codelabs.developers.google.com /codelabs/keras-flowers-tpu/

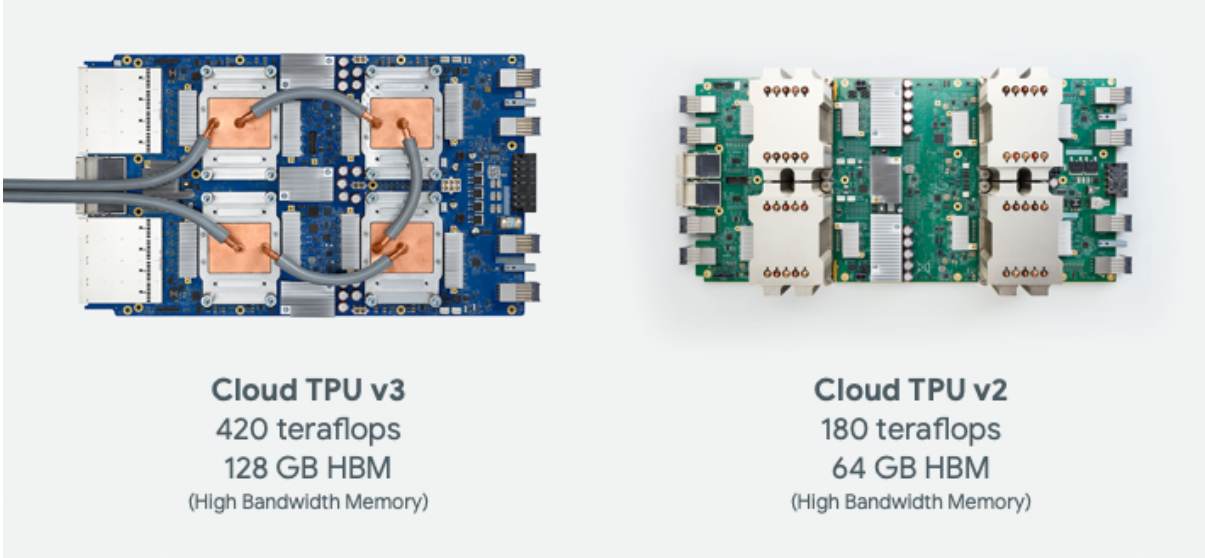
Keras and modern convnets, on TPUs

46-58 minutes

---

3. [INFO] What are Tensor Processing Units (TPUs) ?

In a nutshell



The code for training a model on TPU in Keras:

```
# detect the TPU
tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.config.experimental_connect_to_cluster(tpu)
tf.tpu.experimental.initialize_tpu_system(tpu)
strategy = tf.distribute.experimental.TPUStrategy(tpu)

# use TPUStrategy scope to define model
with strategy.scope():
  model = tf.keras.Sequential( ... )
  model.compile( ... )

# train model normally on a tf.data.Dataset
model.fit(training_dataset, epochs=EPOCHS, steps_per_epoch=...)
```

We will use TPUs today to build and optimize a flower classifier at interactive speeds (minutes per training run).

If this is old news to you SKIP ✗ to the next exercise otherwise READ ON

Why TPUs ?

Modern GPUs are organized around programmable "cores", a very flexible architecture that allows them to handle a variety of tasks such as 3D rendering, deep learning, physical simulations, etc.. TPUs on the

other hand pair a classic vector processor with a dedicated matrix multiply unit and excel at any task where large matrix multiplications dominate, such as neural networks.

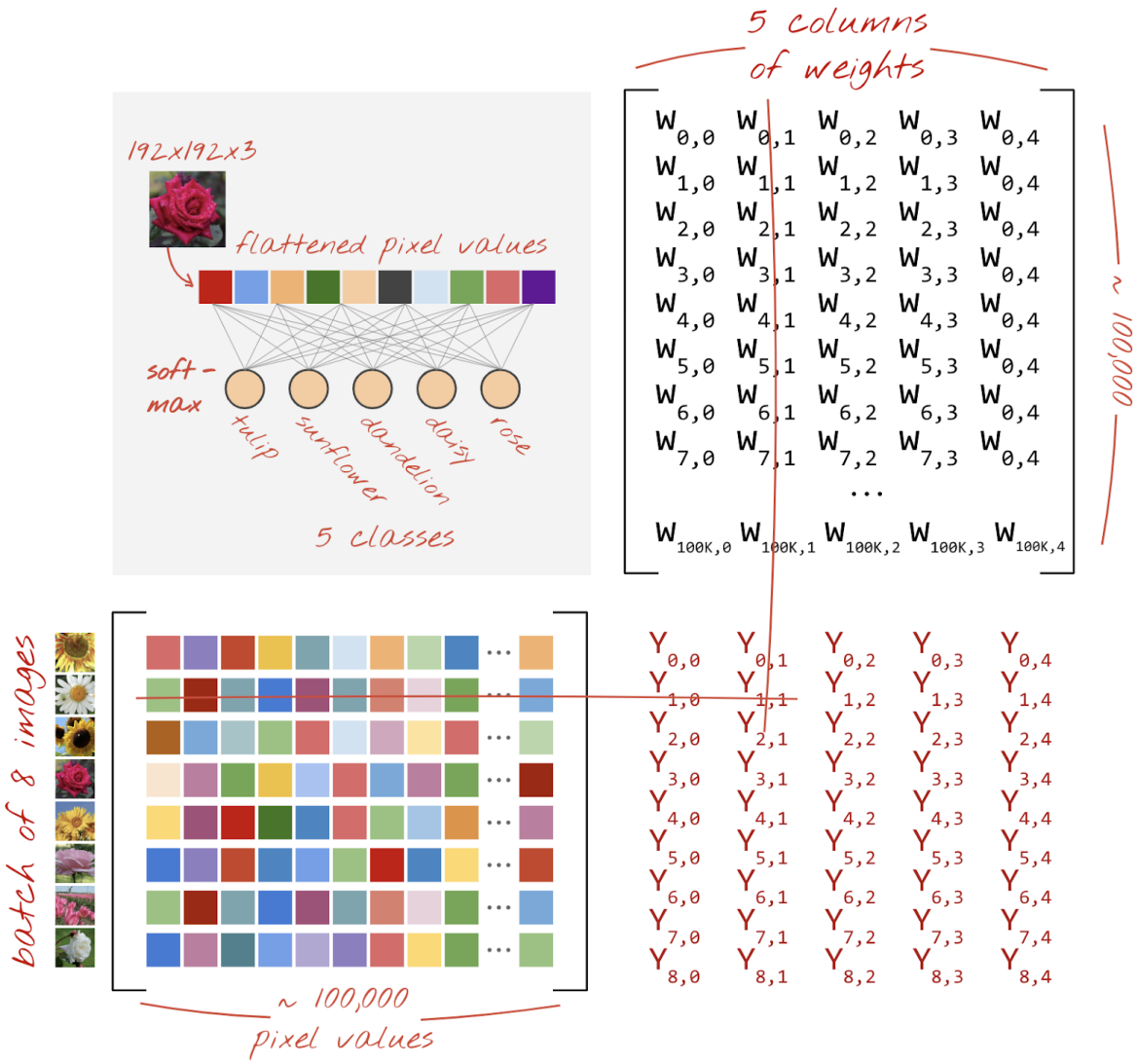
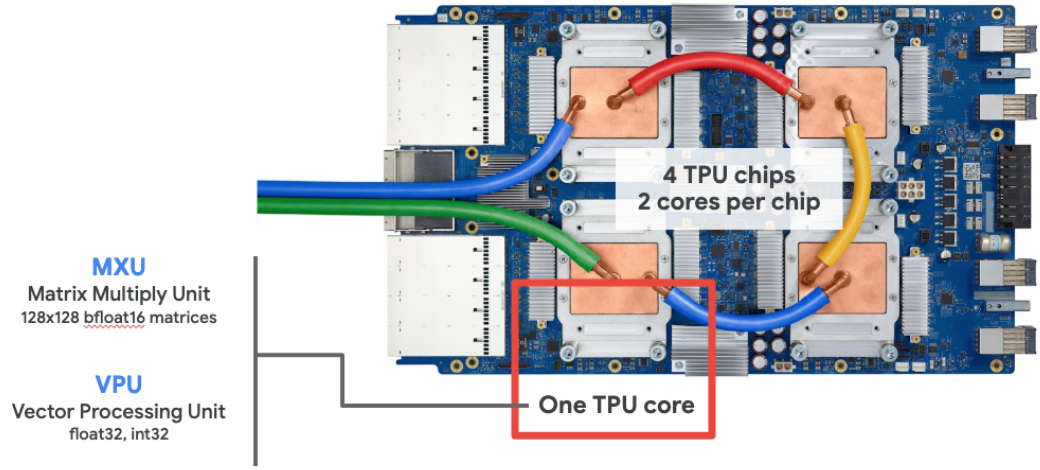


Illustration: a dense neural network layer as a matrix multiplication, with a batch of eight images processed through the neural network at once. Please run through one line x column multiplication to verify that it is indeed doing a weighted sum of all the pixels values of an image. Convolutional layers can be represented as matrix multiplications too although it's a bit more complicated (explanation here, in section 1).

The hardware

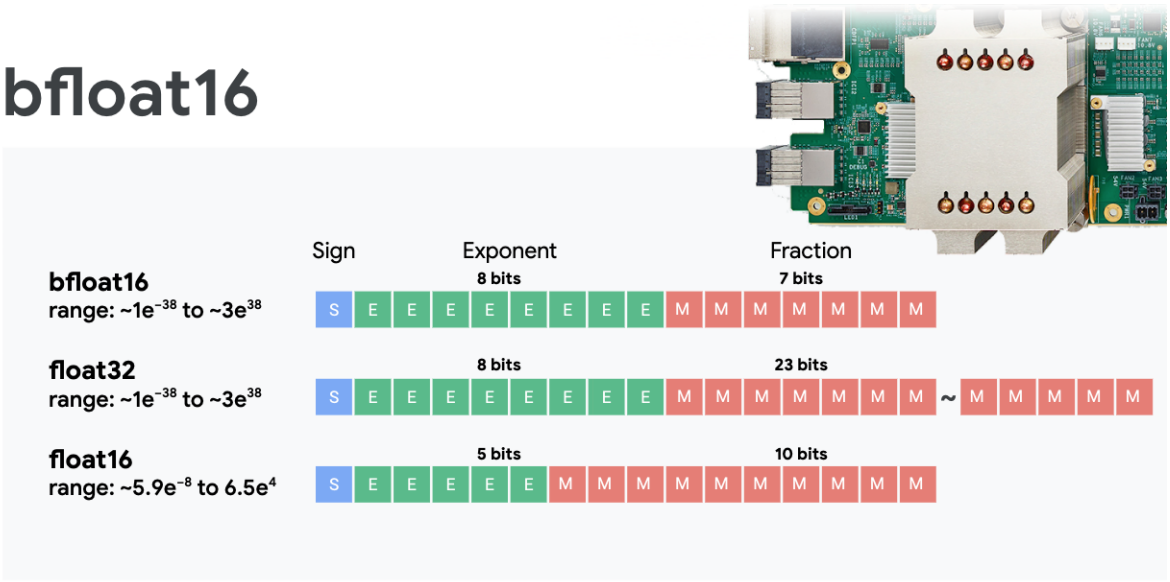
MXU and VPU

A TPU v2 core is made of a Matrix Multiply Unit (MXU) which runs matrix multiplications and a Vector Processing Unit (VPU) for all other tasks such as activations, softmax, etc. The VPU handles float32 and int32 computations. The MXU on the other hand operates in a mixed precision 16-32 bit floating point format.



Mixed precision floating point and bfloat16

The MXU computes matrix multiplications using bfloat16 inputs and float32 outputs. Intermediate accumulations are performed in float32 precision.



Neural network training is typically resistant to the noise introduced by a reduced floating point precision. There are cases where noise even helps the optimizer converge. 16-bit floating point precision has traditionally been used to accelerate computations but float16 and float32 formats have very different ranges. Reducing the precision from float32 to float16 usually results in over and underflows. Solutions exist but additional work is typically required to make float16 work.

That is why Google introduced the bfloat16 format in TPUs. bfloat16 is a truncated float32 with exactly the same exponent bits and range as float32. This, added to the fact that TPUs compute matrix multiplications in mixed precision with bfloat16 inputs but float32 outputs, means that, typically, no code changes are necessary to benefit from the performance gains of reduced precision.

Systolic array

The MXU implements matrix multiplications in hardware using a so-called "systolic array" architecture in which data elements flow through an array of hardware computation units. (In medicine, "systolic" refers to heart contractions and blood flow, here to the flow of data.)

