    units = hp.Int(name="units", min_value=16, max_value=64, step=16)
    model = keras.Sequential([
        layers.Dense(units, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
    model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
return model
```

Sample hyperparameter values from the hp object. After sampling, these values (such as the "units" variable here) are just regular Python constants.

Different kinds of hyperparameters are available: Int, Float, Boolean, Choice.

The function returns a compiled model.

Use KerasTuner for Hyper-parameter Optimization

units $\in \{16, 32, 48, 64\}$

Use KerasTuner for Hyper-parameter Optimization

To specify a search space, define a model-building function (see the next listing). It takes an hp argument, from which you can sample hyperparameter ranges, and it returns a compiled Keras model.

Listing 13.1 A KerasTuner model-building function

```
from tensorflow import keras
from tensorflow.keras import layers

def build_model(hp):
    units = hp.Int(name="units", min_value=16, max_value=64, step=16)
    model = keras.Sequential([
        layers.Dense(units, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
    model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
return model
```

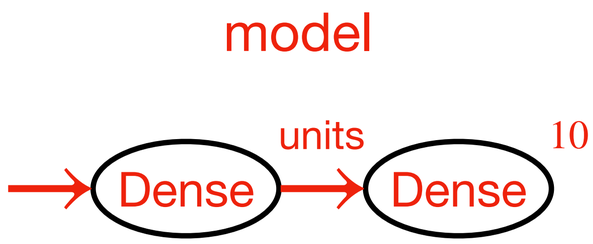
Sample hyperparameter values from the hp object. After sampling, these values (such as the "units" variable here) are just regular Python constants.

Different kinds of hyperparameters are available: Int, Float, Boolean, Choice.

The function returns a compiled model.

Use KerasTuner for Hyper-parameter Optimization

units $\in \{16, 32, 48, 64\}$



Use KerasTuner for Hyper-parameter Optimization

To specify a search space, define a model-building function (see the next listing). It takes an `hp` argument, from which you can sample hyperparameter ranges, and it returns a compiled Keras model.

Listing 13.1 A KerasTuner model-building function

```
from tensorflow import keras
from tensorflow.keras import layers

def build_model(hp):
    units = hp.Int(name="units", min_value=16, max_value=64, step=16)
    model = keras.Sequential([
        layers.Dense(units, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
    model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
return model
```

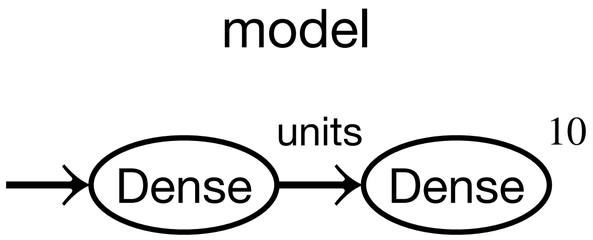
Sample hyperparameter values from the hp object. After sampling, these values (such as the "units" variable here) are just regular Python constants.

Different kinds of hyperparameters are available: Int, Float, Boolean, Choice.

The function returns a compiled model.

Use KerasTuner for Hyper-parameter Optimization

units $\in \{16, 32, 48, 64\}$



optimizer $\in$ {rmsprop, adam}

Use KerasTuner for Hyper-parameter Optimization

If you want to adopt a more modular and configurable approach to model-building, you can also subclass the `HyperModel` class and define a `build` method, as follows.

Listing 13.2 A KerasTuner `HyperModel`

```
import kerastuner as kt

class SimpleMLP(kt.HyperModel):
    def __init__(self, num_classes):
        self.num_classes = num_classes

    def build(self, hp):
        units = hp.Int(name="units", min_value=16, max_value=64, step=16)
        model = keras.Sequential([
            layers.Dense(units, activation="relu"),
            layers.Dense(self.num_classes, activation="softmax")
        ])
        optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
        model.compile(
            optimizer=optimizer,
            loss="sparse_categorical_crossentropy",
            metrics=["accuracy"])
        return model

