Google Cloud

Partner Certification Academy

Professional Machine Learning Engineer

The information in this presentation is classified:

Google confidential & proprietary

⚠️ This presentation is shared with you under NDA.

- Do **not** record or take screenshots of this presentation.
- Do **not** share or otherwise distribute the information in this presentation with anyone **inside** or **outside** of your organization.

Thank you!

Google Cloud

Source Materials

Some of this program's content has been sourced from the following resources:

- Google Cloud certification site
- Google Cloud documentation
- Google Cloud console
- Google Cloud courses and workshops
- Google Cloud white papers
- Google Cloud Blog
- Google Cloud YouTube channel
- Google Cloud samples
- Google codelabs
- Google Cloud partner-exclusive resources

Google Cloud

Google Cloud Skills Boost for Partners

- Professional Machine Learning Engineer Certification
- Cloud Skills Boost for Partners Professional Machine Learning Engineer Learning Path
- Partner Learning Services Instructor-Led PMLE Curriculum

Google Cloud Partner Advantage

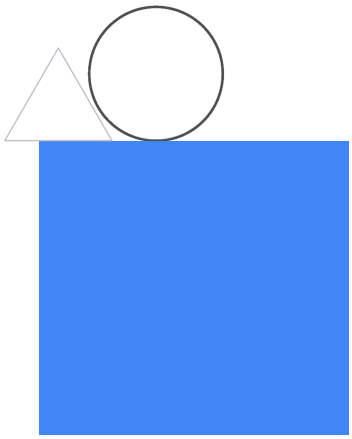
- Best practices for implementing machine learning on Google Cloud
- Artificial Intelligence
- End-to-End MLOps Go-to-Market Kit

Session Logistics

- When you have a question, please:
  - Click the Raise hand button in Google Meet.
  - Or add your question to the Q&A section of Google Meet.
  - Please note that answers may be deferred until the end of the session.
- These slides are available in the Student Lecture section of your Qwiklabs classroom.
- The session is **not recorded.**
- Google Meet does not have persistent chat.
  - If you get disconnected, you will lose the chat history.
  - Please copy any important URLs to a local text file as they appear in the chat.

Google Cloud Partner Learning Programs

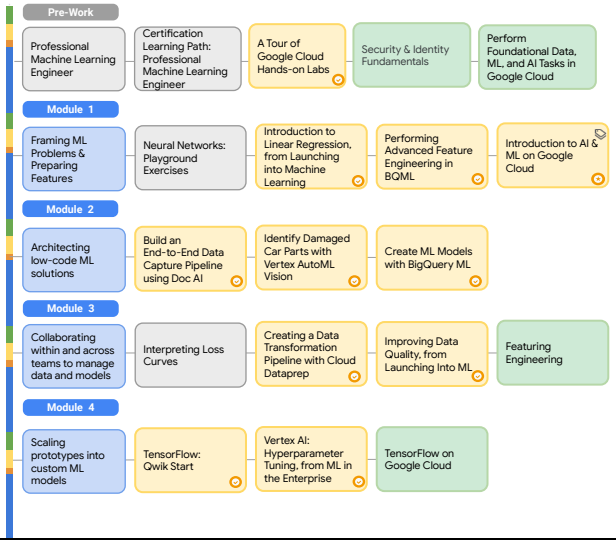
- Partner Certification Academy
- Partner Delivery Readiness Index (DRI)
- Cloud Skills Boost for Partners
- Partner Advantage



PARTNER CERTIFICATION ACADEMY

Professional Machine Learning Engineer

On-demand Course
Hands-on Skill Badge
Hands-on Lab
Resource
Live Virtual Workshop
required to earn exam voucher



A Professional Machine Learning Engineer builds, evaluates, productionizes, and optimizes ML models by using Google Cloud technologies and knowledge of proven models and techniques. The ML Engineer:

- handles large, complex datasets and creates repeatable, reusable code.
- considers responsible AI and fairness throughout the ML model development process, and collaborates closely with other job roles to ensure long-term success of ML-based applications.
- has strong programming skills and experience with data platforms and distributed data processing tools.
- is proficient in the areas of model architecture, data and ML pipeline creation, and metrics interpretation.
- is familiar with foundational concepts of MLOps, application development, infrastructure management, data engineering, and data governance.
- makes ML accessible and enables teams across the organization.

By training, retraining, deploying, scheduling, monitoring, and improving models, the ML Engineer designs and creates scalable, performant solutions.

Recommended candidate:

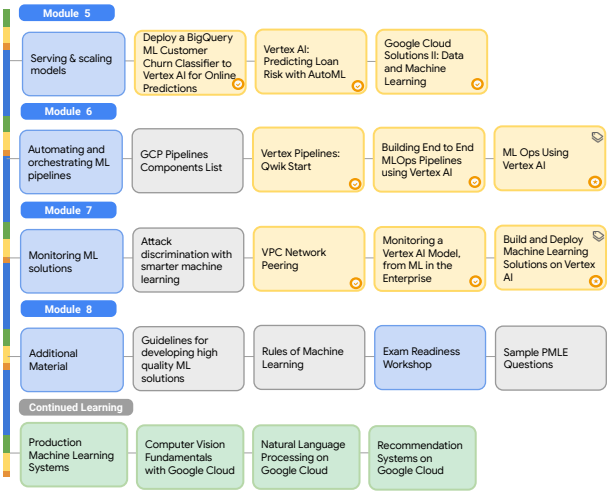
- Has in-depth experience setting up cloud environments for an organization
- Has experience deploying services and solutions based on business requirements

Google Cloud

PARTNER CERTIFICATION ACADEMY

Professional Machine Learning Engineer

On-demand Course
Hands-on Skill Badge
Hands-on Lab
Resource
Live Virtual Workshop
required to earn exam voucher



A Professional Machine Learning Engineer builds, evaluates, productionizes, and optimizes ML models by using Google Cloud technologies and knowledge of proven models and techniques. The ML Engineer:

- handles large, complex datasets and creates repeatable, reusable code.
- considers responsible AI and fairness throughout the ML model development process, and collaborates closely with other job roles to ensure long-term success of ML-based applications.
- has strong programming skills and experience with data platforms and distributed data processing tools.
- is proficient in the areas of model architecture, data and ML pipeline creation, and metrics interpretation.
- is familiar with foundational concepts of MLOps, application development, infrastructure management, data engineering, and data governance.
- makes ML accessible and enables teams across the organization.

By training, retraining, deploying, scheduling, monitoring, and improving models, the ML Engineer designs and creates scalable, performant solutions.

Recommended candidate:

- Has in-depth experience setting up cloud environments for an organization
- Has experience deploying services and solutions based on business requirements

Google Cloud

Learner Commitment

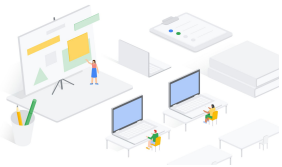
Each week, learners are to complete the learning path's course content, Cloud Skills Boost for Partner Quests/Challenge Labs and material that the mentor has recommended that will support learning.

- **Workshop Day:** Meet for the cohort's weekly 'general session'. **(≅ 2 hours)**
- **During the week:** Complete the week's course, perform hands-on labs, review any additional material suggested material for the week. **(≅ 8 - 16 hours)**

➔ **Important:** Learners must allocate time between each weekly session to study and familiarize themselves with any prerequisite knowledge they may lack. It is also recommended that learners complete the next week's course prior to the scheduled workshop.

Path to Service Excellence

Certification



Advanced Solutions Training

Delivery Readiness Index

Certification is just one step on your professional journey. Google Cloud also offers our partners access to advanced solutions training, and a new quality-focused program called Delivery Readiness Index (DRI) to help you achieve service excellence with your customers.

Benchmark your skills with DRI

DRI helps to benchmark partner proficiency and capability at any point during the customer journey however should be used primarily as a lead measure to predict and prepare for partner delivery success.

DRI assesses and analyzes Partner Consultant GCP proficiency by creating a DRI Profile inclusive of their GCP knowledge, skills, and experience.

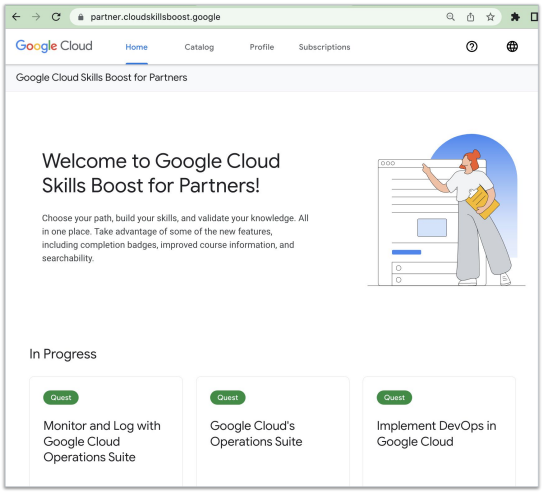
With the DRI insights, we can prescriptively advise the partner project team on the ground and bridge niche capability gaps.

DRI also takes action. For partner consultants, DRI generates a tailored L&D plan that prescribes personalized learning, training, and skill development to build GCP proficiency.

Google Cloud Skills Boost for Partners

https://partner.cloudskillsboost.google/

- On-demand course content
- Hands-on labs
- Skill Badges
- **FREE** to Google Cloud Partners!

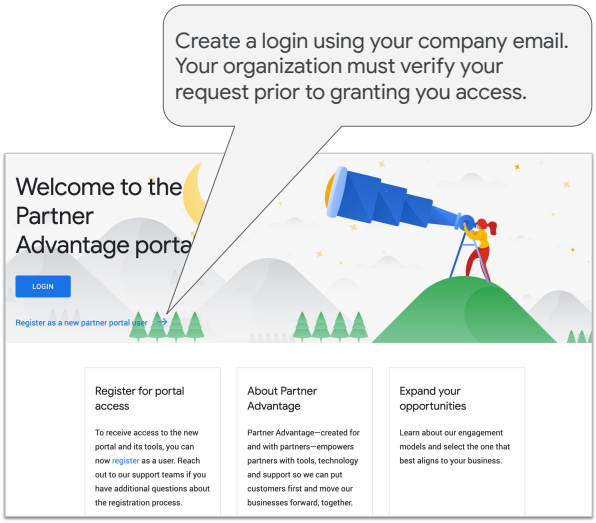


Proprietary + Confidential

Google Cloud

Partner Advantage

- Resources for Google Cloud partner organizations:
  - Recent announcements
  - Solutions/role-based training
  - Live/pre-recorded webinars on various topics
    - Partner Advantage Live Webinars
- Complements the certification self-study material presented on Google Cloud Skills Boost for Partners
- Helpful Links:
  - Getting started on Partner Advantage
  - Join Partner Advantage
  - Get help accessing Partner Advantage



https://www.partneradvantage.goog

Google Cloud

The getting started link:
https://support.google.com/googlecloud/topic/9198654#zippy=

Note the top section, **"Getting Started & User Guides"** and two key documents → Direct Partners to this if they need to enroll into Partner Advantage

1. Logging in to the Partner Advantage Portal - Quick Reference Guide
2. Enrolling in the Partner Advantage Program - Quick Reference Guide

Focus from this point on:

Some context on enrolling in PA:
Access to Partner Portal is given in 2 ways

- Partner Admin Led: Partner Administrator at Partner Company can set up users
- User Led: User can go through Self Registration
  - https://www.partneradvantage.goog/GCPPRM/s/partneradvantageportallogin?language=en_US
  - Or directly to the User Registration Form, https://www.partneradvantage.goog/GCPPRM/s/partnerselfregistration?language=en_US

Please Note

- After a user self-registers, they receive an email that essentially states:

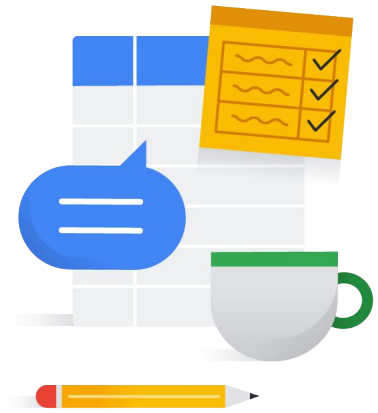
    - “Hi {Partner Name}, you are one step away from joining the Google Cloud Partner Advantage Community. Please click to continue with the user registration process. See you in the cloud, The Partner Advantage Team
- Once registered, they can access limited content until their **Partner Administrator approves the user**
- Their Partner Administrator also receive an email notifying them that a member of their organization has registered themselves on their organization’s Google Cloud Partner Advantage account.
    - It also states that this user has limited access to the portal
    - They are provided instructions on how to review and provision the appropriate access for the user that has registered
- Once their admin approves the user, they receive an email that states:
    - Hi {User Name}, Your Partner Administrator has updated your access to the Google Cloud Partner Advantage portal. You have been granted edit access to additional account information on the portal on behalf of your organization to help build your business. For additional access needs, please work with your Partner Administrator. See you in the cloud, The Partner Advantage Team

The net takeaway is, on the Support Page (the first link on this slide) Google Cloud Partner Advantage Support, there’s a section **“Issue accessing Partner Advantage Portal? Click here for troubleshooting steps”**

- The source of their issue can be related to the different items shown
- Additionally, there’s a Partner Administrator / Partner Adminstrator Team at their partner organization that has to approve their access.. Until that step is completed, they will have access issues/limitation. They will need to identify who this person or team is at their organization

Program issues or concerns?

- Problems with **accessing** Cloud Skills Boost for Partners
  - cloud-partner-training@google.com
- Problems with **a lab** (locked out, etc.)
  - support@qwiklabs.com
- Problems with accessing Partner Advantage
  - https://support.google.com/googlecloud/topic/9198654



- Problems with accessing **Cloud Skills Boost for Partners**
  - cloud-partner-training@google.com
- Problems with a **lab** (locked out, etc.)
  - support@qwiklab.com
- Problems with accessing **Partner Advantage**
  - https://support.google.com/googlecloud/topic/9198654

Module 4

Scaling prototypes into custom ML models, Part 1

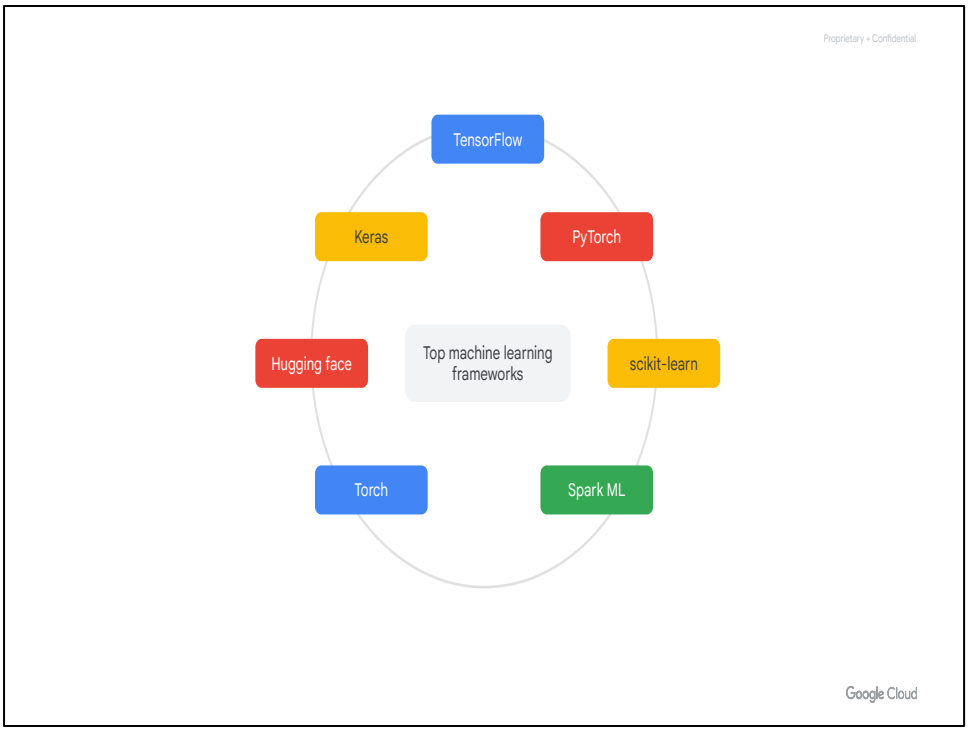
Module Agenda

01 Building custom models

02 Training custom models

03 Hyperparameter tuning on Google Cloud

Building custom models



Source: Machine Learning on Google Cloud v3.0

There are many different machine learning frameworks to help solve business requirements and machine learning problems.

Scikit Learn is a common one to begin with for baseline models.
TensorFlow, optionally with the Keras API to add a simpler layer with which to experiment with architectures, is another example.
PyTorch is another common framework.

You can work with any of these on GCP

TensorFlow is an open-source, high-performance library for numerical computation that uses directed graphs.

TensorFlow Explanation:

Open-source: TensorFlow is freely available for anyone to use, modify, and distribute. This means that developers can access its source code and contribute to its development, making it better over time.

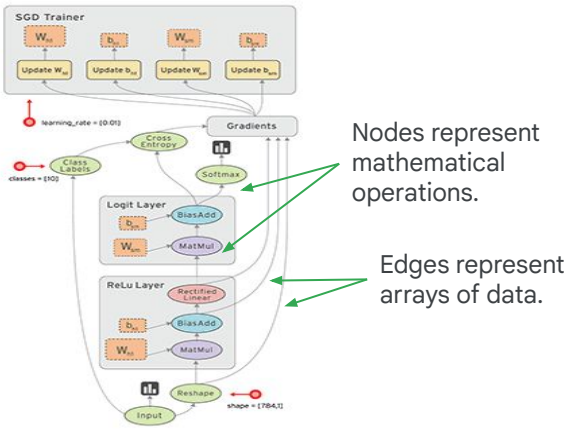
High-performance: TensorFlow is designed to work efficiently and quickly, even with large amounts of data or complex calculations.

Library for numerical computation: TensorFlow is a collection of tools, functions, and algorithms that help programmers perform mathematical operations, especially those needed for AI and machine learning tasks.

Directed graphs: In TensorFlow, computations are represented as a series of steps called a directed graph. Each step (or node) in the graph represents a mathematical operation, while the edges between nodes represent the flow of data between these operations. This graphical representation makes it easier to understand and optimize the computational process.

So, putting it all together, TensorFlow is a free and efficient software library that helps programmers perform complex mathematical operations using a directed graph representation, making it particularly well-suited for

What is a directed graph?



TensorFlow Explanation:
is a software library that makes it easier for people to create and work with ML Models. It helps in building and training ML systems, specifically for tasks that involve

- recognizing patterns,
    - like understanding images

- Translating languages.

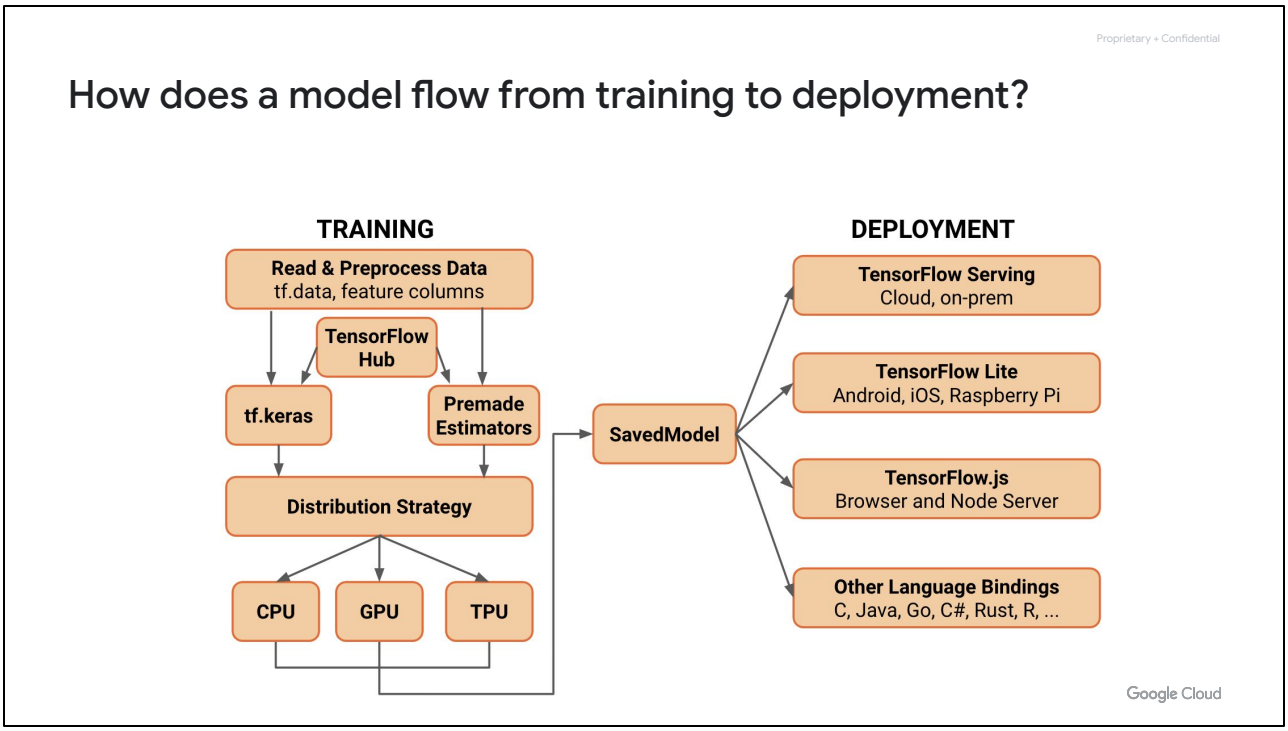
TensorFlow is like a set of building blocks and tools that help users create ML systems without needing to start from scratch.

Connecting the nodes are the edges which are the input and output of the mathematical operations. The *edges represent arrays of data flowing towards the output*.

Starting from the bottom are arrays of raw input data. Sometimes we will need to reshape it before feeding it into a layers of a neural network (like the ReLu layer here). More on ReLu later. Once inside that ReLu layer the weight is multiplied across the array of data in a MatMul or matrix multiplication. Then a bias term is added and the data flows through to the activation function.

Wow -- ReLu, activation functions? Don't worry, let's start with the basics. I kept mentioning an array of data flowing around -- what exactly does that mean?

GIF: https://www.tensorflow.org/images/tensors_flowing.gif



1. **Load your data using tf.data. T**raining data is read using input pipelines which are created using `tf.data`. Feature characteristics, for example bucketing and feature crosses are described using `tf.feature_column`. Convenient input from in-memory data (for example, NumPy) is also supported.
2. **Build, train and validate your model with tf.keras,** or use Premade Estimators. Keras integrates tightly with the rest of TensorFlow so you can access TensorFlow's features whenever you want. A set of standard packaged models (for example, linear or logistic regression, gradient boosted trees, random forests) are also available to use directly (implemented using the `tf.estimator` API). If you're not looking to train a model from scratch, you'll soon be able to use transfer learning to train a Keras or Estimator model using modules from TensorFlow Hub.
3. **Run and debug with eager execution, then use tf.function for the benefits of graphs**. TensorFlow 2.0 runs with eager execution by default for ease of use and smooth debugging. Additionally, the `tf.function` annotation transparently translates your Python programs into TensorFlow graphs. This process retains all the advantages of 1.x

1. TensorFlow graph-based execution: Performance optimizations, remote execution and the ability to serialize, export and deploy easily, while adding the flexibility and ease of use of expressing programs in simple Python.
2. **Use Distribution Strategies for distributed training**. For large ML training tasks, the Distribution Strategy API makes it easy to distribute and train models on different hardware configurations without changing the model definition. Since TensorFlow provides support for a range of hardware accelerators like CPUs, GPUs, and TPUs, you can enable training workloads to be distributed to single-node/multi-accelerator as well as multi-node/multi-accelerator configurations, including TPU Pods. Although this API supports a variety of cluster configurations, templates to deploy training on Kubernetes clusters in on-prem or cloud environments are provided.
3. **Export to SavedModel.** TensorFlow will standardize on SavedModel as an interchange format for TensorFlow Serving, TensorFlow Lite, TensorFlow.js, TensorFlow Hub, and more.

TensorFlow contains multiple abstraction layers

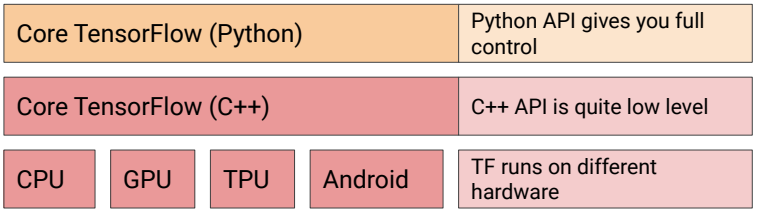
The lowest level of abstraction is the layer that is implemented to target the different hardware platforms. Unless your company makes hardware, it's unlikely that you'll do much at this level but it does exist.

TensorFlow contains multiple abstraction layers

The next level is the TensorFlow C++ API. This is how you can write a custom TensorFlow operation. You would implement the function you want in C++ and register it as a TensorFlow operation. (You can find more details on the Tensorflow documentation on "extending an op" https://www.tensorflow.org/extend/adding_an_op).