The basic element of a matrix multiplication is a dot product between a line from one matrix and a column from the other matrix (see illustration at the top of this section). For a matrix multiplication Y=X*W, one element of the result would be:

```
Y[2,0] = X[2,0]*W[0,0] + X[2,1]*W[1,0] + X[2,2]*W[2,0] + ... + X[2,n]*W[n,0]
```

On a GPU, one would program this dot product into a GPU "core" and then execute it on as many "cores" as are available in parallel to try and compute every value of the resulting matrix at once. If the resulting matrix is 128x128 large, that would require 128x128=16K "cores" to be available which is typically not possible. The largest GPUs have around 4000 cores. A TPU on the other hand uses the bare minimum of hardware for the compute units in the MXU: just `bfloat16 x bfloat16 => float32` multiply-accumulators, nothing else. These are so small that a TPU can implement 16K of them in a 128x128 MXU and process this matrix multiplication in one go.

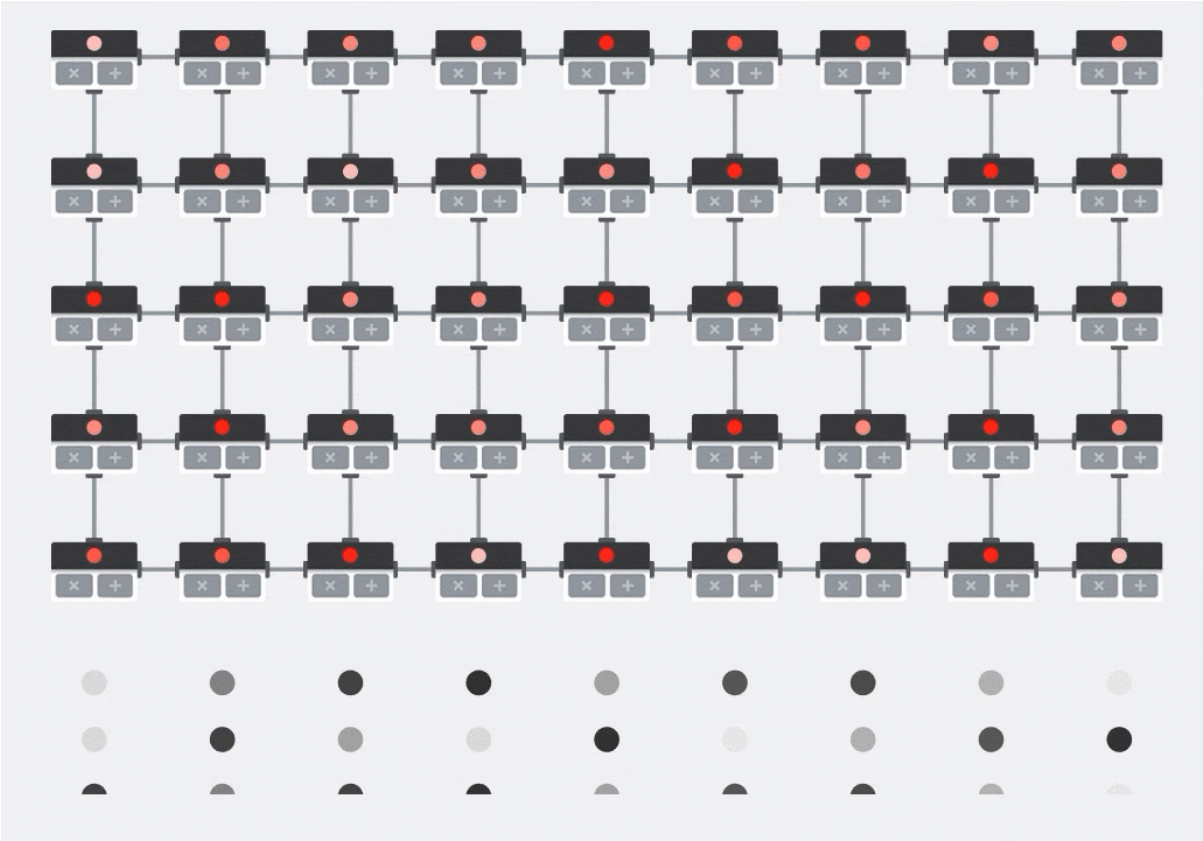


Illustration: the MXU systolic array. The compute elements are multiply-accumulators. The values of one matrix are loaded into the array (red dots). Values of the other matrix flow through the array (grey dots). Vertical lines propagate the values up. Horizontal lines propagate partial sums. It is left as an exercise to the user to verify that as the data flows through the array, you get the result of the matrix multiplication coming out of the right side.

In addition to that, while the dot products are being computed in an MXU, intermediate sums simply flow between adjacent compute units. They do not need to be stored and retrieved to/from memory or even a register file. The end result is that the TPU systolic array architecture has a significant density and power advantage, as well as a non-negligible speed advantage over a GPU, when computing matrix multiplications.

Cloud TPU

When you request one "Cloud TPU v2" on Google Cloud Platform, you get a virtual machine (VM) which has a PCI-attached TPU board. The TPU board has four dual-core TPU chips. Each TPU core features a VPU (Vector Processing Unit) and a 128x128 MXU (MatriX multiply Unit). This "Cloud TPU" is then usually connected through the network to the VM that requested it. So the full picture looks like this:

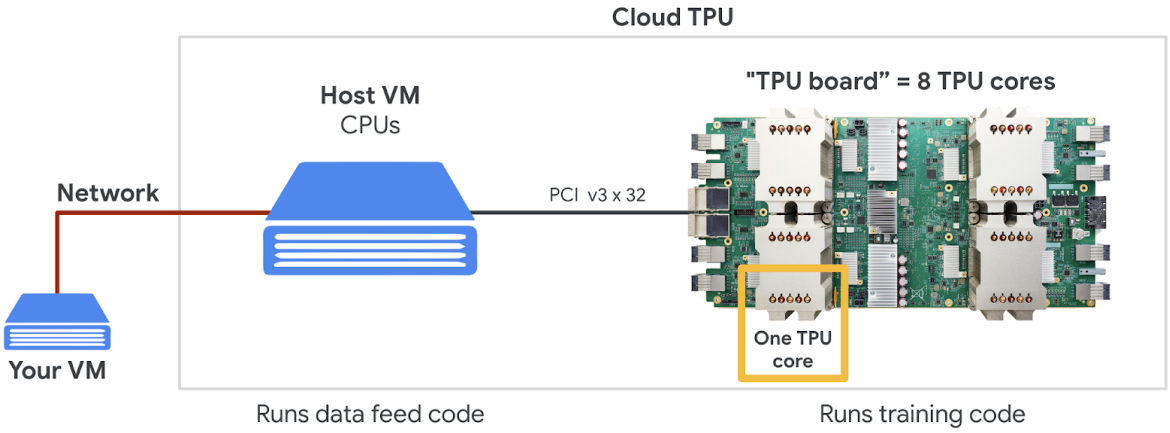


Illustration: your VM with a network-attached "Cloud TPU" accelerator. "The Cloud TPU" itself is made of a VM with a PCI-attached TPU board with four dual-core TPU chips on it.

TPU pods

In Google's data centers, TPUs are connected to a high-performance computing (HPC) interconnect which can make them appear as one very large accelerator. Google calls them pods and they can encompass up to 512 TPU v2 cores or 2048 TPU v3 cores..

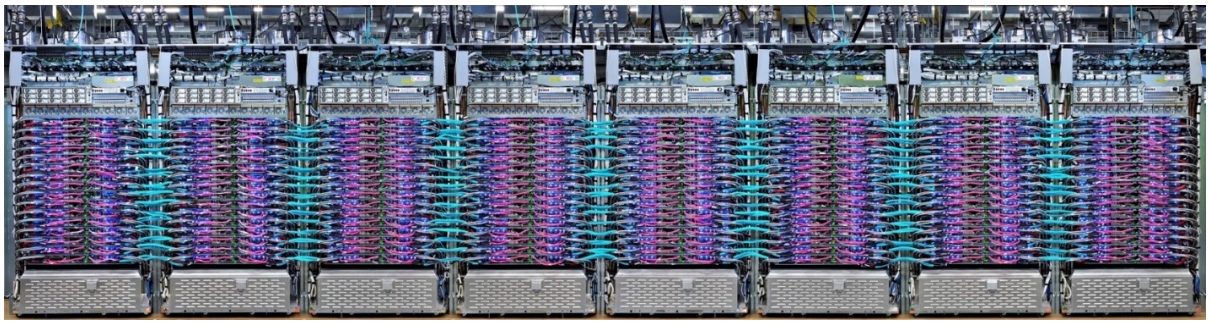


Illustration: a TPU v3 pod. TPU boards and racks connected through HPC interconnect.

During training, gradients are exchanged between TPU cores using the all-reduce algorithm (good explanation of all-reduce here). The model being trained can take advantage of the hardware by training on large batch sizes.

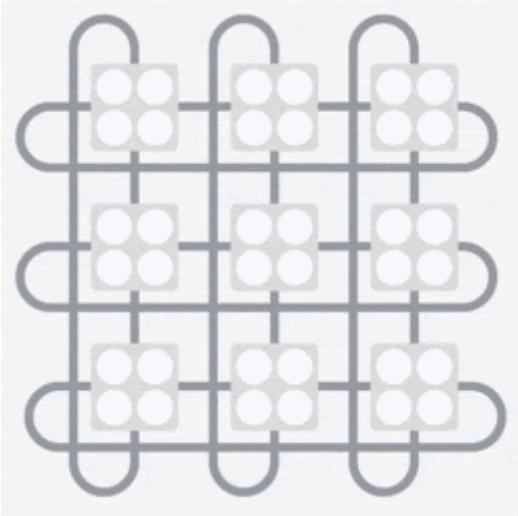


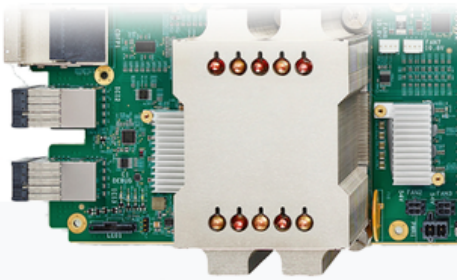
Illustration: synchronization of gradients during training using the all-reduce algorithm on Google TPU's 2-D toroidal mesh HPC network.

The software

Large batch size training

The ideal batch size for TPUs is 128 data items per TPU core but the hardware can already show good utilization from 8 data items per TPU core. Remember that one Cloud TPU has 8 cores.

In this code lab, we will be using the Keras API. In Keras, the batch you specify is the global batch size for the entire TPU. Your batches will automatically be split in 8 and ran on the 8 cores of the TPU.



TPU batch size

ideal 128 per TPU core

interesting from 8 per core

DONE!

scales from

8 to 2048 cores

For additional performance tips see the TPU Performance Guide. For very large batch sizes, special care might be needed in some models, see LARSOptimizer for more details.

Under the hood: XLA

Tensorflow programs define computation graphs. The TPU does not directly run Python code, it runs the computation graph defined by your Tensorflow program. Under the hood, a compiler called XLA (accelerated Linear Algebra compiler) transforms the Tensorflow graph of computation nodes into TPU machine code. This compiler also performs many advanced optimizations on your code and your memory layout. The compilation happens automatically as work is sent to the TPU. You do not have to include XLA in your build chain explicitly.

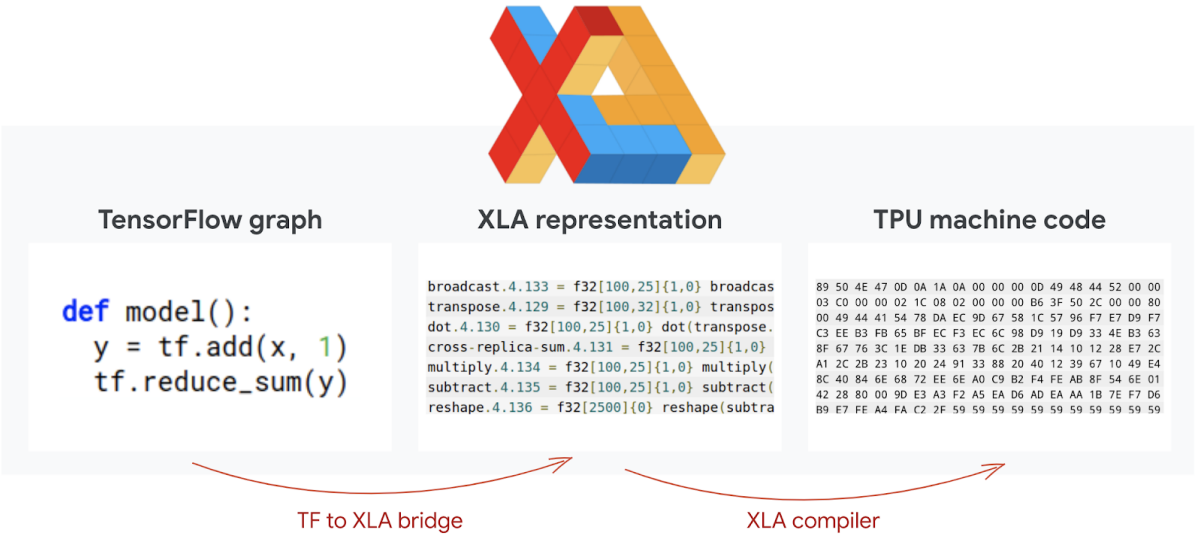


Illustration: to run on TPU, the computation graph defined by your Tensorflow program is first translated to an XLA (accelerated Linear Algebra compiler) representation, then compiled by XLA into TPU

machine code.