hypermodel = SimpleMLP(num_classes=10)
```

The build() method is identical to our prior build_model() standalone function.

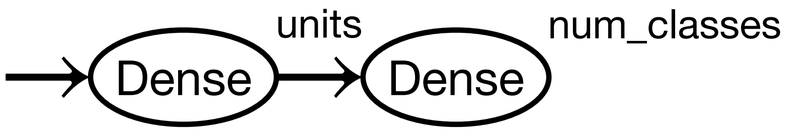
Thanks to the object-oriented approach, we can configure model constants as constructor arguments (instead of hardcoding them in the model-building function).

Use KerasTuner for Hyper-parameter Optimization

units $\in \{16, 32, 48, 64\}$

num_classes

model



optimizer $\in$ {rmsprop, adam}

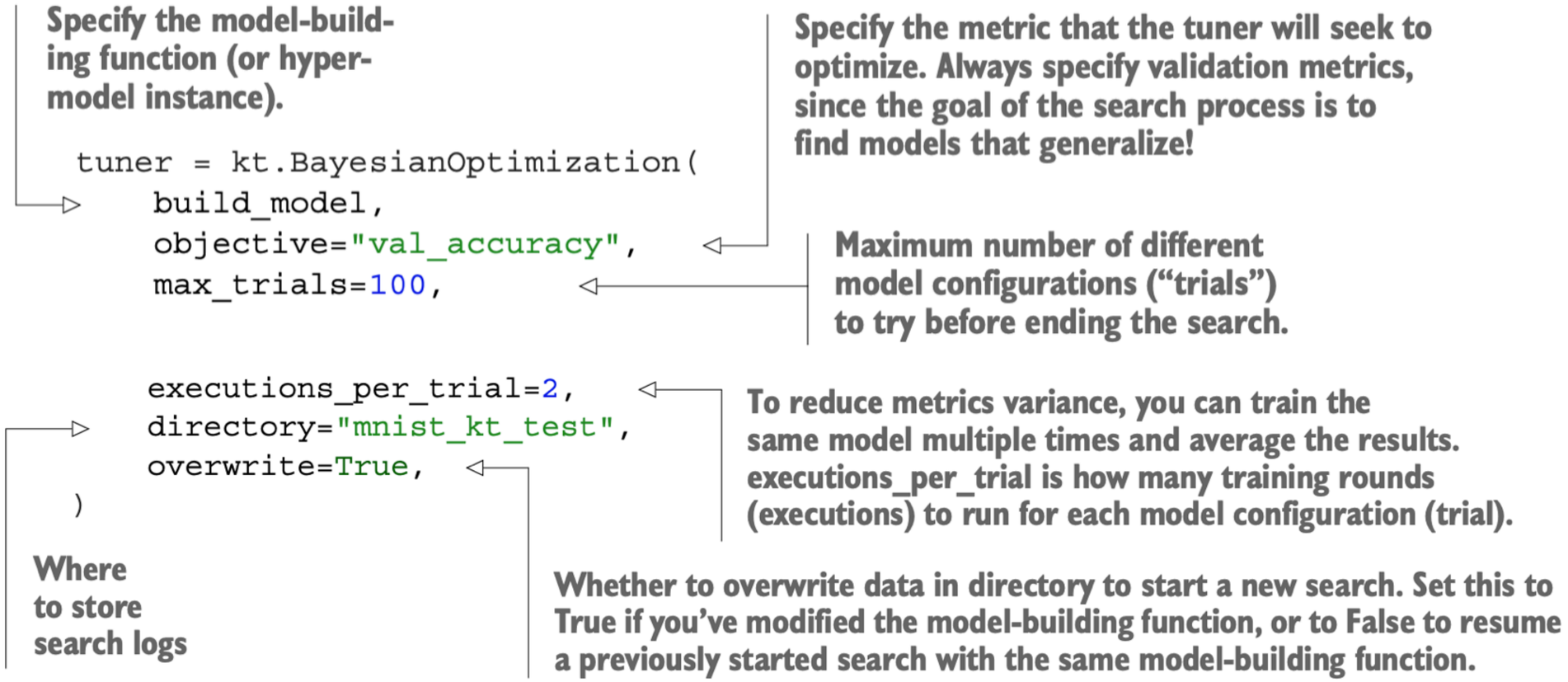
Use KerasTuner for Hyper-parameter Optimization

The next step is to define a "tuner." Schematically, you can think of a tuner as a `for` loop that will repeatedly

- Pick a set of hyperparameter values
- Call the model-building function with these values to create a model
- Train the model and record its metrics

KerasTuner has several built-in tuners available—`RandomSearch`, `BayesianOptimization`, and `Hyperband`. Let's try `BayesianOptimization`, a tuner that attempts to make smart predictions for which new hyperparameter values are likely to perform best given the outcomes of previous choices:

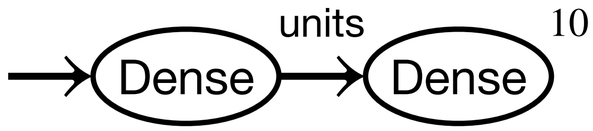
Use KerasTuner for Hyper-parameter Optimization



Use KerasTuner for Hyper-parameter Optimization

units $\in \{16, 32, 48, 64\}$

model



optimizer $\in$ {rmsprop, adam}

tuner: BayesianOptimization

You can display an overview of the search space via search_space_summary( ):