TensorFlow will then give you a Python wrapper that you can use just like you would use an existing function. Assuming you're not an ML researcher, you don't have to do this. But if you ever need to implement your own custom op, you would do it in C++ and it's not too hard. TensorFlow is extensible that way.

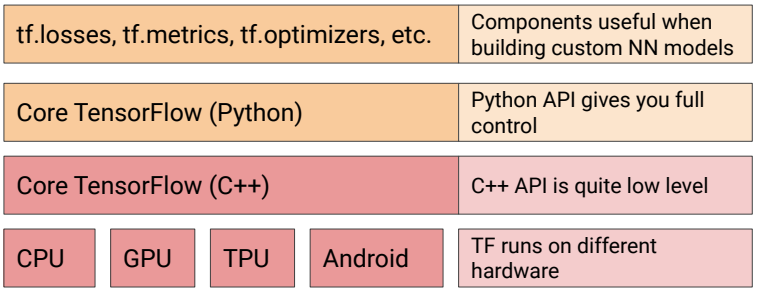
TensorFlow contains multiple abstraction layers



The Core Python API is what contains much of the numeric processing code. Add, subtract, divide, matrix multiply, etc. Creating variables, creating tensors, getting the shape or dimension of a tensor.

All that core, basic, numeric processing stuff is in the Python API.

TensorFlow contains multiple abstraction layers



Then, there are sets of Python modules that have high-level representation of useful neural network components.

Let's say you are interested in creating a new layer of hidden neurons with a ReLU activation function. You can do that by using tf.layers.

If you want to compute the RMSE on data as it comes in, you can use tf.metrics.

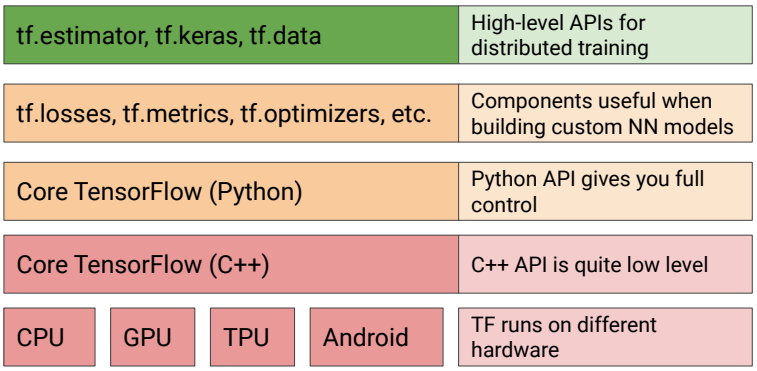
To compute cross-entropy-with-logits, for example, which is a common loss measure in classification problems, you can use tf.losses.

These modules provide components that are useful when building 'custom' Neural Network models.

Why are ‘custom’ NN models emphasized? Because you often don’t need a custom NN model. Many times, you’re quite happy to go with a relatively standard way of training, evaluating and serving models.

You don’t need to customize the way you train. You’re going to use one of the family of gradient-descent-based optimizers and you’re going to backpropagate the weights and do this iteratively. In that case, don’t write the low-level session loop. Just use an estimator or a high-level API such as Keras!

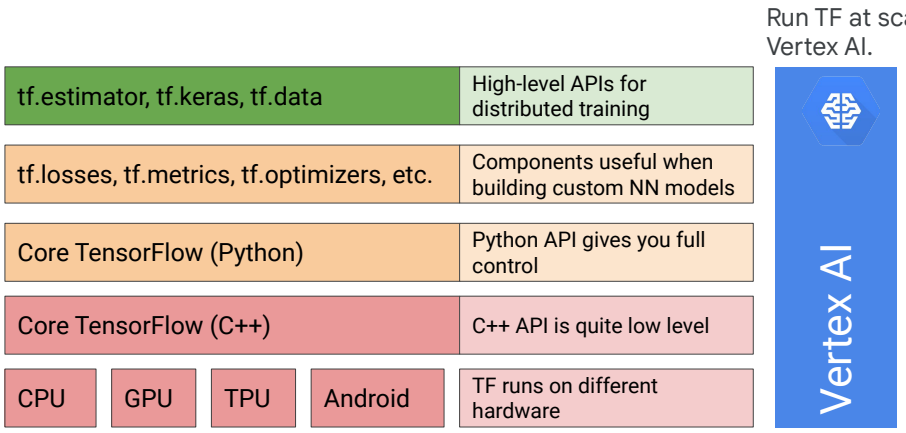
TensorFlow contains multiple abstraction layers



The high-level APIs allow you to easily do distributed training, data preprocessing and model definition, compilation and training. It knows how to evaluate, how to create a checkpoint, how to save a model, how to set it up for TensorFlow serving. It comes with everything done in a sensible way that fits most ML models in production.

If you see example TensorFlow code out on the internet that does not use the Estimator API, ignore that code. Walk away. It's not worth it. You'll have to write a lot of code to do device placement, and memory management, and distribution. Let the high-level API do it for you.

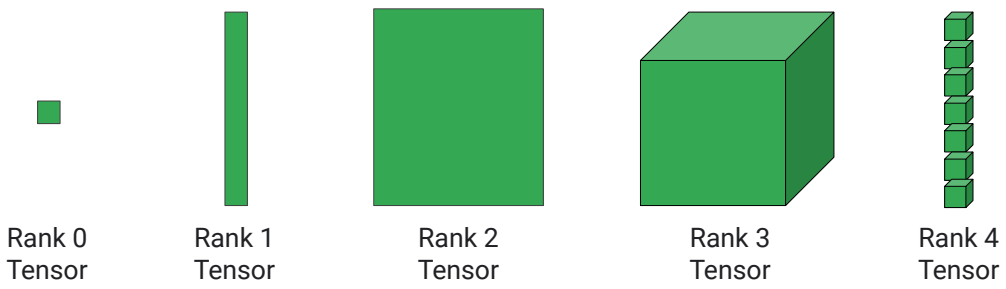
TensorFlow contains multiple abstraction layers



So, those are the TensorFlow levels of abstraction. Cloud AI Platform is orthogonal to this hierarchy -- meaning it cuts across all low-level and high-level APIs.

Regardless of which abstraction level you are writing your TensorFlow code at, CAIP (Cloud AI Platform) gives you a managed service. It's fully hosted TensorFlow. So you can run TF on the cloud on a cluster of machines without having to install any software or manage any servers.

A tensor is an N-dimensional array of data



Well that's actually where TensorFlow gets it's name from!

Starting on the far left -- the simplest piece of data we can have is a scalar. That's a number like "3" or "5". It's what we call zero dimensional or Rank 0. We're not going to get very far passing around single numbers in our flow so let's upgrade.

Rank 1 or 1 dimensional array is a vector. In physics, a vector is something with magnitude and direction.

But in Computer Science, you use vector to mean 1D arrays like a series of numbers in a list.

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()

A tensor is an N-dimensional array of data. When you create a tensor, you will specify its SHAPE. Occasionally, you'll not specify the shape completely. For example, the first element of the shape could be variable, but that special case will be ignored for now. Understanding the SHAPE of your data -- or often times the shape it SHOULD be -- is the first essential part of your machine learning flow.

Here, you're going to create a tf.constant(3). This is 0-rank tensor. Just a number. A scalar. The shape when you look at the Tensor debug output will be simply open-parenthesis close-parenthesis. It's zero-rank. To better understand why there isn't a number in those parentheses let's upgrade to the next level.

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)

If you passed in a bracketed list like 3, 5, 7 to tf.constant instead -- you would now be the proud owner of a one-dimensional tensor (otherwise known as a vector)

Now, you have a one-dimensional tensor. A vector.

It's grow horizontally (think like on the X-Axis) by three units. Nothing on the Y Axis yet since we're still in 1 dimension here. That's why the shape is (3, nothing).

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

Level up!

Now we have a matrix of numbers or a two dimensional array. Take a look at the shape (2,3) that means we have two rows and three columns of data. The first row being that original vector of [3,5,7] which also has three elements in length (that's where the three columns of data comes from).

You can think of a matrix as essentially a stack of 1 D tensors. The first tensor is the vector [3,5,7], and the second 1D tensor that is being stacked is the vector [4,6,8].

Okay so we've got height and width. Let's get more complex!

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

	3D Tensor	3	`tf.constant([[[3, 5, 7],[4, 6, 8]],`
			`[[1, 2, 3],[4, 5, 6]]`
			`])`
				(2, 2, 3)

What does 3D look like? It's a 2D tensor with another 2D tensor on top of it.

Here you can see we're stacking the 3,5,7 matrix on top of the 1,2,3 matrix. We started with two 2x3 matrices so our resulting shape of the 3D tensor is now 2 comma 2 comma 3.

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

	3D Tensor	3	`tf.constant([[[3, 5, 7],[4, 6, 8]],`
			`[[1, 2, 3],[4, 5, 6]]`
			`])`
				(2, 2, 3)

	nD Tensor	n	`x1 = tf.constant([2, 3, 4])`
			`x2 = tf.stack([x1, x1])`
			`x3 = tf.stack([x2, x2, x2, x2])`
			`x4 = tf.stack([x3, x3])`
			`...`
				(3,)
				(2, 3)
				(4, 2, 3)
				(2, 4, 2, 3)

Of course, you can do the stacking in code itself instead of counting parentheses.

Take the example here.

- Our x1 variable is a tf constant constructed from a simple list [2,3,4]. That makes it a vector of length 3.
- X2 is constructed by stacking x1 on top of x1. So, that makes it a 2x3 matrix.
- X3 is constructed by stacking 4 x2s on top of each other. Since each x2 was a 2x3 matrix, this makes X3 a 3D tensor of shape 4x2x3.
- X4 is constructed by stacking x3 on top of x3. That makes it two 4x2x3 tensors, or the final result which is a 4D tensor of shape 2 comma 4 comma 2 comma 3.

A tensor is an N-dimensional array of data

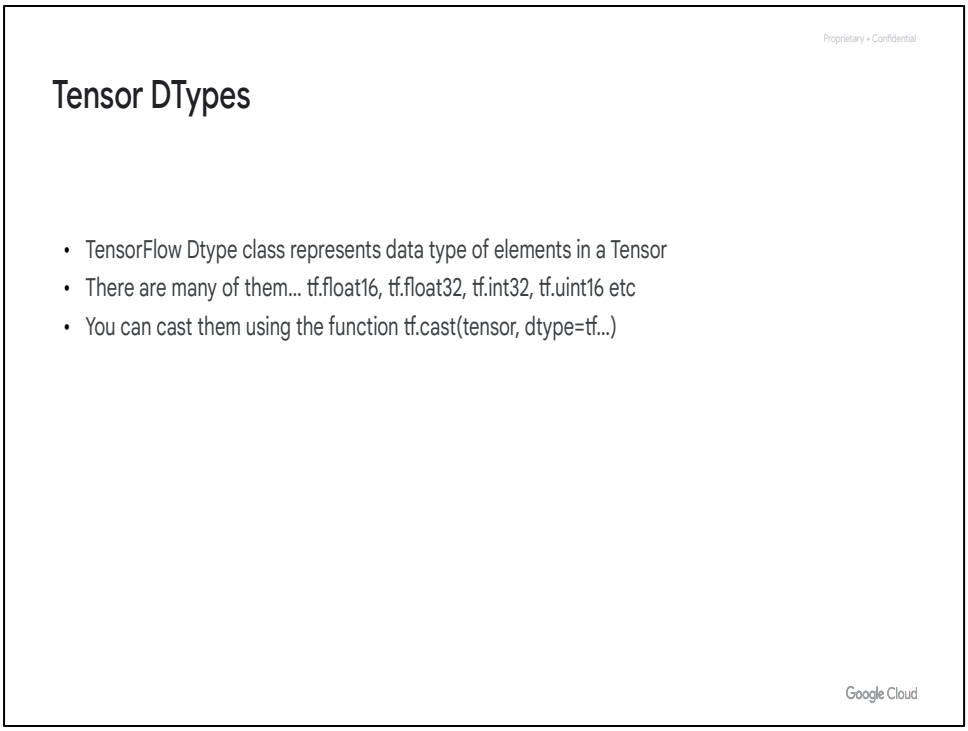
They behave like numpy n-dimensional arrays except that

- tf.constant produces constant tensors
- tf.Variable produces tensors that can be modified

If you've worked with arrays of data before like with Numpy they're similar expect for these two points.

tf.constant will produce tensors with constant values and tf.variable produces tensors with variable values (or ones that can be modified). This will prove super useful later when we need to adjust the model weights during the training phase of our ML project. Those weights can simply be a modifiable tensor array.

Let's take a look at the syntax for each as you will become a ninja in combining, slicing, and reshaping tensors as you see fit.



DTypes are used to specify the type of data a tensor can hold. Some common DTypes in TensorFlow are:

tf.float16, tf.float32, tf.float64: These represent floating-point numbers (decimals) with varying levels of precision. The higher the number, the more precise the value.

tf.int8, tf.int16, tf.int32, tf.int64: These represent signed integer values (whole numbers, including negative numbers) with varying levels of precision. The higher the number, the larger the range of values it can hold.

tf.uint8, tf.uint16: These represent unsigned integer values (whole numbers, only non-negative) with varying levels of precision. The higher the number, the larger the range of values it can hold.

tf.bool: This represents boolean values, which are either True or False.

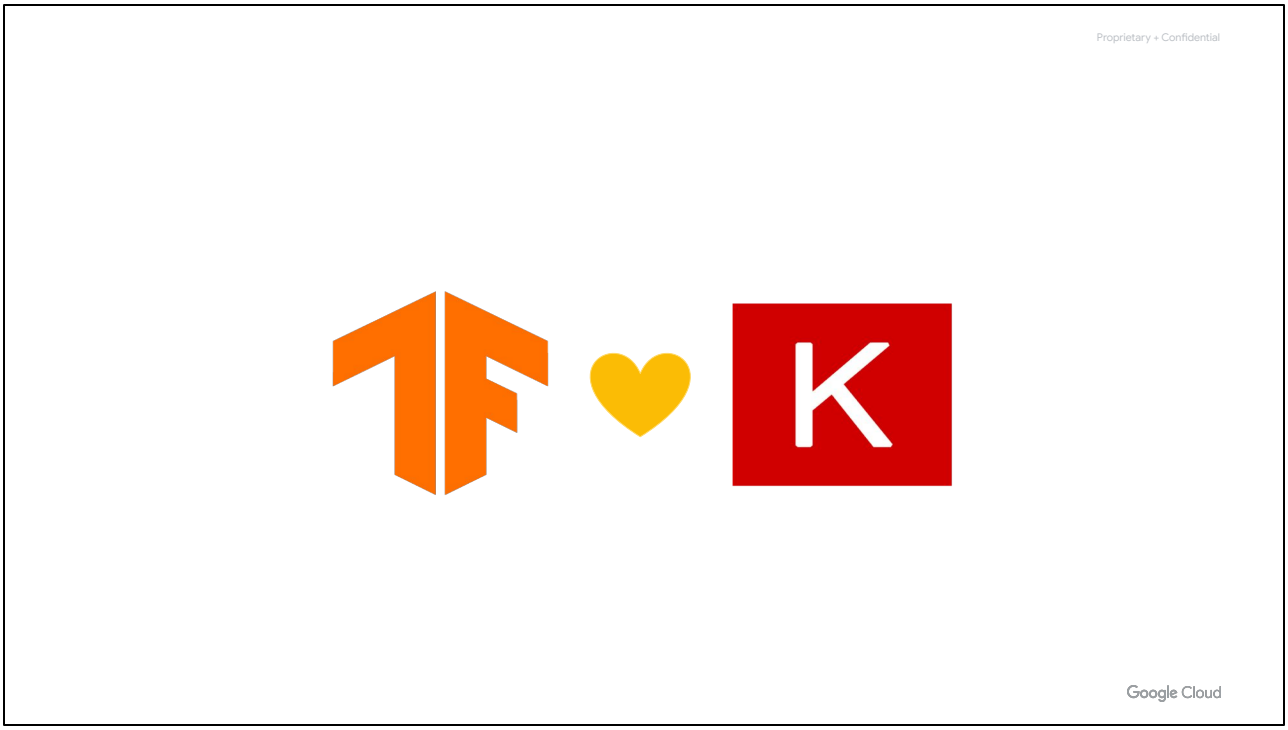
tf.string: This represents text data, like words or sentences.

Building Neural Networks with Keras

So far in the course our data has been loaded in batch form from pre-existing structured databases like Cloud SQL or BigQuery. But what about if the data is on thousands of sensors world-wide? And what if they send new data messages every minute? How can you harness such upstream data sources for your analysis?

Welcome to the module on Real-time IoT Dashboards where we will highlight and solve the challenges of streaming data processing.

LAB: Classifying Images with a NN and DNN Model:
https://partner.cloudskillsboost.google/catalog_lab/5003

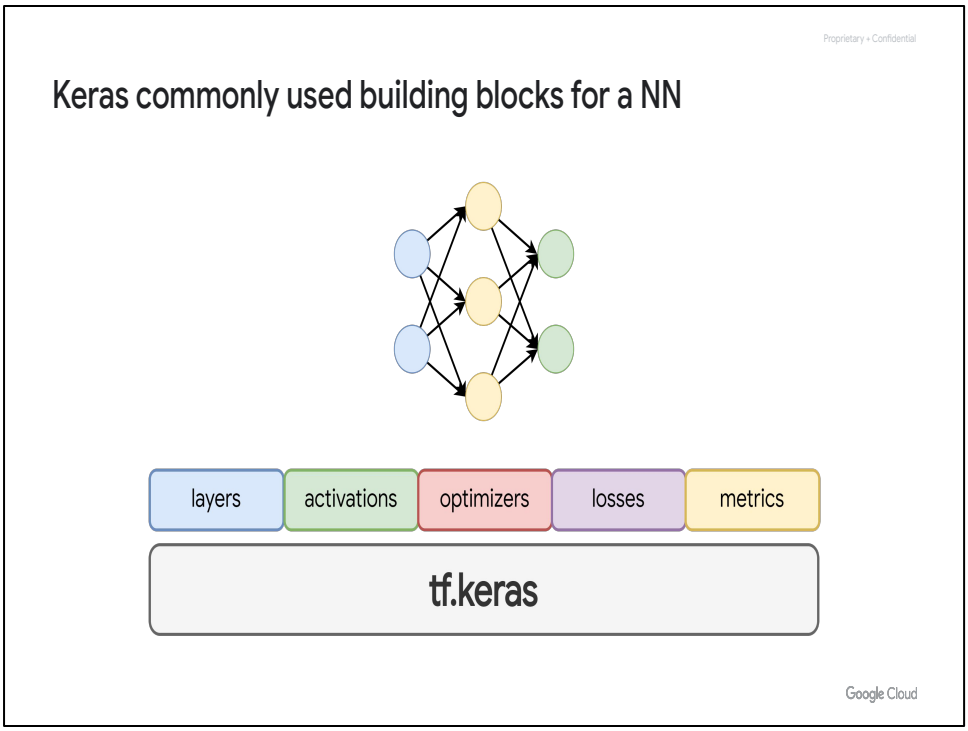


tf.keras is TensorFlow's high-level API for building and training deep learning models. It's used for fast prototyping, state-of-the-art research, and production, with three key advantages:

- **It is user-friendly**
  Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.

- **It is modular and composable**
  Keras models are made by connecting configurable building blocks together, with few restrictions.

- **It is easy to extend**
  Write custom building blocks to express new ideas for research. Create new layers, metrics, loss functions, and develop state-of-the-art models.



with Keras, you can easily assemble layers and components to build complex neural network models without having to code every detail from the ground up.

- It's designed to be user-friendly and intuitive, making the process of designing and training neural networks simpler and more accessible.

simple analogy: imagine building a multi-story house.

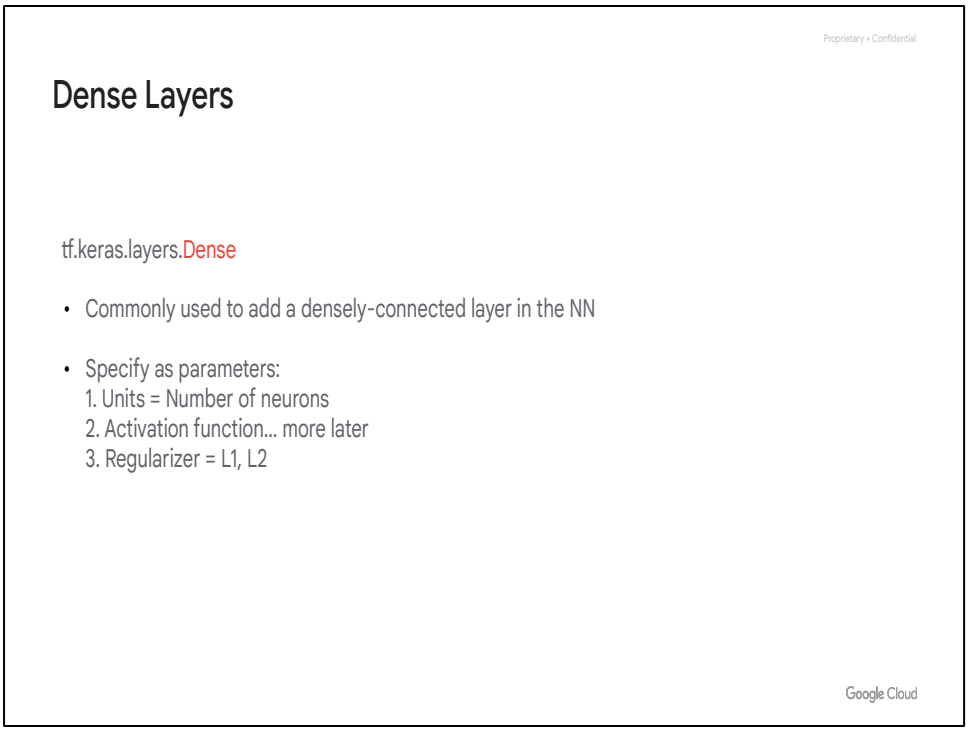
- **Layers:** These are the floors of our house. Each floor has a specific layout and purpose. In Keras, layers are the building blocks of the neural network. Just like adding floors to a house, you stack layers to make your network deeper.

- **Activations:** Think of these as the interior design style of each floor. They determine how the inputs (furniture placements) are transformed to outputs (final look). In neural networks, activation functions decide how to transform the data that passes through a layer, often adding non-linearity.

- **Optimizers:** This is like the construction supervisor who determines the best way to add or modify parts of the house to meet your vision. In Keras, optimizers decide how to adjust the network's weights based on the error it made, making the model better over time.

- **Losses:** This represents the difference between the dream house you've envisioned (target) and the current state of the house (model's prediction). In Keras, loss functions measure how far off our predictions are from the actual outcomes, guiding the model's improvement.