Using TPUs in Keras

TPUs are supported through the Keras API as of Tensorflow 2.1. Keras support works on TPUs and TPU pods. Here is an example that works on TPU, GPU(s) and CPU:

```
# TPU detection
try:
  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
except ValueError:
  tpu = None

# TPUStrategy for distributed training
if tpu:
  tf.config.experimental_connect_to_cluster(tpu)
  tf.tpu.experimental.initialize_tpu_system(tpu)
  strategy = tf.distribute.experimental.TPUStrategy(tpu)
else: # default strategy that works on CPU and single GPU
  strategy = tf.distribute.get_strategy()

# use TPUStrategy scope to define model
with strategy.scope():
  model = tf.keras.Sequential( ... )
  model. compile( ... )

# train model normally on a tf.data.Dataset
model.fit(training_dataset, epochs=EPOCHS, steps_per_epoch=...)
```

In this code snippet:

- `TPUClusterResolver()` finds the TPU on the network. It works without parameters on most Google Cloud systems (AI Platform jobs, Colaboratory, Kubeflow, Deep Learning VMs created through the 'ctpu up' utility). These systems know where their TPU is thanks to a TPU_NAME environment variable. If you create a TPU by hand, eithert set the TPU_NAME env. var. on the VM you are using it from, or call `TPUClusterResolver` with explicit parameters: `TPUClusterResolver(tp_uname, zone, project)`
- `TPUStrategy` is the part that implements the distribution and the "all-reduce" gradient synchronization algorithm.
- The strategy is applied through a scope. The model must be defined within the strategy scope().
- The `tpu_model.fit` function expects a tf.data.Dataset object for input for TPU training.

Common TPU porting tasks

- While there are many ways to load data in a Tensorflow model, for TPUs, the use of the `tf.data.Dataset` API is required.
- TPUs are very fast and ingesting data often becomes the bottleneck when running on them. There are tools you can use to detect data bottlenecks and other performance tips in the TPU Performance Guide.
- int8 or int16 numbers are treated as int32. The TPU does not have integer hardware operating on less than 32 bits.
- Some Tensorflow operations are not supported. The list is here. The good news is that this limitation only applies to training code i.e. the forward and backward pass through your model. You can still use all Tensorflow operations in your data input pipeline as it will be executed on CPU.
- `tf.py_func` is not supported on TPU.

4. Loading Data

We will be working with a dataset of flower pictures. The goal is to learn to categorize them into 5 flower types. Data loading is performed using the `tf.data.Dataset` API. First, let us get to know the API.

Hands-on

Please open the following notebook, execute the cells (Shift-ENTER) and follow the instructions wherever you see a "WORK REQUIRED" label.

Fun with tf.data.Dataset (playground).ipynb

Additional information

About the "flowers" dataset

The dataset is organised in 5 folders. Each folder contains flowers of one kind. The folders are named sunflowers, daisy, dandelion, tulips and roses. The data is hosted in a public bucket on Google Cloud Storage. Excerpt:

```
gs://flowers-public/sunflowers/5139971615_434ff8ed8b_n.jpg
gs://flowers-public/daisy/8094774544_35465c1c64.jpg
gs://flowers-public/sunflowers/9309473873_9d62b9082e.jpg
gs://flowers-public/dandelion/19551343954_83bb52f310_m.jpg
gs://flowers-public/dandelion/14199664556_188b37e51e.jpg
gs://flowers-public/tulips/4290566894_c7f061583d_m.jpg
gs://flowers-public/roses/3065719996_c16ecd5551.jpg
gs://flowers-public/dandelion/8168031302_6e36f39d87.jpg
gs://flowers-public/sunflowers/9564240106_0577e919da_n.jpg
gs://flowers-public/daisy/14167543177_cd36b54ac6_n.jpg
```

Why tf.data.Dataset?

Keras and Tensorflow accept Datasets in all of their training and evaluation functions. Once you load data in a Dataset, the API offers all the common functionalities that are useful for neural network training

data:

```
dataset = ... # load something (see below)
dataset = dataset.shuffle(1000) # shuffle the dataset with a buffer of
1000
dataset = dataset.cache() # cache the dataset in RAM or on disk
dataset = dataset.repeat() # repeat the dataset indefinitely
dataset = dataset.batch(128) # batch data elements together in batches of
128
AUTO = tf.data.experimental.AUTOTUNE
dataset = dataset.prefetch(AUTO) # prefetch next batch(es) while training
```

You can find performance tips and Dataset best practices in this article. The reference documentation is here.

tf.data.Dataset basics

Data usually comes in multiple files, here images. You can create a dataset of filenames by calling:

```
filenames_dataset = tf.data.Dataset.list_files('gs://flowers-
public/*/*.jpg')
# The parameter is a "glob" pattern that supports the * and ? wildcards.
```

You then "map" a function to each filename which will typically load and decode the file into actual data in memory:

```
def decode_jpeg(filename):
  bits = tf.io.read_file(filename)
  image = tf.image.decode_jpeg(bits)
  return image

image_dataset = filenames_dataset.map(decode_jpeg)
# this is now a dataset of decoded images (uint8 RGB format)
```

To iterate on a Dataset:

```
for data in my_dataset:
  print(data)
```

Datasets of tuples

In supervised learning, a training dataset is typically made of pairs of training data and correct answers. To allow this, the decoding function can return tuples. You will then have a dataset of tuples and tuples will be returned when you iterate on it. The values returned are Tensorflow tensors ready to be consumed by your model. You can call `.numpy()` on them to see raw values:

```
def decode_jpeg_and_label(filename):
  bits = tf.read_file(filename)
  image = tf.image.decode_jpeg(bits)
  label = ... # extract flower name from folder name
  return image, label

image_dataset = filenames_dataset.map(decode_jpeg)
# this is now a dataset of (image, label) pairs

for image, label in dataset:
  print(image.numpy().shape, label.numpy())
```

Conclusion:loading images one by one is slow !

As you iterate on this dataset, you will see that you can load something like 1-2 images per second. That is too slow! The hardware accelerators we will be using for training can sustain many times this

rate. Head to the next section to see how we will achieve this.

Solution

Here is the solution notebook. You can use it if you are stuck.

 **Fun with tf.data.Dataset (solution).ipynb**

What we've covered

- 🤔 tf.data.Dataset.list_files
- 🤔 tf.data.Dataset.map
- 🤔 Datasets of tuples
- 😀 iterating through Datasets

Please take a moment to go through this checklist in your head.

5. Loading data fast

The Tensor Processing Unit (TPU) hardware accelerators we will be using in this lab are very fast. The challenge is often to feed them data fast enough to keep them busy. Google Cloud Storage (GCS) is capable of sustaining very high throughput but as with all cloud storage systems, initiating a connection costs some network back and forth. Therefore, having our data stored as thousands of individual files is not ideal. We are going to batch them in a smaller number of files and use the power of tf.data.Dataset to read from multiple files in parallel.

Read-through

The code that loads image files, resizes them to a common size and then stores them across 16 TFRecord files is in the following notebook. Please quickly read through it. Executing it is not necessary since properly TFRecord-formatted data will be provided for the rest of the codelab.

 **Flower pictures to TFRecords.ipynb**

Ideal data layout for optimal GCS throughput

The TFRecord file format

Tensorflow's preferred file format for storing data is the protobuf-based TFRecord format. Other serialization formats would work too but you can load a dataset from TFRecord files directly by writing:

```
filenames = tf.io.gfile.glob(FILENAME_PATTERN)
dataset = tf.data.TFRecordDataset(filenames)
dataset = dataset.map(...) # do the TFRecord decoding here - see below
```

For optimal performance, it is recommended to use the following more complex code to read from multiple TFRecord files at once. This code will read from N files in parallel and disregard data order in favor of reading speed.

```
AUTO = tf.data.experimental.AUTOTUNE
ignore_order = tf.data.Options()
ignore_order.experimental_deterministic = False
```

```
filenames = tf.io.gfile.glob(FILENAME_PATTERN)
dataset = tf.data.TFRecordDataset(filenames, num_parallel_reads=AUTO)
dataset = dataset.with_options(ignore_order)
dataset = dataset.map(...) # do the TFRecord decoding here - see below
```

TFRecord cheat sheet

Three types of data can be stored in TFRecords: **byte strings** (list of bytes), 64 bit **integers** and 32 bit **floats**. They are always stored as lists, a single data element will be a list of size 1. You can use the following helper functions to store data into TFRecords.

writing byte strings

```
# warning, the input is a list of byte strings, which are themselves lists
of bytes
def _bytestring_feature(list_of_bytestrings):
  return
tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))
```

writing integers

```
def _int_feature(list_of_ints): # int64
  return
tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))
```

writing floats

```
def _float_feature(list_of_floats): # float32
  return
tf.train.Feature(float_list=tf.train.FloatList(value=list_of_floats))
```

writing a TFRecord, using the helpers above

```
# input data in my_img_bytes, my_class, my_height, my_width, my_floats
with tf.python_io.TFRecordWriter(filename) as out_file:
  feature = {
    "image": _bytestring_feature([my_img_bytes]), # one image in the list
    "class": _int_feature([my_class]),            # one class in the list
    "size": _int_feature([my_height, my_width]),  # fixed length (2) list
of ints
    "float_data": _float_feature(my_floats)       # variable length  list
of floats
  }
  tf_record =
tf.train.Example(features=tf.train.Features(feature=feature))
  out_file.write(tf_record.SerializeToString())
```

To read data from TFRecords, you must first declare the layout of the records you have stored. In the declaration, you can access any named field as a fixed length list or a variable length list:

reading from TFRecords

```
def read_tfrecord(data):
  features = {
    # tf.string = byte string (not text string)
    "image": tf.io.FixedLenFeature([], tf.string), # shape [] means
scalar, here, a single byte string
    "class": tf.io.FixedLenFeature([], tf.int64),  # shape [] means
scalar, i.e. a single item
    "size": tf.io.FixedLenFeature([2], tf.int64),  # two integers
    "float_data": tf.io.VarLenFeature(tf.float32)  # a variable number of
floats
```

```
    }

    # decode the TFRecord
    tf_record = tf.parse_single_example(data, features)

    # FixedLenFeature fields are now ready to use
    sz = tf_record['size']

    # Typical code for decoding compressed images
    image = tf.image.decode_jpeg(tf_record['image'], channels=3)

    # VarLenFeature fields require additional sparse.to_dense decoding
    float_data = tf.sparse.to_dense(tf_record['float_data'])

    return image, sz, float_data

# decoding a tf.data.TFRecordDataset
dataset = dataset.map(read_tfrecord)
# now a dataset of triplets (image, sz, float_data)
```

Useful code snippets:

reading single data elements

```
tf.io.FixedLenFeature([], tf.string)   # for one byte string
tf.io.FixedLenFeature([], tf.int64)    # for one int
tf.io.FixedLenFeature([], tf.float32)  # for one float
```

reading fixed size lists of elements

```
tf.io.FixedLenFeature([N], tf.string)   # list of N byte strings
tf.io.FixedLenFeature([N], tf.int64)    # list of N ints
tf.io.FixedLenFeature([N], tf.float32)  # list of N floats
```

reading a variable number of data items

```
tf.io.VarLenFeature(tf.string)   # list of byte strings
tf.io.VarLenFeature(tf.int64)    # list of ints
tf.io.VarLenFeature(tf.float32)  # list of floats
```

A VarLenFeature returns a sparse vector and an additional step is required after decoding the TFRecord:

```
dense_data = tf.sparse.to_dense(tf_record['my_var_len_feature'])
```

It is also possible to have optional fields in TFRecords. If you specify a default value when reading a field, then the default value is returned instead of an error if the field is missing.

```
tf.io.FixedLenFeature([], tf.int64, default_value=0) # this field is
optional
```

What we've covered

- 🤔 sharding data files for fast access from GCS
- 😓 how to write TFRecords. (You forgot the syntax already? That's OK, bookmark this page as a cheat sheet)
- 🤔 loading a Dataset from TFRecords using TFRecordDataset

Please take a moment to go through this checklist in your head.

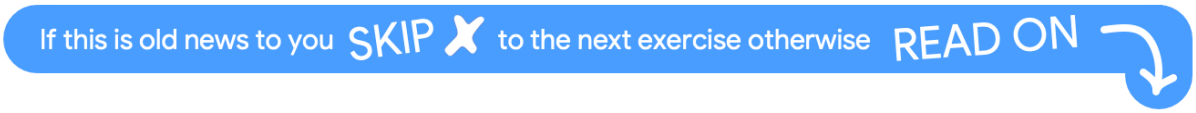
6. [INFO] Neural network classifier 101

In a nutshell

If all the terms in **bold** in the next paragraph are already known to you, you can move to the next exercise. If your are just starting in deep learning then welcome, and please read on.