```
>>> tuner.search_space_summary()
Search space summary
Default search space size: 2
units (Int)
{"default": None,
 "conditions": [],
 "min_value": 16 ,
 "max_value": 64 ,
 "step": 16 ,
 "sampling": None}
optimizer (Choice)
{"default": "rmsprop",
 "conditions": [],
 "values": ["rmsprop", "adam"],
 "ordered": False}
```

Use KerasTuner for Hyper-parameter Optimization

Objective maximization and minimization

For built-in metrics (like accuracy, in our case), the *direction* of the metric (accuracy should be maximized, but a loss should be minimized) is inferred by KerasTuner. However, for a custom metric, you should specify it yourself, like this:

```
objective = kt.Objective(
    name="val_accuracy",
    direction="max")
tuner = kt.BayesianOptimization(
    build_model,
    objective=objective,
    ...
)
```

The metric's name, as found in epoch logs

The metric's desired direction: "min" or "max"

Use KerasTuner for Hyper-parameter Optimization

Finally, let's launch the search. Don't forget to pass validation data, and make sure not to use your test set as validation data—otherwise you'd quickly start overfitting to your test data, and you wouldn't be able to trust your test metrics anymore:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape((-1, 28 * 28)).astype("float32") / 255
x_test = x_test.reshape((-1, 28 * 28)).astype("float32") / 255
x_train_full = x_train[:]
y_train_full = y_train[:]
```

Reserve these for later.

Set these aside as a validation set.

```
num_val_samples = 10000
x_train, x_val = x_train[:-num_val_samples], x_train[-num_val_samples:]
y_train, y_val = y_train[:-num_val_samples], y_train[-num_val_samples:]
callbacks = [
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=5),
]
tuner.search(
    x_train, y_train,
    batch_size=128,
    epochs=100,
    validation_data=(x_val, y_val),
    callbacks=callbacks,
    verbose=2,
)
```

This takes the same arguments as fit() (it simply passes them down to fit() for each new model).

Use a large number of epochs (you don't know in advance how many epochs each model will need), and use an EarlyStopping callback to stop training when you start overfitting.

Use KerasTuner for Hyper-parameter Optimization

The preceding example will run in just a few minutes, since we're only looking at a few possible choices and we're training on MNIST. However, with a typical search space and dataset, you'll often find yourself letting the hyperparameter search run overnight or even over several days. If your search process crashes, you can always restart it—just specify `overwrite=False` in the tuner so that it can resume from the trial logs stored on disk.

Once the search is complete, you can query the best hyperparameter configurations, which you can use to create high-performing models that you can then retrain.

Listing 13.3 Querying the best hyperparameter configurations

```
top_n = 4
best_hps = tuner.get_best_hyperparameters(top_n)
```

Returns a list of HyperParameter objects, which you can pass to the model-building function

Use KerasTuner for Hyper-parameter Optimization

Before we can train on the full training data, though, there's one last parameter we need to settle: the optimal number of epochs to train for. Typically, you'll want to train the new models for longer than you did during the search: using an aggressive `patience` value in the `EarlyStopping` callback saves time during the search, but it may lead to under-fit models. Just use the validation set to find the best epoch:

```
def get_best_epoch(hp):
    model = build_model(hp)
    callbacks=[
        keras.callbacks.EarlyStopping(
            monitor="val_loss", mode="min", patience=10)
    ]

    history = model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=100,
        batch_size=128,
        callbacks=callbacks)
    val_loss_per_epoch = history.history["val_loss"]
    best_epoch = val_loss_per_epoch.index(min(val_loss_per_epoch)) + 1
    print(f"Best epoch: {best_epoch}")
    return best_epoch
```

Note the very high patience value.