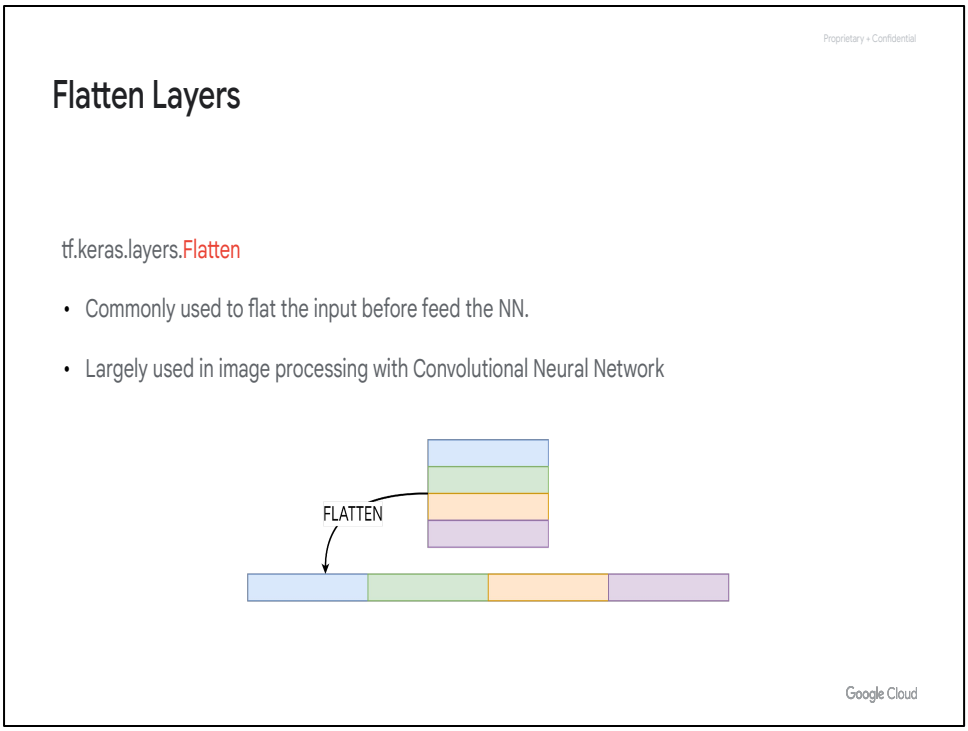
- **Metrics:** These are like the quality checks you do while visiting the construction site. You might check if the walls are straight or if the rooms are the size you wanted. In Keras, metrics are used to evaluate the performance of a model during training and testing. Unlike the loss, they're designed for human interpretation and might not be used to guide training.



tf.keras.layers.Dense: is a core component of neural networks in TensorFlow Keras API

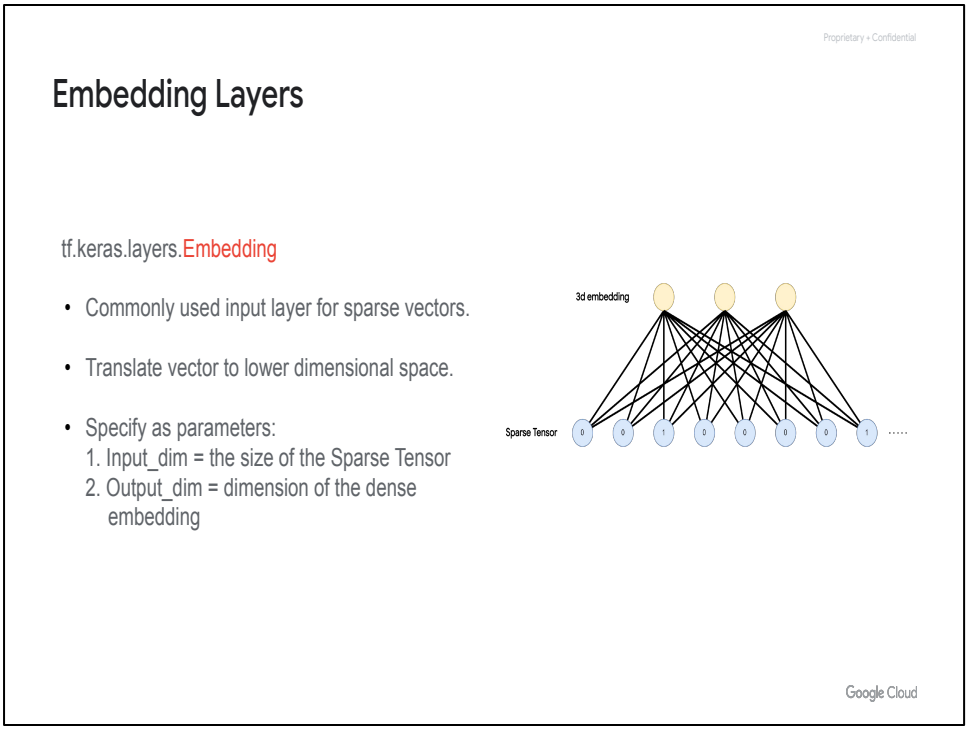
Simple Explanation:

- tf.keras.layers.Dense allows data to be transformed in a way that the neural network can learn and extract patterns from, aiding in tasks like classification, regression, etc.



tf.Keras.layers:

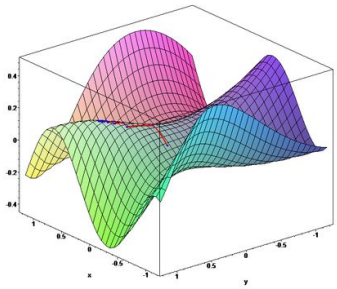
- A layer provided by TensorFlow Keras API that reshapes the input data without changing its volume. It's often used to flatten the output of a convolutional layer to be fed into a dense (fully connected) layer in neural networks, especially in Convolutional Neural Networks (CNNs).



Keras_layers_embedding:

- A layer provided by TensorFlow's Keras API to create word embeddings. Word embeddings are a form of word representation that captures the semantic meaning of words based on their context in sentences. They translate high-dimensional categorical data (like words) into lower-dimensional continuous vectors.

Stochastic Gradient Descent



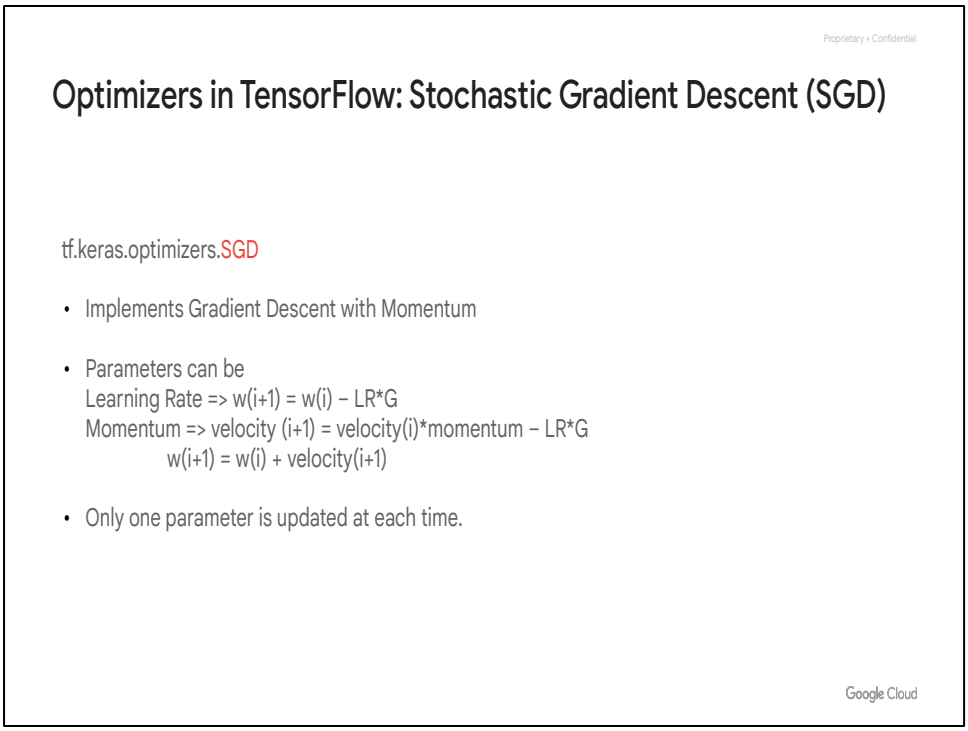
A mathematical technique to minimize loss. Gradient descent iteratively adjusts weights and biases, gradually finding the best combination to minimize loss.

https://developers.google.com/machine-learning/glossary#gradient-descent

Stochastic Gradient Descent
(SGD) is an optimization algorithm used to train machine learning models, including those built with TensorFlow. The main goal of an optimizer is to minimize the loss function, which represents the difference between the model's predictions and the true values.

SGD
is a variation of the Gradient Descent algorithm. In Gradient Descent, the model's parameters are updated in the direction of the steepest decrease of the loss function. This is done by computing the gradients (partial derivatives) of the loss function with respect to each parameter. However, Gradient Descent computes the gradients using the entire dataset, which can be computationally expensive for large datasets.

Stochastic Gradient Descent addresses this issue by updating the model's parameters using only a small subset of the dataset, called a mini-batch, in each iteration. This makes the optimization process faster and more efficient, while still converging to a solution close to the one obtained with Gradient Descent.



Stochastic Gradient Descent
(SGD) is an optimization algorithm used to train machine learning models, including those built with TensorFlow. The main goal of an optimizer is to minimize the loss function, which represents the difference between the model's predictions and the true values.

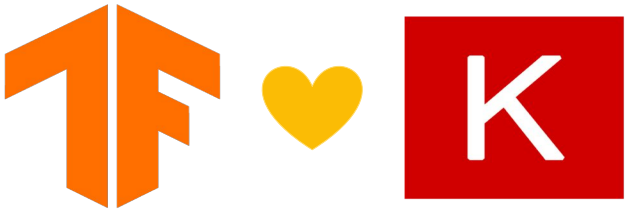
SGD
is a variation of the Gradient Descent algorithm. In Gradient Descent, the model's parameters are updated in the direction of the steepest decrease of the loss function. This is done by computing the gradients (partial derivatives) of the loss function with respect to each parameter. However, Gradient Descent computes the gradients using the entire dataset, which can be computationally expensive for large datasets.

Stochastic Gradient Descent addresses this issue by updating the model's parameters using only a small subset of the dataset, called a mini-batch, in each iteration. This makes the optimization process faster and more efficient, while still converging to a solution close to the one obtained with Gradient Descent.

There are other optimisers like

- tf.keras.optimizers.Adagrad

Loss functions in TensorFlow



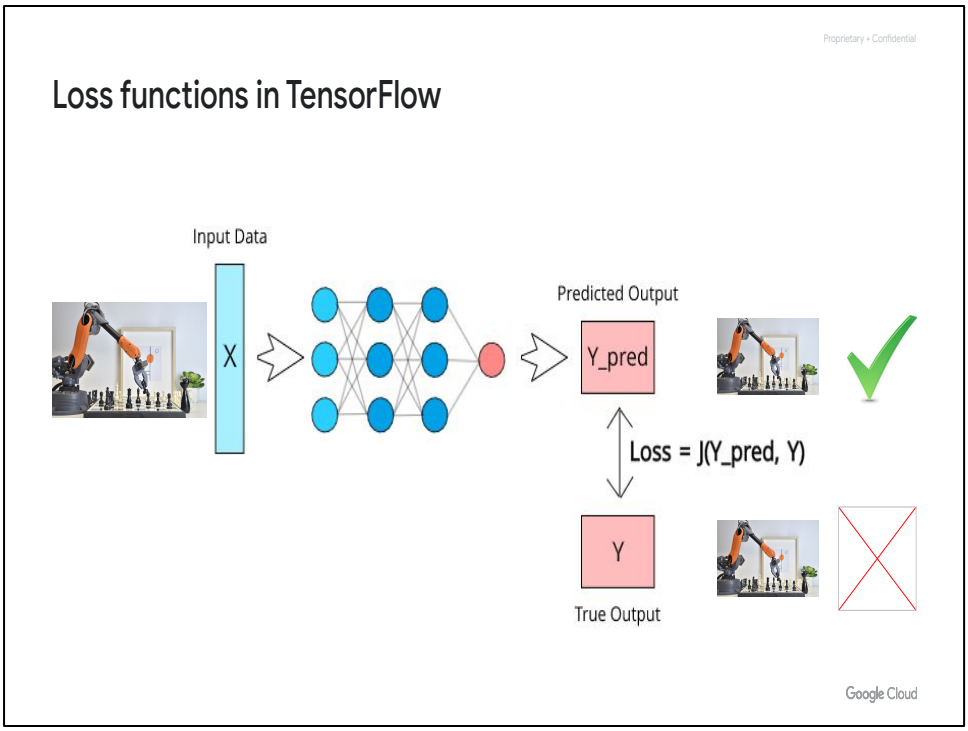
A loss function

tf.keras is TensorFlow's high-level API for building and training deep learning models. It's used for fast prototyping, state-of-the-art research, and production, with three key advantages:

- **It is user-friendly**
  Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.

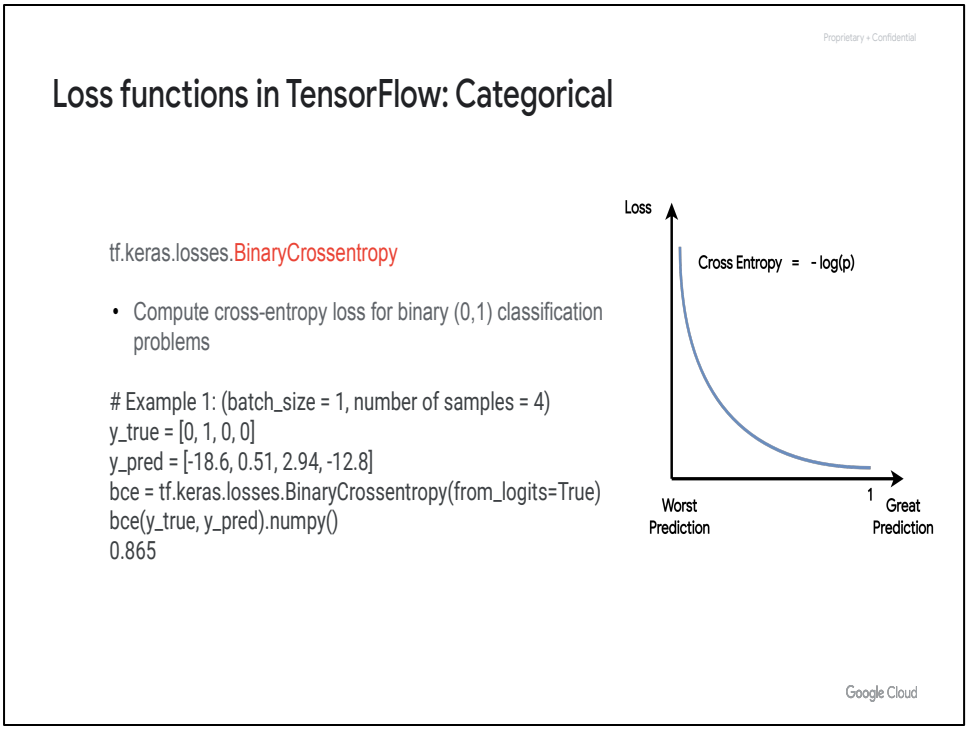
- **It is modular and composable**
  Keras models are made by connecting configurable building blocks together, with few restrictions.

- **It is easy to extend**
  Write custom building blocks to express new ideas for research. Create new layers, metrics, loss functions, and develop state-of-the-art models.



A loss function,
in simplistic terms, is like a scoring system that measures how well a machine learning model is performing.

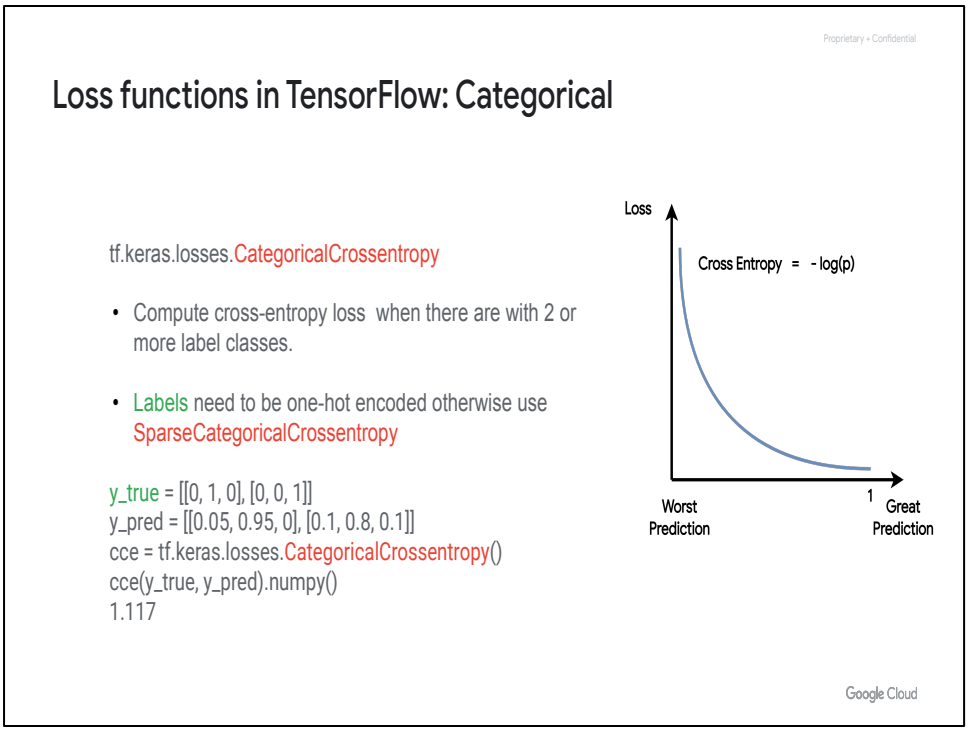
Imagine you're teaching a robot to play a game. The loss function is like the difference between the robot's current move and the ideal move it should make. The higher the difference, the worse the robot is playing, and the lower the difference, the better it's playing. The goal is to help the robot improve its game by minimizing this difference or "loss" over time.



Categorical loss functions
are used when dealing with problems where you need to classify data into multiple categories or classes. Imagine you have a bunch of fruit pictures and you want to teach a computer to recognize them as apples, bananas, or oranges.

Using the categorical loss function, the computer will predict probabilities for each fruit category. For example, it might predict that a certain image is 80% likely to be an apple, 10% likely to be a banana, and 10% likely to be an orange. The loss function will then compare these predictions to the actual labels (e.g., the image is 100% an apple) and calculate a "loss" value based on the difference.

The goal is to minimize this loss value, which means the computer gets better at predicting the correct fruit category. A common categorical loss function is the "categorical cross-entropy," which is used to measure the difference between the predicted probabilities and the actual labels.



Binary Crossentropy is a type of loss function specifically used for binary classification problems, where there are only two possible outcomes or classes (e.g., true or false, positive or negative, 1 or 0). In other words, it's used when you want to teach a machine learning model to make a decision between two choices.

Imagine you want to teach a computer to recognize if an email is spam or not. For each email, the model predicts a probability value, representing how likely it thinks the email is spam (e.g., 0.8 means 80% chance of being spam).

The Binary Crossentropy loss function
compares the model's predicted probability with the actual label (1 for spam, 0 for not spam). The loss function calculates a "loss" value based on the difference between the predicted probability and the actual label, penalizing the model more when the predictions are further away from the truth.

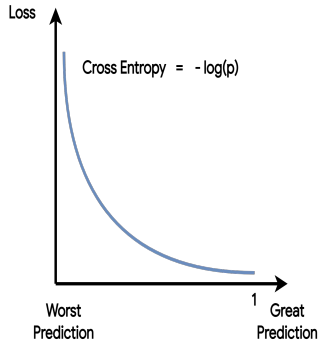
The goal is to minimize this loss value, which means the model gets better at correctly classifying emails as spam or not spam. As the model improves, the loss value decreases, indicating better performance.

Loss functions in TensorFlow: Categorical

tf.keras.losses.SparseCategoricalCrossentropy

- Similar to the CategoricalCrossentropy but labels are expected to be interger

```
y_true = [1, 2]
y_pred = [[0.05, 0.95, 0], [0.1, 0.8, 0.1]]
scce = tf.keras.losses.SparseCategoricalCrossentropy()
scce(y_true, y_pred).numpy()
1.117
```



tf.keras.losses.SparseCategoricalCrossentropy
is a loss function provided by TensorFlow's Keras library, specifically designed for multi-class classification problems where the classes are mutually exclusive. It is a variation of the Categorical Crossentropy loss function, with a key difference in how the true labels are represented.

Loss functions in TensorFlow: Regression

tf.keras.losses.MeanAbsoluteError

- Computes the mean of absolute difference between labels and predictions

tf.keras.losses.MeanSquaredError

- Computes the mean squared error between labels and predictions

Model Metrics in TensorFlow

tf.keras.metrics.Accuracy

- It tells you how often the predictions are equal to labels
- The metric creates two local variables (total, count) that are used to compute the frequency

tf.keras.metrics.AUC

- Approximates the Area Under the Curve of the ROC Precision/Recall
- It measures the quality of binary classifiers

tf.keras.metrics.RootMeanSquaredError

- It computes the RSME between prediction and label

Stacking layers with a Keras Sequential model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


model = Sequential([
    Input(shape=(64,))
    Dense(units=32, activation="relu", name="hidden1"),
    Dense(units=8, activation="relu", name="hidden2"),
    Dense(units=1, activation="linear", name="output")
])
```

The Keras sequential model stacks layers on the top of each other.

The batch size is omitted. Here the model expects batches of vectors with 64 components.

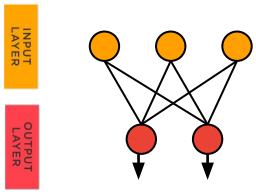
A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.

Sequential models are not advisable if:

- The model you're building has multiple inputs or multiple outputs
- Any of the layers of the model has multiple inputs or multiple outputs
- The model needs to do layer sharing
- The model has a non-linear topology such as a residual connection or multi-branches

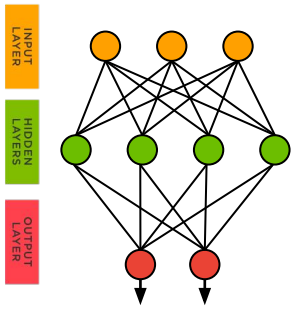
Training custom models

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = \
    keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(3,)),
  tf.keras.layers.Dense(2, activation='softmax')
])
```



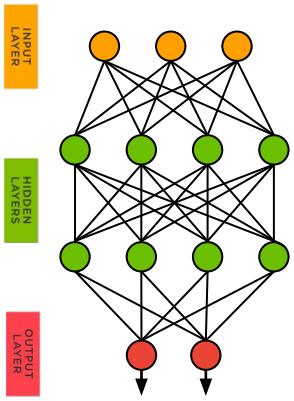
Here is a code snippet with all the steps put together: model definition, compilation, fitting and evaluation.

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = \
    keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(3,)),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])
```



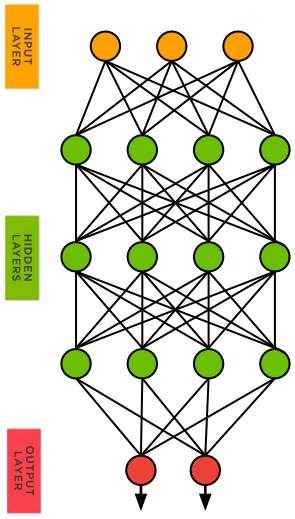
Here is a code snippet with all the steps put together: model definition, compilation, fitting and evaluation.

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = \
    keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(3,)),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])
```



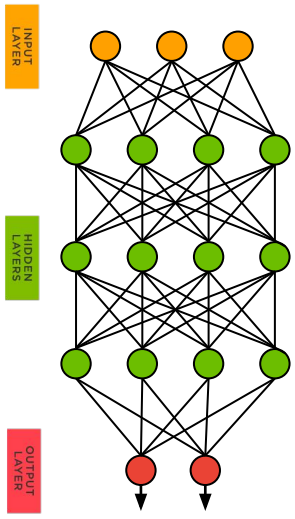
Here is a code snippet with all the steps put together: model definition, compilation, fitting and evaluation.