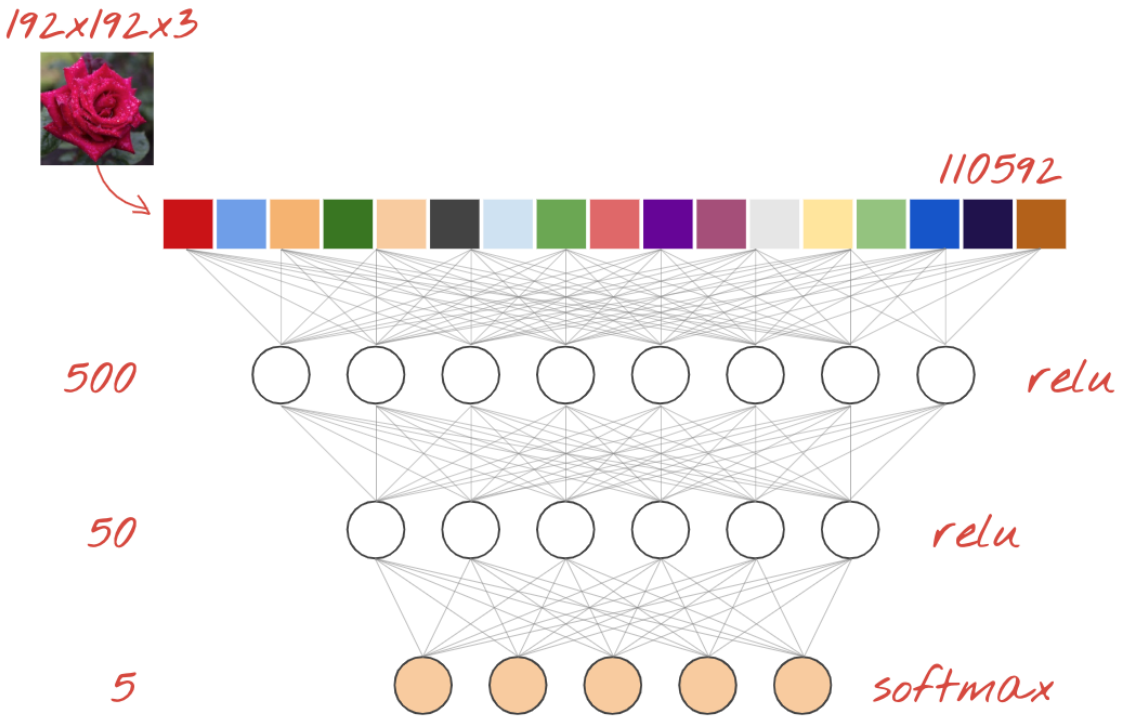
For models built as a sequence of layers Keras offers the Sequential API. For example, an image classifier using three dense layers can be written in Keras as:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[192, 192, 3]),
    tf.keras.layers.Dense(500, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(5, activation='softmax') # classifying into 5
classes
])

# this configures the training of the model. Keras calls it "compiling"
the model.
model.compile(
  optimizer='adam',
  loss= 'categorical_crossentropy',
  metrics=['accuracy']) # % of correct answers