Train Final Model

Finally, train on the full dataset for just a bit longer than this epoch count, since you're training on more data; 20% more in this case:

Train using both training and validation data

```
def get_best_trained_model(hp):
    best_epoch = get_best_epoch(hp)
    model.fit(
        x_train_full, y_train_full,
        batch_size=128, epochs=int(best_epoch * 1.2))
    return model

best_models = []
for hp in best_hps:
    model = get_best_trained_model(hp)
    model.evaluate(x_test, y_test)
    best_models.append(model)
```

Use KerasTuner for Hyper-parameter Optimization

Note that if you're not worried about slightly underperforming, there's a shortcut you can take: just use the tuner to reload the top-performing models with the best weights saved during the hyperparameter search, without retraining new models from scratch:

```
best_models = tuner.get_best_models(top_n)
```

Crafting the right search space for hyper-parameters

1. Human experience

2. Automated tools

> KerasTuner attempts to provide *premade search spaces* that are relevant to broad categories of problems, such as image classification. Just add data, run the search, and get a pretty good model. You can try the hypermodels `kt.applications.HyperXception` and `kt.applications.HyperResNet`, which are effectively tunable versions of Keras Applications models.

Quiz questions:

1. What is hyper-parameter tuning?

2. How to use KerasTuner to tune hyper-parameters?

Roadmap of this lecture:

1. hyperparameter tuning

2. Mixed-precision training

3. Train Keras on multiple GPUs or on a TPU

Scaling up model training

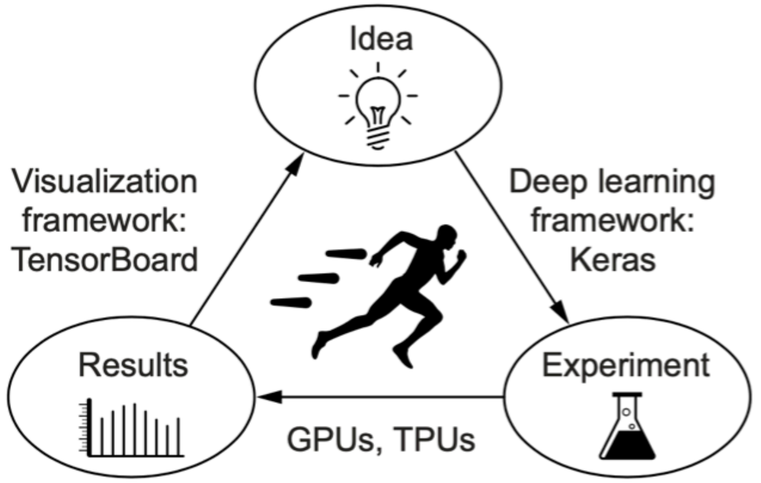


Figure 13.1 The loop of progress

In this section, you'll learn about three ways you can train your models faster:

- Mixed-precision training, which you can use even with a single GPU
- Training on multiple GPUs
- Training on TPUs

Speed up training on GPU with mixed precision

There are three of levels of precision you'd typically use:

- Half precision, or `float16`, where numbers are stored on 16 bits
- Single precision, or `float32`, where numbers are stored on 32 bits
- Double precision, or `float64`, where numbers are stored on 64 bits

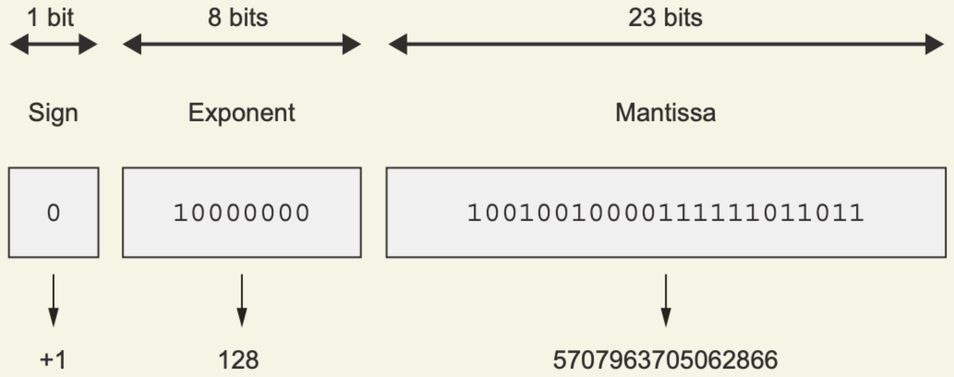
A note on floating-point encoding

A counterintuitive fact about floating-point numbers is that representable numbers are not uniformly distributed. Larger numbers have lower precision: there are the same number of representable values between `2 ** N` and `2 ** (N + 1)` as there are between 1 and 2, for any *N*.