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = \
    keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(3,)),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])
```



Here is a code snippet with all the steps put together: model definition, compilation, fitting and evaluation.

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = \
    keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Input(shape=(3,)),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(2, activation='softmax')
])

# Configure and train
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

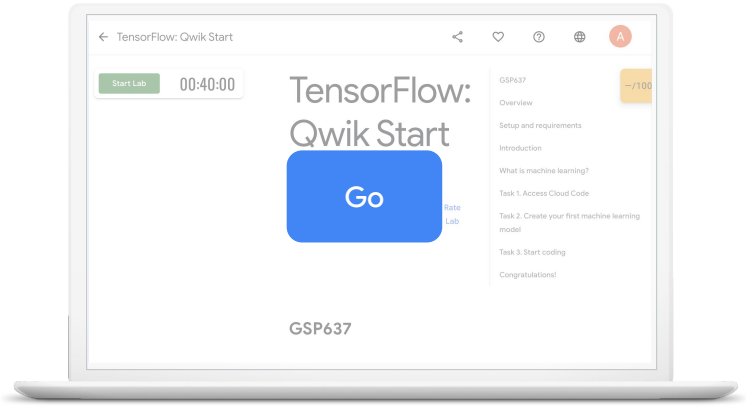


Here is a code snippet with all the steps put together: model definition, compilation, fitting and evaluation.

Recommended Lab

TensorFlow: Qwik Start

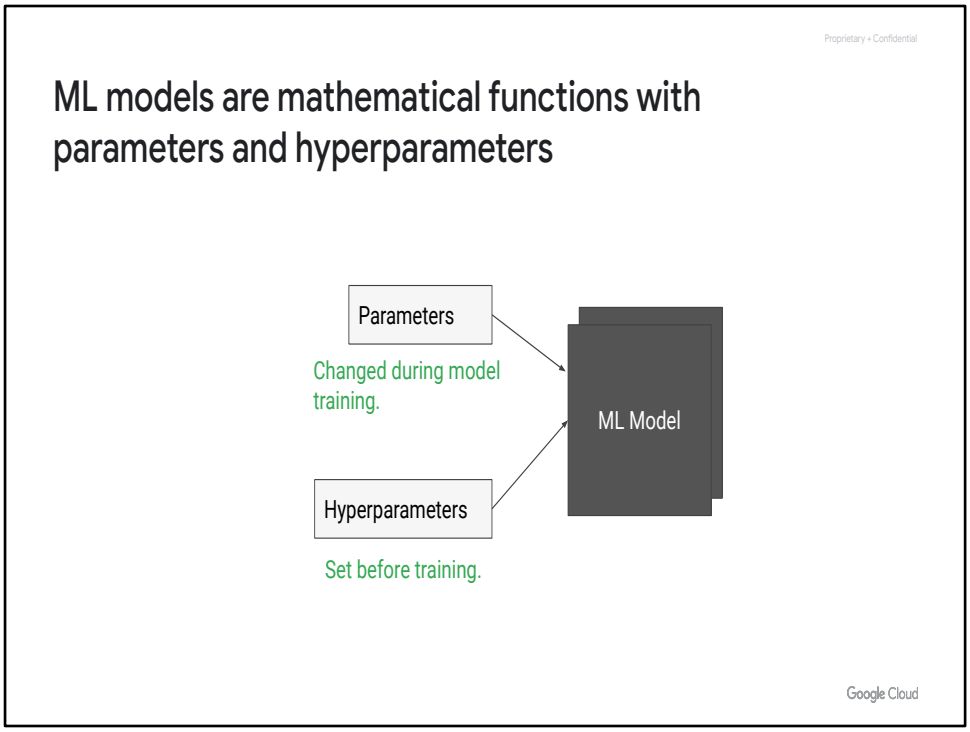
From the Course
Intermediate ML: TensorFlow on Google Cloud



TensorFlow: Qwik Start

Hyperparameter tuning on Google Cloud

- [Course] Machine Learning in the Enterprise
- - LAB: VertexAI Hyperparameter Tuning



This should look familiar to you. You should have seen it in “Launching into ML”.

Remember we said: ML models are mathematical functions with parameters and hyper-parameters.

A parameter is a real-valued variable that changes during model training, like all those weights and biases that we’ve come to know so well.

A hyper-parameter, on the other hand, is a setting that we set *before* training and it doesn’t change afterwards.

Examples of hyper-parameter are: learning rate, regularization rate, batch size, number of hidden layers in a neural network and the number of neurons in each layer.

Now that we are clear about the difference between parameters and hyper parameters, let’s shift our attention to hyper_parameters.

Since, we know parameters are going to be adjusted by the training algorithm. Our job is to set the hyper-parameters right.

We have several knobs that are dataset-dependent

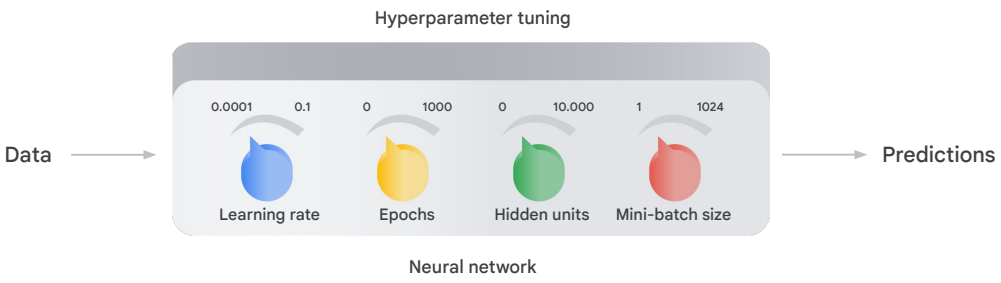
Image source: https://pixabay.com/en/acoustic-guitar-guitar-336479/ (cc0)

Turns out model performance is very sensitive to learning rate and batch size.

Doesn't it feel like tuning a musical instrument?

Told you there is some art involved!

Hyperparameter tuning



The hyperparameters are knobs that act as the network-human interface.

Examples of hyperparameters are the optimizer, its learning rate, epochs, regularization parameters, the number of hidden layers in a DNN, and their sizes. Setting hyperparameters to their optimal values for a given dataset can make a huge difference in model quality.

the learning rate
in machine learning is like the size of the steps a computer takes to learn from data. Imagine you're trying to find the lowest point in a valley by walking blindfolded. Each step you take gets you closer to the lowest point. The learning rate determines how big or small each step is. If the steps are too big, you might overshoot the lowest point; if they're too small, it'll take a long time to reach it. So, it's important to find the right balance for the learning rate to help the computer learn efficiently.

Epoch:
in machine learning is like one complete cycle of learning. Imagine you're studying flashcards to learn a new language. Each time you go through the entire stack of cards, you complete one cycle of learning or "epoch." In machine learning, an epoch is when the computer processes the entire

dataset once, adjusting its understanding after each example. Multiple epochs are often used to help the computer learn more effectively, as it goes through the data multiple times, refining its understanding with each cycle.

hidden units
Each hidden unit focuses on different aspects of the data, and by working together, they enable the computer to learn complex patterns and make accurate decisions.

Analogy:
in machine learning are like tiny helpers inside a computer's brain that work together to solve a problem. Imagine you're trying to solve a jigsaw puzzle, and you have a group of friends who each focus on finding pieces for a specific part of the picture. Each friend represents a hidden unit, and they all contribute to solving the puzzle by sharing their findings. In machine learning, hidden units are part of the network's "brain" (neural network) that process and transform the input data to help the computer understand and make predictions about the problem at hand.

the mini-batch size

In machine learning, the mini-batch size is the number of data points or examples the computer processes at once before updating its understanding. A larger mini-batch size means the computer processes more data at once, which can be faster but might require more memory. A smaller mini-batch size means the computer processes fewer data points at once, which can lead to a more refined learning process but might take more time. The choice of mini-batch size depends on the specific problem, available resources, and desired learning speed.

Analogy:
in machine learning is like the number of questions you practice at once while studying for an exam. Instead of reviewing all questions in one go, you break them into smaller groups and learn from each group separately. This makes the learning process more manageable and efficient.

Learning rate controls the size of the weight adjustment at each training step

If too small, training will take a long time

If too large, training will bounce around

Default learning rate in Estimator's LinearRegressor is smaller of 0.2 or 1/sqrt(num_features) -- this assumes that your feature and label values are small numbers

Image source: https://pixabay.com/en/scale-justice-weight-health-2634795/ (cc0)

So, let's recap our findings.

Once again learning rate controls the size of the step in the weight space.

- If the steps are too small, training will take a long time.
- On the other hand, if the steps are too large, we will bounce around and could even miss the optimal point.

A learning rate of 0.001 means a step size equal 1/1000 of the input space. This could be too small of a learning rate when you have a large optimization surface.

For instance, the default value for LinearRegressor estimator in TF library is set to 0.2 or 1 over square root of the number of features. This assumes your feature and label values are small numbers.

Andrej Karpathy recommends a learning rate on small model of 1e-3 (0.001), or on bigger models 3e-4 (0.0003).

The batch size controls the number of samples that gradient is calculated on.

If too small, training will bounce around

If too large, training will take a very long time

Batch sizes of 32, 64, 128, 256, or 512 are fairly common.

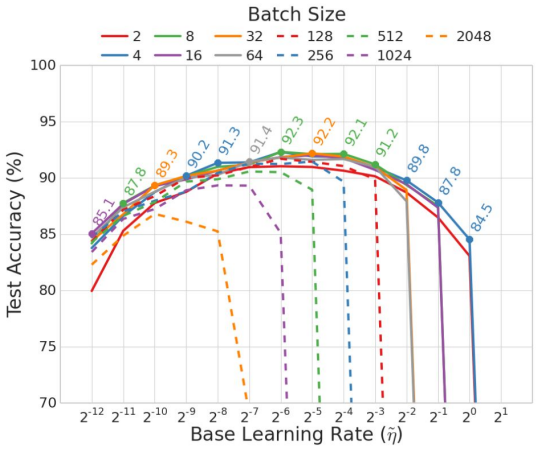
The other knob being batch size controls the number of samples that gradient is calculated on.

If batch size is too small, we could be bouncing around because the batch may not be a good enough representation of the input.

On the other hand, If batch size is too large, training will take a very long time.

As a rule of thumb, 40-100 tends to be a good range for batch size. It can go up to as high as 500.

Larger batch sizes require smaller learning rates



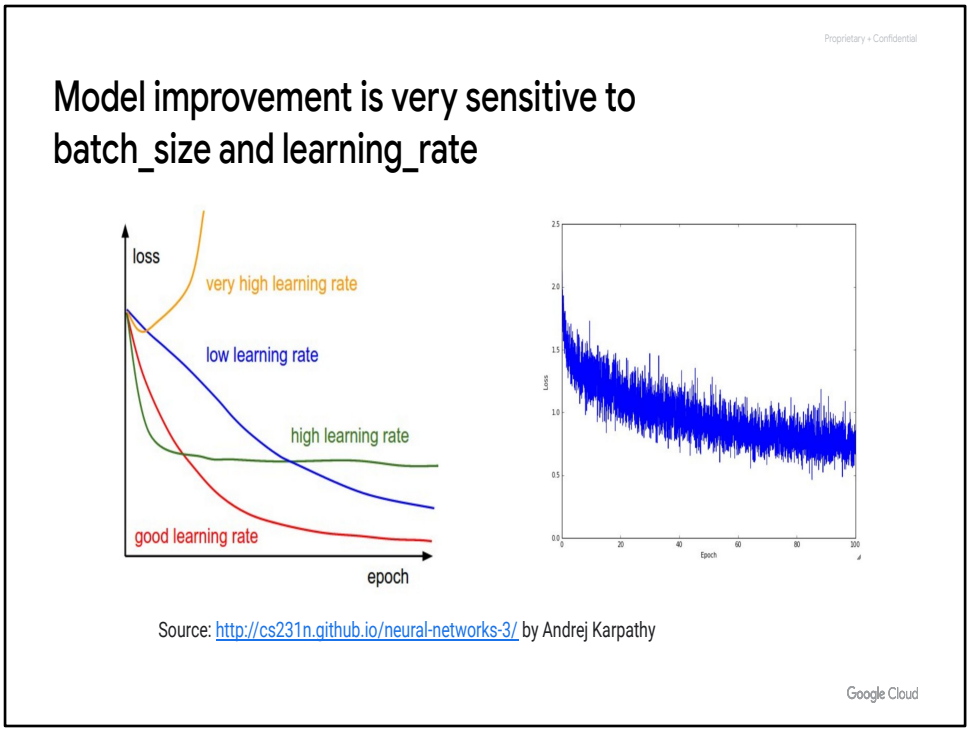
Revisiting Small Batch Training for Deep Neural Networks, Masters and Luschi, 2018

Larger batch sizes require smaller learning rates.

However, deep learning is still an art, and there are no hard and fast rules for picking a learning rate and batch size.

In fact, recent research suggests small mini-batch sizes provide more up-to-date gradient calculations, which yields more stable and reliable training.

And in experimental results for the CIFAR-10, CIFAR-100 and ImageNet datasets - the best performance has been consistently obtained for mini-batch sizes between m = 2 and m = 32. So it may be better to start with a smaller batch size.



In the previous module, we manually played with some of those hyper-parameters. For instance, we learnt that batch_size and learning_rate matter! Here, I have some graph's from Andrej Karpathy's great article that I recommend you review at your leisure. He visualizes the problem so well.

As you see on the left, at a low learning rate, like the blue graph here, improvement is linear. But you often don't get the best possible performance.

At a high learning rate, like the green graph here, you get exponential improvement at first, but you often don't find the best possible performance. At a very high learning rate, like this yellow graph, you can get completely lost!

There's often a goldilocks learning rate, like this red one here, but good luck finding it!

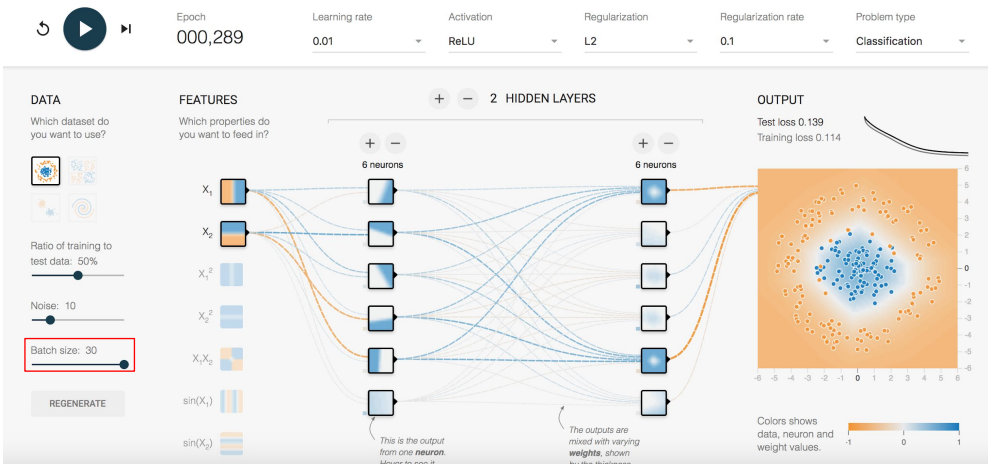
Now let's see what these graphs tell us about batch_size.

On the right, you see a very noisy loss curve, and that's due to the small batch_size. From previous module, you should remember that setting the batch-size too large can dramatically slow things down.

One thing to note though: these graphs are by epoch, but unfortunately, TensorFlow doesn't know much about epochs. You'll have to figure out the epoch by calculating how many steps of given batch-size will equate 1 epoch.

In other words, you need to find out how many steps of given batch-size will be required to travers your dataset once.

Experimenting with hyperparameters manually



Starting with learning rate.

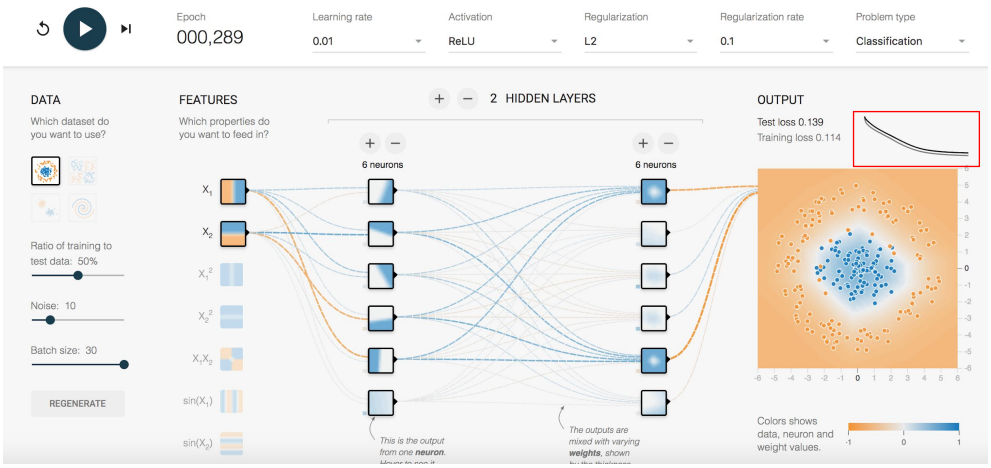
Remember learning rate controls the size of the step in the weight space.

Keeping batch size = 30 and all other parameters constant,

Experimenting with hyperparameters manually

I first set the learning rate to 0.01.

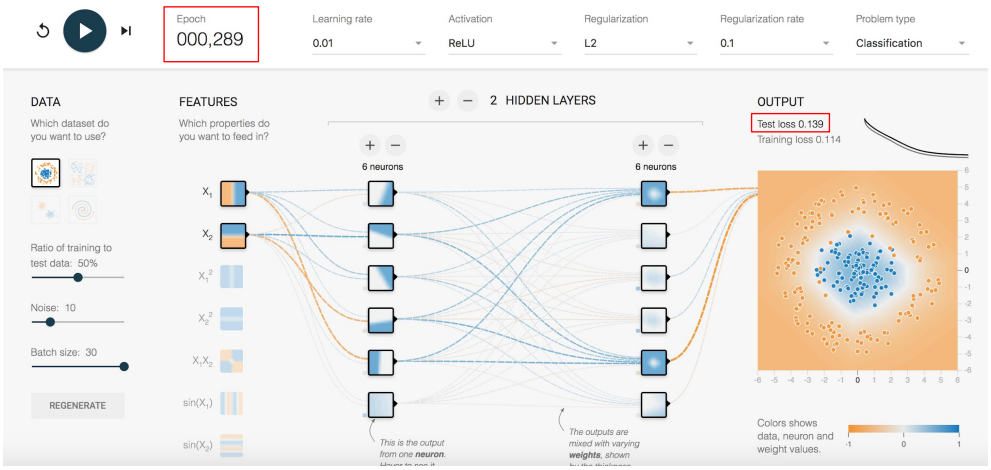
Experimenting with hyperparameters manually



Tensorflow playground uses random starting points, so your result might be different than mine.

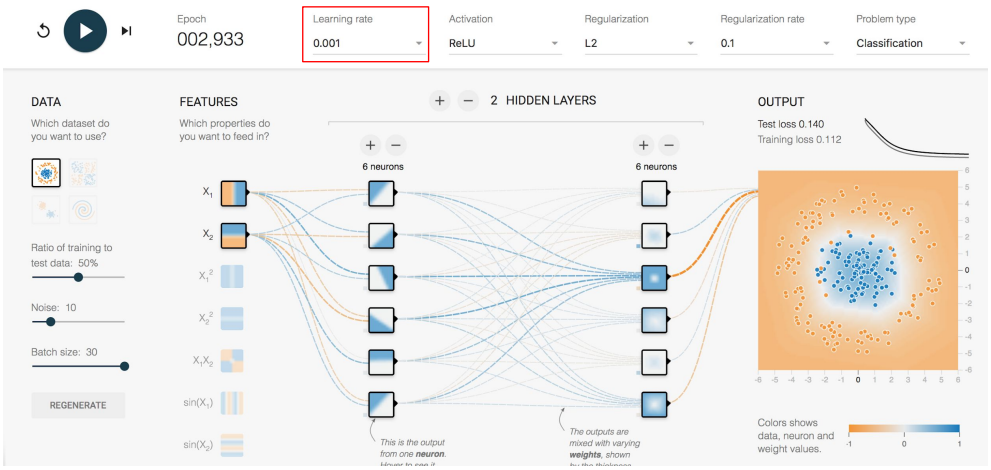
You may notice funny bounces on the loss curve, but it converges pretty fast.

Experimenting with hyperparameters manually



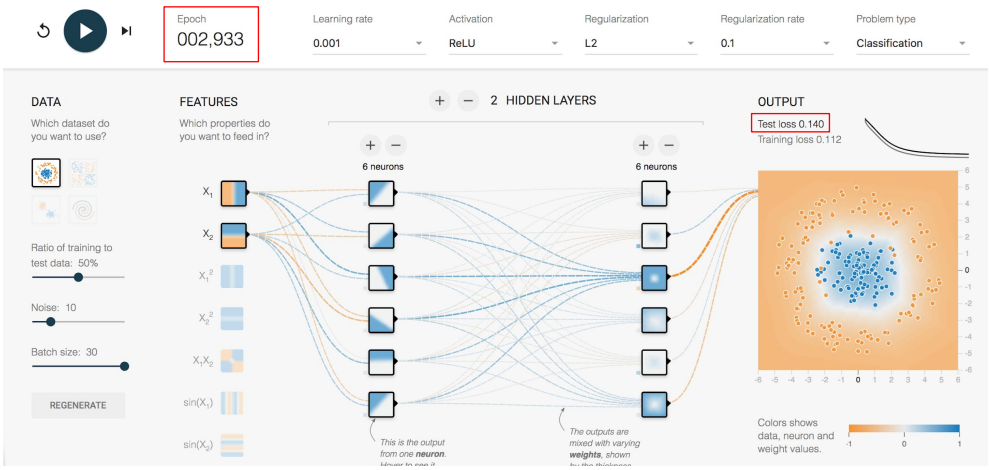
In my case I got to 0.139 loss value on test data in less than 300 epochs.

Experimenting with hyperparameters manually



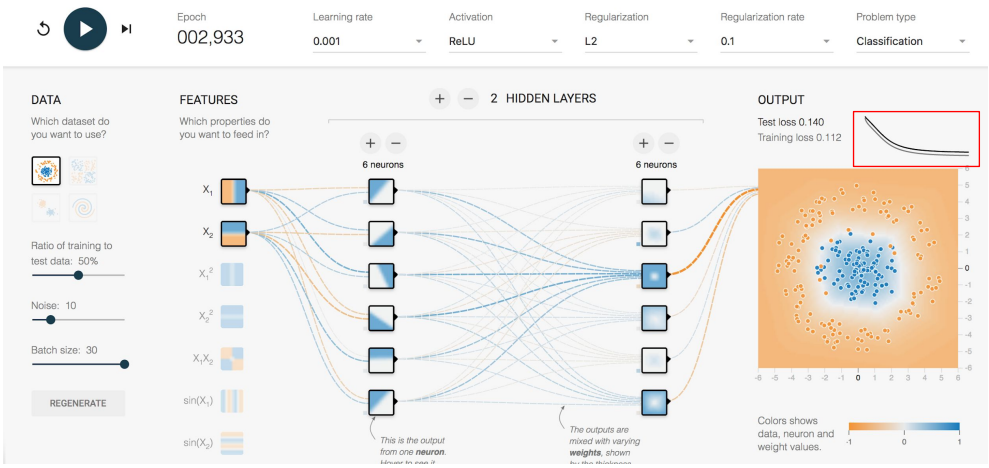
By changing the learning rate to 0.001, I saw much slower performance.

Experimenting with hyperparameters manually



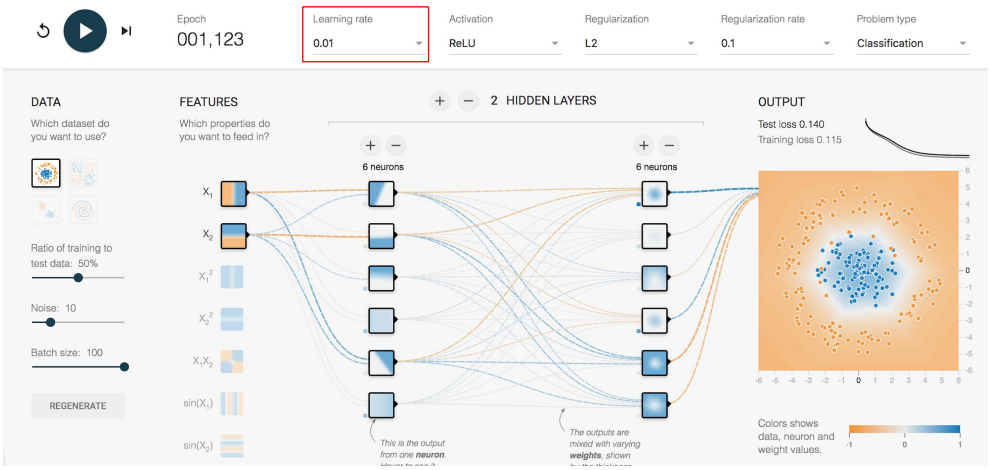
In my case, it took almost 3000 epochs to reach a test loss comparable to the previous experiment.

Experimenting with hyperparameters manually



On the bright side, you should not see any crazy bounces on the loss curve. It should converge slowly but smoothly.

Experimenting with hyperparameters manually

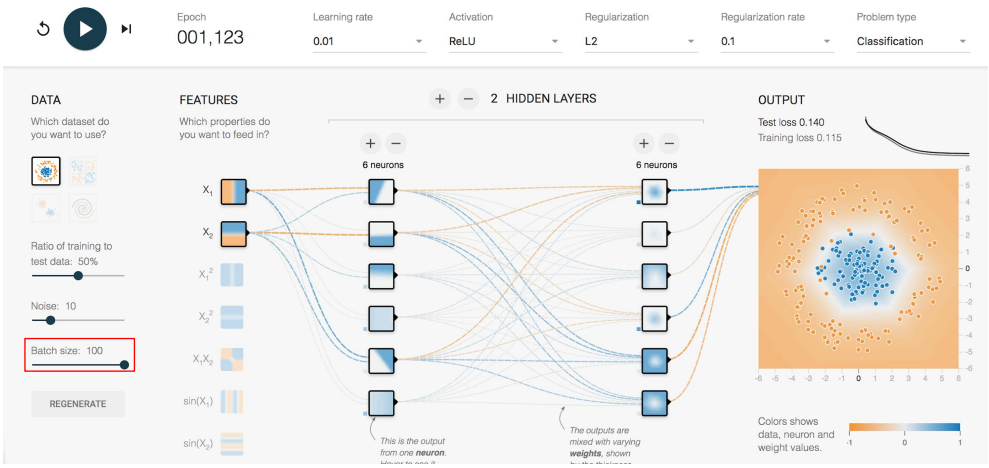


Now let's experiment with the effects of batch size.

Remember batch size controls the number of samples that the gradient is calculated on.

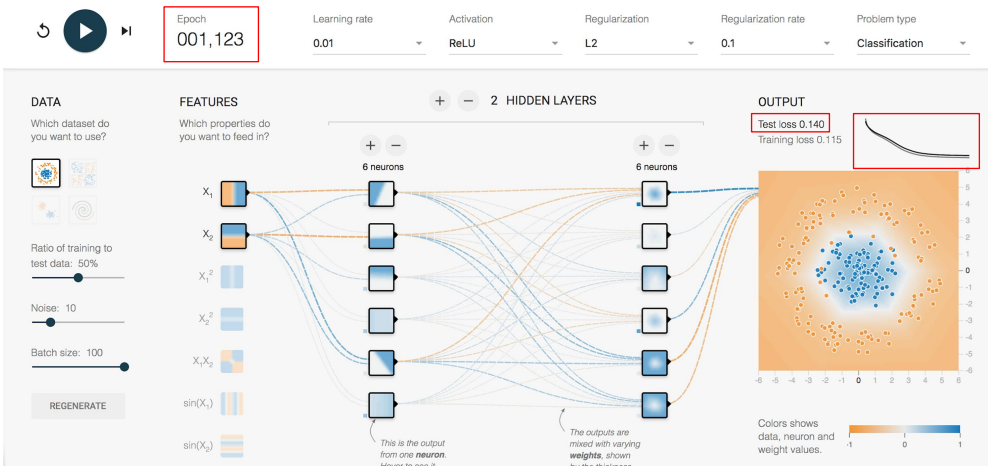
Keeping learning rate as 0.01 and all the other parameters constant, I first tried batch size = 100.

Experimenting with hyperparameters manually



If you are playing along, you may be scratching your head at this point as how to increase batch size beyond 30. Don't worry it's not broken; it's by design. The UI doesn't allow you to go beyond 30, but you can change it in the URL.

Experimenting with hyperparameters manually

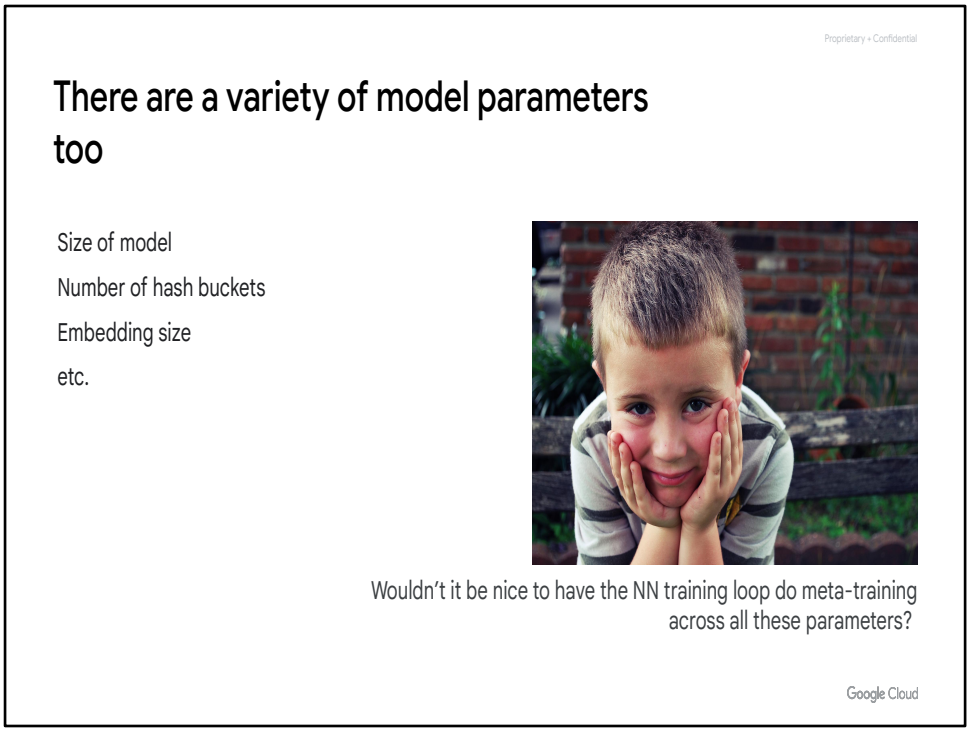


With batch size = 100, I noticed rather slow convergence; it took more than 1000 epochs to reach to similar loss value as previous experiments. But there were no noisy steps.

Experimenting with hyperparameters manually

When reducing the batch size to 5, I got very fast results. Basically, in only 65 epochs I reached similar test loss as previous experiments.

But there were some noisy steps visible on the loss curve.



Thinking about all the knobs and levers and finding the goldilocks combination that is data dependant sounds like a daunting task!

Just think about the permutations! You could automate it using any number of grid search algorithms,

but the search for the right combination can take forever and burn many hours of computational resources.

Wouldn't it be nice to have a training loop do meta-training on all these hyper-parameters and find the setting that is just-right?

Dimensionality:

In machine learning, dimensionality refers to the number of features or attributes that are used to represent each data point in a dataset. For example, in a tabular dataset, each row corresponds to a data point, and the columns represent different features or variables. The total number of columns (features) is the dimensionality of the dataset.