# train the model
model.fit(dataset, ... )
```



Dense neural network

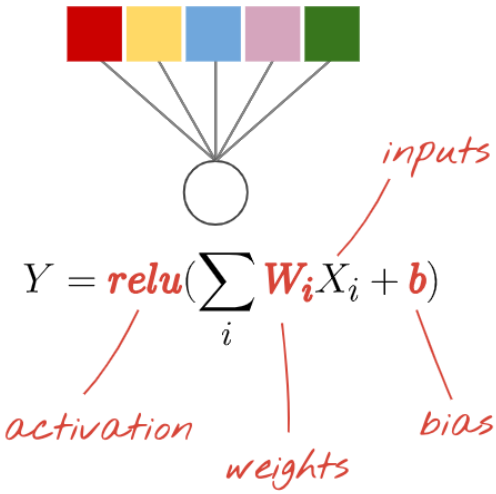
This is the simplest neural network for classifying images. It is made of "neurons" arranged in layers. The first layer processes input data and feeds its outputs into other layers. It is called "dense" because each neuron is connected to all the neurons in the previous layer.

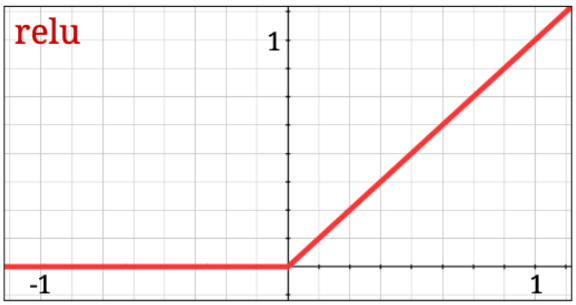


You can feed an image into such a network by flattening the RGB values of all of its pixels into a long vector and using it as inputs. It is not the best technique for image recognition but we will improve on it later.

Neurons, activations, RELU

A "neuron" computes a weighted sum of all of its inputs, adds a value called "bias" and feeds the result through a so called "activation function". The weights and bias are unknown at first. They will be initialized at random and "learned" by training the neural network on lots of known data.





The most popular activation function is called RELU for Rectified Linear Unit. It is a very simple function as you can see on the graph above.

Softmax activation

The network above ends with a 5-neuron layer because we are classifying flowers into 5 categories (rose, tulip, dandelion, daisy, sunflower). Neurons in intermediate layers are activated using the classic RELU activation function. In the last layer though, we want to compute numbers between 0 and 1

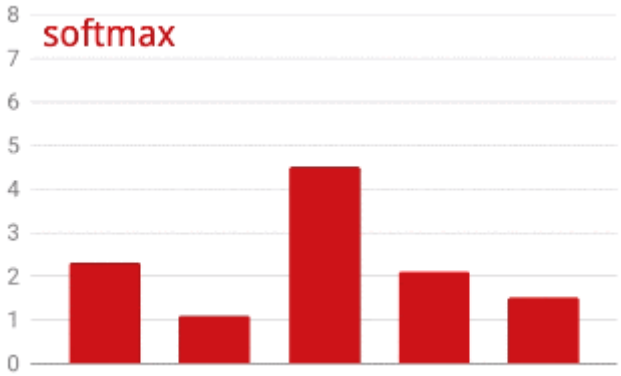
representing the probability of this flower being a rose, a tulip and so on. For this, we will use an activation function called "softmax".

Applying softmax on a vector is done by taking the exponential of each element and then normalising the vector, typically using the L1 norm (sum of absolute values) so that the values add up to 1 and can be interpreted as probabilities.

$$softmax(L_n) = \frac{e^{L_n}}{||e^{L}||}$$

weighted sum+bias

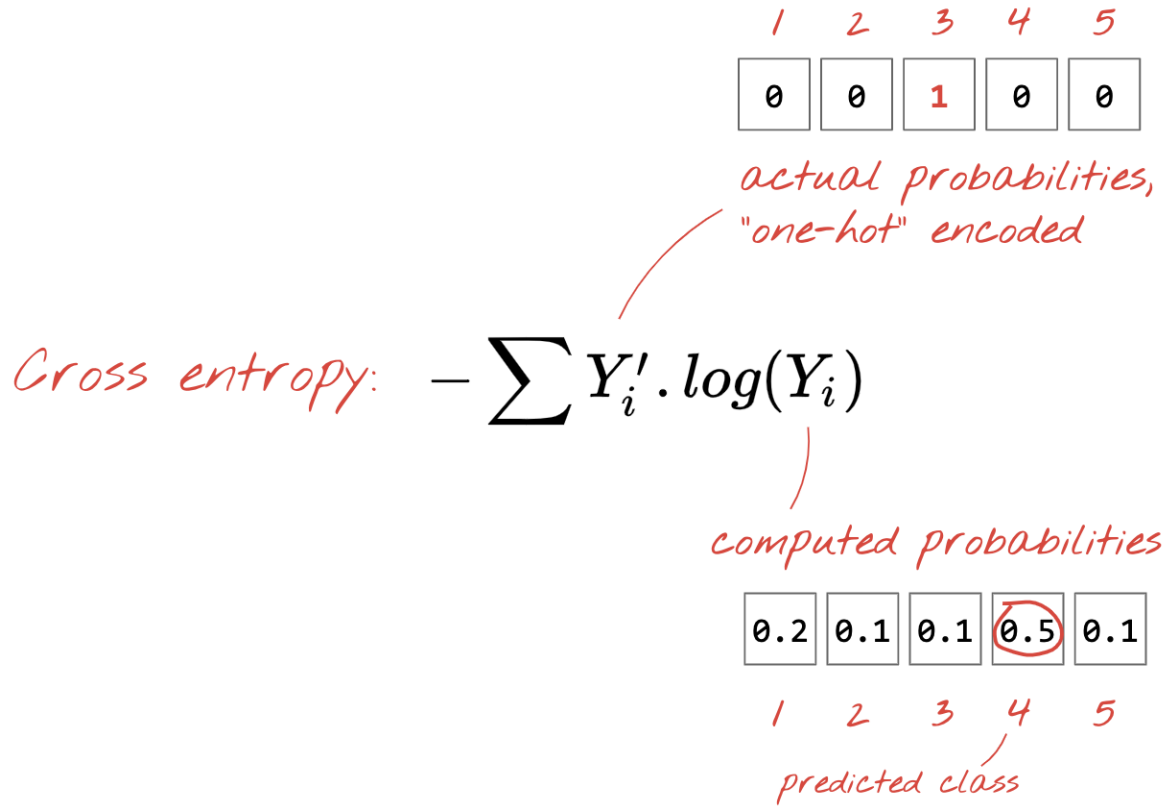
L1 norm



Cross-entropy loss

Now that our neural network produces predictions from input images, we need to measure how good they are, i.e. the distance between what the network tells us and the correct answers, often called "labels". Remember that we have correct labels for all the images in the dataset.

Any distance would work, but for classification problems the so-called "cross-entropy distance" is the most effective. We will call this our error or "loss" function:

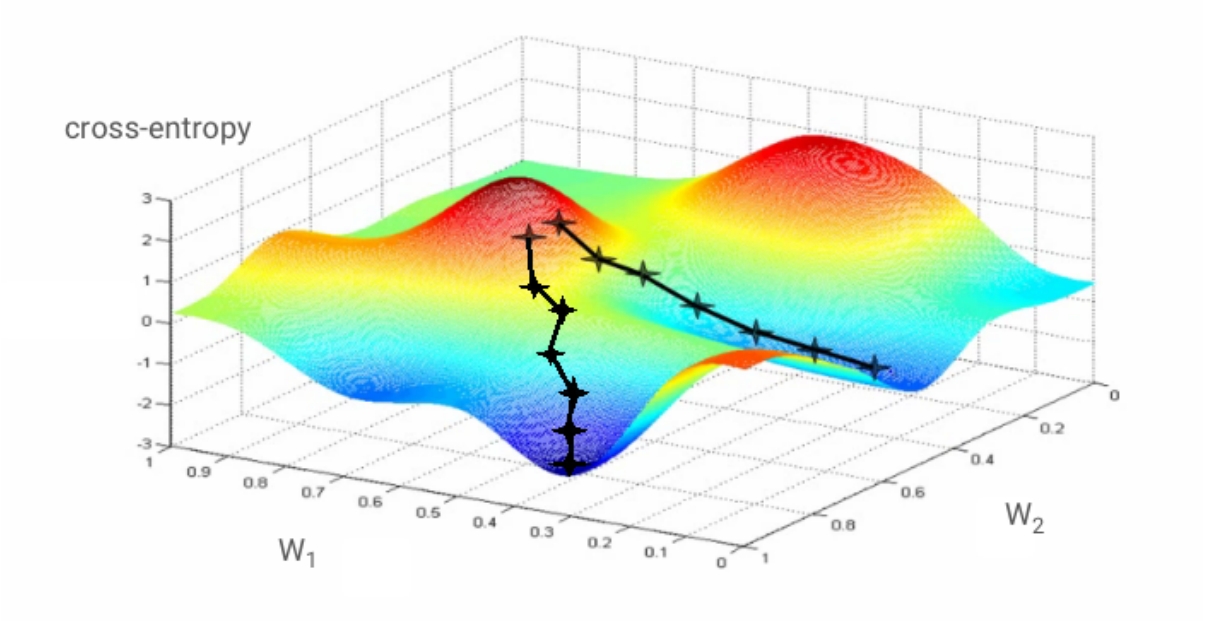


Gradient descent

"Training" the neural network actually means using training images and labels to adjust weights and biases so as to minimise the cross-entropy loss function. Here is how it works.

The cross-entropy is a function of weights, biases, pixels of the training image and its known class.

If we compute the partial derivatives of the cross-entropy relatively to all the weights and all the biases we obtain a "gradient", computed for a given image, label, and present value of weights and biases. Remember that we can have millions of weights and biases so computing the gradient sounds like a lot of work. Fortunately, Tensorflow does it for us. The mathematical property of a gradient is that it points "up". Since we want to go where the cross-entropy is low, we go in the opposite direction. We update weights and biases by a fraction of the gradient. We then do the same thing again and again using the next batches of training images and labels, in a training loop. Hopefully, this converges to a place where the cross-entropy is minimal although nothing guarantees that this minimum is unique.



Mini-batching and momentum

You can compute your gradient on just one example image and update the weights and biases immediately, but doing so on a batch of, for example, 128 images gives a gradient that better represents the constraints imposed by different example images and is therefore likely to converge towards the solution faster. The size of the mini-batch is an adjustable parameter.

This technique, sometimes called "stochastic gradient descent" has another, more pragmatic benefit: working with batches also means working with larger matrices and these are usually easier to optimise on GPUs and TPUs.

The convergence can still be a little chaotic though and it can even stop if the gradient vector is all zeros. Does that mean that we have found a minimum? Not always. A gradient component can be zero on a minimum or a maximum. With a gradient vector with millions of elements, if they are all zeros, the probability that every zero corresponds to a minimum and none of them to a maximum point is pretty small. In a space of many dimensions, saddle points are pretty common and we do not want to stop at them.

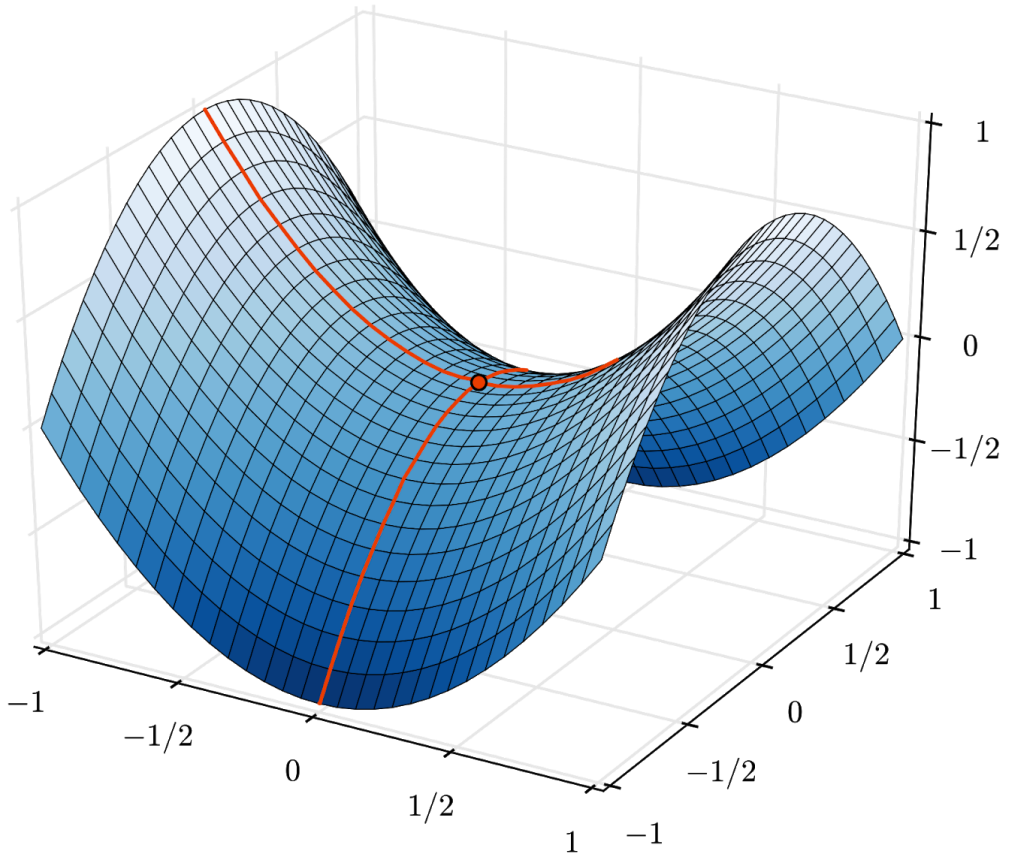


Illustration: a saddle point. The gradient is 0 but it is not a minimum in all directions. (Image attribution Wikimedia: By Nicoguaro - Own work, CC BY 3.0)

The solution is to add some momentum to the optimization algorithm so that it can sail past saddle points without stopping.

Glossary

batch or **mini-batch**: training is always performed on batches of training data and labels. Doing so helps the algorithm converge. The "batch" dimension is typically the first dimension of data tensors. For example a tensor of shape [100, 192, 192, 3] contains 100 images of 192x192 pixels with three values per pixel (RGB).

cross-entropy loss: a special loss function often used in classifiers.

dense layer: a layer of neurons where each neuron is connected to all the neurons in the previous layer.

features: the inputs of a neural network are sometimes called "features". The art of figuring out which parts of a dataset (or combinations of parts) to feed into a neural network to get good predictions is called "feature engineering".

labels: another name for "classes" or correct answers in a supervised classification problem

learning rate: fraction of the gradient by which weights and biases are updated at each iteration of the training loop.

logits: the outputs of a layer of neurons before the activation function is applied are called "logits". The term comes from the "logistic function" a.k.a. the "sigmoid function" which used to be the most popular activation function. "Neuron outputs before logistic function" was shortened to "logits".

loss: the error function comparing neural network outputs to the correct answers

neuron: computes the weighted sum of its inputs, adds a bias and feeds the result through an activation function.

one-hot encoding: class 3 out of 5 is encoded as a vector of 5 elements, all zeros except the 3rd one which is 1.

relu: rectified linear unit. A popular activation function for neurons.

sigmoid: another activation function that used to be popular and is still useful in special cases.

softmax: a special activation function that acts on a vector, increases the difference between the largest component and all others, and also normalizes the vector to have a sum of 1 so that it can be interpreted as a vector of probabilities. Used as the last step in classifiers.

tensor: A "tensor" is like a matrix but with an arbitrary number of dimensions. A 1-dimensional tensor is a vector. A 2-dimensions tensor is a matrix. And then you can have tensors with 3, 4, 5 or more dimensions.

7. Transfer Learning

For an image classification problem, dense layers will probably not be enough. We have to learn about convolutional layers and the many ways you can arrange them.

But we can also take a shortcut! There are fully-trained convolutional neural networks available for download. It is possible to chop off their last layer, the softmax classification head, and replace it with your own. All the trained weights and biases stay as they are, you only retrain the softmax layer you add. This technique is called transfer learning and amazingly, it works as long as the dataset on which the neural net is pre-trained is "close enough" to yours.

Hands-on

Please open the following notebook, execute the cells (Shift-ENTER) and follow the instructions wherever you see a "WORK REQUIRED" label.

Keras Flowers transfer learning (playground).ipynb

Additional information

With transfer learning, you benefit from both advanced convolutional neural network architectures developed by top researchers and from pre-training on a huge dataset of images. In our case we will be transfer learning from a network trained on ImageNet, a database of images containing many plants and outdoors scenes, which is close enough to flowers.

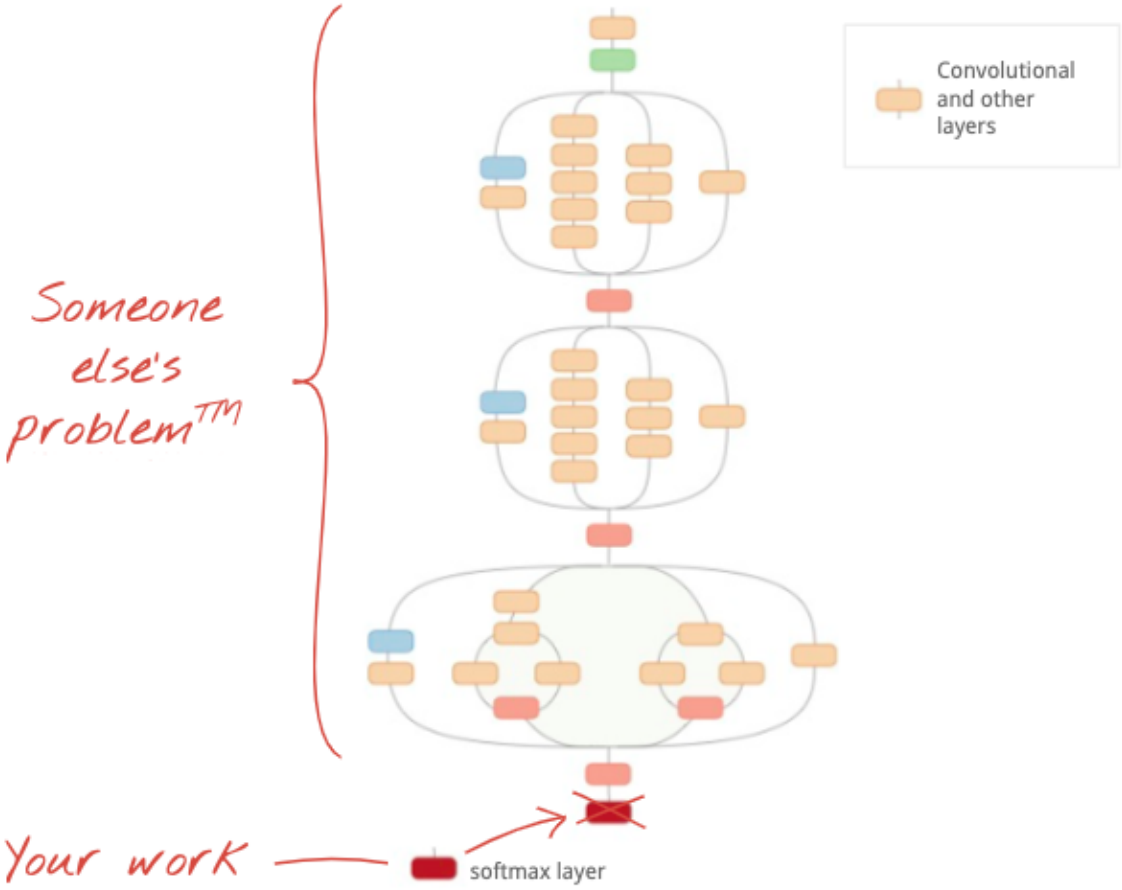


Illustration: using a complex convolutional neural network, already trained, as a black box, retraining the classification head only. This is transfer learning. We will see how these complicated arrangements of convolutional layers work later. For now, it is someone else's problem.

Transfer learning in Keras

In Keras, you can instantiate a pre-trained model from the `tf.keras.applications.*` collection. MobileNet V2 for example is a very good convolutional architecture that stays reasonable in size. By selecting `include_top=False`, you get the pre-trained model without its final softmax layer so that you can add your own:

```
pretrained_model = tf.keras.applications.MobileNetV2(input_shape=
[*IMAGE_SIZE, 3], include_top=False)
pretrained_model.trainable = False