That's because floating-point numbers are encoded in three parts—the sign, the significant value (called the "mantissa"), and the exponent, in the form

```
{sign} * (2 ** ({exponent} - 127)) * 1.{mantissa}
```

For example, here's how you would encode the closest `float32` value approximating Pi:




```
value = +1 * (2 ** (128 - 127)) * 1.5707963705062866
value = 3.1415927410125732
```

The number Pi encoded in single precision via a sign bit, an integer exponent, and an integer mantissa

For this reason, the numerical error incurred when converting a number to its floating-point representation can vary wildly depending on the exact value considered, and the error tends to get larger for numbers with a large absolute value.

Speed up training on GPU with mixed precision

You can, however, use a hybrid approach: that's what mixed precision is about. The idea is to leverage 16-bit computations in places where precision isn't an issue, and to work with 32-bit values in other places to maintain numerical stability. Modern GPUs and TPUs feature specialized hardware that can run 16-bit operations much faster and use less memory than equivalent 32-bits operations. By using these lower-precision operations whenever possible, you can speed up training on those devices by a significant factor. Meanwhile, by maintaining the precision-sensitive parts of the model in single precision, you can get these benefits without meaningfully impacting model quality.

And those benefits are considerable: on modern NVIDIA GPUs, mixed precision can speed up training by up to 3X. It's also beneficial when training on a TPU (a subject we'll get to in a bit), where it can speed up training by up to 60%.

Beware of dtype defaults

Single precision is the default floating-point type throughout Keras and TensorFlow: any tensor or variable you create will be in `float32` unless you specify otherwise. For NumPy arrays, however, the default is `float64`!

Converting a default NumPy array to a TensorFlow tensor will result in a `float64` tensor, which may not be what you want:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> np_array = np.zeros((2, 2))
>>> tf_tensor = tf.convert_to_tensor(np_array)
>>> tf_tensor.dtype
tf.float64
```

Remember to be explicit about data types when converting NumPy arrays:

```
>>> np_array = np.zeros((2, 2))
>>> tf_tensor = tf.convert_to_tensor(np_array, dtype="float32")
>>> tf_tensor.dtype
tf.float32
```

Specify the dtype explicitly.

Note that when you call the Keras `fit()` method with NumPy data, it will do this conversion for you.

Speed up training on GPU with mixed precision

MIXED-PRECISION TRAINING IN PRACTICE

When training on a GPU, you can turn on mixed precision like this:

```
from tensorflow import keras
keras.mixed_precision.set_global_policy("mixed_float16")
```

Typically, most of the forward pass of the model will be done in `float16` (with the exception of numerically unstable operations like softmax), while the weights of the model will be stored and updated in `float32`.

Keras layers have a `variable_dtype` and a `compute_dtype` attribute. By default, both of these are set to `float32`. When you turn on mixed precision, the `compute_dtype` of most layers switches to `float16`, and those layers will cast their inputs to `float16` and will perform their computations in `float16` (using half-precision copies of the weights). However, since their `variable_dtype` is still `float32`, their weights will be able to receive accurate `float32` updates from the optimizer, as opposed to half-precision updates.

Note that some operations may be numerically unstable in `float16` (in particular, softmax and crossentropy). If you need to opt out of mixed precision for a specific layer, just pass the argument `dtype="float32"` to the constructor of this layer.

Quiz questions:

1. What is mix-precision training?

2. How to do mix-precision training?

Roadmap of this lecture:

1. hyperparameter tuning

2. Mixed-precision training

3. Train Keras on multiple GPUs or on a TPU

Multi-GPU training

There are two ways to distribute computation across multiple devices: *data parallelism* and *model parallelism*.

With data parallelism, a single model is replicated on multiple devices or multiple machines. Each of the model replicas processes different batches of data, and then they merge their results.

With model parallelism, different parts of a single model run on different devices, processing a single batch of data together at the same time. This works best with models that have a naturally parallel architecture, such as models that feature multiple branches.

In practice, model parallelism is only used for models that are too large to fit on any single device: it isn't used as a way to speed up training of regular models, but as a way to train larger models. We won't cover model parallelism in these pages; instead we'll focus on what you'll be using most of the time: data parallelism. Let's take a look at how it works.

Multi-GPU training

SINGLE-HOST, MULTI-DEVICE SYNCHRONOUS TRAINING

Once you're able to `import tensorflow` on a machine with multiple GPUs, you're seconds away from training a distributed model. It works like this:

```
strategy = tf.distribute.MirroredStrategy()
print(f"Number of devices: {strategy.num_replicas_in_sync}")
with strategy.scope():
    model = get_compiled_model()
model.fit(
    train_dataset,
    epochs=100,
    validation_data=val_dataset,
    callbacks=callbacks)
```

Create a "distribution strategy" object. MirroredStrategy should be your go-to solution.

Use it to open a "strategy scope."

Everything that creates variables should be under the strategy scope. In general, this is only model construction and compile().

Train the model on all available devices.

Multi-GPU training

When you open a `MirroredStrategy` scope and build your model within it, the `MirroredStrategy` object will create one model copy (replica) on each available GPU. Then, each step of training unfolds in the following way (see figure 13.2):

1. A batch of data (called *global batch*) is drawn from the dataset.
2. It gets split into four different sub-batches (called *local batches*). For instance, if the global batch has 512 samples, each of the four local batches will have 128 samples. Because you want local batches to be large enough to keep the GPU busy, the global batch size typically needs to be very large.

Multi-GPU training

3. Each of the four replicas processes one local batch, independently, on its own device: they run a forward pass, and then a backward pass. Each replica outputs a "weight delta" describing by how much to update each weight variable in the model, given the gradient of the previous weights with respect to the loss of the model on the local batch.
4. The weight deltas originating from local gradients are efficiently merged across the four replicas to obtain a global delta, which is applied to all replicas. Because this is done at the end of every step, the replicas always stay in sync: their weights are always equal.

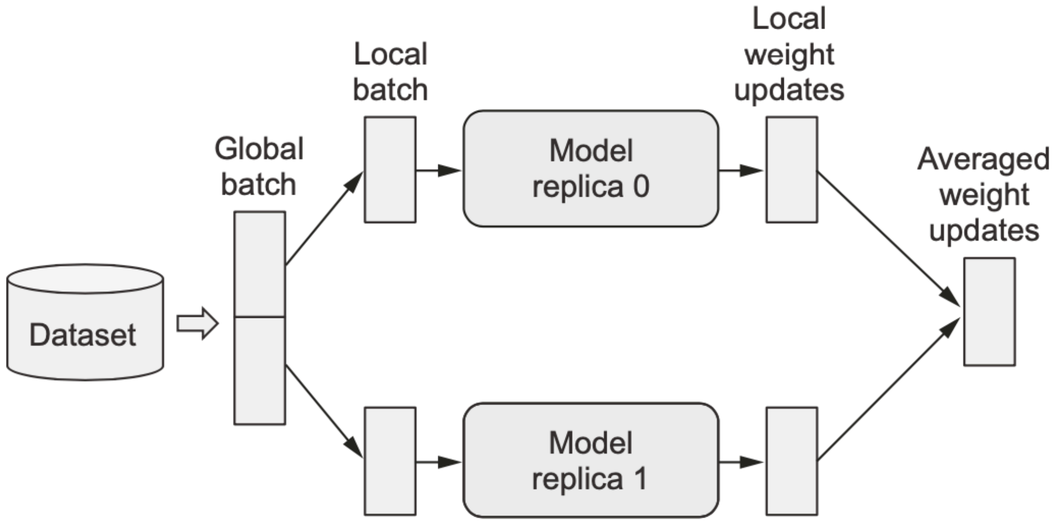


Figure 13.2 One step of `MirroredStrategy` training: each model replica computes local weight updates, which are then merged and used to update the state of all replicas.

Multi-GPU training

`tf.data` performance tips

When doing distributed training, always provide your data as a `tf.data.Dataset` object to guarantee best performance. (Passing your data as NumPy arrays also works, since those get converted to `Dataset` objects by `fit()`). You should also make sure you leverage data prefetching: before passing the dataset to `fit()`, call `dataset.prefetch(buffer_size)`. If you aren't sure what buffer size to pick, try the `dataset.prefetch(tf.data.AUTOTUNE)` option, which will pick a buffer size for you.

In an ideal world, training on *N* GPUs would result in a speedup of factor *N*. In practice, however, distribution introduces some overhead—in particular, merging the weight deltas originating from different devices takes some time. The effective speedup you get is a function of the number of GPUs used:

- With two GPUs, the speedup stays close to 2x.
- With four, the speedup is around 3.8x.
- With eight, it's around 7.3x.

This assumes that you're using a large enough global batch size to keep each GPU utilized at full capacity. If your batch size is too small, the local batch size won't be enough to keep your GPUs busy.

TPU training

Training on a TPU does involve jumping through some hoops, but it's worth the extra work: TPUs are really, really fast. Training on a TPU V2 will typically be 15x faster than training an NVIDIA P100 GPU. For most models, TPU training ends up being 3x more cost-effective than GPU training on average.

Using a TPU via Google Colab

You can actually use an 8-core TPU for free in Colab. In the Colab menu, under the Runtime tab, in the Change Runtime Type option, you'll notice that you have access to a TPU runtime in addition to the GPU runtime.

When you're using the GPU runtime, your models have direct access to the GPU without you needing to do anything special. This isn't true for the TPU runtime; there's an extra step you need to take before you can start building a model: you need to connect to the TPU cluster.

It works like this:

```
import tensorflow as tf
tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect()
print("Device:", tpu.master())
```

TPU training

Much like in the case of multi-GPU training, using the TPU requires you to open a distribution strategy scope—in this case, a `TPUStrategy` scope. `TPUStrategy` follows the same distribution template as `MirroredStrategy`—the model is replicated once per TPU core, and the replicas are kept in sync.

Here's a simple example.

Listing 13.4 Building a model in a `TPUStrategy` scope

```
from tensorflow import keras
from tensorflow.keras import layers

strategy = tf.distribute.TPUStrategy(tpu)
print(f"Number of replicas: {strategy.num_replicas_in_sync}")

def build_model(input_size):
    inputs = keras.Input((input_size, input_size, 3))
    x = keras.applications.resnet.preprocess_input(inputs)
    x = keras.applications.resnet.ResNet50(
        weights=None, include_top=False, pooling="max")(x)
```

TPU training

```
    outputs = layers.Dense(10, activation="softmax")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

with strategy.scope():
    model = build_model(input_size=32)
```

TPU training

We're almost ready to start training. But there's something a bit curious about TPUs in Colab: it's a two-VM setup, meaning that the VM that hosts your notebook runtime isn't the same VM that the TPU lives in. Because of this, you won't be able to train from files stored on the local disk (that is to say, on the disk linked to the VM that hosts the notebook). The TPU runtime can't read from there. You have two options for data loading:

- Train from data that lives in the memory of the VM (not on disk). If your data is in a NumPy array, this is what you're already doing.
- Store the data in a Google Cloud Storage (GCS) bucket, and create a dataset that reads the data directly from the bucket, without downloading locally. The TPU runtime can read data from GCS. This is your only option for datasets that are too large to live entirely in memory.

In our case, let's train from NumPy arrays in memory—the CIFAR10 dataset:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
model.fit(x_train, y_train, batch_size=1024)
```

Note that TPU training, much like multi-GPU training, requires large batch sizes to make sure the device stays well-utilized.

You'll notice that the first epoch takes a while to start—that's because your model is getting compiled to something that the TPU can execute. Once that step is done, the training itself is blazing fast.

TPU training

Leveraging step fusing to improve TPU utilization

Because a TPU has a lot of compute power available, you need to train with very large batches to keep the TPU cores busy. For small models, the batch size required can get extraordinarily large—upwards of 10,000 samples per batch. When working with enormous batches, you should make sure to increase your optimizer learning rate accordingly; you're going to be making fewer updates to your weights, but each update will be more accurate (since the gradients are computed using more data points), so you should move the weights by a greater magnitude with each update.

There is, however, a simple trick you can leverage to keep reasonably sized batches while maintaining full TPU utilization: *step fusing*. The idea is to run multiple steps of training during each TPU execution step. Basically, do more work in between two round trips from the VM memory to the TPU. To do this, simply specify the `steps_per_execution` argument in `compile()`—for instance, `steps_per_execution=8` to run eight steps of training during each TPU execution. For small models that are underutilizing the TPU, this can result in a dramatic speedup.

Quiz questions:

1. How to train a model on multiple GPUs?

2. How to train a model on TPU?