High dimensionality refers to datasets with a large number of features, which can lead to computational challenges and the "curse of dimensionality." The curse of dimensionality refers to problems that arise when working with high-dimensional data, such as increased computational complexity, increased data storage requirements, and difficulty in visualizing and interpreting the data.

Reducing dimensionality is a common preprocessing step in machine learning to overcome the challenges of high-dimensional data. Techniques like feature selection and feature extraction methods are used to reduce the number of features while retaining relevant information. Dimensionality reduction methods aim to transform the data into a lower-dimensional space while preserving the most important patterns and relationships.

- **Analogical Explanation (The Data Book Analogy):**

Imagine you have a large book containing information about various animals. Each page in the book represents a different animal, and the content on each page describes the characteristics of that animal. The book is organized such that each animal has its own set of attributes written down. For example, the attributes might include the animal's size, weight, height, speed, and various other characteristics.

Now, the dimensionality of this "data book" corresponds to the number of attributes mentioned for each animal. If the book is quite thick and each animal has many attributes listed, the dimensionality is high. On the other hand, if the book is relatively thin and each animal has only a few attributes mentioned, the dimensionality is low.

As you can imagine, having a high-dimensional data book can be challenging to manage. It would be cumbersome to flip through all the pages and compare animals based on all their attributes. So, to make things more manageable, you decide to create a summary booklet. In this summary booklet, you extract the most important and distinguishing attributes for each animal. By doing this, you effectively reduce the dimensionality of the data, making it easier to work with and understand.

Embedding size:
In machine learning, an embedding size refers to the dimensionality of the learned representation for each item (e.g., word, image, or user) in a dataset. It is a hyperparameter that determines the number of features or dimensions in the embedded space. When data is fed into a machine learning model, it is often represented in a high-dimensional space, which can be computationally expensive and lead to overfitting. Embeddings help to reduce the dimensionality while preserving relevant information.

For instance, in natural language processing (NLP), words from a vocabulary are typically represented as one-hot encoded vectors, where each word is a sparse vector with a length equal to the vocabulary size. This can be very high-dimensional, especially for large vocabularies. By using word embeddings, each word is mapped to a dense vector of a fixed length, which is the embedding size. The embedding captures the semantic meaning of the word and enables the model to better understand relationships between words.

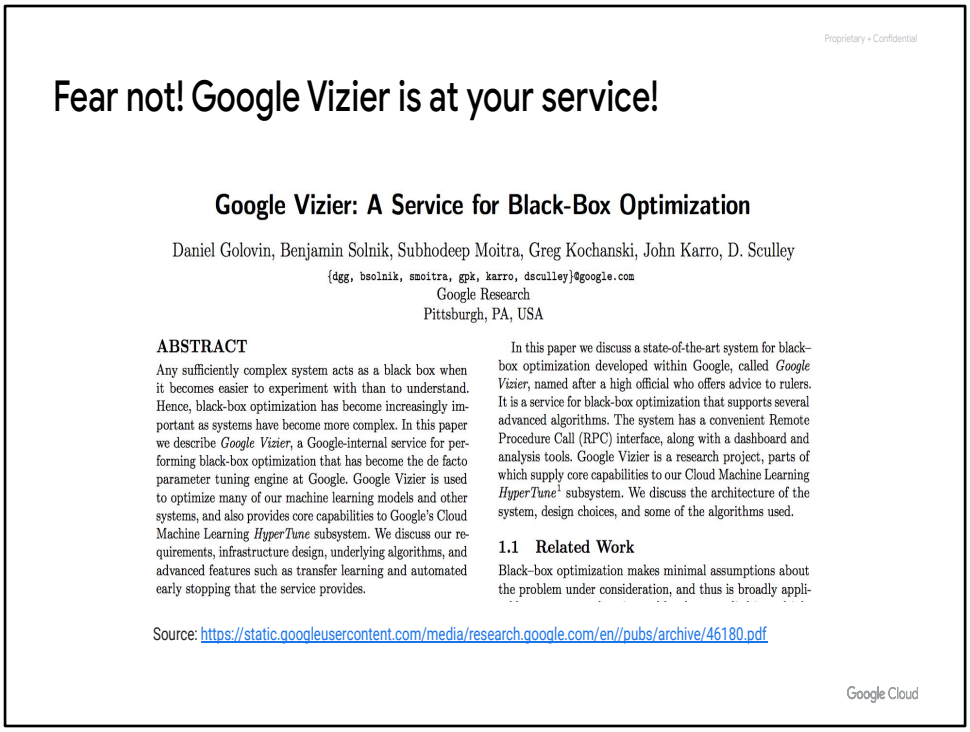
- **Analogical Explanation (The Address Analogy):**

Imagine you have a massive city with countless streets, and each street is named

after a unique word from a vast language. Navigating this city is challenging because the street names (words) are all over the place, and it takes a long time to travel between different streets.

To simplify navigation and make the city more organized, the city planners decide to create a new, smaller city. In this new city, they build a limited number of avenues (dimensions), each labeled with a number (index). Each avenue represents a specific aspect of the words in the original city. For example, one avenue might represent the sentiment of words, another avenue might represent their category, and so on.

Now, instead of referring to the entire street name (word), you only need to remember the avenue number (index) it corresponds to. This is the concept of word embedding. The embedding size is the number of avenues (dimensions) in the new city. By mapping each word to its corresponding avenue numbers (embedding), you can represent and understand the relationships between words more efficiently. Traveling around this new city is faster and less cumbersome, similar to how machine learning models can process data more effectively using embeddings.



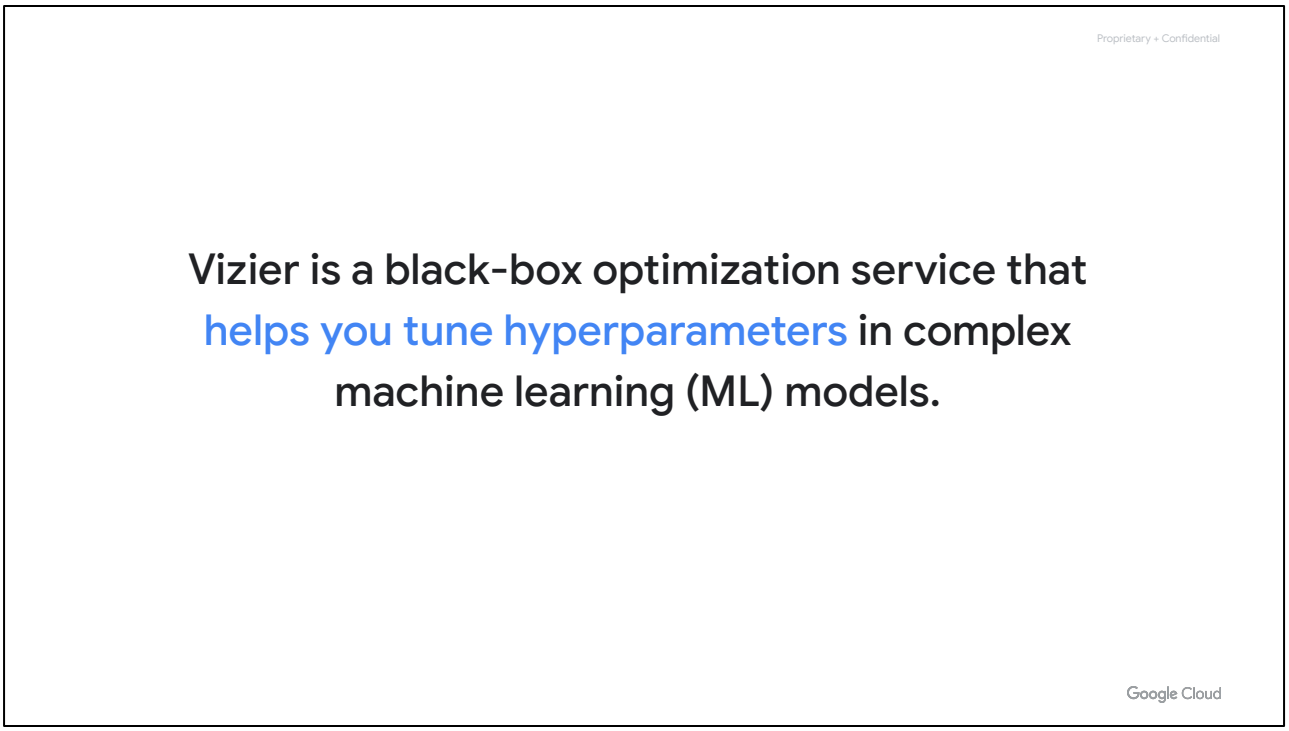
Fear not! Google Vizier is at your service!

For the most part, we will enjoy automagic hyperparameter tuning that is powered by Google Vizier algorithm, without needing to know about the details.

If you are curious to know what's going on inside the black-box, I'd recommend reviewing the research paper via the link that's on the screen.

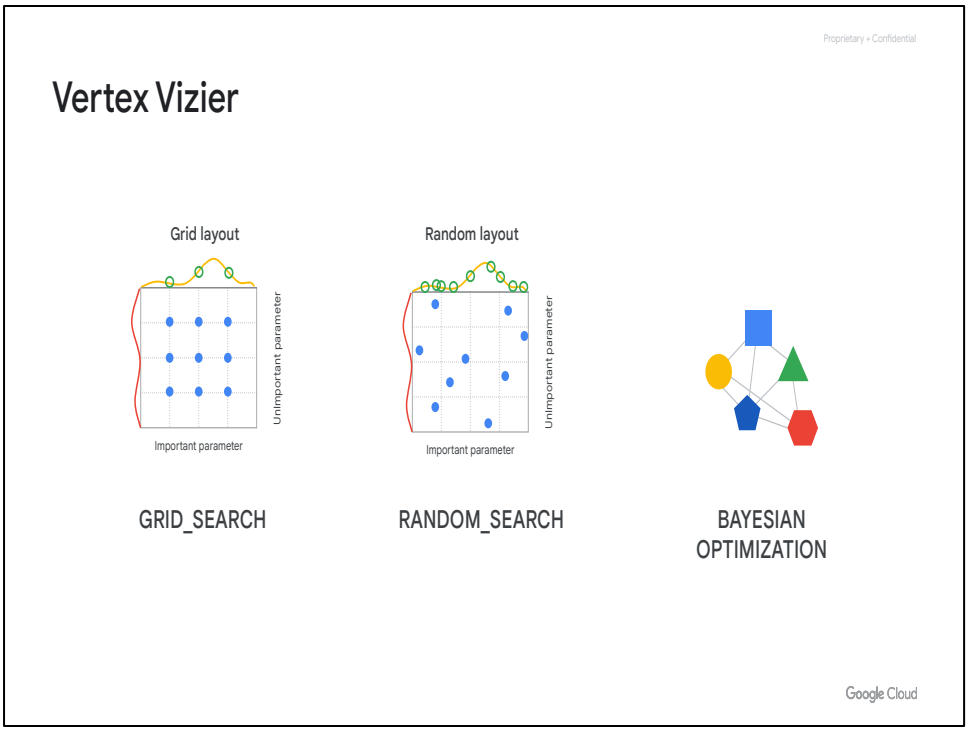
All you need to know is that Cloud AI Platform takes the burden away! You just need to configure your job properly and let AI Platform do the heavy lifting.

Let's see what it takes to get some hyper-parameters tuned for us the automagic way!



Source: Machine Learning on Google Cloud v3.0

To summarize, Vertex Vizier is a black-box optimization service that helps you tune hyperparameters in complex machine learning (ML) models.

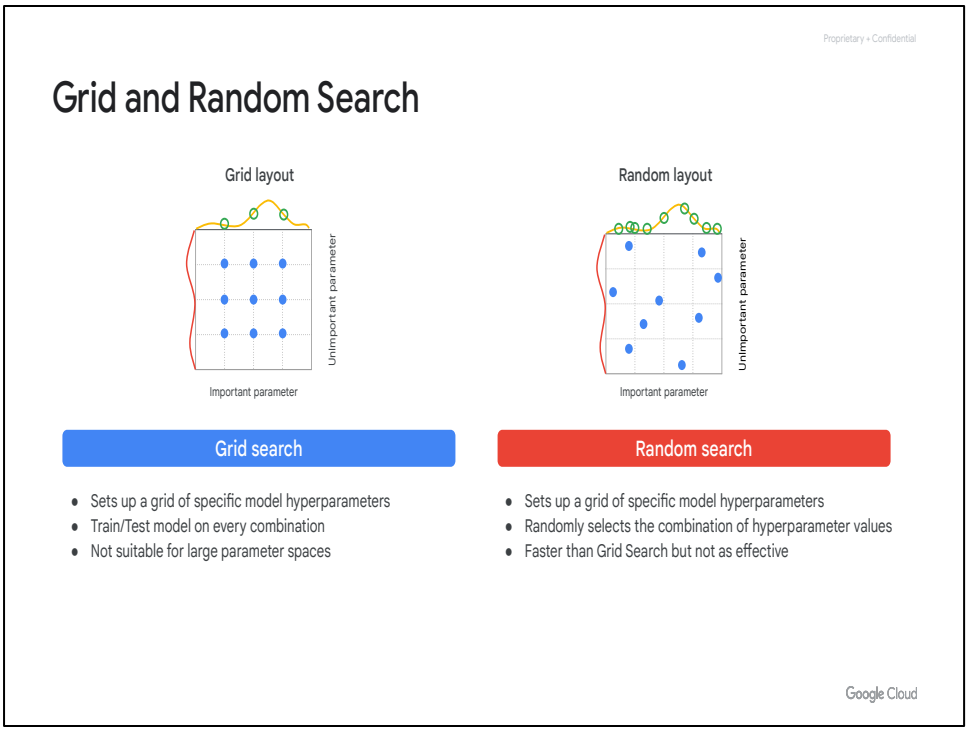


Source: Machine Learning on Google Cloud v3.0

Vertex Vizier offers Grid Search, Random Search, and Bayesian Optimization. If you do not specify an algorithm, Vizier uses the default algorithm. The default algorithm applies Bayesian optimization to arrive at the optimal solution with a more effective search over the parameter space.
The Grid Search option is useful if you want to specify a quantity of trials that is greater than the number of points in the feasible space.

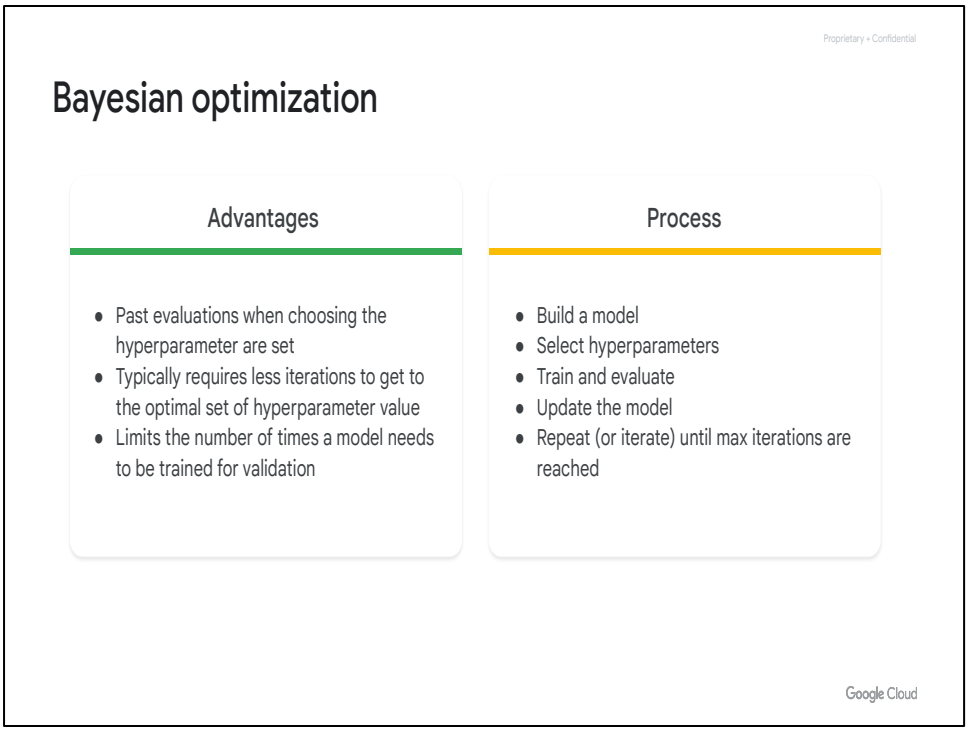
To use grid search, all parameters must be of type INTEGER, CATEGORICAL, or DISCRETE.

The Random Search option provides a simple random search within the feasible space.



Source: Machine Learning on Google Cloud v3.0

Previously, we identified two hyperparameter tuning methods: Grid Search and Random Search. Grid Search is a very traditional technique for hyperparameter tuning. In the Grid Search method, we can set up a grid of specific model hyperparameters and then train/test our problem statement model on every combination of values.
In Random Search, we set up a grid of specific model hyperparameter values (the same as with the grid search), but here we select the combination of hyperparameter values randomly. A Random Search tuning technique is faster than a Grid Search, but a Grid Search is more effective than a Random Search because a Random Search misses a few combinations.
Both Grid Search and Random Search are time-consuming techniques because they roam the full space of available parameter values in an isolated way without paying attention to past results. And neither Grid nor Random Search uses prior information from the past experiments to select the next set of hyperparameter value combinations.



Source: Machine Learning on Google Cloud v3.0

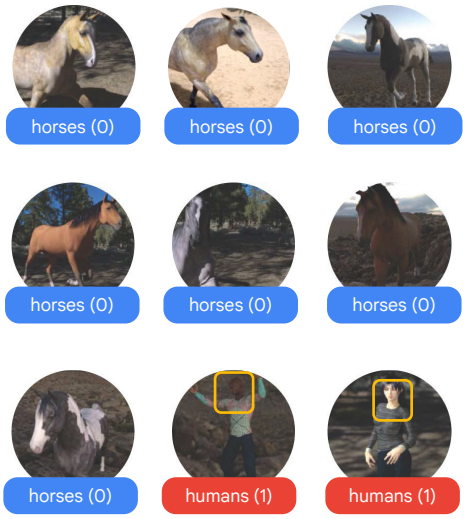
Bayesian optimization is another method of hyperparameter tuning that takes into account past evaluations when choosing which hyperparameter set to evaluate next.

This approach typically requires fewer iterations to get to the optimal set of hyperparameter values, most notably because it disregards those areas of the parameter space that it believes won't produce useful results.

This in turn limits the number of times a model needs to be trained for validation, because only those settings that are expected to generate a higher validation score are passed through for evaluation.

We optimize on a single objective

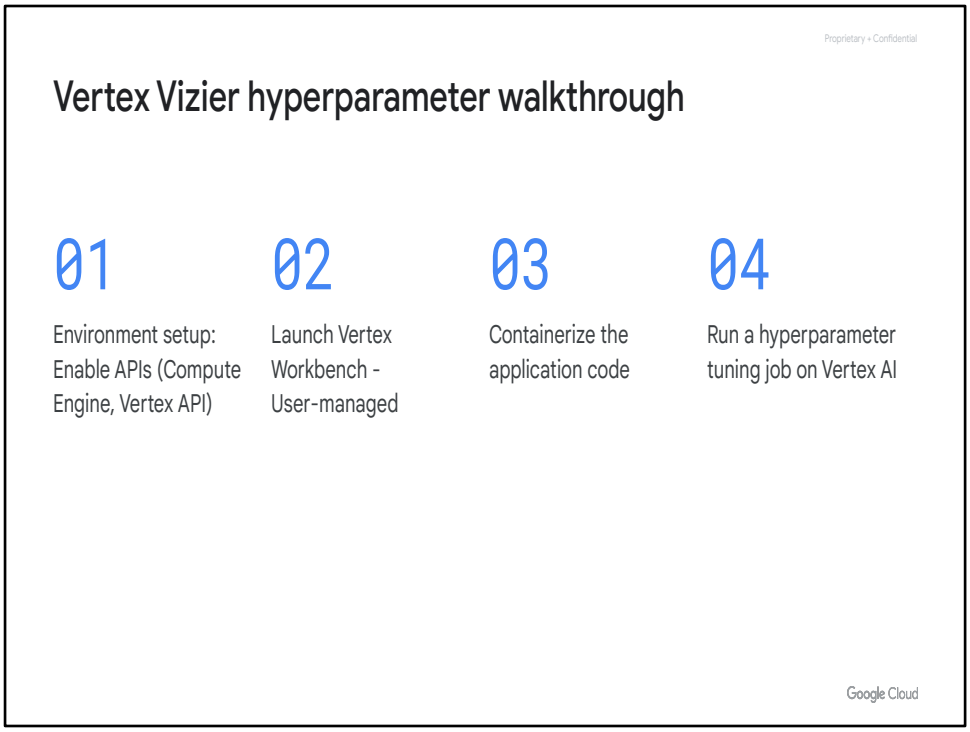
For example accuracy on classification of "Horses" or "Humans"



Source: Machine Learning on Google Cloud v3.0

Let's review an example of the single objective metric.

The dataset is an image classification model trained on the horses or humans dataset from TensorFlow Datasets.



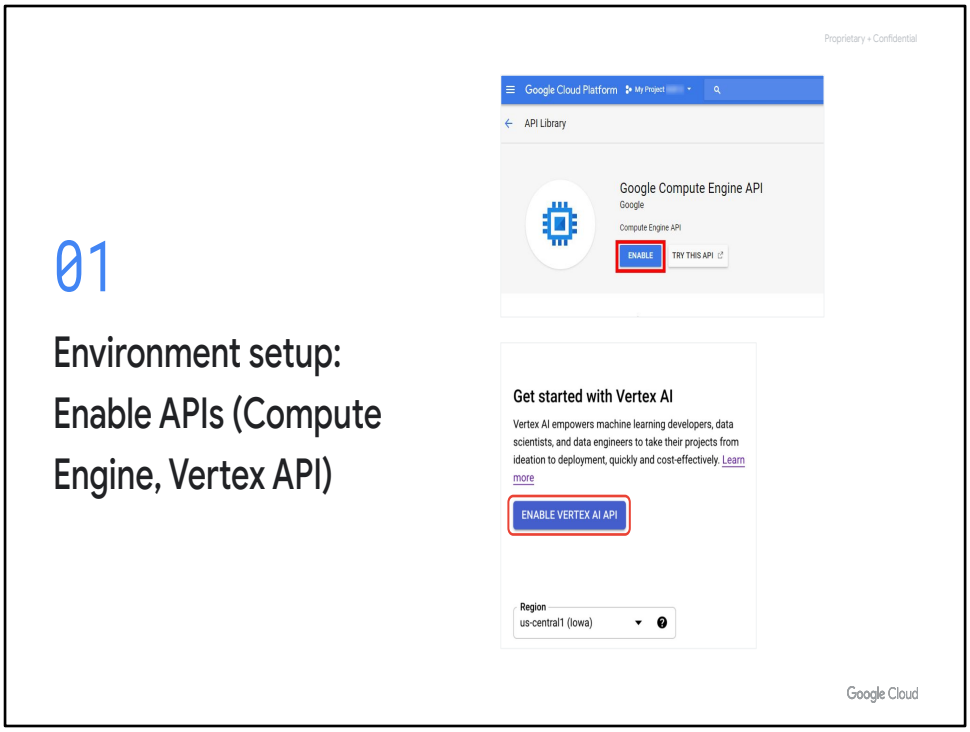
Source: Machine Learning on Google Cloud v3.0

Here's an overview of how to accomplish hyperparameter tuning using Vertex Vizier.

First, set up your environment by enabling the Compute Engine API and the Vertex API.

Next, launch the Vertex Workbench,
containerize the application code, and run a hyperparameter tuning job on Vertex AI.

Now, let’s explore each step in more detail.

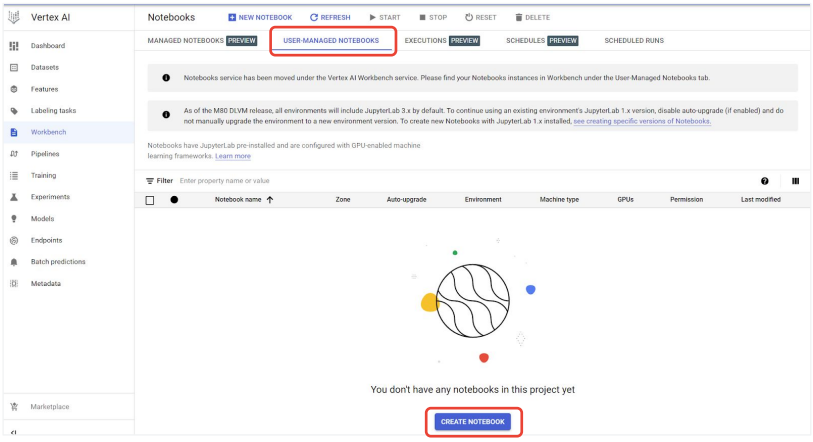


Source: Machine Learning on Google Cloud v3.0

Enable the Compute Engine API and the Vertex API to set up your environment.

02

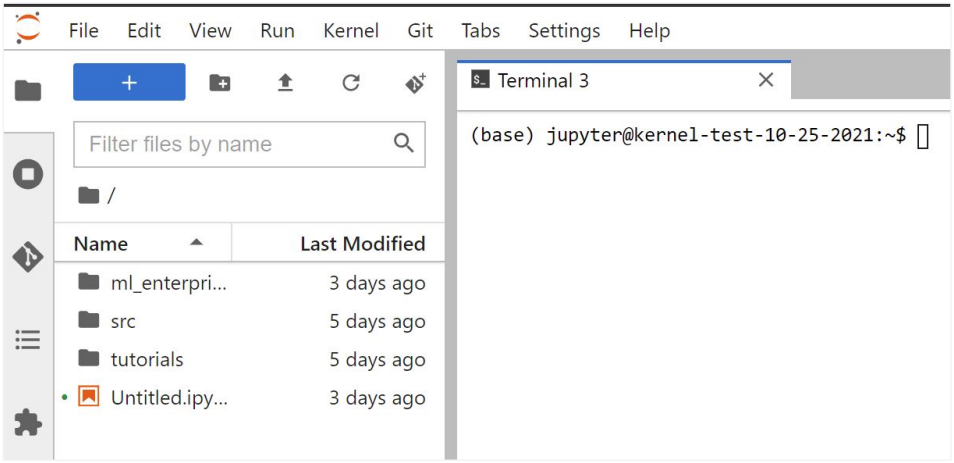
Launch Vertex Workbench - User-managed



Source: Machine Learning on Google Cloud v3.0

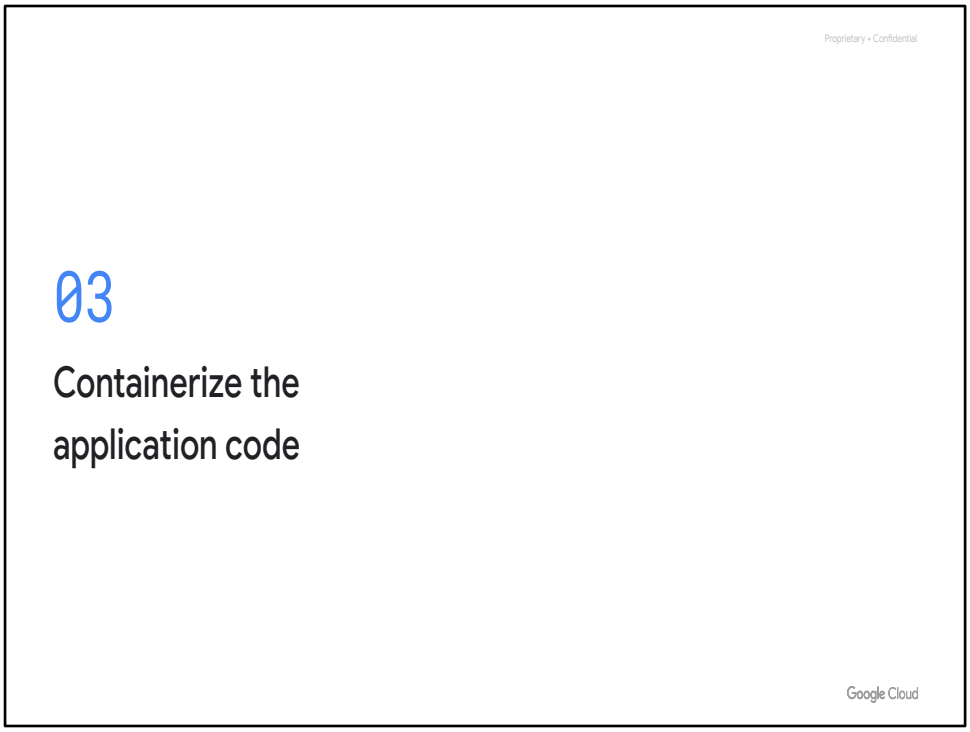
To create a user-managed notebook, select Workbench from the Navigation menu. In the lab, you'll need to wait for Notebook to spin up.

User-managed notebook



Source: Machine Learning on Google Cloud v3.0

After the user-managed Notebook is opened and you've launched a Notebook, you'll need to launch a terminal window so that you can

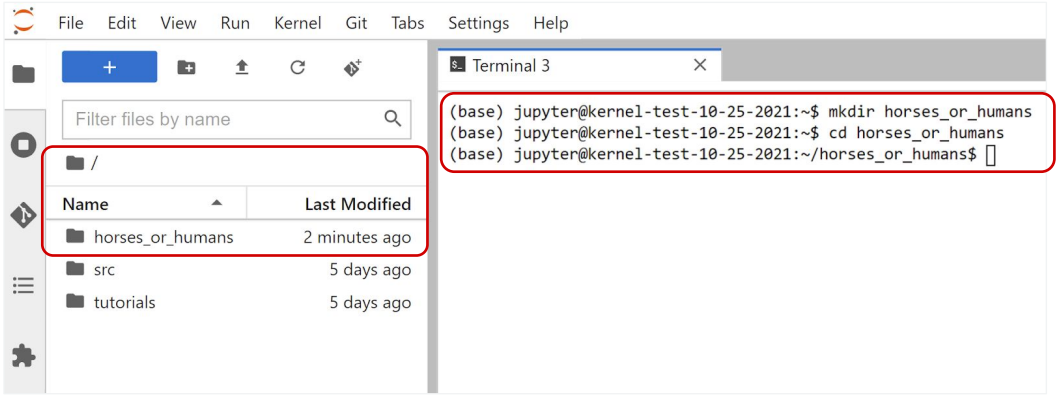


Source: Machine Learning on Google Cloud v3.0

containerize the application code.

You'll submit this hyperparameter tuning job to Vertex by putting your training application code in a Docker container and pushing this container to Container Registry. Using this approach, you can tune hyperparameters for a model built with any framework.

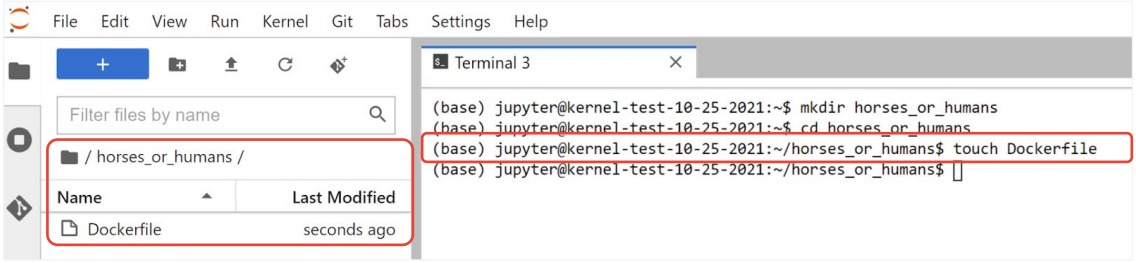
Create a new directory



Source: Machine Learning on Google Cloud v3.0

In the terminal window, create a new directory called horses_or_humans and open it with “change directory.”

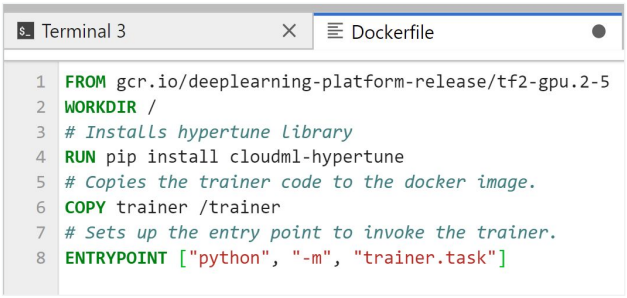
Create a Dockerfile



Source: Machine Learning on Google Cloud v3.0

The first step in containerizing your code is to create a Dockerfile. In the Dockerfile, you'll include all the commands needed to run the image. It will install all the necessary libraries, including the CloudML Hypertune library, and set up the entry point for the training code. From your Terminal, create an empty Dockerfile.

This Dockerfile uses the Deep Learning Container TensorFlow Enterprise 2.5 GPU Docker image

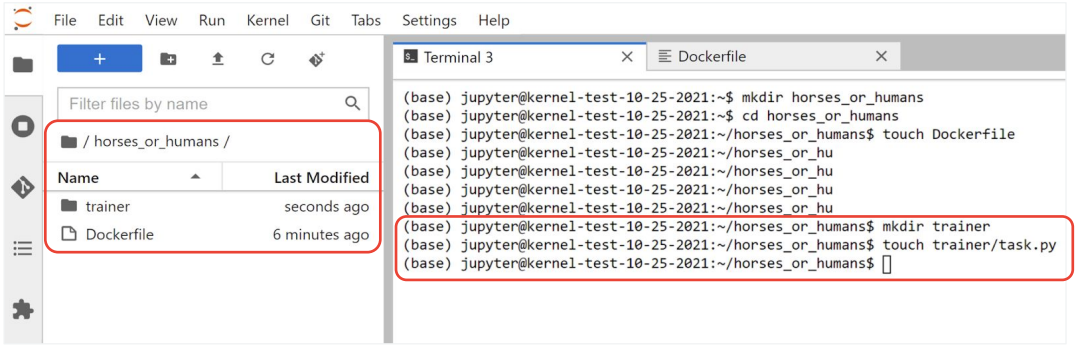


Source: Machine Learning on Google Cloud v3.0

Open the **Dockerfile** from the left menu and copy the commands you'll need to run the image.

This Dockerfile uses the Deep Learning Container TensorFlow Enterprise 2.5 GPU Docker image. The Deep Learning Containers on Google Cloud come with many common ML and data science frameworks pre-installed. After downloading that image, this Dockerfile sets up the entrypoint for the training code. You haven't created these files yet: in the next step, you'll add the code for training and tuning the model.

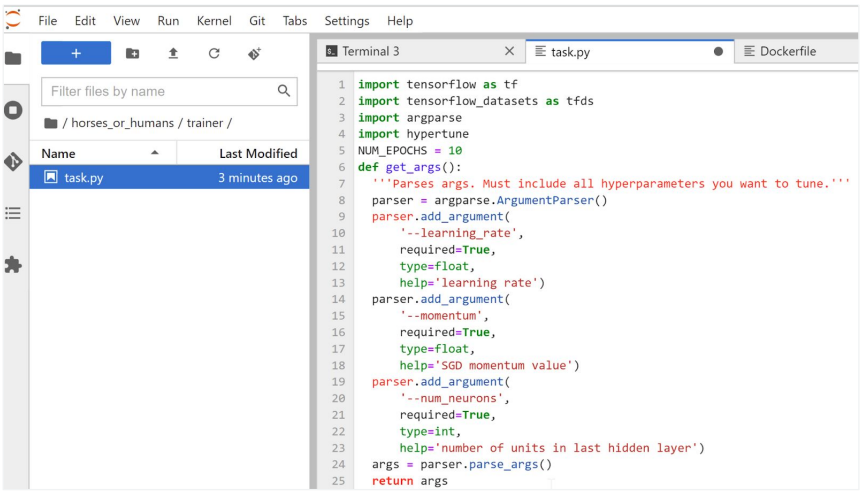
Add the model code



Source: Machine Learning on Google Cloud v3.0

From your Terminal, create a directory for the training code and a Python file where you'll add the code.

Open the task.py file



Source: Machine Learning on Google Cloud v3.0

Next, open the **task.py** file you just created and copy the code into the file.

Before you build the container, let's take a deeper look at the code. A few components are specific to using the hyperparameter tuning service.

Code specific to hyperparameter tuning

The script imports the hypertune library. Note that the Dockerfile from step 1 included instructions to pip install this library.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import argparse
import hypertune
NUM_EPOCHS = 10
def get_args():
    '''Parses args. Must include all hyperparameters you want to tune.'''
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--learning_rate',
        required=True,
        type=float,
        help='learning rate')
    parser.add_argument(
        '--momentum',
        required=True,
        type=float,
        help='SGD momentum value')
    parser.add_argument(
        '--num_neurons',
        required=True,
        type=int,
        help='number of units in last hidden layer')
    args = parser.parse_args()
    return args
def preprocess_data(image, label):
    '''Resizes and scaled images.'''
    image = tf.image.resize(image, 150,150))
```