model = tf.keras.Sequential([
    pretrained_model,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(5, activation='softmax')
])
```

Also notice the `pretrained_model.trainable = False` setting. It freezes the weights and biases of the pre-trained model so that you train your softmax layer only. This typically involves relatively few weights and can be done quickly and without necessitating a very large dataset. However if you do have lots of data, transfer learning can work even better with `pretrained_model.trainable = True`. The pre-trained weights then provide excellent initial values and can still be adjusted by the training to better fit your problem.

Finally, notice the `Flatten()` layer inserted before your dense softmax layer. Dense layers work on flat vectors of data but we do not know if that is what the pretrained model returns. That's why we need to

flatten. In the next chapter, as we dive into convolutional architectures, we will explain the data format returned by convolutional layers.

You should get close to 75% accuracy with this approach.

Solution

Here is the solution notebook. You can use it if you are stuck.

 **Keras Flowers transfer learning (solution).ipynb**

What we've covered

- 🤔 How to write a classifier in Keras
- 🤓 configured with a softmax last layer, and cross-entropy loss
- 😈 Transfer learning
- 🤔 Training your first model
- 🧐 Following its loss and accuracy during training

Please take a moment to go through this checklist in your head.

8. [INFO] Convolutional neural networks

In a nutshell

If all the terms in **bold** in the next paragraph are already known to you, you can move to the next exercise. If your are just starting with convolutional neural networks please read on.

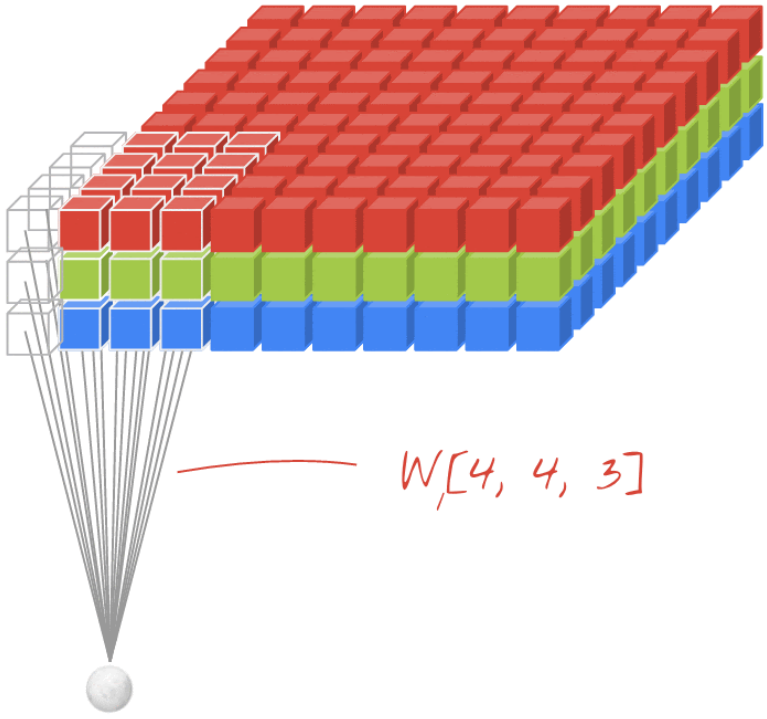


Illustration: filtering an image with two successive filters made of 4x4x3=48 learnable weights each.

This is how a simple convolutional neural network looks in Keras:

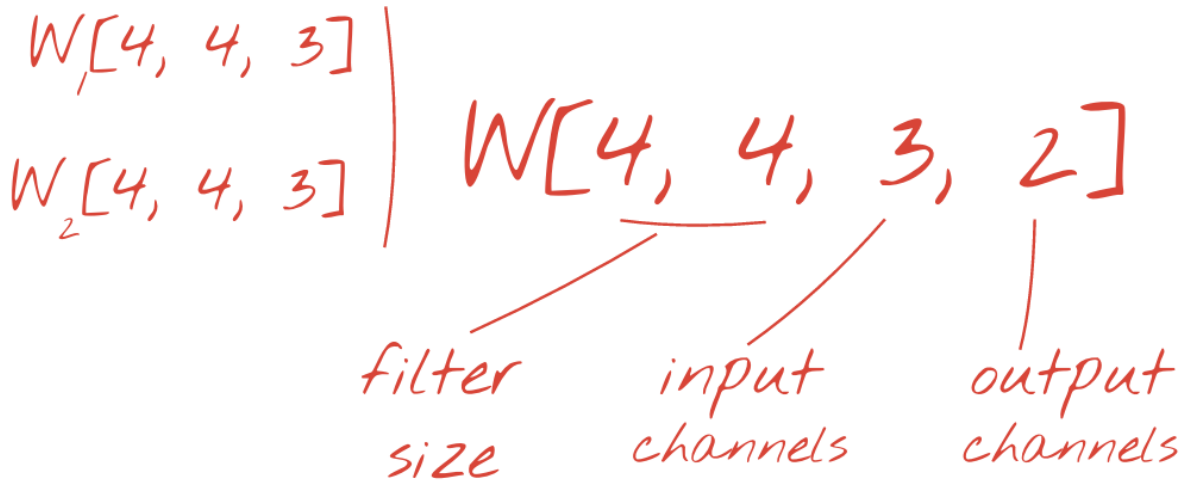
```
model = tf.keras.Sequential([
  # input: images of size 192x192x3 pixels (the three stands for RGB
channels)
  tf.keras.layers.Conv2D(kernel_size=3, filters=24, padding='same',
activation='relu', input_shape=[192, 192, 3]),
  tf.keras.layers.Conv2D(kernel_size=3, filters=24, padding='same',
activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=2),
  tf.keras.layers.Conv2D(kernel_size=3, filters=12, padding='same',
activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=2),
  tf.keras.layers.Conv2D(kernel_size=3, filters=6, padding='same',
activation='relu'),
  tf.keras.layers.Flatten(),
  # classifying into 5 categories
  tf.keras.layers.Dense(5, activation='softmax')
])

model.compile(
  optimizer='adam',
  loss= 'categorical_crossentropy',
  metrics=['accuracy'])
```

If this is old news to you SKIP ✗ to the next exercise otherwise READ ON

Convolutional neural nets 101

In a layer of a convolutional network, one "neuron" does a weighted sum of the pixels just above it, across a small region of the image only. It adds a bias and feeds the sum through an activation function, just as a neuron in a regular dense layer would. This operation is then repeated across the entire image using the same weights. Remember that in dense layers, each neuron had its own weights. Here, a single "patch" of weights slides across the image in both directions (a "convolution"). The output has as many values as there are pixels in the image (some padding is necessary at the edges though). It is a filtering operation, using a filter of 4x4x3=48 weights.

However, 48 weights will not be enough. To add more degrees of freedom, we repeat the same operation with a new set of weights. This produces a new set of filter outputs. Let's call it a "channel" of outputs by analogy with the R,G,B channels in the input image.



The two (or more) sets of weights can be summed up as one tensor by adding a new dimension. This gives us the generic shape of the weights tensor for a convolutional layer. Since the number of input and output channels are parameters, we can start stacking and chaining convolutional layers.

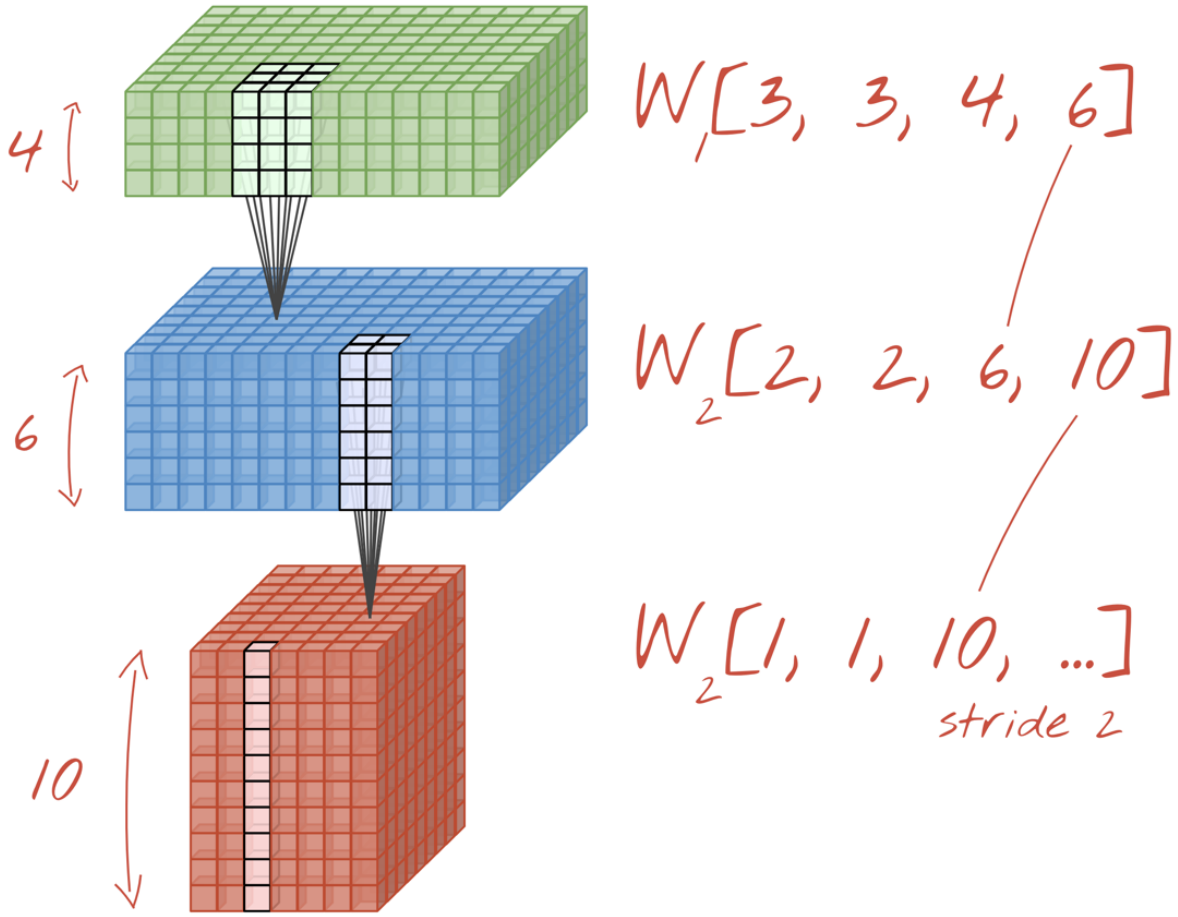


Illustration: a convolutional neural network transforms "cubes" of data into other "cubes" of data.

Strided convolutions, max pooling

By performing the convolutions with a stride of 2 or 3, we can also shrink the resulting data cube in its horizontal dimensions. There are two common ways of doing this:

- Strided convolution: a sliding filter as above but with a stride >1
- Max pooling: a sliding window applying the MAX operation (typically on 2x2 patches, repeated every 2 pixels)

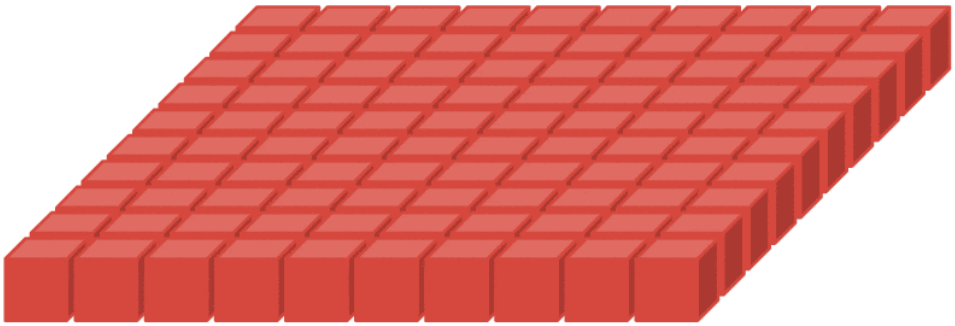


Illustration: sliding the computing window by 3 pixels results in fewer output values. Strided convolutions or max pooling (max on a 2x2 window sliding by a stride of 2) are a way of shrinking the data cube in the horizontal dimensions.

Convolutional classifier

Finally, we attach a classification head by flattening the last data cube and feeding it through a dense, softmax-activated layer. A typical convolutional classifier can look like this:

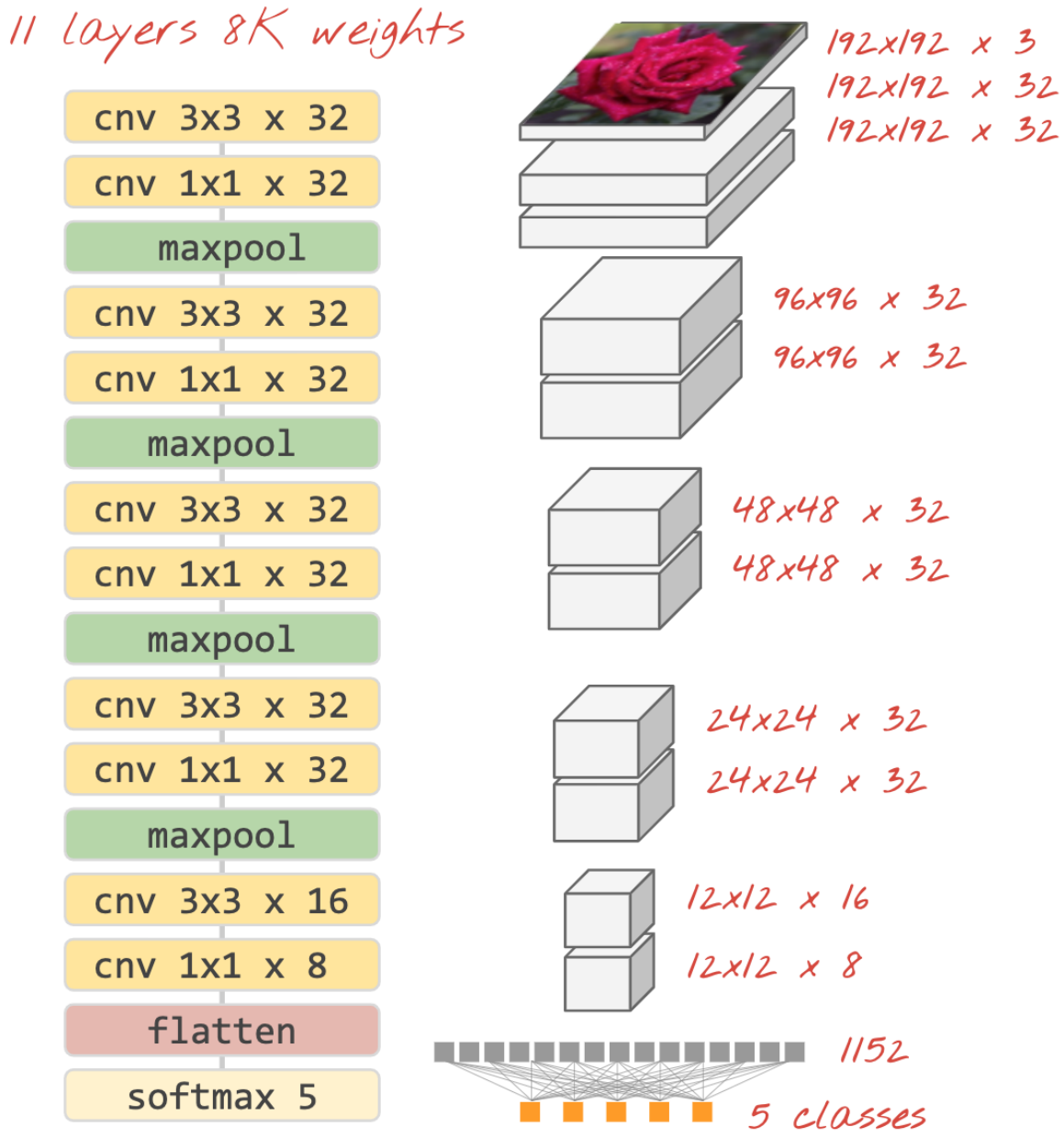


Illustration: an image classifier using convolutional and softmax layers. It uses 3x3 and 1x1 filters. The maxpool layers take the max of groups of 2x2 data points. The classification head is implemented with a dense layer with softmax activation.

In Keras

The convolutional stack illustrated above can be written in Keras like this:

```
model = tf.keras.Sequential([
  # input: images of size 192x192x3 pixels (the three stands for RGB
channels)
  tf.keras.layers.Conv2D(kernel_size=3, filters=32, padding='same',
activation='relu', input_shape=[192, 192, 3]),
  tf.keras.layers.Conv2D(kernel_size=1, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=2),
  tf.keras.layers.Conv2D(kernel_size=3, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.Conv2D(kernel_size=1, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=2),
```

```
  tf.keras.layers.Conv2D(kernel_size=3, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.Conv2D(kernel_size=1, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=2),
  tf.keras.layers.Conv2D(kernel_size=3, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.Conv2D(kernel_size=1, filters=32, padding='same',
activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=2),
  tf.keras.layers.Conv2D(kernel_size=3, filters=16, padding='same',
activation='relu'),
  tf.keras.layers.Conv2D(kernel_size=1, filters=8, padding='same',
activation='relu'),
  tf.keras.layers.Flatten(),
  # classifying into 5 categories
  tf.keras.layers.Dense(5, activation='softmax')
])

model.compile(
  optimizer='adam',
  loss= 'categorical_crossentropy',
  metrics=['accuracy'])
```

9. Your custom convnet

Hands-on

Let us build and train a convolutional neural network from scratch. Using a TPU will allow us to iterate very fast. Please open the following notebook, execute the cells (Shift-ENTER) and follow the instructions wherever you see a "WORK REQUIRED" label.

Keras_Flowers_TPU (playground).ipynb

The goal is to beat the 75% accuracy of the transfer learning model. That model had an advantage, having been pre-trained on a dataset of millions of images while we only have 3670 images here. Can you at least match it?

Additional information

How many layers, how big?

Selecting layer sizes is more of an art than a science. You have to find the right balance between having too few and too many parameters (weights and biases). With too few weights, the neural network cannot represent the complexity of flower shapes. With too many, it can be prone to "overfitting", i.e. specializing in the training images and not being able to generalize. With a lot of parameters, the model will also be slow to train. In Keras, the `model.summary()` function displays the structure and parameter count of your model:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 192, 192, 16)      448
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 192, 192, 30)      4350
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 96, 96, 30)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 96, 96, 60)        16260
_________________________________________________________________
```

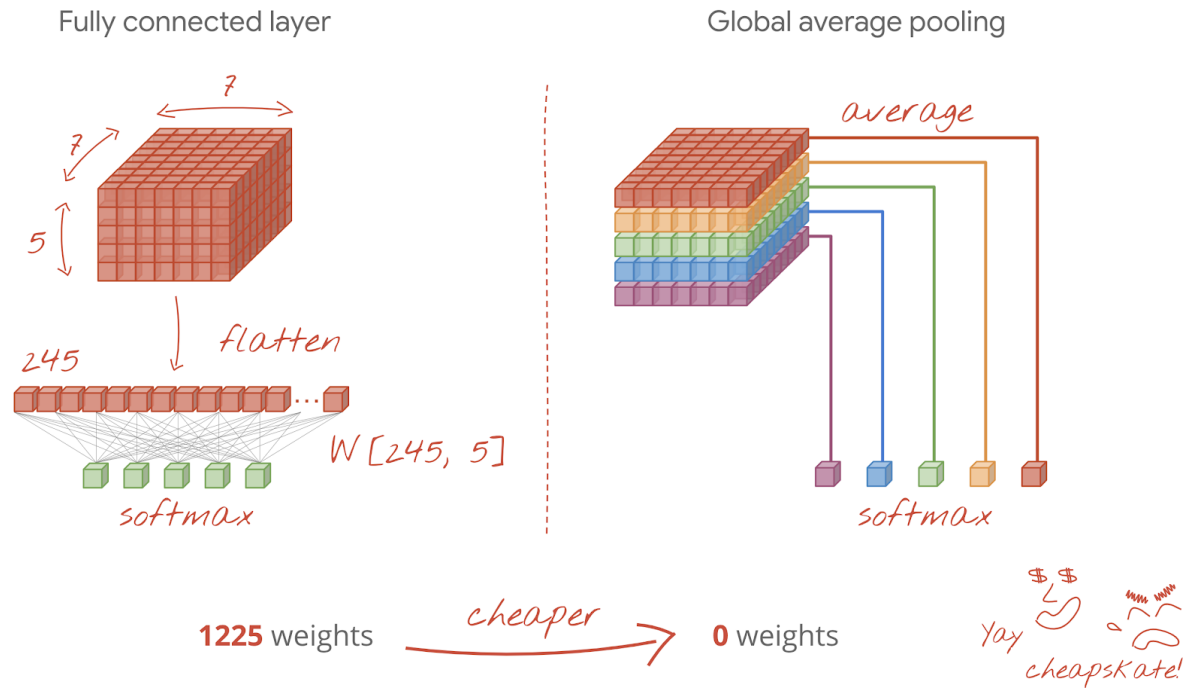
```
...
_________________________________________________________________
global_average_pooling2d (Gl (None, 130)               0
_________________________________________________________________
dense (Dense)                (None, 90)                11790
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 455
=================================================================
Total params: 300,033
Trainable params: 300,033
Non-trainable params: 0
_________________________________________________________________
```

A couple of tips:

- Having multiple layers is what makes "deep" neural networks effective. For this simple flower recognition problem, 5 to 10 layers make sense.
- Use small filters. Typically 3x3 filters are good everywhere.
- 1x1 filters can be used too and are cheap. They do not really "filter" anything but compute linear combinations of channels. Alternate them with real filters. (More about "1x1 convolutions" in the next section.)
- For a classification problem like this, downsample frequently with max-pooling layers (or convolutions with stride >1). You do not care where the flower is, only that it is a rose or a dandelion so losing x and y information is not important and filtering smaller areas is cheaper.
- The number of filters usually becomes similar to the number of classes at the end of the network (why ? see "global average pooling" trick below). If you classify into hundreds of classes, increase the filter count progressively in consecutive layers. For the flower dataset with 5 classes, filtering with only 5 filters would not be enough. You can use the same filter count in most layers, for example 32 and decrease it towards the end.
- The final dense layer(s) is/are expensive. It/they can have more weights than all the convolutional layers combined. For example, even with a very reasonable output from the last data cube of 24x24x10 data points, a 100 neuron dense layer would cost 24x24x10x100=576,000 weights !!! Try to be thoughtful, or try global average pooling (see below)

Global average pooling

Instead of using an expensive dense layer at the end of a convolutional neural network, you can split the incoming data "cube" into as many parts as you have classes, average their values and feed these through a softmax activation function. This way of building the classification head costs 0 weights. In Keras, the syntax is `tf.keras.layers.GlobalAveragePooling2D()`



Solution

Here is the solution notebook. You can use it if you are stuck.

Keras_Flowers_TPU (solution).ipynb

What we've covered

- 🤔 Played with convolutional layers
- 🤓 Experimented with max pooling, strides, global average pooling, ...
- 😀 iterated on a real-world model fast, on TPU

Please take a moment to go through this checklist in your head.

11. Squeezenet

A simple way of putting these ideas together has been showcased in the "Squeezenet" paper. The authors suggest a very simple convolutional module design, using only 1x1 and 3x3 convolutional layers.

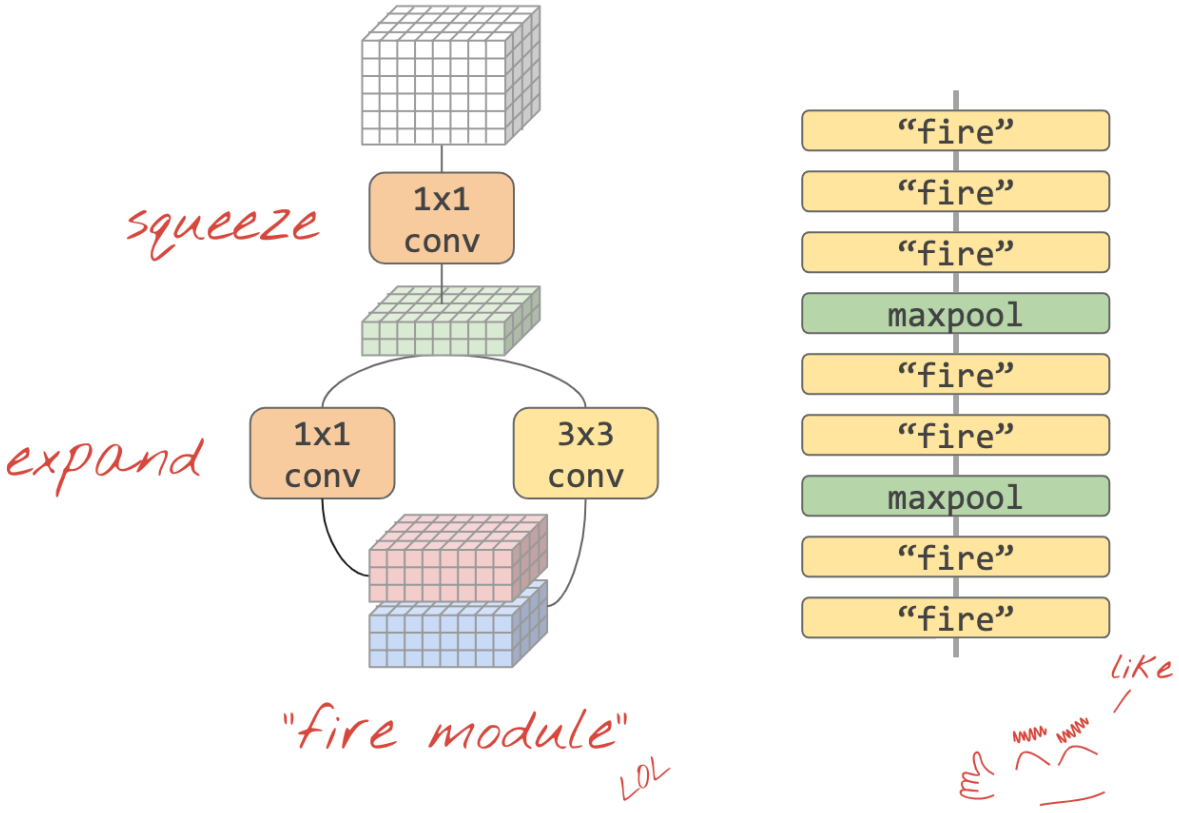


Illustration: squeezenet architecture based on "fire modules". They alternate a 1x1 layer that "squeezes" the incoming data in the vertical dimension followed by two parallel 1x1 and 3x3 convolutional layers that "expand" the depth of the data again.

Hands-on

Continue in your previous notebook and build a squeezenet-inspired convolutional neural network. You will have to change the model code to the Keras "functional style".


Keras_Flowers_TPU (playground).ipynb

Additional info

It will be useful for this exercise to define a helper function for a squeezenet module:

```
def fire(x, squeeze, expand):
  y = l.Conv2D(filters=squeeze, kernel_size=1, padding='same',
activation='relu')(x)
  y1 = l.Conv2D(filters=expand//2, kernel_size=1, padding='same',
activation='relu')(y)
  y3 = l.Conv2D(filters=expand//2, kernel_size=3, padding='same',
activation='relu')(y)
  return tf.keras.layers.concatenate([y1, y3])

# this is to make it behave similarly to other Keras layers
def fire_module(squeeze, expand):
  return lambda x: fire(x, squeeze, expand)

# usage:
x = l.Input(shape=[192, 192, 3])
y = fire_module(squeeze=24, expand=48)(x) # typically, squeeze is less
than expand
y = fire_module(squeeze=32, expand=64)(y)
...
model = tf.keras.Model(x, y)
```

The objective this time is to hit 80% accuracy.

Things to try

Start with a single convolutional layer, then follow with "`fire_modules`", alternating with `MaxPooling2D(pool_size=2)` layers. You can experiment with 2 to 4 max pooling layers in the network and also with 1, 2 or 3 consecutive fire modules between the max pooling layers.

In fire modules, the "squeeze" parameter should typically be smaller than the "expand" parameter. These parameters are actually numbers of filters. They can range from 8 to 196, typically. You can experiment with architectures where the number of filters gradually increases through the network, or straightforward architectures where all fire modules have the same number of filters.