Source: Machine Learning on Google Cloud v3.0

The script imports the hypertune library. Note that the Dockerfile from Step 1 included instructions to pip install this library.

Code specific to hyperparameter tuning

The function get_args() defines a command-line argument for each hyperparameter you want to tune.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import argparse
import hypertune
NUM_EPOCHS = 10
def get_args():
    '''Parses args. Must include all hyperparameters you want to tune.'''
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--learning_rate',
        required=True,
        type=float,
        help='learning rate')
    parser.add_argument(
        '--momentum',
        required=True,
        type=float,
        help='SGD momentum value')
    parser.add_argument(
        '--num_neurons',
        required=True,
        type=int,
        help='number of units in last hidden layer')
    args = parser.parse_args()
    return args
def preprocess_data(image, label):
    '''Resizes and scaled images.'''
    image = tf.image.resize(image, 150,150))
```


Source: Machine Learning on Google Cloud v3.0

The function get_args() defines a command-line argument for each hyperparameter you want to tune. In this example, the hyperparameters that will be tuned are

Code specific to hyperparameter tuning

Learning rate

```
import tensorflow as tf
import tensorflow_datasets as tfds
import argparse
import hypertune
NUM_EPOCHS = 10
def get_args():
    '''Parses args. Must include all hyperparameters you want to tune.'''
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--learning_rate',
        required=True,
        type=float,
        help='learning rate')
    parser.add_argument(
        '--momentum',
        required=True,
        type=float,
        help='SGD momentum value')
    parser.add_argument(
        '--num_neurons',
        required=True,
        type=int,
        help='number of units in last hidden layer')
    args = parser.parse_args()
    return args
def preprocess_data(image, label):
    '''Resizes and scaled images.'''
    image = tf.image.resize(image, 150,150))
```


Source: Machine Learning on Google Cloud v3.0

learning rate, the momentum value in the optimizer,

Code specific to hyperparameter tuning

Number of neurons in the last hidden layer of the model

```
import tensorflow as tf
import tensorflow_datasets as tfds
import argparse
import hypertune
NUM_EPOCHS = 10
def get_args():
    '''Parses args. Must include all hyperparameters you want to tune.'''
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--learning_rate',
        required=True,
        type=float,
        help='learning rate')
    parser.add_argument(
        '--momentum',
        required=True,
        type=float,
        help='SGD momentum value')
    parser.add_argument(
        '--num_neurons',
        required=True,
        type=int,
        help='number of units in last hidden layer')
    args = parser.parse_args()
    return args
def preprocess_data(image, label):
    '''Resizes and scaled images.'''
    image = tf.image.resize(image, 150,150))
```


Source: Machine Learning on Google Cloud v3.0

and the number of neurons in the last hidden layer of the model; but you can experiment with others.

Code specific to hyperparameter tuning

The value passed in those arguments is then used to set the corresponding hyperparameter in the code.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import argparse
import hypertune
NUM_EPOCHS = 10
def get_args():
    '''Parses args. Must include all hyperparameters you want to tune.'''
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--learning_rate',
        required=True,
        type=float,
        help='learning rate')
    parser.add_argument(
        '--momentum',
        required=True,
        type=float,
        help='SGD momentum value')
    parser.add_argument(
        '--num_neurons',
        required=True,
        type=int,
        help='number of units in last hidden layer')
    args = parser.parse_args()
    return args
def preprocess_data(image, label):
    '''Resizes and scaled images.'''
    image = tf.image.resize(image, 150,150))
```


Source: Machine Learning on Google Cloud v3.0

The value passed in those arguments is then used to set the corresponding hyperparameter in the code.

If you have a model trained for three epochs with validation data and provided accuracy as a metric, the History.history attribute would look similar

```
  x = tf.keras.layers.MaxPooling2D((2, 2))(x)
  x = tf.keras.layers.Conv2D(64, (3, 3), activation='relu')(x)
  x = tf.keras.layers.MaxPooling2D((2, 2))(x)
  x = tf.keras.layers.Flatten()(x)
  x = tf.keras.layers.Dense(num_neurons, activation='relu')(x)
  outputs = tf.keras.layers.Dense(1, activation='sigmoid')(x)
  model = tf.keras.Model(inputs, outputs)
  model.compile(
      loss='binary_crossentropy',
      optimizer=tf.keras.optimizers.SGD(learning_rate, momentum=momentum),
      metrics=['accuracy'])
  return model
def main():
  args = get_args()
  train_data, validation_data = create_dataset()
  model = create_model(args.num_neurons, args.learning_rate, args.momentum)
  history = model.fit(train_data, epochs=NUM_EPOCHS, validation_data=validation_data)
  #DEFINE METRIC
  hp_metric = history.history['val_accuracy'][-1]
  hpt = hypertune.HyperTune()
  hpt.report_hyperparameter_tuning_metric(
       hyperparameter_metric_tag='accuracy',
       metric_value=hp_metric,
       global_step=NUM_EPOCHS)
  if __name__ == "__main__":
       main()
```


Source: Machine Learning on Google Cloud v3.0

At the end of the main() function, the hypertune library is used to define the metric you want to optimize. In TensorFlow, the keras model.fit method returns a History object. The History.history attribute is a record of training loss values and metrics values at successive epochs. If you pass validation data to model.fit, the History.history attribute will also include validation loss and metrics values. For example, if you trained a model for three epochs with validation data and provided accuracy as a metric, the History.history attribute would look similar

to this dictionary.

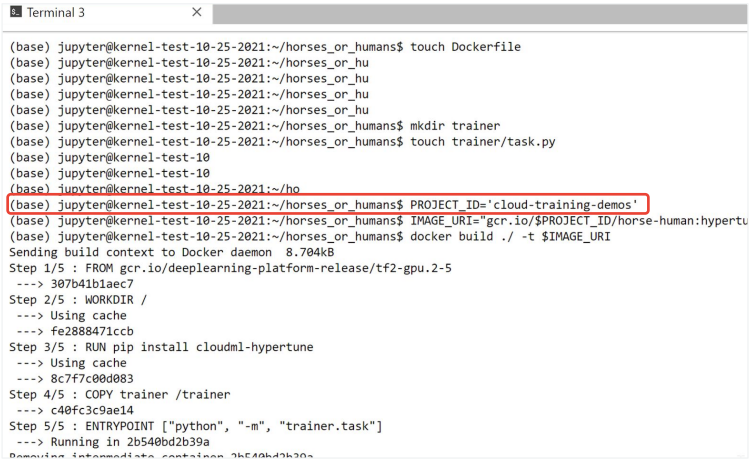
```
{
 "accuracy": [
   0.7795261740684509,
   0.9471358060836792,
   0.9870933294296265
 ],
 "loss": [
   0.6340447664260864,
   0.16712145507335663,
   0.04546636343002319
 ],
 "val_accuracy": [
   0.3795261740684509,
   0.4471358060836792,
   0.4870933294296265
 ],
 "val_loss": [
   2.044623374938965,
   4.100203514099121,
   3.0728273391723633
 ]
```


Source: Machine Learning on Google Cloud v3.0

to the dictionary on the slide, which shows the JSON output of a model trained for three epochs with validation data and accuracy as metrics.

Build the container

You can get your project ID by running gcloud config list --format 'value(core.project)' in your terminal.

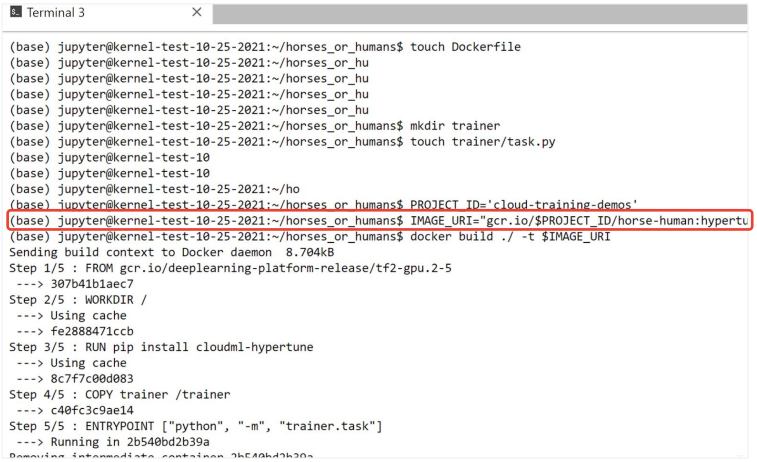


Source: Machine Learning on Google Cloud v3.0

Now that we've explored the code, let's build the container. From your Terminal, define an environment variable for your project,but replace your-cloud-project with your project ID.

Build the container

You can get your project ID by running gcloud config list --format 'value(core.project)' in your terminal.

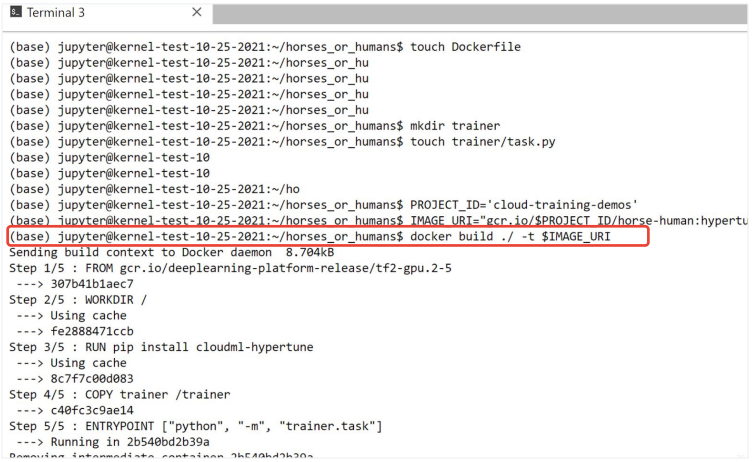


Source: Machine Learning on Google Cloud v3.0

Define a variable with the URI of your container image in Container Registry.

Build the container

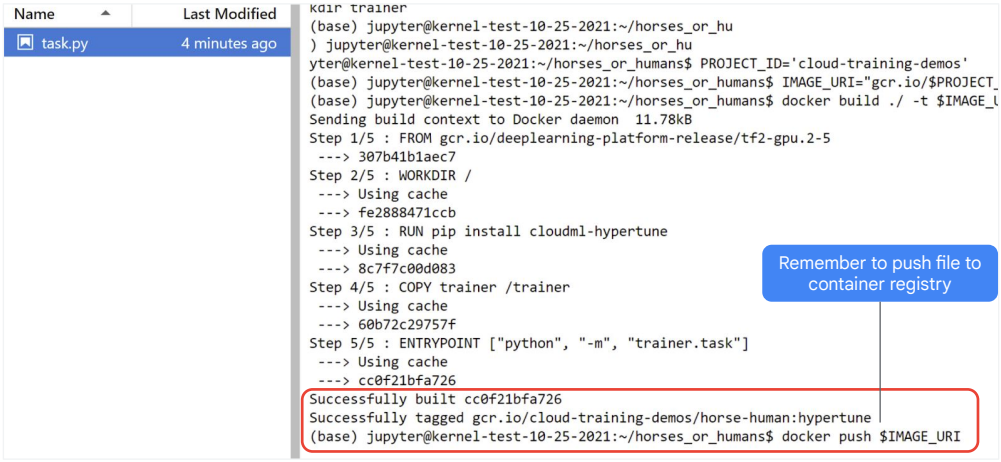
You can get your project ID by running gcloud config list --format 'value(core.project)' in your terminal.



Source: Machine Learning on Google Cloud v3.0

Then build the container by running docker build file in the root of your **horses_or_humans** directory.

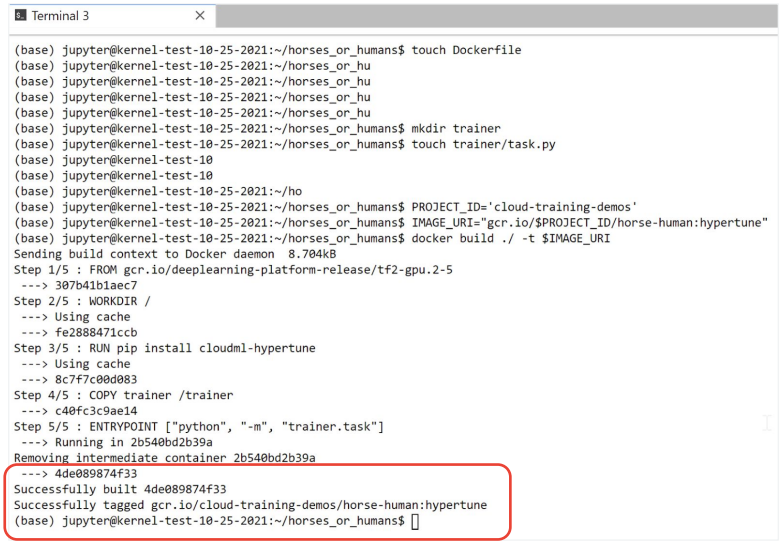
Push the file to the Container Registry after the build



Source: Machine Learning on Google Cloud v3.0

Now, push the file to the container registry after the build.

Push to Container Registry



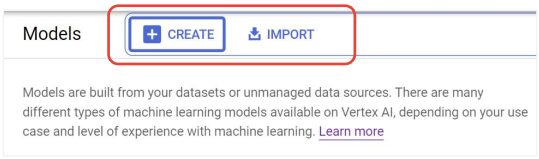
Source: Machine Learning on Google Cloud v3.0

With the container pushed to Container Registry, you're ready to kick off a custom model hyperparameter tuning job.

Source: Machine Learning on Google Cloud v3.0

Although our walkthrough uses a custom container on Container Registry, you can also run a hyperparameter tuning job with the Pre-built containers.

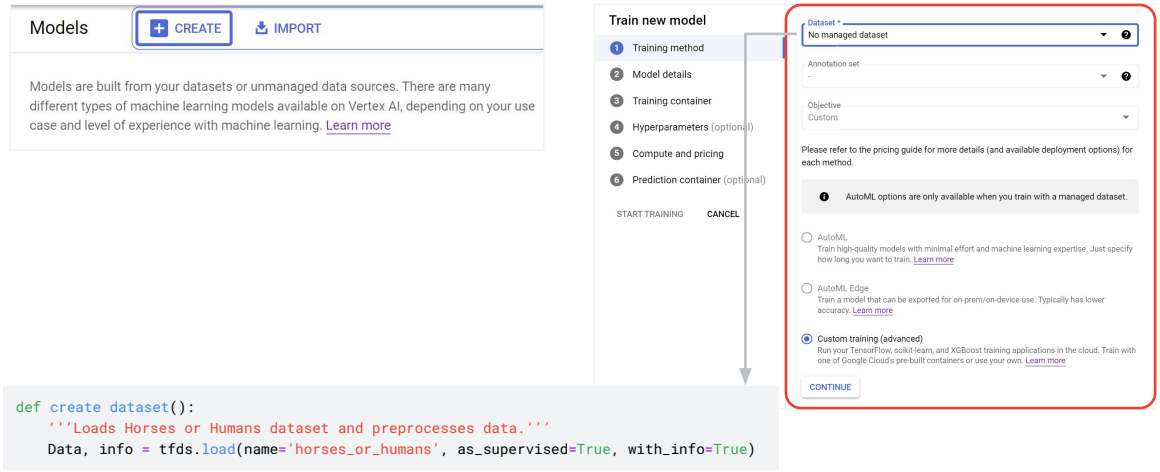
Configure training job



Source: Machine Learning on Google Cloud v3.0

Now you're ready to train your model. All you'll need to do is configure your training job. When a model is selected from the Vertex AI navigation menu, two options are offered: create and import. Select create to begin configuring your training job.

Configure training job



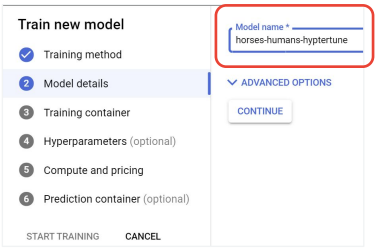

```
def create_dataset():
    '''Loads Horses or Humans dataset and preprocesses data.'''
    Data, info = tfds.load(name='horses_or_humans', as_supervised=True, with_info=True)
```


Source: Machine Learning on Google Cloud v3.0

Under Dataset, select “No managed dataset.” This is selected is because a function created the dataset to load and split the data and is contained in the task.py file.

After you select “Custom training (advanced)” as your training method, you can continue to configure more options.

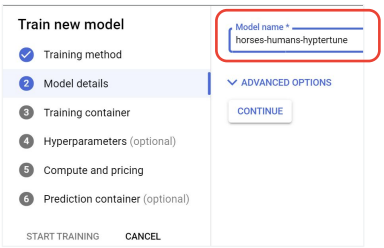
Configure training job

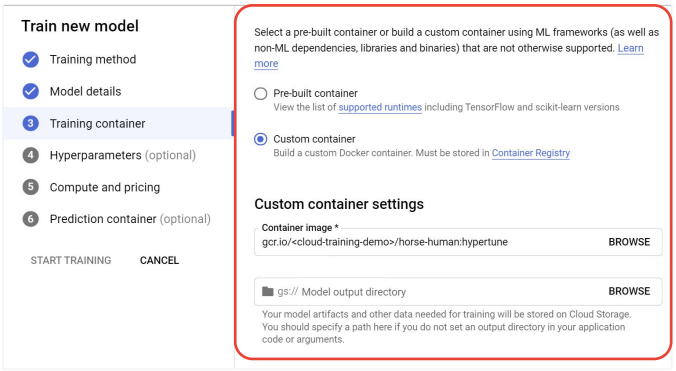


Source: Machine Learning on Google Cloud v3.0

In the Model details window, enter h**orses-humans-hyptertune** for **Model name**. Click **Continue**, which takes you to the

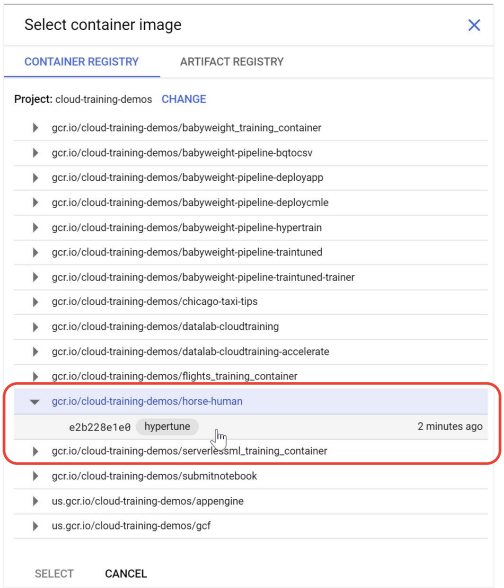
Select the container image





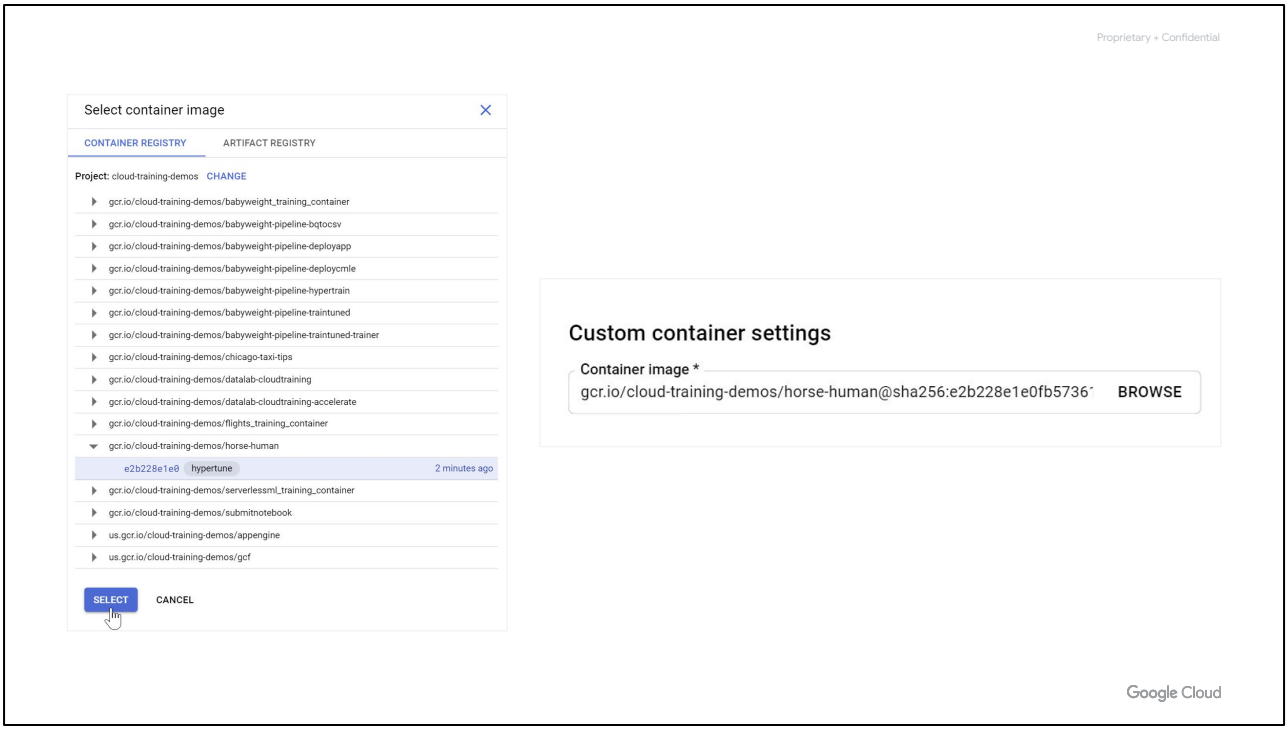
Source: Machine Learning on Google Cloud v3.0

Training container page. In the Training container step, select Custom container. In the Custom container settings, for Container image, browse to find your



Source: Machine Learning on Google Cloud v3.0

Container image.



Source: Machine Learning on Google Cloud v3.0

Select the horse-human:hypertune image, and then click Continue.

Configure three hyperparameters

```
import tensorflow as tf
import tensorflow_datasets as tfds
import argparse
import hypertune
NUM_EPOCHS = 10
def get_args():
    '''Parses args. Must include all hyperparameters you want to tune.'''
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--learning_rate',
        required=True,
        type=float,
        help='learning rate')
    parser.add_argument(
        '--momentum',
        required=True,
        type=float,
        help='SGD momentum value')
    parser.add_argument(
        '--num_neurons',
        required=True,
        type=int,
        help='number of units in last hidden layer')
    args = parser.parse_args()
    return args
def preprocess_data(image, label):
    '''Resizes and scaled images.'''
    image = tf.image.resize(image, 150,150))
```

Learning rate

Momentum

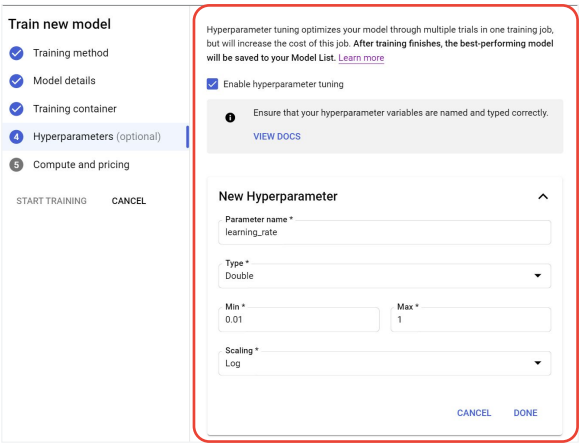
Number of neurons

Source: Machine Learning on Google Cloud v3.0

For our project, we'll configure three hyperparameters: learning rate, momentum, and number of neurons. Recall that these are the parameters set in the training application code of the task.py file.

Configure hyperparameter tuning job - Learning rate

- For **Paramenter name,** enter `learning_rate`
- For **Type**, select **Double**
- For **Min**, enter `0.01`, and for **Max,** enter `1`
- For **Scaling**, select **Log**
- Click **DONE**



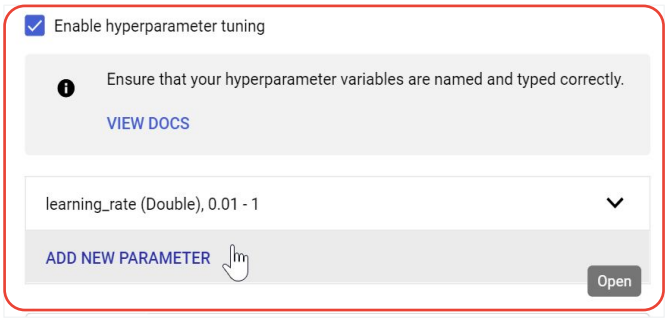
Source: Machine Learning on Google Cloud v3.0

On the Hyperparameter page, select **Enable hyperparameter tuning** to start configuring the hyperparameters. Essentially, you'll need to add the hyperparameters that you set as command line arguments in the training application code. When adding a hyperparameter, you first need to provide the name.

This should match the argument name that you passed to argparse. For **Parameter name**, enter learning_rate. For **type**, select **Double**. For **Min**, enter 0.01, and for **Max**, enter 1. For **Scaling**, select **Log.**

Click **DONE**. After adding the learning_rate hyperparameter, add parameters for momentum and num_neurons.

After adding the learning_rate hyperparameter, add parameters for momentum and num_neurons

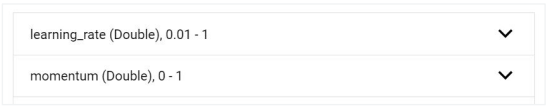


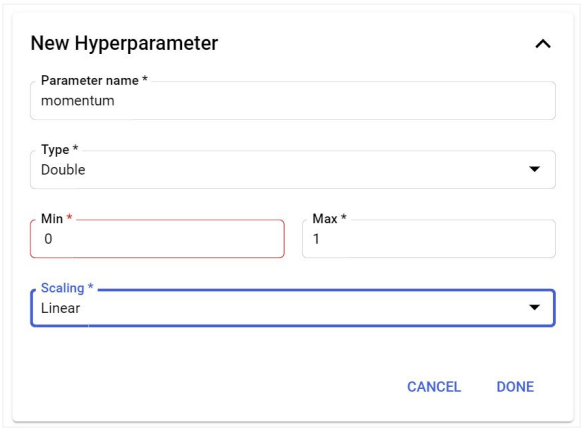
Source: Machine Learning on Google Cloud v3.0

- For **momentum**:
  - Click **ADD NEW PARAMETER**.

Momentum

- Click **ADD NEW PARAMETER**
- For **Parameter name**, enter `momentum`
- For **Type**, select **Double**
- For **Min**, enter `0` for **Min**, and for **Max,** enter `1`
- For **Scaling,** select **Linear**
- Click **DONE**



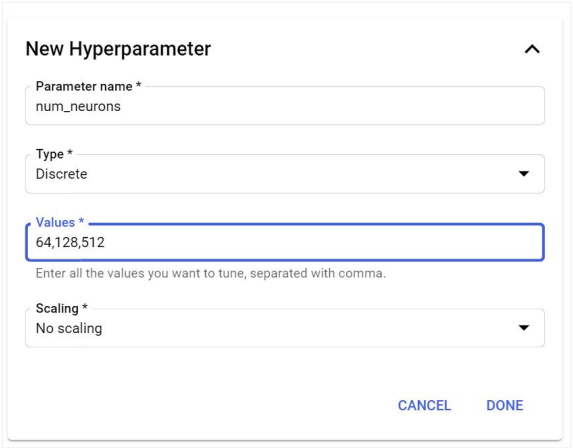


Source: Machine Learning on Google Cloud v3.0

- For **Parameter name**, enter momentum.
- For **Type,** select **Double**.
- For **Min**, enter 0, and for **Max**, enter 1.
- For **Scaling**, select **Linear**.
- Click **DONE**.

Number of neurons

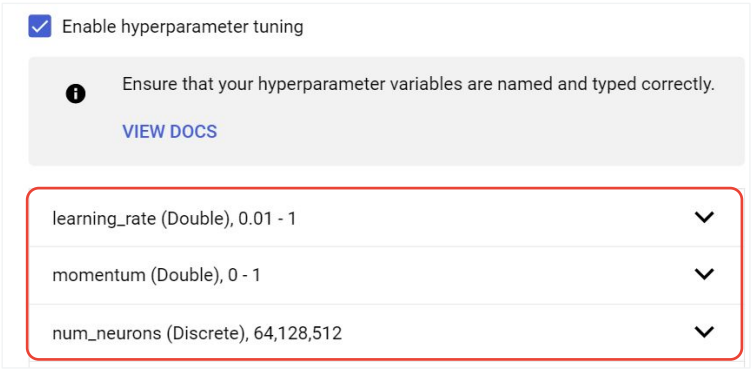
- Click **ADD NEW PARAMETER**
- For **Parameter name,** enter `num_neurons`
- For **Type,** select **Discrete**
- For **Values,** enter `64`, `128`, `512`
- For **Scaling**, select **No Scaling**
- Click **DONE**



Source: Machine Learning on Google Cloud v3.0

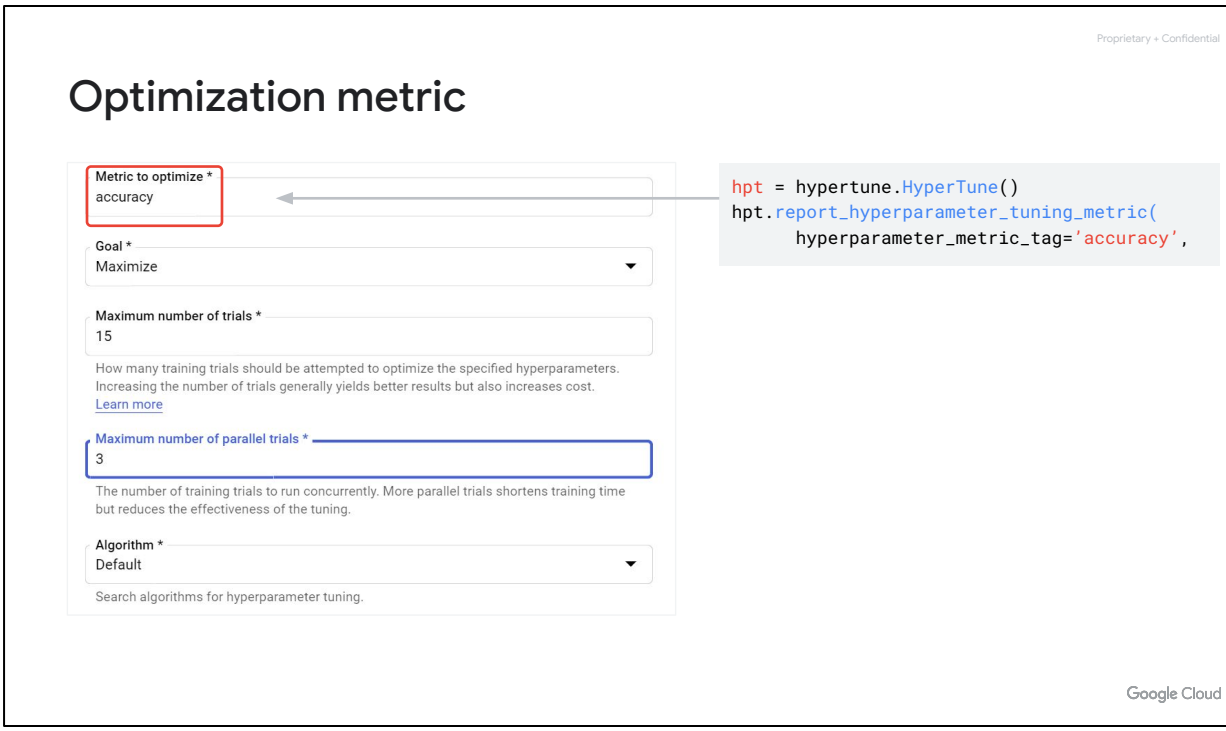
- For **num_neurons**:
  - Click **ADD NEW PARAMETER**.
  - For **Parameter name,** enter num_neurons.
  - For **Type**, select **Discrete**.
  - For **Values,** enter 64, 128, 512.
  - For **Scaling,** select **No scaling**.
  - Click **DONE**.

Three configured hyperparameters



Source: Machine Learning on Google Cloud v3.0

To summarize, the three hyperparameters you configured are: learning_rate, momentum, and the number of neurons.

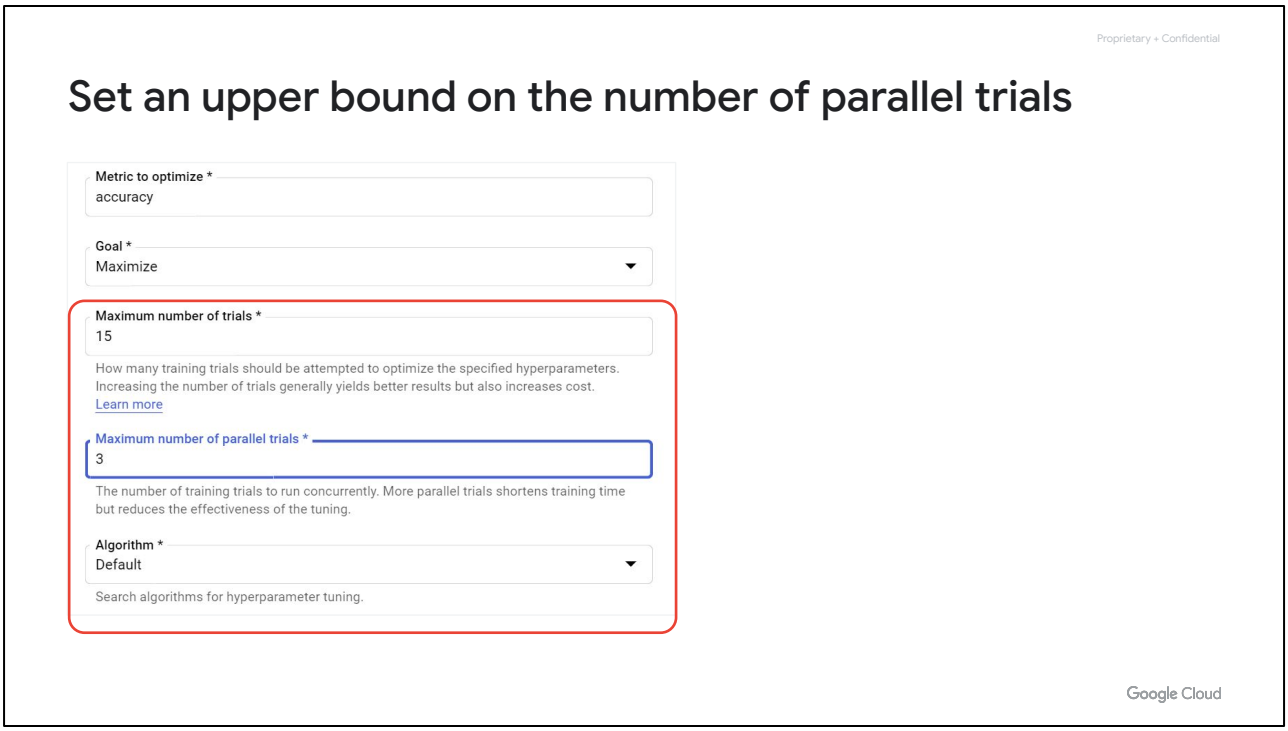


Source: Machine Learning on Google Cloud v3.0

After adding the hyperparameters, you'll next provide both the metric you want to optimize and the goal.