Here is an example:

```
x = tf.keras.layers.Input(shape=[*IMAGE_SIZE, 3]) # input is 192x192
pixels RGB

y = tf.keras.layers.Conv2D(kernel_size=3, filters=32, padding='same',
activation='relu')(x)
y = fire_module(24, 48)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(24, 48)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(24, 48)(y)
y = tf.keras.layers.GlobalAveragePooling2D()(y)
y = tf.keras.layers.Dense(5, activation='softmax')(y)

model = tf.keras.Model(x, y)
```

At this point, you might notice that your experiments are not going so well and that the 80% accuracy objective seems remote. Time for a couple more cheap tricks.

Batch Normalization

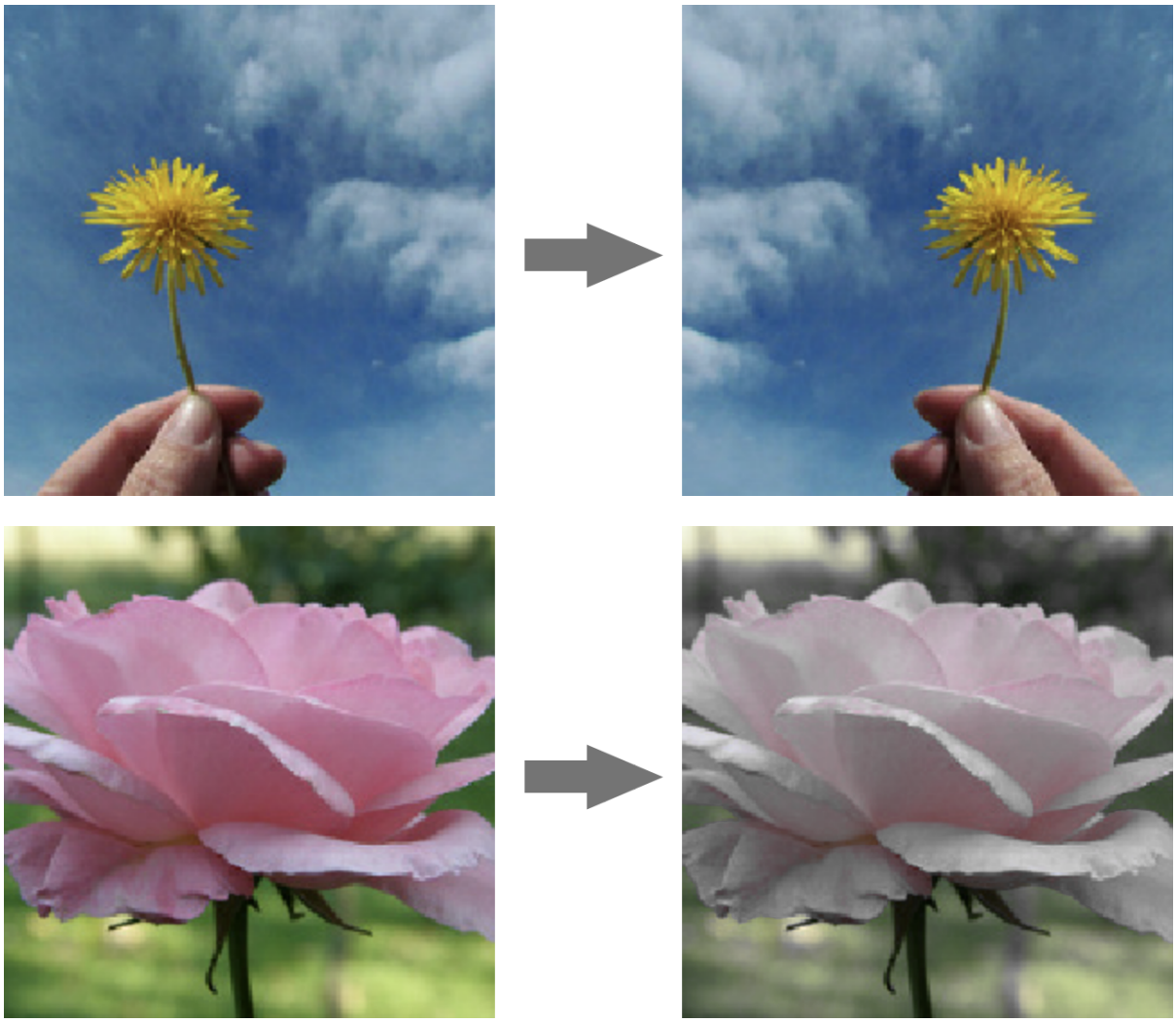
Batch norm will help with the convergence problems you are experiencing. There will be detailed explanations about this technique in the next workshop, for now, please use it as a black box "magic" helper by adding this line **after every convolutional layer** in your network, including the layers inside of your fire_module function:

```
y = tf.keras.layers.BatchNormalization(momentum=0.9)(y)
# please adapt the input and output "y"s to whatever is appropriate in
your context
```

The momentum parameter has to be decreased from its default value of 0.99 to 0.9 because our dataset is small. Never mind this detail for now.

Data augmentation

You will get a couple more percentage points by augmenting the data with easy transformations like left-right flips of saturation changes:



This is very easy to do in Tensorflow with the tf.data.Dataset API. Define a new transformation function for your data:

```
def data_augment(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_saturation(image, lower=0, upper=2)
    return image, label
```

Then use it in you final data transformation (cell "training and validation datasets", function "get_batched_dataset"):

```
dataset = dataset.repeat() # existing line
# insert this
if augment_data:
  dataset = dataset.map(data_augment, num_parallel_calls=AUTO)
dataset = dataset.shuffle(2048) # existing line
```

Do not forget to make the data augmentation optional and to add the necessary code to make sure **only the training dataset is augmented**. It makes no sense to augment the validation dataset.

80% accuracy in 35 epochs should now be within reach.

Solution

Here is the solution notebook. You can use it if you are stuck.

 **Keras_Flowers_TPU_squeezenet.ipynb**

What we've covered

- 🤔 Keras "functional style" models
- 🤓 Squeezenet architecture
- 🤓 Data augmentation with tf.data.datset

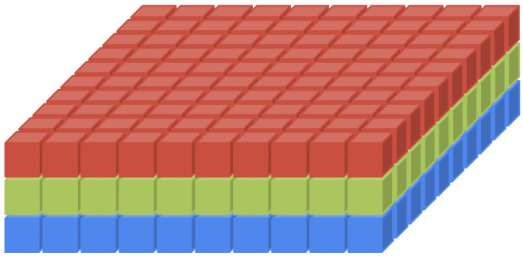
Please take a moment to go through this checklist in your head.

12. Xception fine-tuned

Separable convolutions

A different way of implementing convolutional layers had been gaining popularity recently: depth-separable convolutions. I know, it's a mouthful, but the concept is quite simple. They are implemented in Tensorflow and Keras as `tf.keras.layers.SeparableConv2D`.

A separable convolution also runs a filter on the image but it uses a distinct set of weights for each channel of the input image. It follows with a "1x1 convolution", a series of dot products resulting in a weighted sum of the filtered channels. With new weights each time, as many weighted recombinations of the channels are computed as necessary.



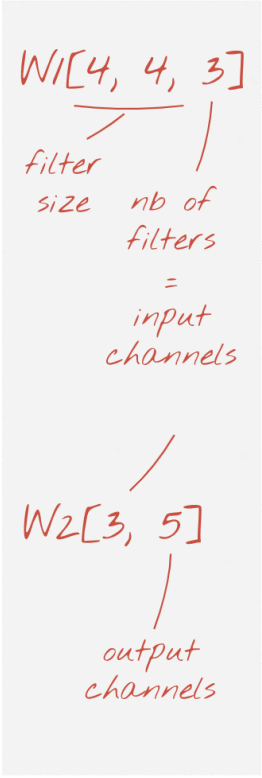


Illustration: separable convolutions. Phase 1: convolutions with a separate filter for each channel. Phase 2: linear recombinations of channels. Repeated with a new set of weights until the desired number of output channels is reached. Phase 1 can be repeated too, with new weights each time but in practice it is rarely so.

Separable convolutions are used in most recent convolutional networks architectures: MobileNetV2, Xception, EfficientNet. By the way, MobileNetV2 is what you used for transfer learning previously.

They are cheaper than regular convolutions and have been found to be just as effective in practice. Here is the weight count for the example illustrated above:

Convolutional layer: 4 x 4 x 3 x 5 = 240

Separable convolutional layer: 4 x 4 x 3 + 3 x 5 = 48 + 15 = 63

It is left as an exercise for the reader to compute than number of multiplications required to apply each style of convolutional layer scales in a similar way. Separable convolutions are smaller and much more computationally effective.

Hands-on

Restart from the "transfer learning" playground notebook but this time select Xception as the pre-trained model. Xception uses separable convolutions only. Leave all weights trainable. We will be fine-tuning the pre-trained weights on our data instead of using the pre-trained layers as such.

Keras Flowers transfer learning (playground).ipynb

Goal: accuracy > 95% (No, seriously, it is possible!)

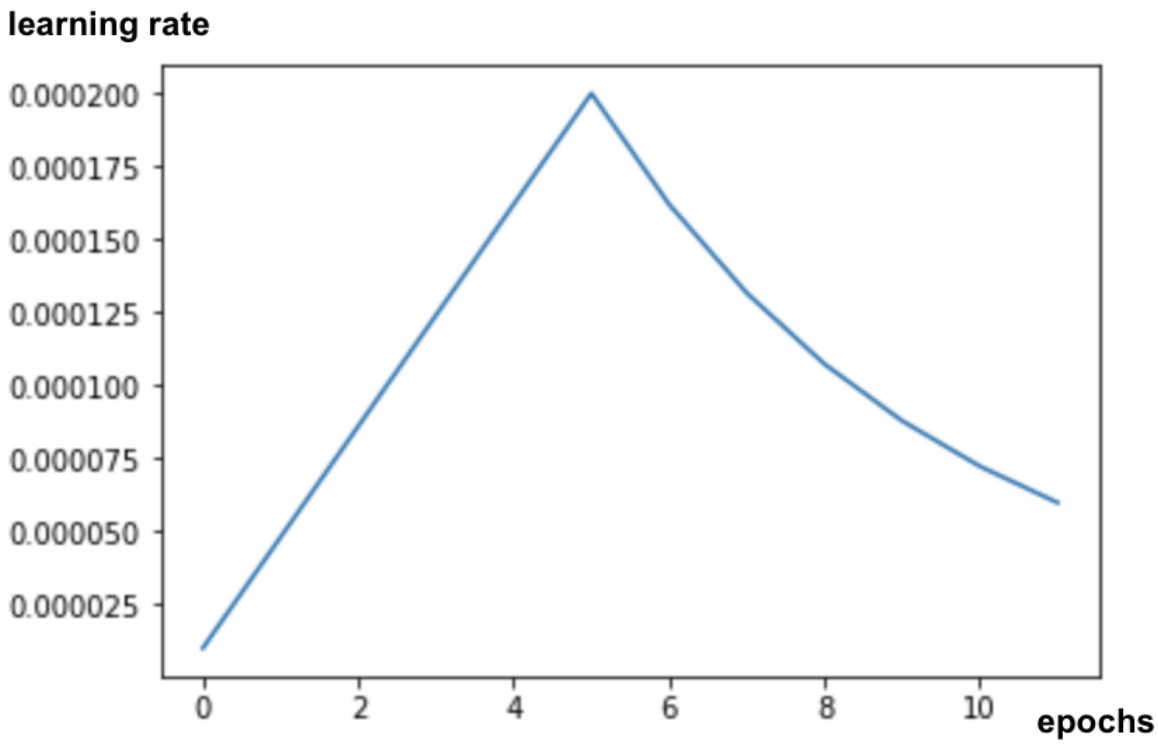
This being the final exercise, it requires a bit more code and data science work.

Additional info on fine-tuning

Xception is available in the standard pre-trained models in tf.keras.application.*
Do not forget to leave all weights trainable this time.

```
pretrained_model = tf.keras.applications.Xception(input_shape=
[*IMAGE_SIZE, 3],
                                                  include_top=False)
pretrained_model.trainable = True
```

To get good results when fine-tuning a model, you will need to pay attention to the learning rate and use a learning rate schedule with a ramp-up period. Like this:



Starting with a standard learning rate would disrupt the pre-trained weights of the model. Starting progressively preserves them until the model has latched on your data is able to modify them in a sensible way. After the ramp, you can continue with a constant or an exponentially decaying learning rate.

In Keras, the learning rate is specified through a callback in which you can compute the appropriate learning rate for each epoch. Keras will pass the correct learning rate to the optimizer for each epoch.

```
def lr_fn(epoch):
  lr = ...
  return lr

lr_callback = tf.keras.callbacks.LearningRateScheduler(lr_fn,
verbose=True)

model.fit(..., callbacks=[lr_callback])
```

Solution

Here is the solution notebook. You can use it if you are stuck.

 **07_Keras_Flowers_TPU_xception_fine_tuned_best.ipynb**

What we've covered

- 🤔 Depth-separable convolution
- 🤓 Learning rate schedules
- 😈 Fine-tuning a pre-trained model.

Please take a moment to go through this checklist in your head.