1. For **Metric to optimize,** enter **accuracy**.
2. For **Goal,** select **Maximize**.

The Vertex AI Hyperparameter tuning service will run multiple trials of your training application with the values configured in the previous steps. You'll need to put an upper bound on the number of trials the service will run. More trials generally leads to better results, but there will be a point of diminishing returns after which additional trials have little or no effect on the metric you're trying to optimize. It is a best practice to start with a smaller number of trials and get a sense of how effective your chosen hyperparameters are before scaling up to a large number of trials.



Source: Machine Learning on Google Cloud v3.0

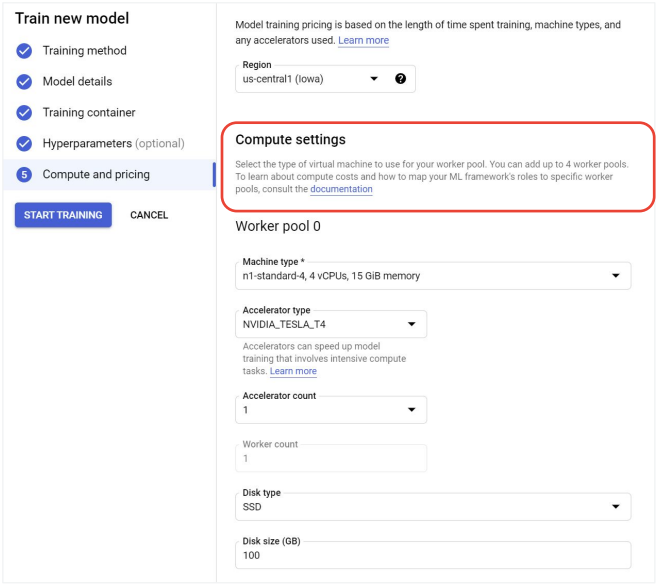
You'll also need to set an upper bound on the number of parallel trials. Increasing the number of parallel trials will reduce the amount of time the hyperparameter tuning job takes to run; however, it can reduce the effectiveness of the job overall. This is because the default tuning strategy uses results of previous trials to inform the assignment of values in subsequent trials. If you run too many trials in parallel, some trials will start without the benefit of the result from the trials still running.

3. For demonstration purposes, you can set the **Maximum number of trials** to 15 and the **maximum number of parallel trials** to 3. You can experiment with different numbers, but this can result in a longer tuning time and higher cost.

Select **Default** as the **Algorithm**, which will use Google Vizier to perform Bayesian optimization for hyperparameter tuning. Click Continue.

Configure compute

Note:
Using the GPU is completely optional. The hyperparameter tuning job will just take a little longer to complete if you do not use the GPU.



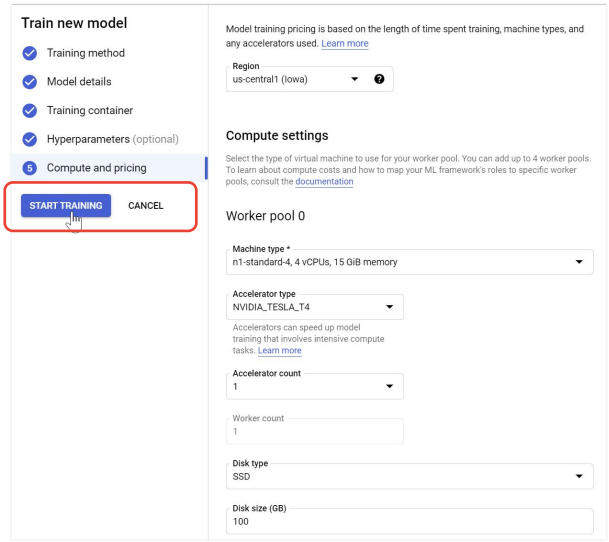
Source: Machine Learning on Google Cloud v3.0

On the **Compute and pricing** page, leave the selected region as-is and configure **Worker pool 0** as follows:

- For **Machine type**, select **Standard** > **n1-standard-4**.
- For **Accelerator type**, select **NVIDIA_TESLA_T4**.
- For **Accelerator count**, select **1**.
- For **Disk type**, select **SSD** .
- For **Disk size (GB)**, enter 100.

Start training

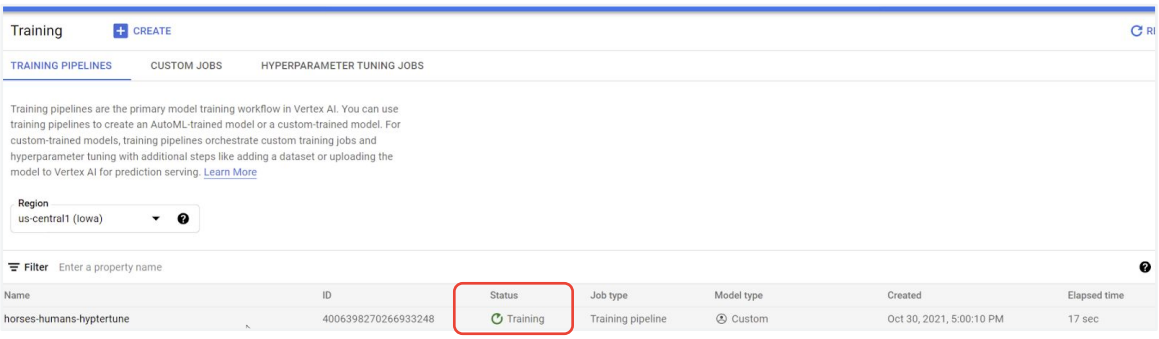
Note: The hyperparameter tuning job will take about 30-40 minutes to complete.



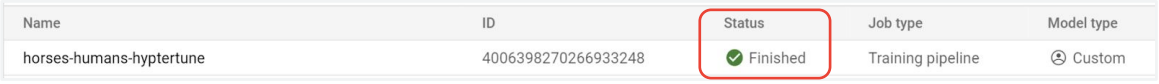
Source: Machine Learning on Google Cloud v3.0

Click “Start Training” to kick off your training job.

Training started



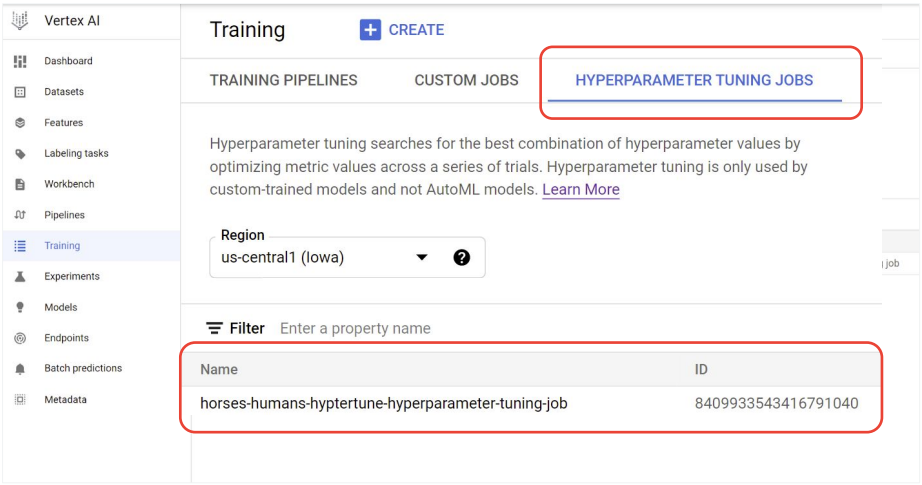
Training finished



Source: Machine Learning on Google Cloud v3.0

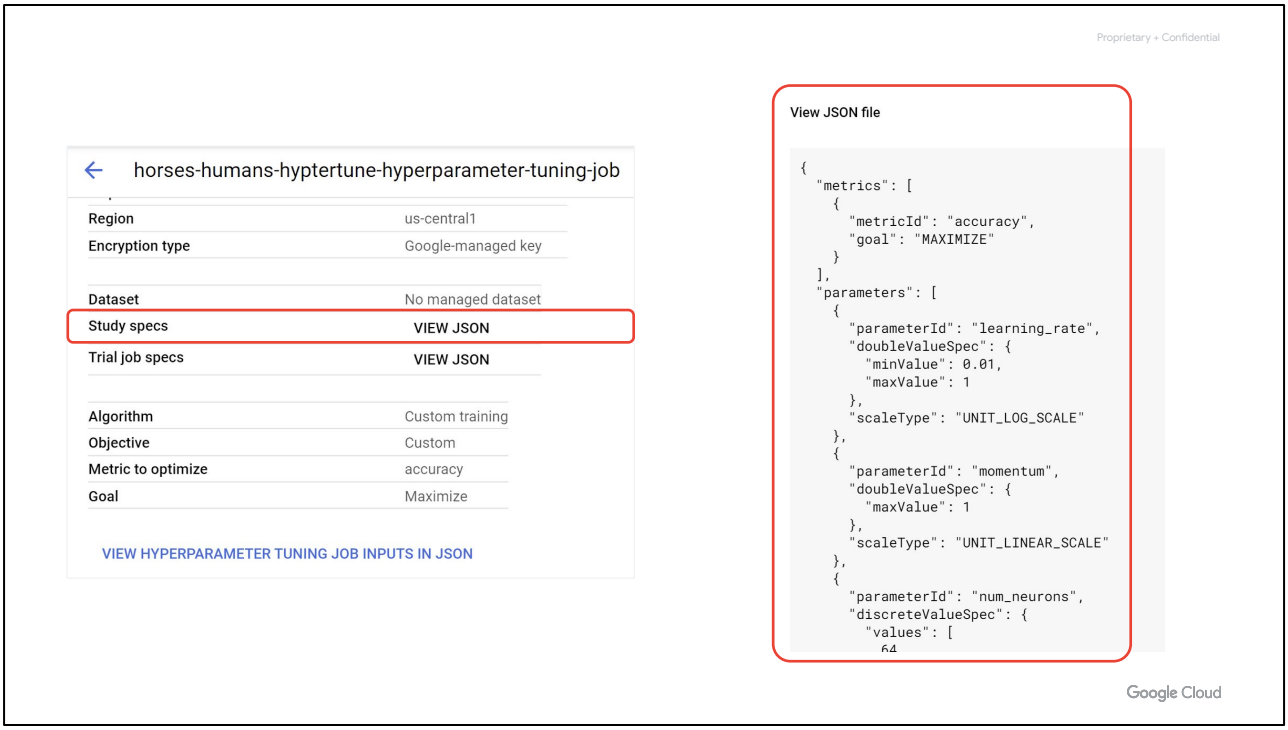
Note that training has started, which is indicated by the ‘in progress” training indicator on the model line. After training is completed, a green check mark “finished” appears on the model line.

Select hyperparameter training jobs



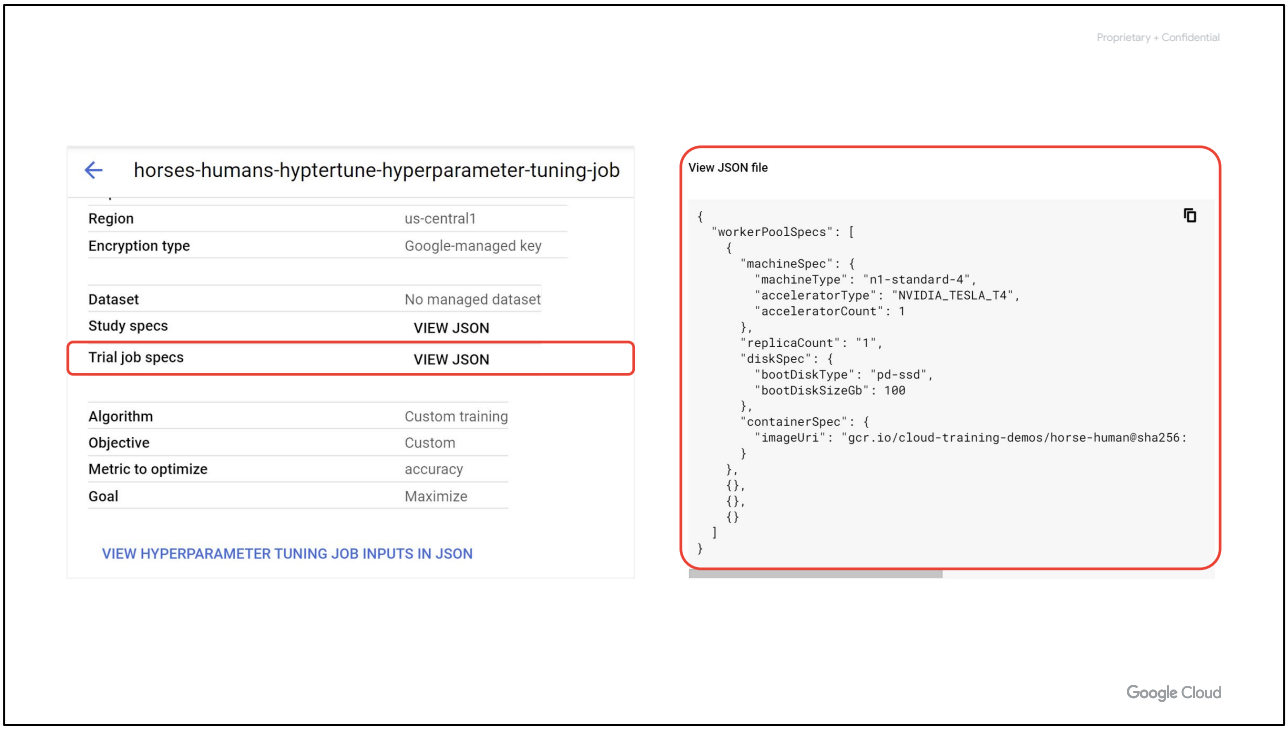
Source: Machine Learning on Google Cloud v3.0

To see the results of your training job, select hyperparameter training jobs and then click on the job.



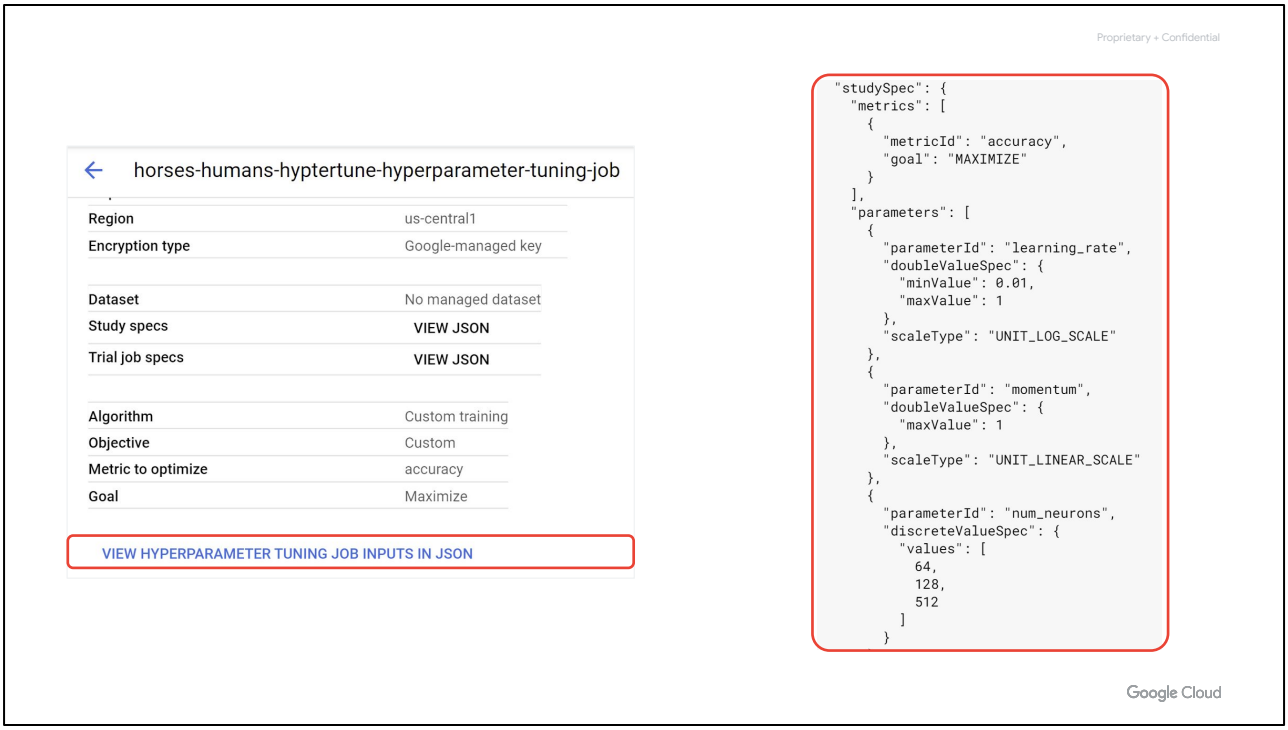
Source: Machine Learning on Google Cloud v3.0

The study specs can be viewed in JSON.



Source: Machine Learning on Google Cloud v3.0

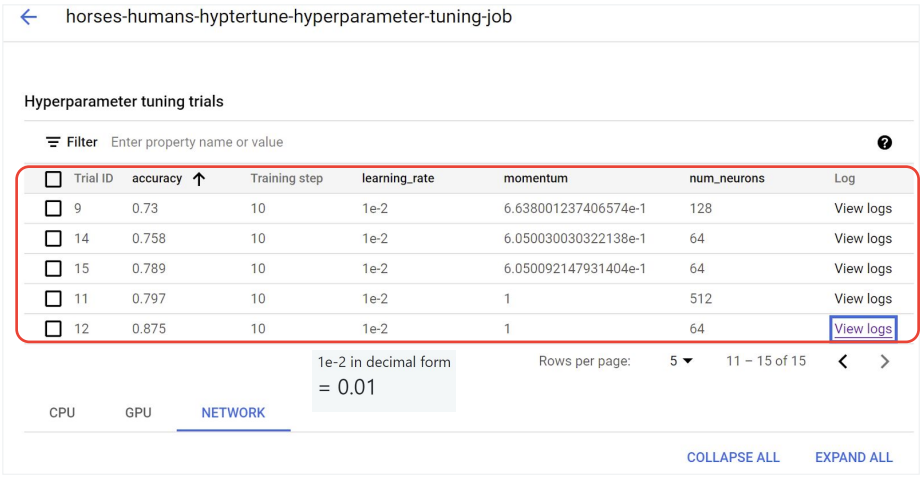
And also the trialspecs.



Source: Machine Learning on Google Cloud v3.0

You can also view the inputs in JSON.

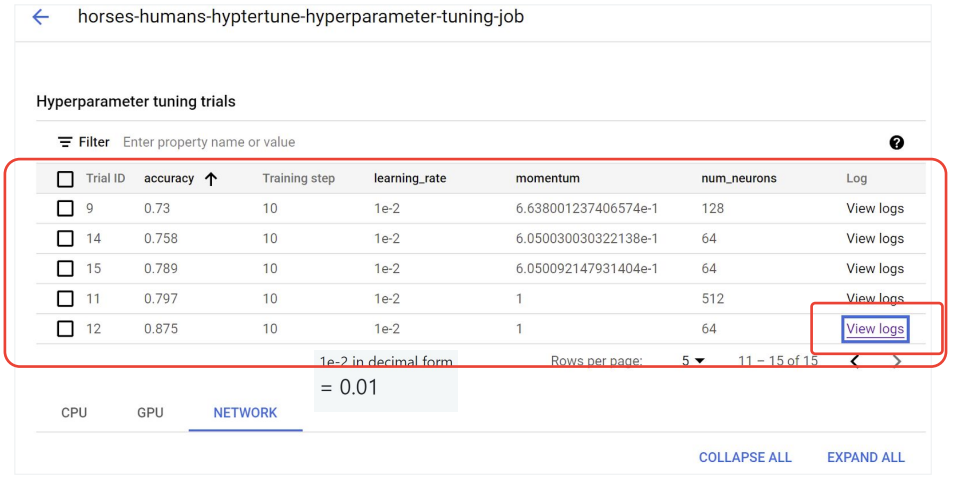
Hyperparameter results



Source: Machine Learning on Google Cloud v3.0

Here are the hyperparameter results. Note that setting the learning rate to .01, momentum to 1, and the number of neurons to 64 results in the highest validation accuracy.

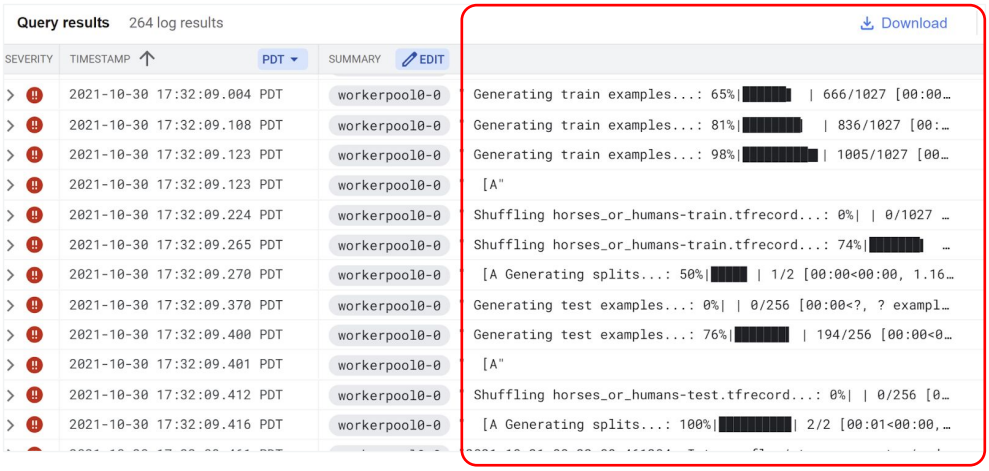
Hyperparameter results



Source: Machine Learning on Google Cloud v3.0

Click view logs of the trial ID to see details of the training job.

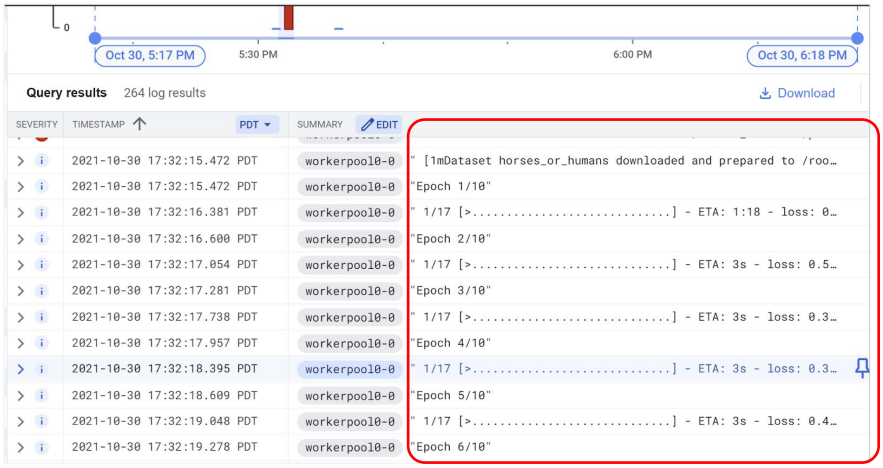
Training job logs



Source: Machine Learning on Google Cloud v3.0

The logs show the “progression” through the training. Training data is generated and split. Tested data is generated and split.

Training job logs



Source: Machine Learning on Google Cloud v3.0

The number of epochs are also shown.

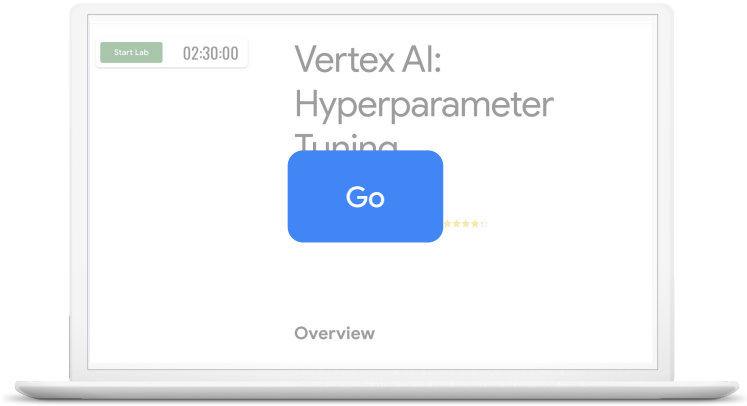
That completes our walkthrough.

Recommended Lab

Vertex AI: Hyperparameter Tuning

From the Course
Machine Learning in the Enterprise

Select this lab from the section "Vertex Vizier Hyperparameter Tuning"



Questions and answers

Thank you for attending this training!

We love your feedback! Please take a minute to complete the survey and help us improve our courses.


Google Cloud